



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

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.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

A MODEL AND A METHODOLOGY FOR DISTRIBUTED DEBUGGING

Ioakim Hamantzoglou

A Thesis

in

The Department

of

Computer Science

**Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada**

July 1988

© Ioakim Hamantzoglou, 1988.

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author, (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-44825-3

ABSTRACT

A MODEL AND A METHODOLOGY FOR DISTRIBUTED DEBUGGING

IOAKIM HAMAMTZOGLU

Reliability is one of the main motivations for the development of distributed systems. Software reliability is an important part of system reliability. One approach to reliable software is fault intolerance, whose objective is the prevention of fault occurrence. Fault intolerance advocates the use of a well-planned process for software development. Debugging is an essential element of this process. Given that we have evidence of an error, debugging is the activity of diagnosing the cause of the problem and correcting the software to remove it.

Debugging in distributed systems poses new problems, due to true concurrency, system complexity and lack of total control. These problems make traditional debugging techniques inadequate for distributed programs. In this thesis, we discuss problems related to distributed debugging, and propose solutions in the framework of a model for distributed computations based on partial ordering. We present an approach for distributed debugging, based on the comparison of actual system behavior against its synchronization specification. Finally, we propose and discuss the use of certain debugging tools, and present the high level specifications of a distributed debugger.

To my parents

Lisa and Minas

ACKNOWLEDGEMENTS

I would, first and foremost, like to thank my supervisor Professor H.F. Li, for his guidance, inspiration and support. Our many discussions proved invaluable, in bringing this thesis to fruition. To Chris, my special thanks for his help, interest, and above all, his friendship. I am very grateful to Professor T. Radhakrishnan for various discussions and advice. Thanks are also due to all members of the Distributed Computing Group, for many challenging and thought-provoking discussions.

Last but not least, thank you Gina for your patience and understanding.

Table of Contents

1. INTRODUCTION	1
1.1 Distributed Systems	1
1.2 Software Reliability	3
1.3 Distributed Debugging	4
1.4 Thesis Outline	6
2. THE ROLLBACK AND RECOVERY KERNEL	8
2.1 The problem of Rollback and Recovery	8
2.2 The VLR Rollback and Recovery Algorithm	12
2.3. The Application Program	15
2.4 The Rollback and Recovery Kernel	18
2.4.1 Design	18
2.4.2 Functional Description	22
2.4.3 Implementation	25
2.5 Global System Information and Distributed Debugging	27
3. MODELS FOR DISTRIBUTED COMPUTATIONS	29
3.1 Model of the Distributed System	29
3.2 The Space-Time Model	30
3.3 The Pomset Model	33
3.3.1 Definitions	35
3.3.2 Model Interpretation	36
3.4 Comparison of the Two Models	40

4. DISTRIBUTED DEBUGGING	42
4.1 Sequential Debugging	42
4.2 Distributed Debugging	44
4.3 Error Classification	47
4.4 Debugging Strategy	48
4.5 Probe Effect	53
5. THE DISTRIBUTED DEBUGGER	57
5.1 Channel Counters	58
5.1.1 Description of Channel Counters	59
5.1.2 Use of Channel Counters	64
5.2 Process Termination Conditions	69
5.3 Timestamps	74
5.3.1 Description of Timestamps	76
5.3.2 Use of timestamps	85
5.4 Implementation Issues	86
5.5 High Level Specifications of the Distributed Debugger	88
5.6 Related Work	97
6. DEBUGGING TWO DISTRIBUTED ALGORITHMS	101
6.1 The VLR Global State Detection Algorithm	102
6.2 A Message Routing Algorithm	113
7. CONCLUSIONS	122
7.1 Suggestions for Future Work	125

REFERENCES	128
----------------------	-----

LIST OF FIGURES

Fig. 2.1.1:	Domino effect	11
Fig. 2.4.1:	The design of the RRK	21
Fig. 3.2.1a:	A three-process DCS	32
Fig. 3.2.1b:	ST diagram of a computation	32
Fig. 5.1.1:	Pomset prefix and corresponding values of channel counters	63
Fig. 5.1.2:	Pomset prefixes	66
Fig. 5.1.3:	Prefixes in the pomset of a composite process	68
Fig. 5.2.1:	Process termination problems	71
Fig. 5.3.1:	ST diagram cuts corresponding to values the same channel counters values	75
Fig. 5.3.2:	ST diagram and event timestamps	77
Fig. 5.3.3:	Timestamp windows and corresponding ST diagrams	84
Fig. 5.5.1:	Basic block of a structure chart	89
Fig. 5.5.2:	Structure chart of the distributed debugger	90
Fig. 5.5.3a:	Structure chart of the master debugger	91
Fig. 5.5.3b:	Structure chart of the master debugger (cont)	92
Fig. 5.5.3c:	Structure chart of the master debugger (cont)	93
Fig. 5.5.4a:	Structure chart of a node debugger	94

Fig. 5.5.4b:	Structure chart of a node debugger (cont)	95
Fig. 5.5.4c:	Structure chart of a node debugger (cont)	96
Fig. 6.1.1:	Fully and partially connected DCSs . . .	106
Fig. 6.1.2	Invalid and valid ST diagrams for the VLR algorithm (scenario 1).	110
Fig. 6.1.3	Reconstructed ST diagram for the VLR algorithm (scenario 2)	114
Fig. 6.2.1:	Reconstructed ST diagram for the message routing algorithm	121

LIST OF ABBREVIATIONS

BSD	Brekley Software Development
CC	Channel Counter
CCP	Current Checkpoint vector of a Process
DCS	Distributed Computer System
DE	Domino Effect
EDM	Error Detection Module
FIFO	First In First Out
ICB	Input Channel Buffers
LAN	Local Area Network
ND	Node Debugger
OS	Operating System
PE	Probe Effect
POMSET	Partially Ordered MultiSET
RC	Response Checkpoint
RCP	Received Checkpoint vector of a Process
RR	Rollback and Recovery
RRK	Rollback and Recovery Kernel
SIC	Self Induced Checkpoint
SRS	Synchronization Requirement Specification
ST	Space Time
STV	Sorted Timestamp Vector
TV	Timestamp Vector
VLR	Venkatesh, Li, Radhakrishnan (State Detection Algorithm)

CHAPTER 1

INTRODUCTION

1.1 Distributed Systems.

A recent trend in computer systems is to distribute computation among several physical processors. This trend is accelerated by technological advances in the fields of microelectronics and communications. Although the amount of distribution varies over a wide spectrum, two schemes are usually employed for building such systems. The first results in systems that comprise several processors that share memory and a clock. Communication takes place through the shared memory. These systems are known as tightly coupled systems.

The second scheme effects the so-called loosely coupled systems, in which processors do not share memory. Instead, they have their own local memories. Communication is achieved via message passing, through various kinds of communication lines, such as high-speed buses, or telephone lines. Such systems are usually referred to as Distributed Computer Systems, or simply Distributed Systems. This is the kind of systems we consider in this thesis.

There are four major reasons for building distributed systems: computation speed-up, resource sharing, increased reliability, and communication. The first objective, that is computation speed-up, can be achieved when a particular computation is divided into a number of concurrently executing subcomputations. A distributed system allows us then to distribute the computation among the various sites, to run it concurrently.

This last observation introduces the notion of a process. Each subcomputation corresponds to a process. A distributed system can be viewed then, as a collection of processes that communicate with each other. User processes share several resources (files, disks, peripherals, etc.) which are managed by operating system, or server, processes. We have used this particular model to depict a distributed system: a DCS is a collection of concurrently executing, communicating processes.

Increased reliability is another motivation for the development of distributed systems. DCSs can provide higher reliability due to their potential to offer greater redundancy. Consequently, when one site fails the remaining sites can potentially continue operation. Reliability in computer systems, however, has two aspects: hardware and software reliability. The latter is one of the most costly performance characteristics to assess, and possibly the most difficult to guarantee.

1.2 Software Reliability.

Reliability is one of the least precise aspects of software scope. Quite a few software reliability measures have been proposed, but they are still considered to be in their developmental stage. It is usually, the nature of software that dictates special measures to ensure reliability. For example, software in an air traffic control system must not, under any circumstances, fail, or human life may be lost. On the other hand, less critical application software, such as an inventory control system, should not fail either, but the impact of failure is considerably less dramatic. Regardless of the application, reliability is considered an essential software characteristic.

Software is increasingly considered as a system element. The costs associated with a software failure are motivating forces for well-planned development. A great deal of software failures can be attributed to residual design faults. Two approaches have been adopted in dealing with such faults: fault intolerance and fault tolerance. The former aspires to the development of errorless programs, while the latter is concerned with acceptable software performance, in spite of the presence of faults.

Both strategies have influenced software engineering. The fault intolerance approach has generated techniques that became standard steps in the process of software development. On the other hand, fault tolerance research has produced facilities that

are embodied in modern programming environments. For example, testing and the invariably following debugging, are considered necessary and planned steps of the software development phase. Likewise, backward error recovery is a well-known method of implementing software fault tolerance, that is incorporated in many computing environments.

There are, however, new issues that must be addressed, before rollback/recovery and debugging are applied in distributed computing. Such issues, that mainly relate to debugging distributed programs, are discussed in this thesis.

1.3 Distributed Debugging.

Almost everyone who has written, even simple programs, has learned something about debugging them. The presence of program errors is an accepted fact between programmers and users alike. The release of flawless software is still considered as a hopelessly optimistic endeavour.

Debugging is an integral part of the software testing phase, and can be described as the process of determining the location of errors and removing them. Finding the cause of a single failure may consume many labour-hours. Therefore, debugging should be of acute interest to anyone concerned with improving programming productivity.

In the literature the use of terms, such as error, fault, and bug, tends to be perplexed. We present the following

definitions, offered in the IEEE Software Glossary, that help clarify their meaning[8]:

Error: A conceptual, syntactic, or clerical discrepancy which results in one or more faults in the software.

Fault: A specific manifestation of an error. A discrepancy in the software which can impair its ability to function as intended. An error may be the cause of several faults.

Finally, a software failure is described as the result of a fault. In the above definitions, we discern the following causal relationship: errors create faults that cause failures. We use the term "bug" as a synonym for error. Following the above clarification, we can define debugging as the process of locating and removing errors, that have manifested themselves by producing faults. Nobody claims that debugging removes all the bugs that exist in a program. The number of discrete states, that even simple programs are capable of attaining, is so enormous, that it is almost inevitable that undetected bugs will remain, even after exhaustive testing and debugging.

Debugging is still regarded an art, by many programmers. This is attributed mainly to the fact that often, the external manifestation of the error (i.e. fault), and the internal cause of the error (i.e. bug), may have no obvious relationship to each other. Hence, debugging is described as "the poorly understood mental process that connects a symptom to a cause of a software

problem" [22]. While this definition may sound true, and appeal to common experience, we believe that if a more formal approach is adopted, debugging can become a more systematic task, and less of a chaotic process. This is especially true in distributed computing.

Distributed programming poses new problems with respect to debugging. They are caused by concurrency, increased complexity, and absence of total control. Their solution requires a new approach, as traditional debugging methods are simply inadequate for distributed programs. Discussion of any debugging method inadvertently leads to the consideration of the debugging tools, that will support it. Debuggers are tools incorporated in the programming environments, and therefore they are affected by distribution.

1.4 Thesis Outline.

In this thesis, we are mostly interested in the process of debugging. The main focus of the thesis is to identify the problems of distributed debugging and propose a solution. The effects of the proposed debugging strategy in the debugging facility are discussed, and high level specifications of a distributed debugger are presented.

The practical experience, that we acquired from the implementation of a distributed algorithm, is presented in chapter 2. The algorithm is a rollback and recovery algorithm

for DCSS. The main aspects of the algorithm are discussed, along with the design of the Rollback and Recovery Kernel (RRK), that is the software package that implements the algorithm. We also describe the design and implementation of a distributed application program, that was used as a test-bed for the performance evaluation of the RRK.

The models that we have used to represent a distributed computation are presented in chapter 3. The main differences between sequential and distributed debugging, the problems of the latter, and their proposed solutions are discussed in chapter 4. The features of a distributed debugger and their capabilities are analyzed in chapter 5. A few example uses of the debugging strategy and the tools of the debugger are offered in chapter 6. We end the thesis, with our conclusions and suggestions for further work, in chapter 7.

CHAPTER 2

THE ROLLBACK AND RECOVERY KERNEL

Practical experience in the design and implementation of distributed programs is not as much as we have for sequential programs. In this chapter, we describe the implementation of a distributed algorithm. Our work was carried out as part of a project that is aiming at the development of a fault tolerant distributed programming environment. The implementation of the rollback and recovery algorithm and its performance evaluation were one of the project objectives. An extended description of the design, implementation, as well as performance evaluation of the algorithm, appears in [18].

2.1 The Problem of Rollback and Recovery.

Increased system reliability is one of the motivations for the development of distributed systems. One approach to higher system reliability is fault tolerance. Fault tolerance can be informally characterized, as a system's capability to continue functioning, even at reduced capacity, in the presence of faults. Distributed systems are intrinsically more complex than

conventional systems with centralized control. This inherent complexity makes them more susceptible to faults.

A large class of system failures is due to design faults. These errors are difficult to deal with as they are unanticipated. Usually, they are handled by discarding the current system state and restoring the system to a previously saved correct state. Rollback and Recovery (RR) [2], is a well known method for preserving the integrity and consistency of fault-tolerant systems. The basic concept is that at various intervals, the system records its state in stable storage (checkpoints). When a fault is detected this previous consistent state can be regenerated and the system resumes execution from that point. In systems under centralized control, it is easy to arrange that a checkpoint of the system state is established at convenient points in time. This cannot be achieved so easily in distributed systems with decentralized control.

In distributed systems, there is no global clock. Consequently, we cannot freeze all processes simultaneously to record the individual component states. Moreover, single processes can no longer be viewed in isolation. An error in one process can possibly contaminate the state of other processes, via message passing. In other words, errors spread in the system. This results in the rollback of one process forcing the rollback of other processes too. Hence, in distributed systems, we must identify, for all processes, a set of process states that constitute a consistent global state. This state is called

recovery line, and in case of an error the whole system can be rolled back to it.

There are two fundamental strategies for approaching the problem of determining a recovery line: the preplanned and the unplanned. Distributed rollback and recovery algorithms are classified into two categories, depending on which of the two methods is used. In the preplanned strategy, checkpoints are recorded according to certain rules, that ensure that all checkpoints belong to a recovery line. In the unplanned strategy, the system tries to deduce a recovery line at the time of rollback. Planned strategies result in a clean system structure at the expense of processing speed and generality of communications. Unplanned strategies sacrifice ease of recovery for speed and generality of communications.

Domino Effect.

A main disadvantage of the unplanned strategies is the so called Domino Effect (DE). DE can be defined as the uncontrollable rollback of all processes. We can illustrate DE with the example of Fig. 2.1.1. In the example, two processes, namely p and q , have independently taken a sequence of checkpoints. The interleaving of messages and checkpoints leaves no consistent set of checkpoints for p and q , except the initial one (X_0, Y_0) . Consequently, if p fails at the designated point, both processes have to be rolled back to the beginning of their

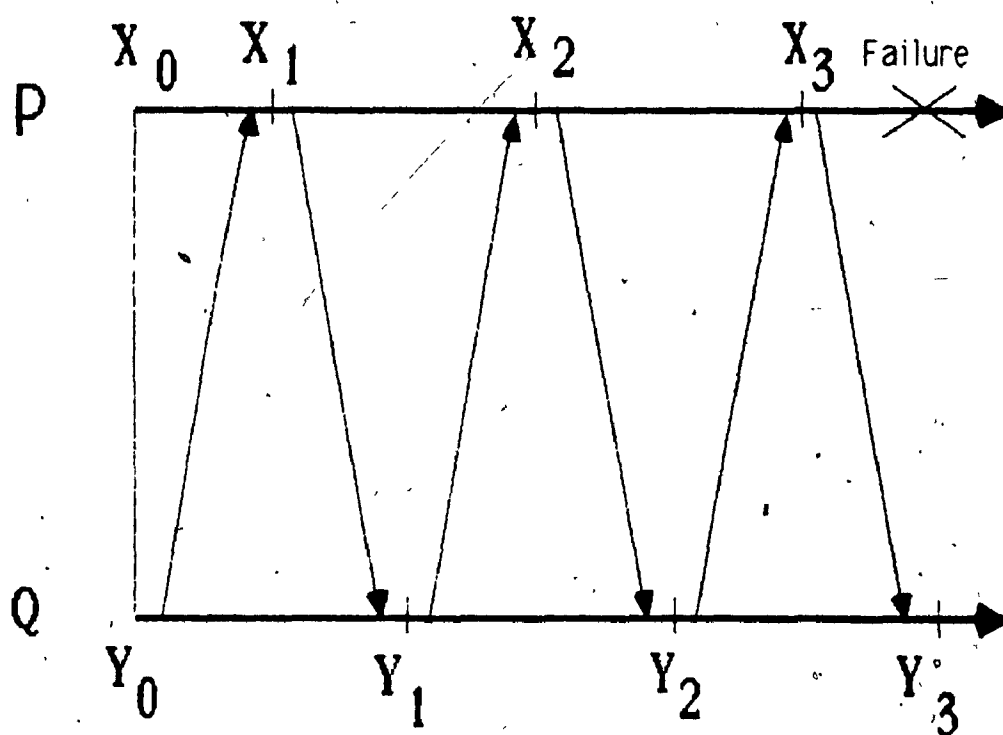


Fig. 2.1.1: Domino effect

computation.

For time critical applications, that require a guaranteed progress rate, DE is an unacceptable behavior. DE is also undesirable in time consuming computations, where we cannot afford to restart the computation from the beginning, each time a fault occurs.

2.2 The VLR Rollback and Recovery Algorithm.

The VLR algorithm was developed by Venkatesh, Li and Radhakrishnan [23]. It belongs to the preplanned category. The algorithm makes use of global information to coordinate the set up of checkpoints. It places no restriction on the application program, to which its operation is transparent. Hence, it can be characterized as a non-intrusive algorithm, as far as checkpointing is concerned. It also ensures DE avoidance. The computing environment is regarded by the algorithm as a collection of asynchronously communicating processes. Communication is accomplished via reliable, FIFO channels. The algorithm is divided in two logical phases: checkpointing and rollback.

Checkpointing.

The algorithm establishes, on behalf of the application process, two kinds of checkpoints:

(i) Self Induced Checkpoints (SICs) which are established following an explicit request from the application process, based on its own local requirements.

(ii) Response Checkpoints (RCs) which are induced by information imported from another process. RCs are said to belong to recovery lines owned by other processes.

All checkpoints are labelled with the following tag: <owner process-id, ckcpnt-id>. Each process¹, counts its SICs with a monotonically increasing counter. It also maintains an N-element vector, N being the number of processes, where it keeps the number of the latest known checkpoint of every other process. This is called the Current Checkpoint Vector (CCP), and it is appended to every application message that is sent out by the process. It represents global state information, as perceived by that process.

Upon receiving a message process P_i first strips the Received Checkpoint Vector (RCP) and compares it with the local CCP. Whenever there are differences, the process updates its CCP as follows. If $RCP(j) > CCP(j)$, for any j , then $CCP(j)$ is set to $RCP(j)$, and a new RC is created and tagged with the label: < $j, RCP(j)$ >. Process P_j is considered the owner of this RC. This way a SIC causes the creation of a RC. On the other hand, when

¹ In the following, process refers to the composition of the application process and the VLR algorithm process.

$RCP(j) < CCP(j)$ this message intersects one, or more, recovery lines, and is stored in all local checkpoints owned by P_j and having an ordinal number greater than $RCP(j)$. If the process is later rolled back to any of those checkpoints, this message will be played back.

Rollback and Recovery.

When a fault is diagnosed, process P_i identifies the most recent of its SICs, say k , that precedes the occurrence of the fault. It loads the state saved in that SIC, deletes all later checkpoints, and loads all messages saved in that checkpoint to its input buffers. It then sends out a specific message $\langle i, k \rangle$, called recovery message, to all its output channels, to inform other processes of its intention to rollback. This process is the initiator of the rollback. The application process is then allowed to resume execution. When another process, say P_j , receives a recovery message, it searches for its r_{th} RC, owned by P_i , where r is the first integer greater than, or equal to, k . If such a checkpoint exists, then process P_j will participate in the rollback. This process performs the same actions as the initiator, the only difference being that it does not send the recovery message to the process that informed it about the rollback.

Another thing that must be taken care of, is purging all prerollback messages. Those messages were the result of

computations that preceded the fault and will be redone. To recognize such messages each process sets its input channels, except the one on which it was informed about rollback, to cautious state. While a channel is on cautious state, the process examines the RCP of arriving messages and purges those for which $RCP(i) \geq k$. When the process receives the recovery message $\langle i, k \rangle$ on that channel, it resets it to the normal state.

A final remark concerning the rollback phase of the algorithm, stems from the observation that in large DCSSs it is probable that more than one processes initiate rollback independently. The VLR algorithm takes care of this situation, by defining the, so called Effective Recovery Line (ERL), which identifies the checkpoint to which each process must be rolled back, as well as the set of messages that should be played back.

2.3. The Application Program.

To evaluate the performance of a RR algorithm, a distributed application program is needed to provide the required testing ground. The program, used in our case, is a simple simulation game, based on the well known "Game of Life" proposed by J.H. Conway [13]. The basic concept of the game is the simulation of the lives of cellular automata, that live and die following a few simple rules. The game is normally played in a two-dimensional grid. Each grid location called a cell, or a pixel, can be either dead or alive. The game was modified to yield a

distributed version, called "N-life".

In "N-life" the grid is partitioned into several adjoining regions, whose control is distributed over several processes. Each process controls the live and death of cells in its own region(s), communicating with its neighboring processes, to import and export information concerning boundary cells. Patterns generated in one region can be expanded to cover other regions too, or they can migrate to other regions retaining their form, and thus travel through the grid, since wrap around connections are employed.

The communication patterns of the program can be easily manipulated, by changing the initial patterns and modifying the task assignment to different processes. Hence, many different message passing behaviors can be realized. This is a worthwhile feature of the application program, that allowed the evaluation of the algorithm performance under variable communication behaviors.

Implementation.

Implementing "N-life" proved a worthy experience and offered valuable insight in the problems of distributed programming. The program of this algorithm is straightforward and its synchronization requirements are simple. The algorithm can be classified as "loosely synchronous", since processes must normally synchronize through various iterations.

The program is partitioned into a set of identical processes, distributed over the physical system. For simplicity there is one process per system node. The number of processes depends on the selected partition of the grid and the particular task assignment that is exercised. A process can be responsible for up to three different regions. Both the number of processes and the topology of process interconnection remain static during a particular execution of the program.

Each process is composed of two communicating layers. The inner layer is responsible for the computation of the algorithm on every iteration, based on the grid image of the previous iteration. The outer layer is responsible for the communication of the process with other processes. It sends all boundary information to neighbor-processes, so that they can calculate the algorithm in their own regions. It also receives any incoming messages and forwards them to the inner layer.

Communication between processes can be realized in different modes, which include: sending out the value of every boundary pixel as a separate message, packing up groups of cells in one message or even communicating asynchronously, only when there are live boundary cells. When the last mode is used, processes that must communicate at a certain iteration, must be synchronized. This may result in temporary blocking of faster processes, that have to wait for slower ones. The interaction between processes ensures deadlock avoidance, by forcing every process to send out any information at the beginning of an iteration, and then wait

for incoming information.

The system that was used for the implementation, was a network of twelve Sun 3/50M-4 workstations, connected through an Ethernet LAN. The OS was an enhanced version of 4.2BSD Berkeley UNIX, and inter-node communication was realized through the "socket" primitive of the internet domain. The code is written in C programming language.

2.4 The Rollback and Recovery Kernel.

2.4.1 Design.

The design and implementation of the Rollback and Recovery Kernel (RRK) demonstrated how tricky distributed programming can be. The implementation of even simple and well-developed algorithms reveals a lot of the implications of process synchronization. The algorithm, in its development phase, was conceived as part of the existing OS kernel. Although the final implementation does not conform to the original view, we retained the name kernel, to refer to the software that implements the VLR algorithm.

The RRK must offer its services to user processes, while at the same time it must be transparent to them. There exist several alternatives in designing such a software package. Three different ways of incorporating the RRK in the system were considered:

1. The first approach is to have the RRK as a part of the OS. It would be added to the UNIX communication layer, where it would have easy access to all messages transmitted through the LAN. This approach would result in minimal overhead and transparency from application processes. On the other hand, it would require modification of the UNIX source code, a major undertaking which was deferred to a later stage. Moreover, this approach is not flexible for the experimental comparison of many RR algorithms.

2. The second alternative is to have the RRK as a separate process on each system node. This would allow the RRK to offer its services to all application processes running on that node. This approach would lead to a very complex design, since the interprocess communication would have to be redirected through the RRK process. Such a modification would be hard to hide from the application processes. Additionally, certain UNIX constraints (such as number of file descriptors that a process can maintain at any time) would limit the RRK support to a small number of processes.

3. The last approach that was considered and finally adopted, was to include the RRK to the code of each application process. The RRK is considered by the application process as a communication support package, that offers new communication primitives. This is not an ideal design, since it is not robust. In case of a major system fault, that causes the abortion of the

application process, the RRK is lost too. It is, however, sufficient for the purpose of evaluating the performance of the algorithm. Moreover, once enough experience has been accumulated and the RRK has taken its final form, it can be moved to the OS layer.

The RRK can be easily incorporated in the application process. The user accesses the RRK, through call of certain communication and checkpointing primitives. The interface of the RRK with the application is clearly defined, and it is the only thing the application programmer should be familiar with in order to make use of the kernel. The RRK makes certain assumptions about the application program. The number of processes, as well as their interconnection topology, should remain static throughout execution. Furthermore, every application process has a unique identification number. All this information should be made known to the RRK.

The RRK package is partitioned into two modules: the checkpointing module and the rollback module, Fig. 2.4.1. The interaction of the two modules is briefly outlined in the following. Incoming messages are received by the rollback module, and filtered for existing control messages. If a message is not a control message, it is forwarded to the checkpoint module, through the Input Channel Buffers (ICBs). When an explicit read request is made from the application process, the checkpoint module extracts the first message from the ICB and processes it, performing all the actions dictated by the VLR

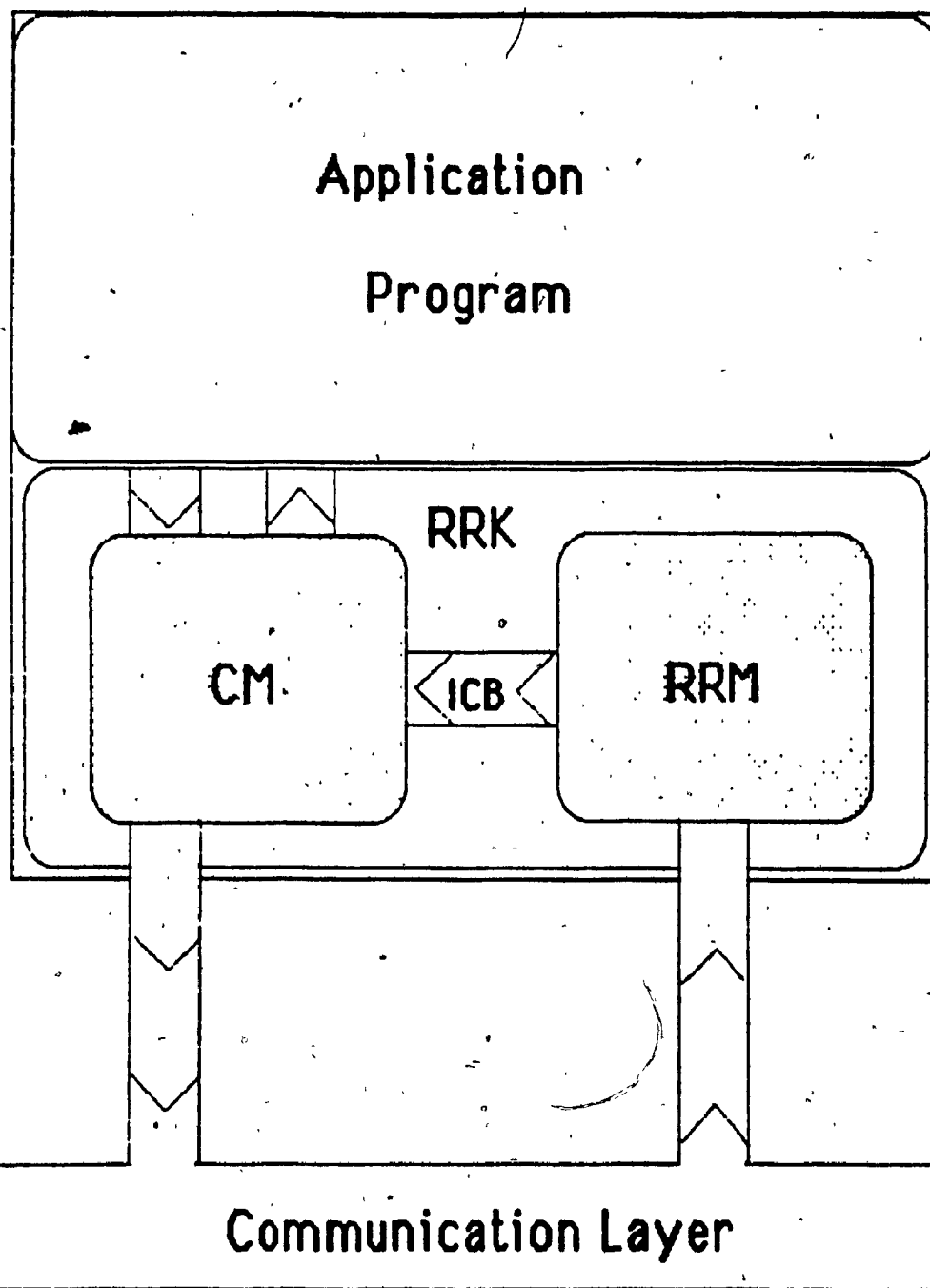


Fig. 2.4.1: The design of the RRK

algorithm. Finally, it delivers the application message to the application process. All outgoing messages pass through the checkpoint module, that stamps them with the local CCP vector.

The checkpoint module contains certain functions that must be rewritten, when a new application program is used with the RRK. These are mainly the functions that save and restore the application process state. Which information exactly constitutes the process state is, apparently, application dependant. The application programmer is responsible for writing and optimising those functions, respecting the RRK guidelines. The other alternative, would be a "brute force" solution, whereby the whole process state is saved with every checkpoint. This approach is obviously not economical.

2.4.2 Functional Description.

The RRK can be regarded as an intermediate layer, between the OS communication layer and the application program layer. It provides the application process with primitives for creating SICS and handling interprocess communication. It also establishes RCs, based on the message exchange pattern. Finally, it can initiate, upon request, rollback to a particular recovery line.

The services offered by the RRK can be divided into two broad categories:

(i) Services available to the application process, through explicit use of communicating and checkpointing primitives.

(ii) Operations which are performed by the RRK transparently and hence cannot be called from the application code.

The first category corresponds to checkpointing functions, while the second to rollback functions.

RRK interfaces.

The RRK interacts explicitly with the application process and the Error Detection Module (EDM), and implicitly with the OS, via system calls. The RRK's interaction with the application process is performed through explicit calls to three communication primitives and one SIC primitive. Its interaction with the EDM is achieved with software signals, that represent the detection of an error. This latter interface must be redefined more clearly, when a sophisticated EDM is developed.

Detailed description.

The RRK operates as a buffer layer, whose function is to intercept all messages. More precisely, message exchange between

two application processes is realized in the following manner:

1. The sender process delivers, to its own RRK layer, an application message.
2. The RRK appends to this message some RRK internal information, creating an RRK-message.
3. This message is forwarded to the OS for transmission through the LAN.
4. The message is received, at the destination node, by the receiver's RRK.
5. The RRK strips the information, that was attached to the message. Based on that information, it performs some operations invisible to the application process.
6. Finally, the receiver's RRK delivers the application message to the application process.

The RRK introduces a new type of messages, the control messages. Control messages are invisible to application processes. They inform the RRK about initiation of a rollback to a particular recovery line. They also help the RRK identify and discard obsolete messages after rollback. Although the processing of application messages is performed in FIFO order, control messages are processed as soon as they arrive. This preferential handling ensures that information concerning rollback is spread quickly through the system, and processes do not lose time performing redundant computations.

RRK services.

The RRK provides primitives for process interaction and checkpoint creation. It enables an application process to:

- Create SICs.
- Send and receive application messages.
- Check the ICBs, for the existence of awaiting messages.

In addition, the RRK performs the following operations, without explicit request:

- Creates RCs.
- Saves messages, that cross established recovery lines.

Finally, when the RRK is informed about a rollback the following actions are taken:

- Rollback the application process.
- Inform other processes about rollback.
- Discard obsolete messages, after rollback.

2.4.3 Implementation.

In its present form, the RRK is implemented as a set of

passive routines; it is not an active task. The RRK code is run on behalf of the calling process, which is the active task. As far as the OS is concerned, the RRK is part of the application process. The functions of the RRK are activated through calls from the application code, as a result of a message receipt, or as a result of a signal from the EDM.

The primitives offered by the RRK are: kread, kwrite, kselect and chckpnt. The first three perform the same actions with the UNIX primitives read, write and select, respectively. Actually, those UNIX primitives must be replaced in the application code, by their RRK namesakes, when the RRK is used. The last function, namely chckpnt, is called from the application code, whenever a SIC must be created.

Future directions.

As it was mentioned at the beginning of this chapter, the experimental work that was presented here, is part of an ongoing project. We are in the process of upgrading the physical system that was used. These changes will bring about modifications, but mainly on implementation aspects of the RRK. Hence, addition of local stable stores to the currently diskless workstations, will result in alterations in the communication subsystem and the checkpointing operations. Finally, it would be desirable to move the RRK into the communication layer, in view of a more robust and efficient implementation.

2.5. Global System Information and Distributed Debugging.

The experience we have with distributed programming is far less than that of sequential programming. The task of synchronizing a varying number of processes was proved to be harder than expected. The intricacies of synchronization may become the source of frustrating situations, especially during debugging.

Approximately eight months were spent for the design and implementation of the application program and the RRR. Almost fifty per cent of this time, was devoted to debugging and testing. For debugging we relied on existing tools, such as conventional sequential debuggers, which proved inefficient for distributed software. The notions of global state and message passing are unknown to sequential programs and hence to conventional debuggers. Furthermore, old concepts such as breakpoints and single-step execution, take on new meaning in the context of distributed programming. Implementing even simple distributed programs reveals the need for a more formal approach to distributed debugging.

The VLR algorithm is an example in the use of global system information for a particular application, namely rollback and recovery. Important events for RR are creations of checkpoints. The CCP represents each process' local picture of the system state. With every incoming message a process collects

information and updates its local picture. This information serves the purpose of tracking down message dependencies from checkpoint create events. It allows the coordination of checkpointing and identification of messages that will be resent, should a rollback occur.

The use of global information can be extended to other applications and distributed debugging is a suitable candidate. In distributed debugging, we must redefine the concepts of program state and execution history. The same basic idea, of attaching information to messages in order to track down event dependencies, can be employed. We need, however, information of finer granularity to be able to mark every significant event. Important events related to process synchronization are send and receive events. Using information about those events, we can completely characterize particular states in the execution of sequential processes. Moreover, compiling such local information from the constituent processes can help reconstruct the system execution history and identify particular system states.

The use of system-wide information and formal process synchronization requirement specification are the backbone of the distributed debugging strategy, that we propose in the next chapters.

CHAPTER 3

MODELS FOR DISTRIBUTED COMPUTATIONS

Models are necessary for the analysis of complex systems. They suppress irrelevant information, and thus offer the simplicity and abstractness that are necessary, in order to handle such systems. In this chapter we discuss the two models, that we used to represent a distributed computation: the Space Time model and the Pomset model. In the beginning, we state some assumptions we have made, concerning the operating characteristics of the distributed system,

3.1 Model of the Distributed System.

We depict a distributed system as a collection of concurrently executing processes. There are certain assumptions, regarding the operating characteristics of the distributed systems we have considered, which are:

- i. Processes communicate only via message passing, through point-to-point unidirectional channels.
- ii. Processes have at their disposal two kinds of communication primitives: send and receive.
- iii. Channels introduce a finite, variable and positive message delay.
- iv. Messages are delivered in a FIFO manner.

Assumption (iii) implies that channels are reliable. It also enforces the cause and effect relationship between the sending and the receipt of a message.

3.2 The Space-Time Model.

The Space-Time (ST) model, or diagram¹, is a graphic model. It is similar to the model presented in [1]. An ST diagram displays the two aspects of a distributed computation: the spatial (vertical axis) and the temporal (horizontal axis).

Each system process is represented by an edge, called a process edge. A process edge also depicts the progression of the

¹ The two terms, ST-model and ST-diagram, are used interchangeably.

process execution along the time axis. Send or receive events, executed by a process, are marked on its edge as event nodes. Edges connecting two event nodes, on two different process lines, represent the transmission of a message, and are called channel edges. The process that executes the send event is referred to as the sender process, and the process that executes the receive event is the receiver.

The ST diagram of a simple computation, in the DCS of Fig. 3.2.1a, is shown in Fig. 3.2.1b. There are three processes, namely P_1 , P_2 and P_3 . The pattern of the interprocess communication, during a certain time interval, is depicted explicitly in the ST diagram.

From an ST diagram, like the one in Fig. 3.2.1b, we can easily deduce any precedence relationships between events. It is obvious that events in a process are totally ordered. Matching send and receive events are directly related due to causality. Moreover, applying transitivity, we can extract any existing precedence relationships between comparable events. Two events that are not comparable (i.e. neither precedes the other) are considered concurrent.

The ST model is especially useful, when we want to illustrate the interprocess communication that took place during a particular execution. Event nodes show the exact instants in time, at which the events occurred. In other words, the ST diagram gives a very clear picture of "what actually happened".

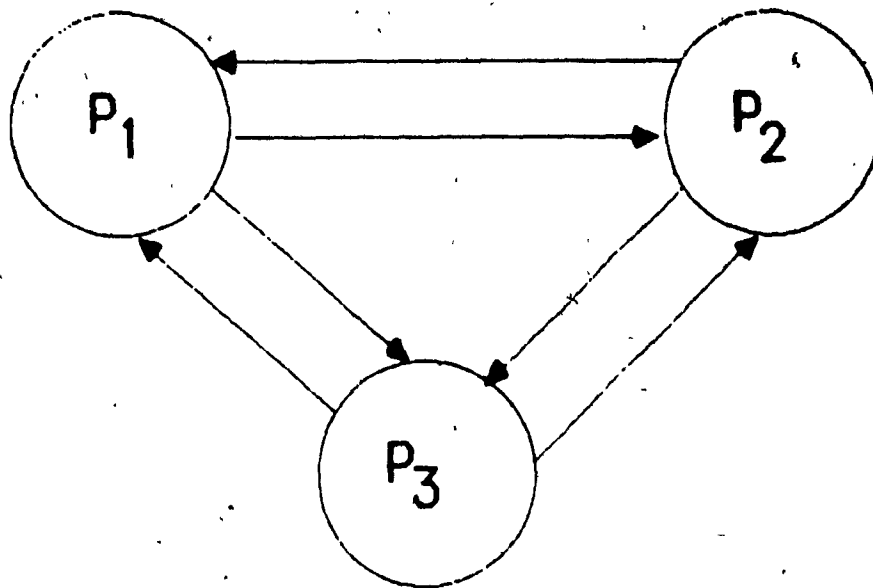


Fig. 3.2.1a: A three-process DCS

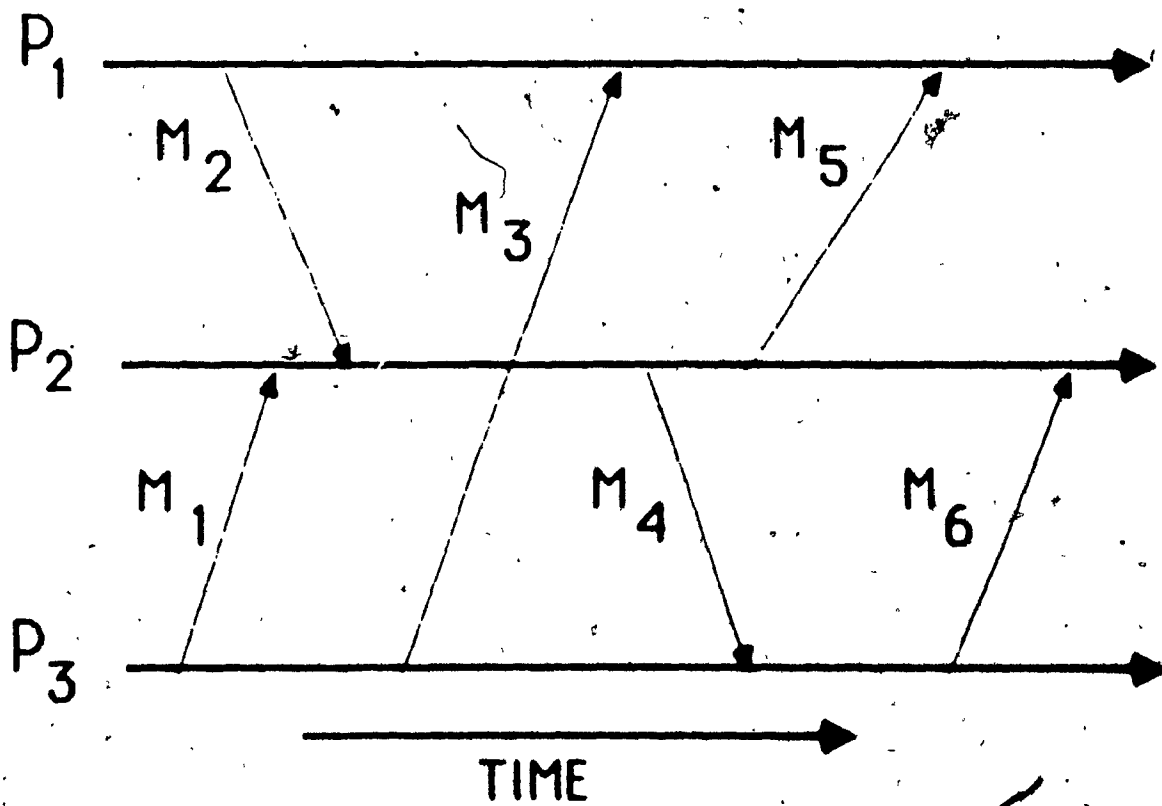


Fig. 3.2.1b: ST diagram of a computation

This attribute makes the ST diagram a valuable tool during the testing and debugging phase of a system.

On the other hand, there are some weaknesses in the ST model, the main one being the way concurrency is expressed. Concurrency is not represented in a simple and obvious manner, and it has to be deduced indirectly. Events on a single process are totally ordered, with respect to time. While this is true for sequential processes, it is not always the case for composite processes. In such processes the ST diagram exercises an arbitrary serialization of concurrent events. Concurrency can also be present in sequential processes when the synchronization requirements specifications are specified. Some synchronization events can be characterized as concurrent by the designer. An ST diagram fails to describe those events and therefore it is not practical as a specification tool.

3.3 The Pomset Model.

The Pomset model was introduced by V.Pratt in [19] and refined by the same author in [20] and [21]. It is an attempt to establish a formal model for concurrent processes, using the concept of partial ordering. The model is general and can be applied to many different applications. In the following, we first present its formalism and proceed with its interpretation in the case of distributed computations. In other words, in the

terminology of formal language theory, we first introduce its syntax and then its semantics.

We begin with an informal discussion of the basic concepts. The basic idea behind the model is the realization that:

"A fundamental concept in a process-oriented model of computation is variety of behavior.. A popular way to model variety is with sets: a program, or process, is modelled as a set of its possible behaviors" [20].

Pomsets represent the different behaviors of a process. The word pomset is an abbreviation from the words Partially Ordered Multiset. We can intuitively grasp the concept of a pomset, generalizing on the notion of a string. A string is a set of symbols from an alphabet. More precisely, we can define a string as a finite linearly ordered multiset. The term multiset is used, instead of set, because multiple occurrences of the same symbol are allowed. For example, the string 011010 is a multiset that contains multiple occurrences of the symbols 0 and 1. A pomset can be regarded as a natural generalization of a string, where linear order is replaced with partial order and the word finite is omitted.

A language is a set of strings. By analogy, a process is a set of pomsets. Another characteristic of the model, is the natural and straightforward way in which process composition can be performed. The process that is realized by a system of

processes is the intersection of the utilizations of its constituent processes. The utilization is an application dependent concept, that adds to the process the knowledge of its interaction with the other processes of the system.

3.3.1 Definitions.

A labelled partial order (lpo) is a 4-tuple (V, Σ, \leq, μ) where:

(i) V is a vertex set, typically modelling events.

(ii) Σ is an alphabet, modelling actions.

(iii) \leq is a partial order, with $e \leq f$ typically interpreted as event e preceding event f in time.

(iv) μ is a labelling function, $\mu : V \rightarrow \Sigma$, assigning symbols to vertices.

A pomset is defined as the isomorphism class of an lpo. Isomorphism class means that the identities of the vertices are not important and they can be treated as anonymous points. A process is defined as a set of pomsets. The Pomset model is a mathematically oriented model. An algebra of pomset operations has been defined in [21]. Below, we cite the definitions of few operations, which are important to our discussion. They are:

Concurrence: The concurrence of two pomsets $p:[V,\Sigma,\leq,\mu]$ and $p':[V',\Sigma',\leq',\mu']$ is the pomset $p||p':[V\vee V',\Sigma\vee\Sigma',\leq\vee\leq',\mu\vee\mu']$. In other words, the concurrence $p||p'$ denotes the process consisting of two concurrently executing processes.

Concatenation: The concatenation $p;p'$, or simply pp' , is as for concurrence except that instead of $\leq\vee\leq'$ the partial order is taken to be $\leq\vee\leq'v(VxV')$. This forces every event of p to precede every event of p' .

Prefix: A pomset q is a prefix of a pomset p , written $q\leq p$, when q is obtainable from p by deleting a subset of the events of p , provided that, if event u is deleted and $u<v$, then v is also deleted.

Colocation: If pomset events are labelled with location id, such as port id or process id, then all events with the same labels are said to be colocated.

3.3.2 Model Interpretation.

A process is characterized by the set of its alternative behaviors. Process behavior is a sequence of events, where the ordering of events conforms to a partial temporal ordering. The events, we consider, are the primitive send and receive events,

observed along the interface of the process with the rest of the system. Additional information about the represented events can be incorporated in the model by annotating the pomsets. More precisely, colocated events can be labelled with identical tags, that identify the particular port (i.e. channel-end) that is associated with each event. Alternatively, events can be tagged according to the type of messages to which they are related.

Processes can be composed hierarchically to form a composite process. The process that is realized by combining two or more processes, is represented by its own set of alternative behaviors (i.e. another set of pomsets). The events, contained in the pomsets, are events observed at the interface of the system of the composed processes with the rest of the world. Events related to the exchange of messages among composed processes, are considered internal events of the composite process and are omitted from its pomsets. The restriction of the composite process behavior to any of the composed processes must correspond to a valid pomsets for those processes. Process composition can be extended to any set of processes, the final result being an entire system considered as just another process.

The above discussion is illustrated with the following example. Imagine again the system of three interconnected processes of Fig. 3.2.1a. Each process communicates with the other two processes through unidirectional channels. Assume that process P_1 sends messages to both P_2 and P_3 , and then receives their responses in the order they arrive. This pattern of

message exchange of process P_1 , is repeated throughout system execution. If the order in which P_1 sends out its messages to P_2 and P_3 is not specified the following pomset represents a valid behavior for process P_1 :

$$B_1: [s_2 ; s_3 ; (r_2 \parallel r_3)]$$

where event labels denote the channels to which events are related. Another possible valid behavior, for process P_1 , is given by the following pomset:

$$B_2: [s_3 ; s_2 ; (r_2 \parallel r_3)]$$

If we assume, that process P_1 executes an interleaved sequence of the above two behaviors, then it is characterized by the following:

$$P_1: \pi \delta (B_1^* \parallel B_2^*)$$

where the symbol π (prefix closure) makes the process abortable by permitting it to get only part of the way through its computation. The symbol δ indicates serialization of colocated events. Let us further assume that process P_2 is characterized by the following behavior:

$$P_2: [(r_1 \parallel r_3) ; s_1]^*$$

In other words, P_2 waits two messages from P_1 and P_3 , and then sends its response to P_1 . Finally, let process P_3 be characterized by the following straightforward behavior:

$$P_3: [r_1 ; s_2 ; s_1]^*$$


Suppose we want to compose processes P_1 and P_2 into one composite process. We can then obtain a pomset that characterizes this process, by composing the behaviors of P_1 and P_2 . The derived pomset is:

$$P_1 \circ P_2: [(s_3 ; r_3^1) \parallel r_3^2]^*$$

where the symbol " \circ " indicates process composition, and superscripts denote distinct ports.

The above process pomsets have been given in the generator space. In other words, the actual pomsets that can be possibly observed would consist of an a priori unknown number of repetitions of those generators. The rules for composing pomsets generators to yield the accurate pomsets, correspond to the usual operations that can be performed on pomsets, as described in [21].

As we can see from the above example, concurrency is inherent in distributed systems. It becomes almost ubiquitous during the system design phase. Even at the level of sequential




processes, events can be specified as concurrent during design, either because they depend on unstable environment parameters (receipt of messages over channels with variable message delay), or because the designer does not want to impose any limitations on the implementation, for example strict ordering of identical events at different ports.

The main strength of the Pomset model, lies in its ability to express and model concurrency in a very simple and efficient manner, that can be clearly understood by both the designer and the implementor of a distributed algorithm. It provides also a straightforward solution to the problem of process composition. These properties make it a useful specification tool in the hands of the system designer, who can specify system processes in a precise and elegant way. Its capability for system specification can be useful in distributed debugging.

3.4 Comparison of the Two Models.

The two models are equivalent in the case of a network of communicating sequential processes, in the sense that the same event relationships can be expressed in both. There is a direct correspondence from cuts in the ST model to pomset prefixes in the Pomset model, as shown in [24]. When, however, we attempt a hierarchical process composition in a system of sequential processes, the ST model loses its comprehensiveness, since it fails to represent concurrency in composite processes.

Another slight difference between the two models, is the variation in the notion of time. In the Pomset model emphasis is given to the precedence relationships between events. It is the sequence of events that is important, not the exact time ordering. In the ST model, on the other hand, a more precise representation of event timing is usually incorporated in the diagram. The peculiarities of each model make them useful for different applications.



CHAPTER 4

DISTRIBUTED DEBUGGING

Most programs, at some stage of their development contain errors. Debugging is the process of locating the source of program errors that manifest their presence by generating faults. Faults lead either to poor program performance, or to incorrect results. In this chapter, we first present the basic ideas of sequential debugging and briefly describe its methods. Then we introduce distributed debugging and its problems. We point out the inherent inadequacies that make conventional debugging methods inefficient for distributed programs. Finally, we propose our strategy for distributed debugging. From this informal discussion we establish the need for tools that should be offered by a distributed debugging facility. These features, along with an outline of the distributed debugger, are presented in the next chapter.

4.1 Sequential Debugging.

The means of software defect removal are several, depending on the software development stage at which they were introduced.

Any of the following techniques can help designers, or programmers, remove flaws [8]: requirements reviews, design reviews, code reviews, static analysis, functional tests, etc. Furthermore other methods, such as program verification, or structured programming, can help in reducing the number of bugs in programs. However, even with these techniques bugs will still remain and debugging will still be needed.

Debugging sequential programs has been practiced and mastered since the dawn of the computer age. Although it is well understood, the definite and general guide to debugging has yet to be written, nor is it likely to be. The reason is that even though a lot of debugging techniques and methods exist, intuition remains still by far, the most widely used debugging tool.

No matter how improvised debugging might be, there is commonality to the debugging procedures followed by programmers. The programmer tries to isolate the area of the bug and eventually locate it. The program input is chosen among the most important or, even better, among those that are already known to produce failures. In other words, a test case is selected. Then the program is executed, and its execution flow is checked using existing debugging tools, such as breakpoints and probes. This method is based on the implicit assumption, that any control over the various program objects (program variables, data structures, etc.) can be inserted at breakpoints. This is possible, since instructions are executed sequentially, and program behavior is

not affected by the amount of time that elapsed between the execution of two consecutive instructions.

Most of today's conventional¹ debuggers offer more or less the same features, which among others include: execution under the control of the debugger, single stepping, position or event breakpointing, continue after breakpointing, stack tracing, printing and changing the value of a variable. These functions, by themselves, are only mechanisms for manipulating software. They do not show what is wrong with the program, nor do they aid the user by stating how they can be used. It is the debugging strategy that directs the programmer to their use.

4.2 Distributed Debugging.

Distributed software, like all software, is prone to errors. Debugging distributed programs is less well understood than sequential debugging, because distributed systems, algorithms, programming languages, and their abstractions are not as well understood as sequential ones. The existing debugging methods are usually ad hoc and tailored to a particular application, or tied to a specific programming environment. Distributed debugging is usually considered an extension of conventional debugging, and is looked at as an implementation problem, rather than a conceptual one. We believe, that existing debugging

¹ Conventional debuggers are debuggers used for sequential programs, as opposed to debuggers for distributed programs.

techniques are inefficient for distributed programs. New methodologies should be developed, that take into account the idiosyncracies of distributed systems.

Problems in Distributed Debugging.

DCSs share some characteristics not to be found in centralized systems. These properties will necessarily affect debugging and can be attributed to the following factors:

- Presence of multiple asynchronous processes: A distributed system is made up of a collection of asynchronously communicating processes, executing concurrently on different processors. The result is a system having many loci of control, an element that makes DCSs harder to understand and more prone to errors.

- Time management: DCSs have no global clock. The lack of a common clock reference makes it impossible to impose a total ordering in the sequence of events occurring in the system. At most, we can accurately recreate the sequence of events that took place locally in sequential processes. Reproducing the exact sequence of events is of tantamount importance in debugging.

- Variable message delay: Communication channels introduce delays that can vary depending on several factors, such as the nature of the channel, the channel traffic, etc. Variable delays can cause nondeterminism, in the sense that the ordering of

events, even on sequential processes, may change over different executions.

- Finally, DCSs tend to be rather large, i.e. with a large number of processors and processes. Largeness, not exclusively a property of DCSs, makes systems too complex for anyone to comprehend how a particular change will affect other parts of the system.

The above aspects of DCSs introduce new problems in the debugging process. Familiar concepts, like program state, breakpointing, and single step execution need redefinition, since they are no longer valid, or useful, in distributed environments. For example, what is the notion of program state and can one be recorded²? Or can we use the concept of breakpointing in DCSs, and how do we go about setting breakpoints across more than one process? Obviously, we should answer questions like these, before we propose any debugging tools.

Typically, DCSs are debugged today as follows: each process in the system is run through a conventional debugger, and the output of each debugger is displayed on a separate terminal [12]. Sometimes, additional debugging tools are added, such as monitoring interprocess communication. Although, nobody argues that such tools may be useful, the problem remains, as with sequential debugging: debugging tools are just mechanisms for

² An illustration of the problems of global states in DCSs, their classification, and algorithms for their recording is presented in [4].

manipulating software, that do not support the user on how to use them. The underlying philosophy is missing.

4.3 Error Classification.

Since debugging is concerned with locating and removing software errors, it is only natural that we want to know more about errors. Quite a few efforts to define and classify bugs have been made, based on the belief that the more we know about a problem the better we can fashion a solution. Error taxonomies are usually bound to a particular model.

Hence, in sequential programs, errors are mainly related to the particular stage of software development at which they were introduced. Consequently, we can have requirement bugs, design bugs and code bugs [8]. On the other hand, in distributed systems errors are usually divided in component bugs and design bugs. This distinction results from the view that a DCS is made up of component processes, that are "glued" together by the design. Component bugs are further divided, depending on whether they generated wrong results, or were generated at a wrong time [9].

The above error distinction of DCSs can be vague. For example, a component bug can be regarded as a design bug, when another level of abstraction is employed. In our study, we have used Pomsets to model DCSs. In this context, we adopt the following error classification. Errors, whose occurrence can be

identified as a violation of the pomset, that specifies the process behavior, are called synchronization errors. Alternatively, bugs which do not result in any pomset violations are considered as computational errors. This classification fits nicely into our model, and expands smoothly to encompass hierarchical process composition.

It can be argued that pomsets do not always contain the same amount of information, since annotation can be used. However, they always include the events that are necessary to fully describe process synchronization. Hence any errors that cannot be linked to those events, do not influence process interaction. We feel that this argument, justifies the above classification.

4.4 Debugging Strategy.

Distributed programs have two facets: the sequential and the synchronization. Consequently, there are two aspects in distributed debugging: isolating and removing computational errors and doing the same for synchronization errors. The first task, namely the removal of computational bugs, is the easier one. It resembles sequential debugging, something that programmers are well familiar with. Methods and tools for conventional debugging are better developed and understood. The new element in distributed programming is the synchronization aspect of processes. Although, we do not underestimate the importance of conventional debugging, we feel that emphasis

should be put in dealing with the synchronization behavior of processes. In the following, we will concentrate on debugging of synchronization errors.

We are proposing a debugging strategy that focuses the programmer's attention at the interaction level of process activity. The "interesting" events at this level are send and receive events. These events provide the information that is necessary to the programmer, in his attempt to gain an understanding of the errors, or at least to perceive where the implementation and the expected behavior differ.

Producing correct distributed software is a complicated process, that begins with the algorithm design phase, winds its way through software design, and ends with implementation, debugging and testing. The common factor in all phases of this development should be the Synchronization Requirements Specification (SRS). SRS is produced by the algorithm developer, refined by the software designer and used by the programmer.

Our basic premise is that debugging should be based on a comparison of the actual process³ behavior against formal process specifications. The tools for specifying and depicting process behaviors, are the two models presented in chapter two. More precisely, the Pomset model is suitable for expressing process specifications; on the other side, the ST model is more efficient

³ In the following, the term "process" refers to any process, be it sequential or composite.

in depicting actual event occurrence, and therefore more appropriate in describing actual process behavior.

The use of the two models allows for a set of coherent solutions to the problems of distributed debugging, that were outlined in the previous sections. The concept of a "cut" suggests an answer to the questions of breakpointing and step execution, while a process state corresponds to what is referred to as "characterization computation" [24] in the ST diagram, or as a pomset prefix in the Pomset model.

Discussion of the Debugging Process.

The synchronization requirements of a process are fully defined using the Pomset model, while actual process behavior is illustrated in the ST diagram, which generally corresponds to a particular serialization of a pomset. During testing and debugging, process behavior should be checked against its specifications. Synchronization errors cause faults that are manifested as violations in pomset specifications.

The comparison of process behavior to process specification, can be performed at any level of abstraction the programmer deems necessary: sequential process level, composite process level (black box debugging), or even system level. Hierarchical process composition allows the programmer to concentrate in suspected interactions and overlook irrelevant ones.

Another advantage of specifying the synchronization requirements of a process, is that their availability enables the user to foresee states in the synchronization behavior of a process. Consequently, he can define a pomset prefix that should be passed by a process (or a system), under proper execution. Since pomset prefixes correspond to pomset cuts, defining a prefix is tantamount to describing a consistent pomset cut. The programmer may command the debugging facility, to suspend execution, once the pre defined prefix is observed. Stated it briefly, a breakpoint in sequential debugging, corresponds to a pomset prefix in distributed debugging.

Failure of a process to reach a specified consistent cut may hint at the presence of errors. It should be noticed however, that due to the presence of non determinism in the execution of parallel programs, this is not always true. In the Pomset model, a process is described by a set of pomsets, rather than by a single pomset. Hence, specifying a prefix in a particular pomset does not ensure, that the prefix is reachable in all possible process behaviors. The alternative is to define a complete set of prefixes, one of which must be reached. If such a set, that accounts for all possible process behaviors, can be defined, failure of the process to reach one of the prefixes indicates the existence of errors.

Once a process has been suspended along a consistent cut, the user can check the sequence of events, that led to the cut. By comparing against the specifications, he can determine whether

it is a valid (i.e. according to specifications) sequence or not. The user can also examine the program state, by exercising application specific global invariants, across a consistent global cut. A global invariant can be informally defined as a property satisfied by all processes throughout execution. Sometimes, process group invariants can be also defined. Invariants are used to reveal the presence of faults that have not yet caused any observable discrepancies in program behavior. Application specific invariants are mainly aimed at debugging computational errors. While their use may sound like error detection, it should be understood that in concurrent programs, testing and debugging are not, really distinct phases. An error in a DCS can spread quickly through the system, and by the time the programmer observes a fault, critical information in the system may have been lost.

After the cut the program can resume execution, until it reaches the next breakpoint. An alternative mode of execution is to execute a process up to its next event. This mode corresponds to single step execution of conventional debugging, and allows the user to have a tighter control during execution, and possibly monitor interprocess communication. It can be argued, that this mode effects a transformation in the semantics of unblocking communication primitives. We believe, however, that the semantics are not changed, since a process still continues execution after a send event, without waiting for the receiver's acknowledgement. From the above discussion it becomes evident

that the debugging facility (i.e., debugger) should provide certain tools to support debugging. These tools are discussed in the next chapter.

4.5 Probe Effect.

The term Probe Effect (PE) is used to describe behaviors observed when delays are introduced into concurrent programs with synchronization errors [10]. PE is manifested through altered program behaviors. It can result in masking of synchronization errors. The PE is apt to be a side effect, when concurrent programs are executed with debuggers, and therefore it should be taken into account in the design of the debugger. In the following, we attempt a characterization of PE without offering any solutions to the problems it introduces in the debugger's design.

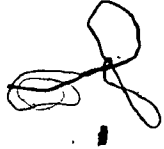
When the debugging operations interfere with the execution of a certain process, the behavior of other processes can also be influenced. The overhead introduced by the debugger can mask synchronization errors, the result being concurrent programs that do not work unless they are being debugged. For PE not to be observed, the debugger's presence should be made transparent to the debugged program. As a debugging side effect, PE is primarily due to channel probes, that aid monitoring of message exchanges between processes. In this case, PE can be described

as the effect of the channel probe on the delivery of a message through the channel. Possible consequences are:

- Delay in the delivery of a message.
- Change in the contents of the message.

The second consequence is easily avoidable, with proper design and implementation, although the user may intentionally wish to change the contents of the message for testing purposes. PE is primarily sensed when it induces an extra delay to message delivery, the reason for that being that in a parallel program the relative speeds of concurrent processes can influence the results of the computation.

A possible scenario, that illustrates the above argument is the following. Imagine a receiver process that has to choose, in a non deterministic way, among a set of messages sent by a group of sender processes. When one of the sending processes is delayed, some communications may be shifted in time. This may result in a change in the set of alternatives considered by the receiver process. The flow of program execution will consistently follow certain paths, while eluding others. As a result, possible errors in the synchronization behavior of processes can be masked out during the testing/debugging phase of the program. These errors will be revealed later, when the program executes in another environment, without the debugger's presence.



We can characterize delay on message delivery due to PE, as follows. Let the delivery delay on channel i be D_i . Assume that it exhibits the same characteristics on all messages transmitted through the channel. Then, we can identify three different kinds of delay:

- a. D_i is a finite random variable, independent of any other delay D_j .
- b. D_i and D_j are related.
- c. D_i is infinite.

The first kind is the most usual in practice. It corresponds to the given example. The second condition can also occur, because the underlying implementations of the probes share internal logical/physical resources. Hence, D_i and D_j can be related through some common process, that unfairly serializes the message handling on channels i and j . For example, the probes on channels i and j may both need to access a common database to record messages for monitoring reasons. If the arbiter process, that regulates access to the database, imposes an arbitrary serialization on simultaneous requests from the probes, the second kind of delay will result. The third condition is met when the probe is shut off, or has mismatch behavior with respect to the process being probed, something we can detect and rectify easily. Alternatively, the user's intervention in program execution can result in an infinite channel delay. The user may

wish to exercise certain execution ordering, that leads to process incompatible send-receive conditions and eventual deadlock.

CHAPTER 5

THE DISTRIBUTED DEBUGGER

In this chapter, we present some of the important features of the distributed debugging facility that we propose. The discussion focuses on issues that were introduced in the previous chapter. Solutions are presented at a conceptual, rather than an implementation level. We discuss only the abstractions of the debugger, while a lot of low level details are omitted. Emphasis is placed in the tools for debugging synchronization errors, although the distributed debugger should also contain conventional tools, used in sequential debugging.

In the first three sections, we present the tools that should be offered by the distributed debugger, and discuss issues related to their use. The suggested tools help exercise the debugging strategy that we advocate. In the next two sections, we attempt a general and brief description of the basic structure of the debugger, and present its high level specifications. Related work is presented in the last section of the chapter.

5.1 Channel Counters.

As it was pointed out in the previous chapter, an important tool in distributed debugging is the ability to set breakpoints across many processes. We claim, that positional breakpoints of sequential programs, correspond to pomset prefixes in distributed programs. They can also be depicted as consistent cuts in the ST diagram of a DCS. The difference is that, pomset prefixes can be defined a priori, while ST diagram cuts can be defined only a posteriori, that is after execution. Hence, it is mainly pomset prefixes that are useful for setting program breakpoints.

In ST diagrams, cuts are described by their characterization computation, which amounts to the computation performed by every process up to the point of the cut. ST diagram cuts also correspond to pomset prefixes, as shown in [24]. The execution of a prefix (i.e. the events that constitute the prefix), or of the characterization computation, results in a process reaching a particular state. The state of the system along the ST diagram cut comprises the set of all component process states, along with the states of all channels. From the preceding discussion follows that, in order for the user to be able to set breakpoints, the debugger should enable him to define consistent pomset prefixes. One way of accomplishing this is by use of Channel Counters.

5.1.1 Description of Channel Counters.

According to the adopted model of a DCS, processes communicate via message passing, through unidirectional point-to-point channels. The debugger consists of a central site, the master site, and local node processes, the node debuggers (see section 5.4). Every node debugger maintains one counter for every channel-end, or port, of the process it controls. Channel counters, of the same application process, are called local channel counters. Counter values increment monotonically and originally are set to zero. They may be reset to zero, if the user wishes¹ so, at breakpoints¹. A counter is incremented every time a message leaves from, or arrives at, the corresponding channel-end.

There are two kinds of counters: colorless and colored. The former are incremented with every message sent, or received, from the associated channel-end. The latter count messages of a particular color. Message colors correspond to event labels of annotated pomsets. The number and the kinds of colors are parameters, defined at the beginning of the debugging session by the programmer. The debugger determines the color of a message by looking up a particular message field. Message colors help the user concentrate in a subset of process interactions.

¹ Messages that are in transit (i.e. have been sent, but not received) at breakpoints should be taken into account, before resetting counter values. Otherwise, counter values might not be consistent anymore.

The value of each counter represents the channel state at that end. Because colorless counters increase their values with each message, they define unique channel states. On the other hand, colored counters change values with messages of a particular color only, and consequently, they define classes of channel states, rather than unique states. A set of local channel counter values, defines the synchronization state of that process. The usefulness of channel counters in describing pomset prefixes is expressed in the following lemma:

Lemma 5.1.1: For every pomset prefix, there is a set of channel counter values, that is necessarily associated with it.

Proof: The truth of the lemma is inferred, by observing the following. Every pomset prefix consists of events, send and receive, associated with channel-ends, or ports. Events associated with a particular port are colocated events. If we count all events, observed at a particular port, and assign their number as the corresponding counter's value, and do the same for all sets of colocated events in the prefix, then this set of channel counter values is associated with the particular pomset prefix, Q.E.D.

Actually, the lemma can be generalized for any pomset, since every pomset can be considered as a prefix of another pomset. The counters can be either local counters, when we consider pomsets of sequential processes, or counters belonging to

separate sequential processes, in the case of composite processes.

The lemma asserts that a pomset prefix can be fully expressed in terms of channel counter values. However, not all values are always necessary, or even useful, in specifying a prefix to the debugger. Consider the following pomset generator that characterizes a sequential process:

$$P: [s_1 ; s_2 ; (r_1 \parallel r_2)]^*$$

where subscripts indicate distinct channel-ends. The prefix that comprises the first two send events, can be expressed in terms of counter values as:

$$CC_1 = 1 \text{ AND } CC_2 = 1$$

If we observe, though, that event s_2 should not happen before s_1 , then we realize that the counter value for port one is redundant. Stated differently, when CC_2 takes the value 1, CC_1 should also have the value of 1, and hence it is sufficient to specify the value for CC_2 in order to describe the particular prefix.

This last remark introduces the use of colored counters, which keep track of events with a particular label (i.e. color).

Definition: The α -restriction of a pomset p , is the pomset derived from p , by deleting all elements of p that do not have

the label α .

Lemma 5.1.2: For every pomset restriction, there is a set of color-sensitive counter values of the same label, that is necessarily associated with the restriction.

Proof: A pomset restriction can be considered as one-label annotated pomset. Its events can be numbered by color-sensitive counters of the same color. Hence, lemma 5.1.1 implies the truth of lemma 5.1.2, Q.E.D.

A pomset prefix can be defined by the restrictions, that correspond to the labels of its last event(s)² and its concurrent events that are not included in the prefix. Consequently, it can be expressed in terms of the values of the colored counters, that correspond to the labels of those restrictions. This is the minimum set of counter values, required to define a pomset prefix unambiguously. In the pomset prefix shown in Fig. 5.1.1, the corresponding channel counter values are:

$$CC_{I/1} = 1 \quad \text{AND} \quad CC_{I/2} = 0$$

where I stands for Input.

² The labels of all concurrent events, that either precede or follow the cut, should be considered.

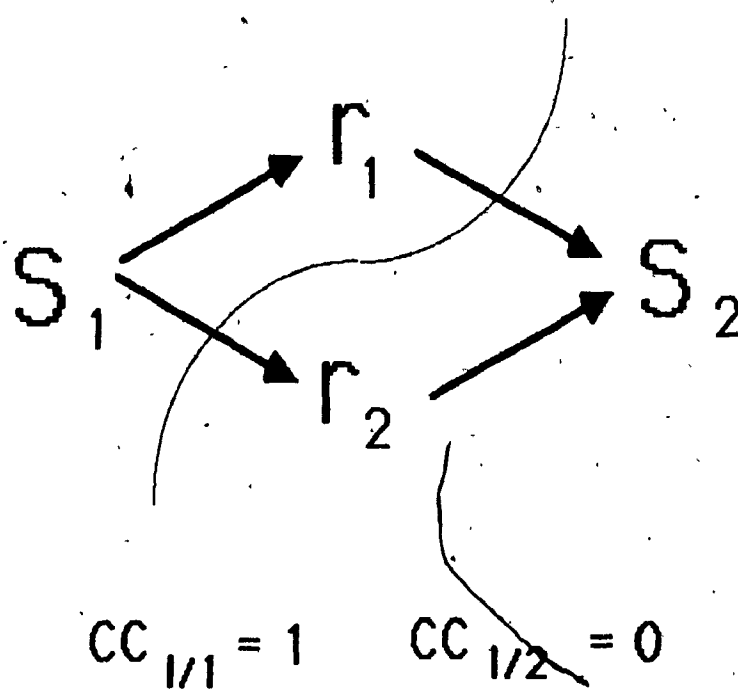


Fig. 5.1.1: Pomset prefix and corresponding values
of channel counters

5.1.2 Use of Channel Counters.

The programmer observes the symptoms of an error (i.e. a fault) and decides to suspend execution once a particular prefix is observed. He can specify the corresponding breakpoint by defining a set of counter values. If the breakpoint corresponds to a unique prefix that should always be observed during execution, then failure to reach it reveals the existence of synchronization errors. It is usually the case though, that such "absolute" prefixes are rare in DCSs, due to non-determinism present during the execution. Normally, the programmer should specify a class of prefixes, any of which could be passed under correct execution. This is usually realized by partial definition: values for some of the counters are specified, while for the rest "don't care" value are assumed. Alternatively, multiple prefixes can be also defined by disjunction.

Once a specified prefix is passed the debugger suspends process execution. The programmer can then examine the internal states of the constituent processes, invoke global invariant tests, or examine the sequence of event occurrence (see following sections) that led to the cut. The execution path should be checked against process specification. From a breakpoint the system can resume execution.

An example that illustrates how counter values define breakpoints, is given below. Imagine a fully connected three process DCS, similar to that of Fig. 3.2.1a. Process P_1 is

characterized by the pomsets that can be generated by the following generator:

$$P_1: [(s_2 \parallel s_3) ; (r_2 \parallel r_3)]^*$$

Different prefixes can be defined in the pomset shown in Fig. 5.1.2. The corresponding sets of channel counter values, that specify each prefix are (CC stands for Channel Counter):

Prefix 1: $CC_{12} = 1$ and $CC_{13} = 0$

Prefix 2: $CC_{12} = CC_{13} = 1$ and $CC_{21} = CC_{31} = 0$

Prefix 3: $CC_{21} = 1$ and $CC_{31} = 0$

Actually, the counter values need not be ones and zeros only. They could be any integer values N and $N-1$, respectively. In prefixes 1 and 3, we define values for some of the counters of process P_1 . The rest assume a "don't care" value. When P_1 is suspended, say after prefix 3 is passed, the values of unspecified counters can also be examined. In this case, they should have the values: $CC_{12} = CC_{21}$ and $CC_{13} = CC_{31}$. Any deviation from these values would signify the presence of synchronization errors. The point, however, is that the values of those counters are not necessary for the specification of the prefix.

The above prefixes are defined in P_1 's pomset only. We can also define prefixes for any composite process, such as $P_1 \circ P_2$.

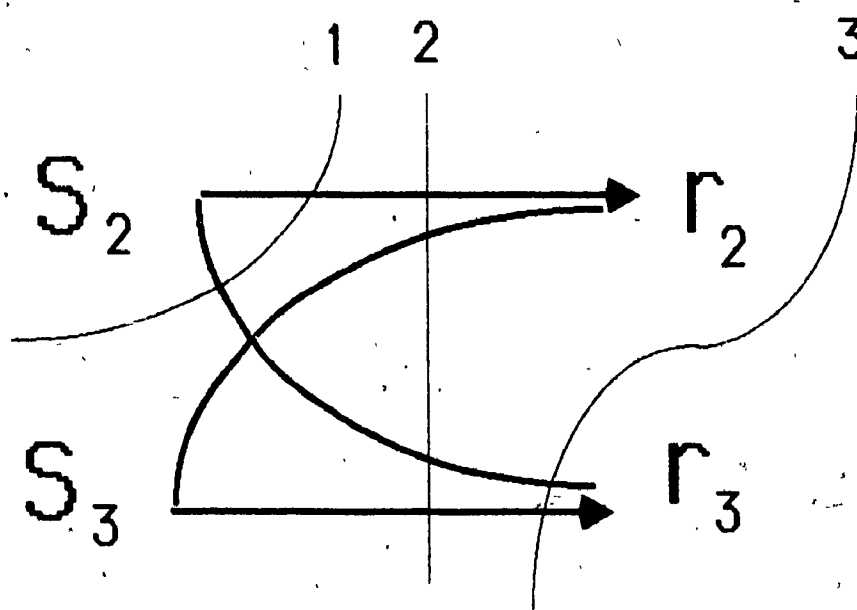


Fig. 5.1.2: Pomset prefixes

Let us assume that process P_2 is characterized by the following behavior:

$$P_2: [(r_1 \parallel r_3) ; s_1]^*$$

Then process $P_1 \otimes P_2$ is characterized by the pomset generator:

$$P_1 \otimes P_2: [(s_3 \vee r_3^1) \parallel r_3^2]^*$$

where superscripts indicate that the two receive events of the same label are observed at different ports. Prefixes that we can define in this pomset are shown in Fig. 5.1.3. They correspond to the following sets of values:

Prefix 4: $CC_{13} = 1$ and $CC_{32} = 0$

Prefix 5: $CC_{13} = CC_{32} = 1$

In the case of the composite process $P_1 \otimes P_2$, we can perceive how non determinism may alter the execution paths of a process. The counter values corresponding to prefix 4 define only one prefix, that might not be reached during execution. The counter values for prefix 5 are more general (actually they represent a set of prefixes) and they describe prefix 4 too.

The presence of synchronization errors in the implementation of processes can cause problems in the operation of the debugger. These problems are discussed in the next section.

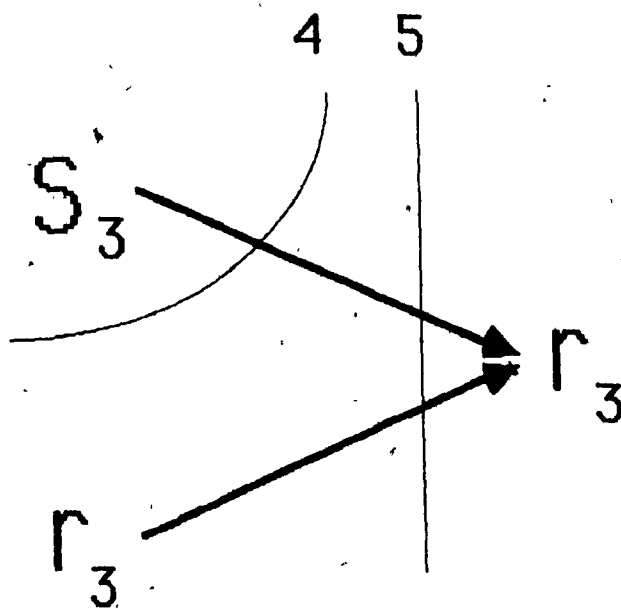


Fig. 5.1.3: Prefixes in the pomset of a composite process

5.2 Process Termination Conditions.

Process termination problems may arise when the distributed debugger is instructed to suspend the execution of a set of processes after a pre specified prefix is passed. It is then possible, that the operation of the debugger interferes with the execution of the application processes, creating pathological situations, such as process starvation. Two reasons can be the cause of such problems. Either the programmer specifies a non consistent cut, or synchronization errors alter process behaviors. We rule out the first case, since it presupposes an error on the programmer's part. The second scenario, however, is likely to occur.

Before giving an example we should describe the procedure followed by the debugger, in suspending the execution of a set of processes after a specified prefix. The user inputs the channel counter values that define the prefix, at the central site. The values are then "projected" to node debuggers. The projection of a prefix definition to a node debugger, consists of the set of values for locally maintained counters. For every counter for which no value is specified the debugger assumes a "don't care" value.

A local debugger reports to the central site when:

1. Its local process is suspended, or

2. The specified condition is irreversibly violated.

The conditions for the suspension of a local process are discussed below. Violation of a specified condition occurs, when a counter exceeds its specified value, while at least one other local counter has not reached its own value. The central debugger compiles the information received from node debuggers, and declares success, or failure, in reaching the specified state.

The following example demonstrates how synchronization errors can cause process termination problems for the debugger, Fig. 5.2.1. Two processes P_1 and P_2 are connected as shown in the figure. The pomsets that specify the two processes are:

$$P_1: a ; b \text{ and } P_2: c ; d$$

Let us now assume, that there are synchronization errors in the implementation of P_2 , and its actual synchronization behavior corresponds to the pomset:

$$P_2: c ; c ; d$$

In other words, P_2 expects two messages from P_1 , before yielding its own response in channel d . The programmer, without suspecting the existing error, specifies the following prefixes

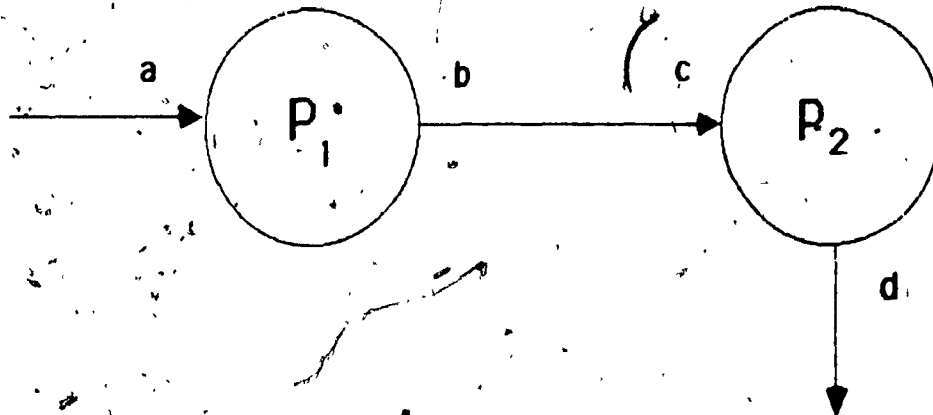


Fig. 5.2.1: Process termination problems

in the process pomsets:

$$P_1: CC_a = 1 \ \& \ CC_b = 1$$

$$P_2: CC_c = 1 \ \& \ CC_d = 1$$

Process P_1 will meet the specified condition, after it has received and sent its first message. Consequently, the local debugger that controls P_1 will suspend its execution. This will leave process P_2 waiting, ad infinitum, for a second message. Process P_2 cannot be suspended, because it has not passed the specified prefix. At this point, the debugger's operation becomes overt to an application process, because of the debugger's interference with program execution (i.e. suspension of P_1). Process P_2 will infinitely block for a second receive event. The example depicts one possible scenario for process termination problems. It becomes clear that any debugger specifications should take into account situations similar to the one presented in the example.

When a local debugger suspends a process, it changes the process status from "running" to "blocked" and the status of its channels to "dead". The debugger sends a message to all "dead" output channels, that notifies other node debuggers of the status change. Likewise, any node debugger upon receipt of such a message marks the corresponding channel as "dead". Messages arriving at the input channels of a suspended process can be saved in buffers, and replayed when the process resumes

execution.

A local debugger should then suspend its process, in the following cases:

1. The process satisfies the specified condition.
2. The process attempts a receive event on a dead channel.
3. The process does not show any signs of activity for a certain amount of time.

The first case corresponds to the successful reaching of the defined prefix. It occurs when every counter attains the required value. The other two cases correspond to failures to reach the pre defined prefix. More precisely, the second condition reveals a process behavior not accounted for in process specifications, provided of course, that the specified cut was consistent. The third case prevents endless "hanging" of the debugger, in case of pathological process behavior, such as infinite looping, sudden death, etc. It should be pointed out, that this last condition is included as a precaution feature, against faulty process implementation rather than possible debugger side-effects. The debugger does introduce a definite overhead in process execution, but it does not modify process behavior, as far as its liveness and safety requirements are concerned. This condition can be implemented as "a time out

mechanism, and ensures that the debugger always returns control to the user, even at the presence of process implementation errors.

It is evident, that suspending a process is a decision taken locally by a node debugger. No cooperation between node debuggers is required, except to disseminate information regarding change of process status.

5.3 Timestamps.

The gist of the proposed debugging strategy, is the comparison of actual process behavior against process specifications. This implies, that the debugger should provide some means for recording the sequence of events, executed by each process. An important problem related to event recording is the identification of concurrent events, among events of a composite process, or events of different processes. Simple event recording on sequential processes is clearly inadequate for identifying concurrency.

Another major issue in distributed debugging, is the reconstruction of the exact sequence of event occurrence that took place. It has been pointed out many times, that non determinism in DCSs, may change the sequence of events over different executions. Therefore even though, a system reaches the same prefix twice, the execution paths that were followed may be different. An example is shown in Fig. 5.3.1. The shown ST

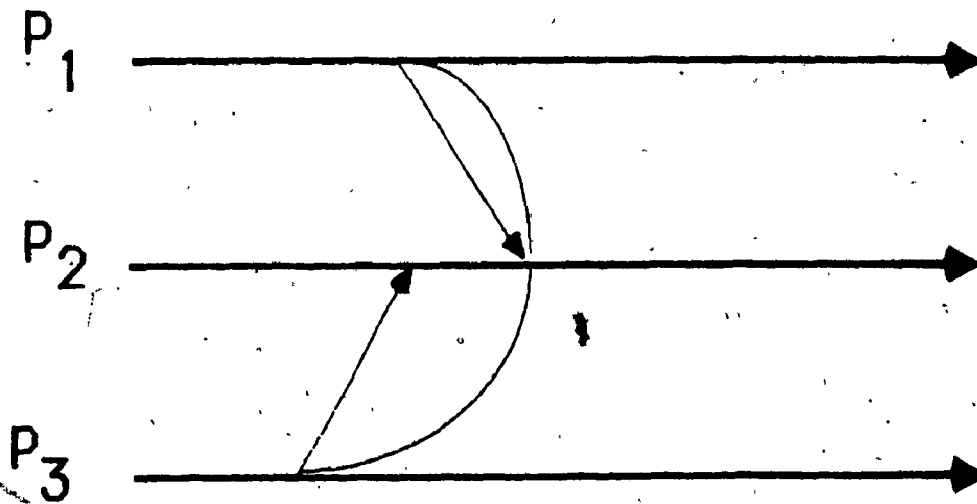
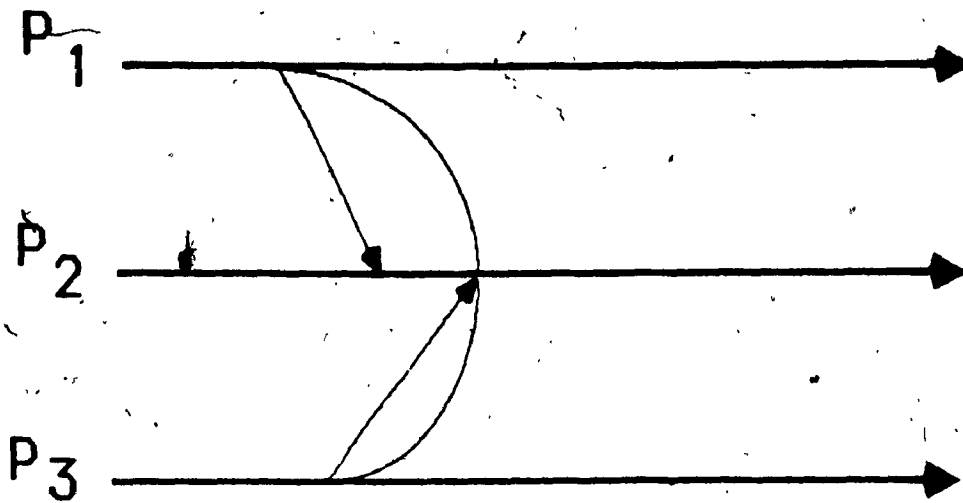


Fig. 5.3.1: ST diagram cuts corresponding to the same channel counter values

diagram cuts are characterized by the same set of CC values:

$$P_1: CC_{12} = 1, \quad P_2: CC_{12} = 1 \text{ AND } CC_{32} = 1, \quad P_3: CC_{32} = 1$$

Obviously, the event sequences that led to each cut are different. It is possible that synchronization, or even computational, errors are masked out and cannot be observed under certain event sequences. Intermittent presence of errors can cause a lot of frustration and delays during debugging.

As a solution to the above problems, we propose the use of timestamps that mark every event in the system. Timestamps have nothing to do with the real time an event occurred. They can be described as logical timestamps that uniquely identify system events. Their use allows the user to infer useful event relationships, including dependency and concurrency. Fig. 5.3.2 shows an example of an hypothetical DCS ST diagram, and the corresponding timestamps for every event.

5.3.1 Description of Timestamps.

The Timestamp Vector (TV) of any process P_i has N elements, N being the number of processes in the system. Its elements are integers, originally set to zero. They take on their values as follows:

- a. Element e_i is incremented by 1 with every event of P_i .

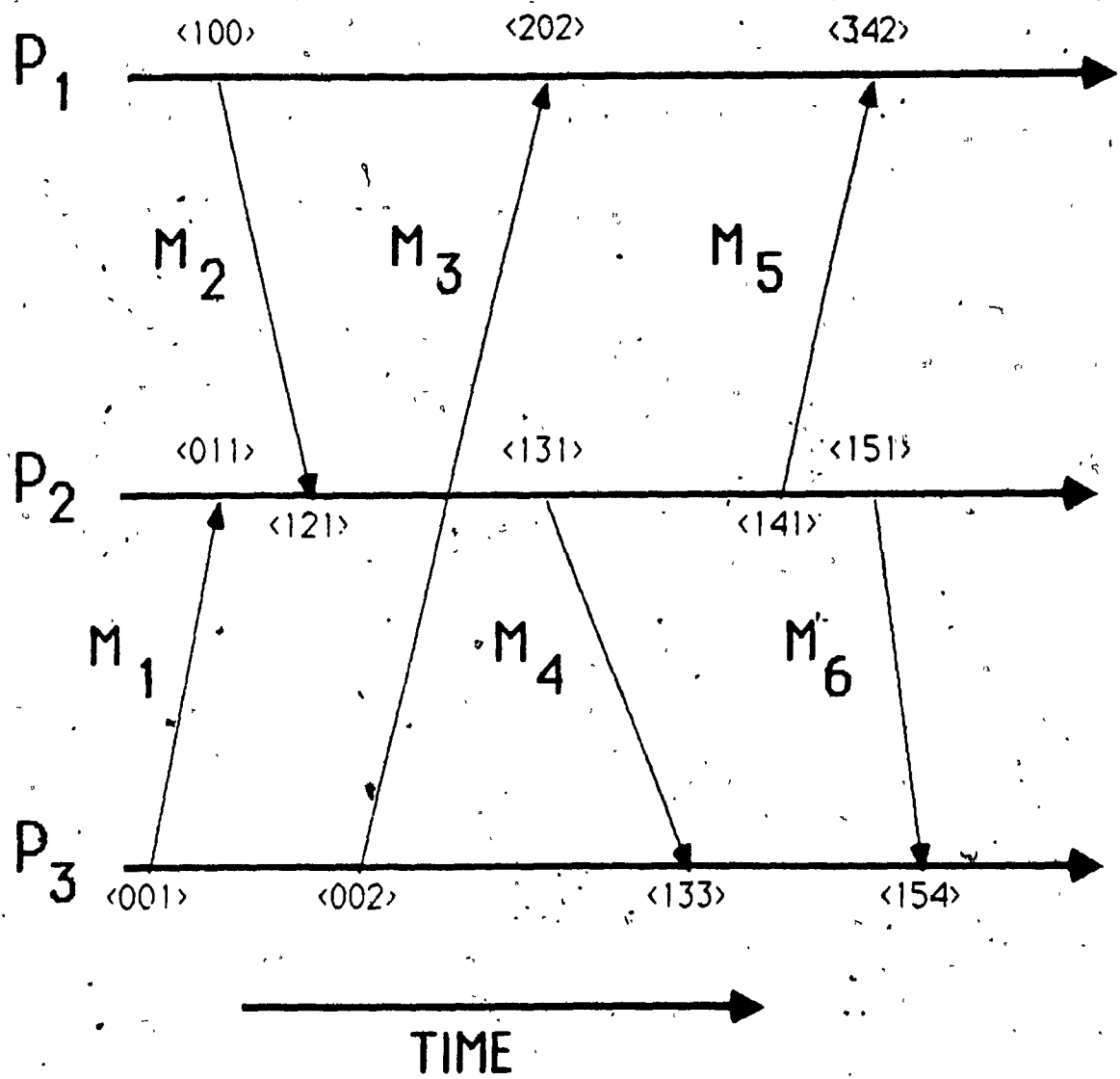


Fig. 5.3.2: ST diagram and event timestamps

- b. For every $j \in \{0, \dots, N\}$ and $j < i$, element e_j is incremented, only with the receipt of a message, iff:

$$e_j' > e_j,$$

i.e. the value e_j' in the received TV is greater than e_j .

From (a) and (b) we can understand how event relationships can be deduced from their TVs. When an event precedes another, knowledge of the first should be reflected in the corresponding entry (i.e. the entry of the first event owner's id) of the second's TV. On the contrary, when no event shows a dependency on the other, we infer that two events are concurrent.

Each local debugger maintains its own copy of the TV. A node debugger counts the send and receive events executed by the process it controls, by incrementing the value of the TV element, that corresponds to the id of this local process. The value of any other element is equal to the ordinal number of the latest event, known to have been executed by the corresponding process. Hence, the value of the i th element in a TV, indicates a particular point in the synchronization behavior of process P_i , as it is perceived by other processes.

Using event timestamps, the ST diagram of a DCS can be built. The following theorem guarantees its uniqueness:

Theorem 5.1: If the whole history of timestamp values on each process is available, then there is only one ST diagram that can be reconstructed.

Before we prove the theorem, we establish the following lemmata:

Lemma 5.3.1: Every event in the ST diagram of a system is identified by a unique timestamp vector value.

Proof: Let E_1 and E_2 be two events, in a three-process DCS. Let also $\langle i_1, j_1, k_1 \rangle$ and $\langle i_2, j_2, k_2 \rangle$ be the corresponding TVs. Then there are two possible cases:

i. E_1 and E_2 belong to the same process, say P_i . Then i_1 must be different from i_2 , by condition (a) in timestamp definition. That is, either $i_1 < i_2$, or $i_1 > i_2$, Q.E.D.

ii. E_1 and E_2 belong to different processes, say P_i and P_j respectively. Let E_3 be the first event on process P_j , that "knows" about E_1 , and let $\langle i_3, j_3, k_3 \rangle$ be its TV. In other words, E_3 is the first event on P_j in whose TV: $i_3 = i_1$. The TVs of E_3 and E_1 differ, since, at least, $j_1 < j_3$. Then, if E_2 precedes E_3 , its TV does not reflect E_1 , and therefore $i_2 < i_3$. Consequently, $i_2 < i_1$ and hence E_2 's TV is different than E_1 's. If E_2 follows E_3 then, the TVs of E_1 and E_2 are different, since $j_1 < j_3 < j_2$, Q.E.D.

Lemma 5.3.2: Events on a sequential process are totally ordered.

Proof: We know that:

- i. No two events can have the same TV value (lemma 5.3.1).
- ii. The element that corresponds to the owner process' id is incremented by one, with every event (TV definition).

From the above, lemma 5.3.2 follows directly, Q.E.D.

Lemma 5.3.3: Send and receive events on a sequential process can be distinguished by their timestamps.

Proof: Let E_1 and E_2 be two consecutive events, on process P_i . Let also $\langle i_1, j_1, k_1 \rangle$ and $\langle i_2, j_2, k_2 \rangle$ be the corresponding TVs. If E_2 is a send event, then its TV differs from E_1 's, only in that $i_2 = i_1 + 1$. No other element can have changed value, according to condition (b), in TV definition. If E_2 is a receive event, then it is the first event in whose TV, the corresponding send event is reflected. Then, both, $i_2 = i_1 + 1$ and, say, $j_2 > j_1$, Q.E.D.

In other words, a send event's TV value differs from the TV value of the previous event, only in the element corresponding to the owner process, which has been incremented by one. The timestamp for a receive event, on the other hand, always reflects the timestamp of the related send event. Incidentally, we should notice, that a receive event is the first event that "learns"

about its matching send event. Only events that directly, or indirectly, follow it may have a knowledge of the existence of that send event in their TVs.

We can now prove theorem 5.1:

Theorem 5.1 Proof: Given a set of TV values, two conditions must be satisfied in order to reconstruct the correct ST diagram:

i. Find the correct order of events on each process.

ii. Match send and receive events correctly.

Condition (i) is guaranteed by lemma 5.3.2. We will prove condition (ii), namely, that no two events (i.e. one send and one receive) can be wrongly matched, given the whole set of TV values on every process. Let E_1 and E_2 be two related events, E_1 being a send event and E_2 its matching receive event. Assume that E_1 and E_2 are not paired, and let E_3 be the send event to which E_2 is wrongly matched. Finally let $\langle i_1, j_1, k_1 \rangle$, $\langle i_2, j_2, k_2 \rangle$ and $\langle i_3, j_3, k_3 \rangle$ be the corresponding TVs. Then, in the resulting ST diagram, there are three possibilities for E_1 and E_2 :

- i. E_1 precedes E_2 , or
- ii. E_1 follows E_2 , or
- iii. E_1 and E_2 are concurrent.

i. This is impossible because, according to lemma 5.3.3, E_2 is the first event in whose TV, the i_{th} element has the value of i_1 . Hence the resulting ST diagram would either be wrong, or have different TV values.

ii. This is also impossible, since E_1 would have to be matched with a receive event that follows E_2 . Then that event should be the first to have a knowledge of E_1 in its TV, again according to lemma 5.3.3. But E_2 's TV value already reflects E_1 , hence we would have a violation of lemma 5.3.3 in the resulting ST diagram.

iii. Finally, E_1 and E_2 cannot be concurrent since E_2 's TV shows a dependency from E_1 .

From (i), (ii) and (iii) we conclude that mismatching any pair of events would result in a ST diagram with different timestamps. Therefore, by matching only correct pairs the authentic ST diagram is produced, Q.E.D.

Below we present an algorithm for the construction of the ST diagram, given the values of the recorded timestamp vectors. The ST diagram is constructed by one process that collects the timestamps from all system processes. The crux of the algorithm is that timestamps can be regarded as N-digit integers. Hence, each sequential process sends to the compiler process a sorted

(in ascending order) vector of integers. The compiler performs the following steps:

1. Mark event timestamps as send and receive (according to lemma 5.3.3).
2. Merge all vectors, and sort the resulting Sorted Timestamp Vector (STV) in ascending order.
3. While the STV is not empty do:
 - i. Match the first send event in STV (of any process P_i) with the first receive event, not belonging to P_i , for which its i th timestamp element is equal to the i th element of the send event's timestamp.
 - ii. Delete the matched pair of timestamps.

Theorem 5.1 guarantees that there is only one ST diagram that can be reconstructed, given the whole set of TV values on each process. If, however, a limited number of timestamps (i.e. a "window") is available for every process, we cannot guarantee a unique ST diagram. This is exemplified in Fig. 5.3.3. If each process saves only the three most recent TV values, both ST diagrams shown in the figure can be reconstructed. We can resolve this problem, if every node debugger, in addition to the

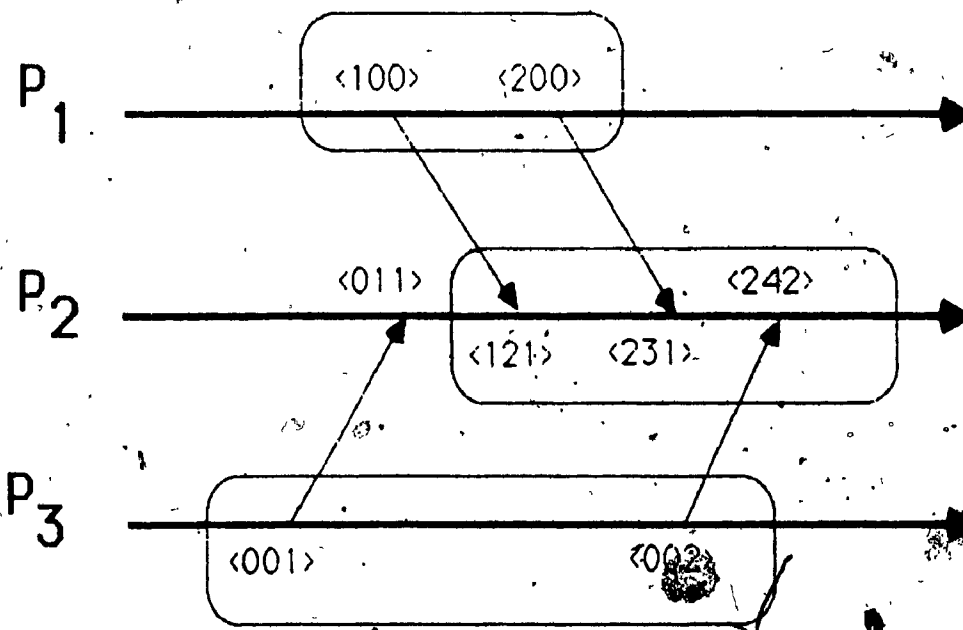
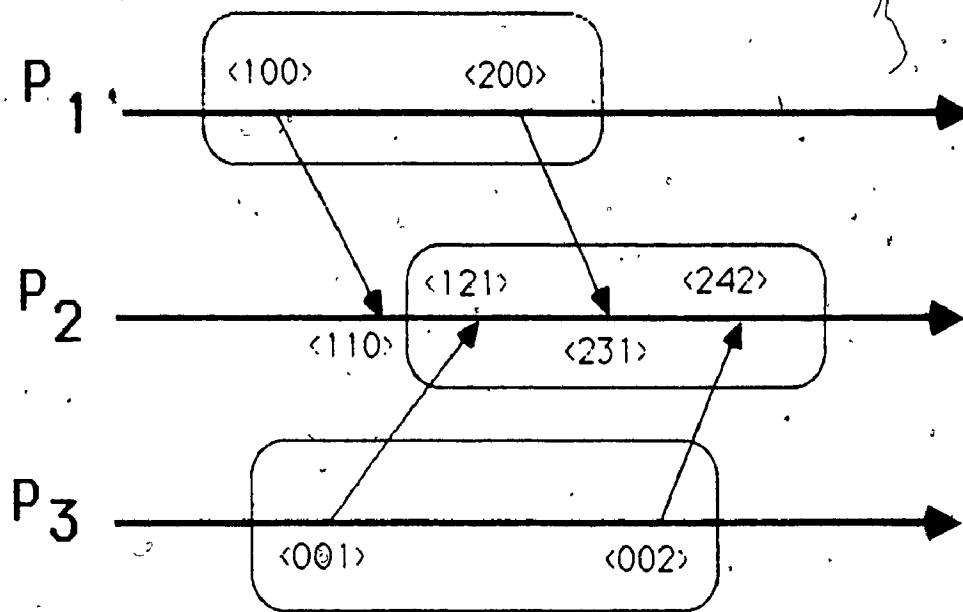


Fig. 5.3.3: Timestamp window and corresponding ST diagrams

last M TV values, saves an extra TV value: the one just preceding the first of the M values.

5.3.2 Use of Timestamps.

A copy of the TV is appended by the node debugger to every outgoing message. On the other end, when the debugger receives an incoming message, it strips off the TV copy that was attached by the sender. The receiver updates its own copy, by setting every element with a lower value to the value of the corresponding element in the received TV. Received TVs are also recorded for later reference. Every node debugger saves all TV instances associated with the last M events, M being a parameter defined at the beginning of the debugging session by the programmer.

Timestamps can be considered as a complement to channel counters. They enable programmers to distinguish between traces for the same cut, based on the sequence of events that led to that cut. When system execution is suspended the values of the recorded TVs can be used to reconstruct the ST diagram of the system, in the window of the last M events executed by every process. Then the sequence of event occurrence can be checked against specification. Synchronization errors, generate faults that can be perceived as violations in the synchronization requirement specification of a process.

5.4 Implementation Issues.

The distributed debugger is conceived as a distributed program that encapsulates the application program. On every node of the system, where an application process is running, there is a debugger process³, that acts as a buffer process between the application process and the communication subsystem. These debugger processes are called node, or local, debuggers. The node debugger spawns the application process, controls its execution and intercepts all its messages. For simplicity, in the ensuing discussion, we assume one application process per node. On a separate node, called the central site, there is a master debugger process. This process, called the master debugger, provides the distributed debugger's user interface.

Every application process has a unique identification number in the system, by which it is known to the debugger. For the present, we assume a fixed topology for the application program. The status of an application process can be either "live" (i.e. running), or "blocked" (i.e. suspended). The channels of a process, whose status becomes blocked, change their status to "dead". A blocked process neither sends nor receives any messages. Messages arriving at "dead" channels are saved in "unlimited" buffers and played back when the blocked process resumes execution.

³ The term "process" refers either to the application process, or to the composite process, formed by the application and the debugger processes.

Node debuggers should offer all the features of conventional sequential debuggers. In addition, they should offer additional tools geared for the distributed environment. Hence a node debugger intercepts messages of the local application process, and processes their TVs. It should also record the contents of messages, for later examination by the programmer. Messages could be saved at one end, either the sending or the receiving, if we assume reliable communication. The node debuggers help also implement the so-called step execution mode. This is done, by requiring from the debugger, which spawns and controls process execution, to request the user's permission for the execution of every send or receive event. Once the permission is granted, the debugger allows the process to continue. Node debuggers communicate among themselves via control messages.

The central site runs on a node with no application processes. It is at this site, that the programmer issues commands to the debugger and receives any produced output. The master debugger also communicates with node debuggers, via control messages. It compiles any received information and reports to the user. The programmer should also be able to access, from the central site, the tools offered by local debuggers. In this case, the central site should act as a smart terminal, multiplexing output from different processes in separate windows.

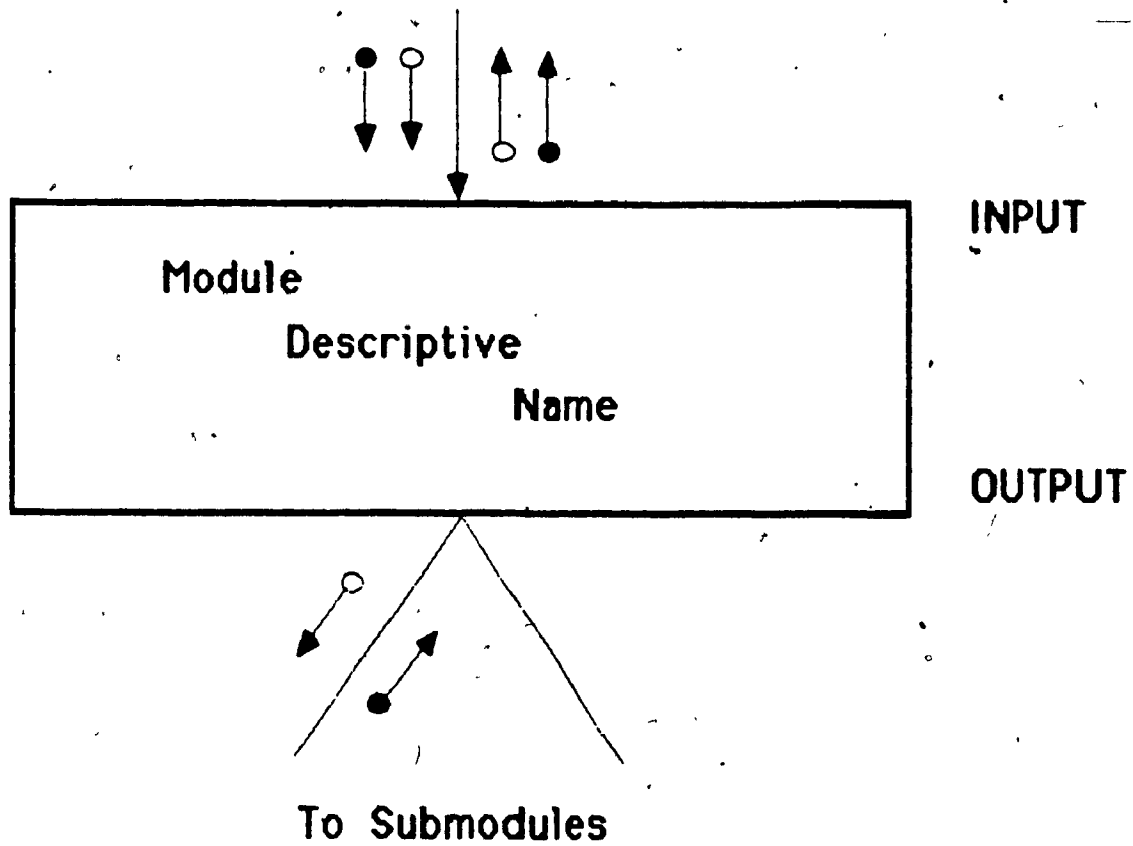
The central site offers the capabilities of setting session parameters (i.e. number and type of message colors, ST diagram

window size, etc), defining breakpoints, granting permissions for the execution of events in step mode. It is also the compiler process, that constructs the ST diagram of the system. Finally, the ability to establish recovery lines during execution and to request rollback to a particular recovery line, can be incorporated in the master debugger's features.

5.5 High Level Specifications of the Distributed Debugger.

In this section, we present the specifications for the main modules of the distributed debugger. The specifications represent a first level functional decomposition of the debugger, and their purpose is to convey a feeling for its structure.

For the specifications we have used structure charts [17]. These are hierarchical diagrams that define the overall architecture of the program, by showing its basic modules and their interrelationships. The basic building block of a structure chart is a module. Modules are represented by rectangles of the type shown in Fig. 5.5.1. Each rectangle has a descriptive name, that explains the task the module performs. Open-circle arrows show data passing between modules, while filled-circle arrows indicate control information passing. At the upper right corner of the rectangle, in a first level module, the input to the module is indicated, and at the lower right corner the output, if any. The specifications for the debugger, are shown in figures 5.5.2 through 5.5.4.



○ → Data Passing
● → Control Information Passing

Fig. 5.5.1: Basic block of a structure chart

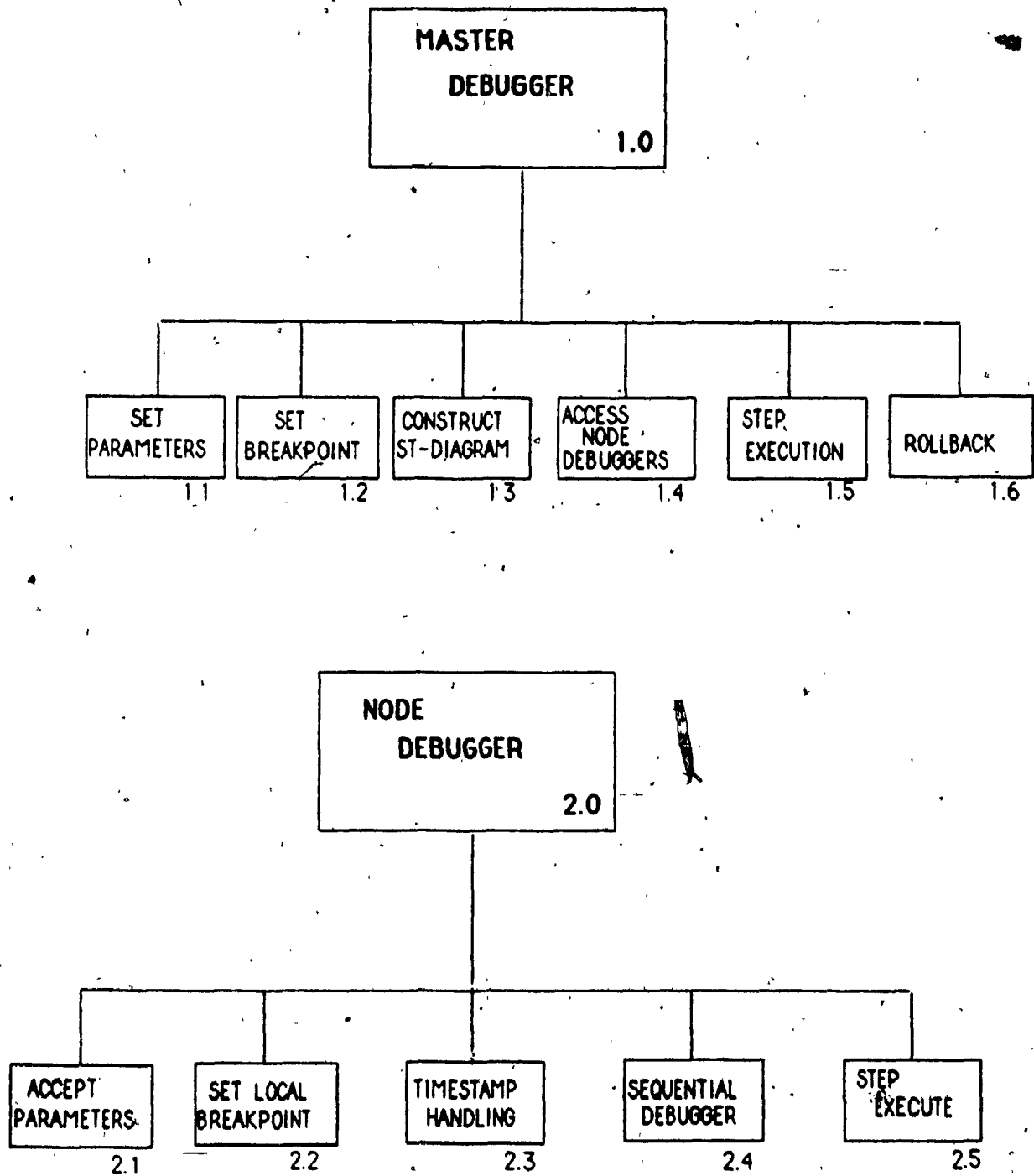


Fig. 5.5.2: Structure chart of the distributed debugger

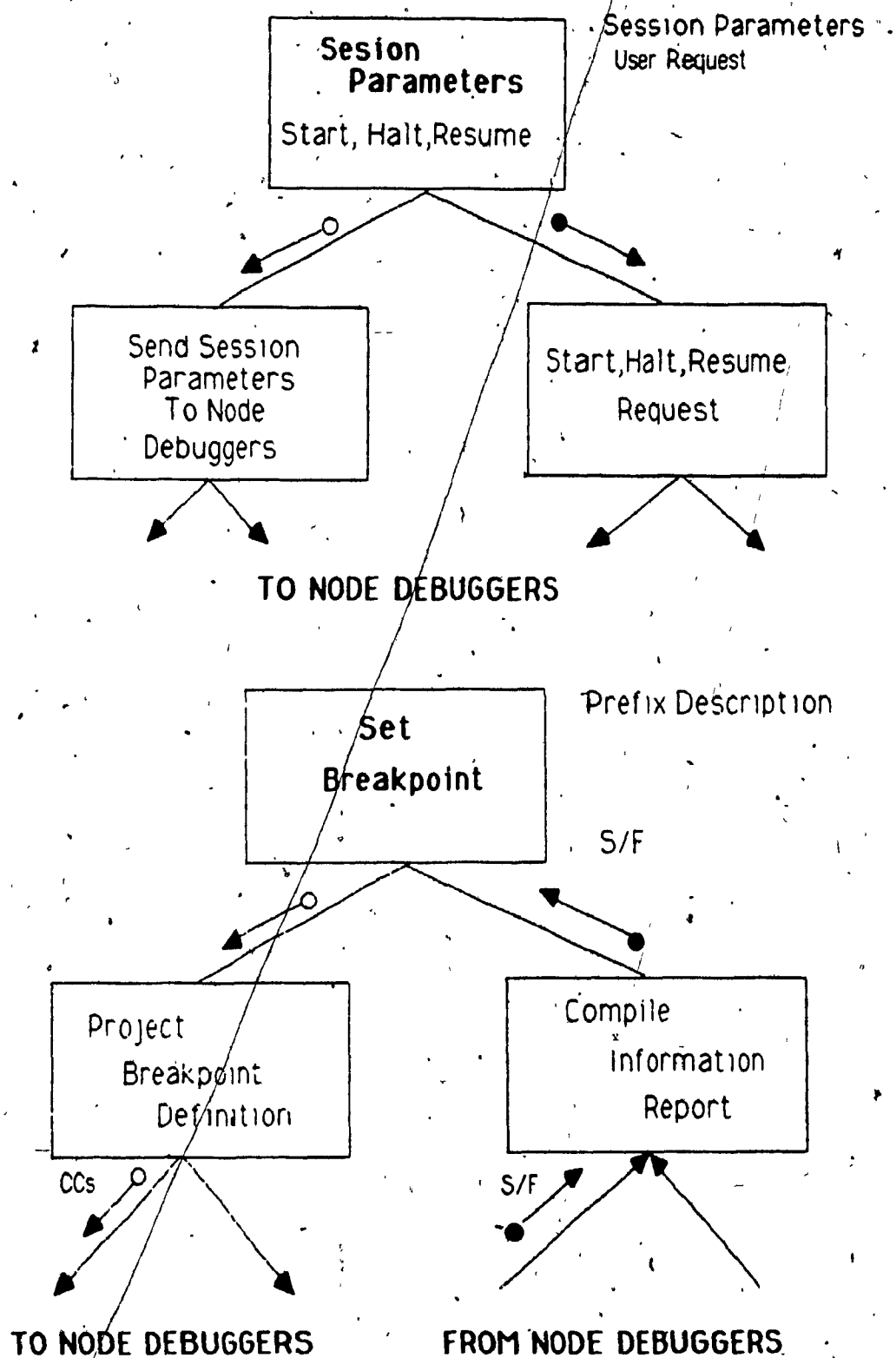


Fig. 5.5.3a: Structure chart of the master debugger

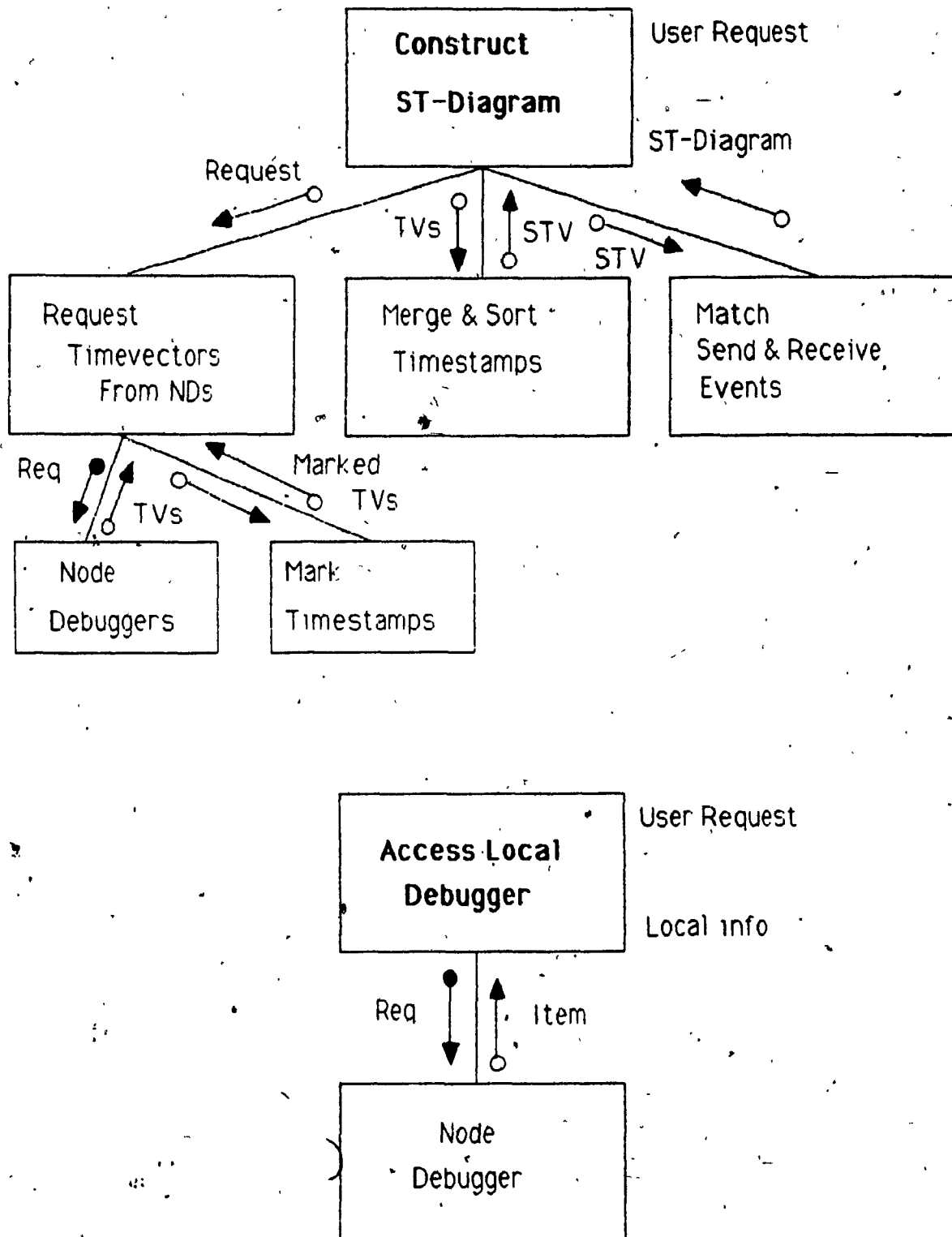


Fig. 5.5.3b: Structure chart of the master debugger (cont)

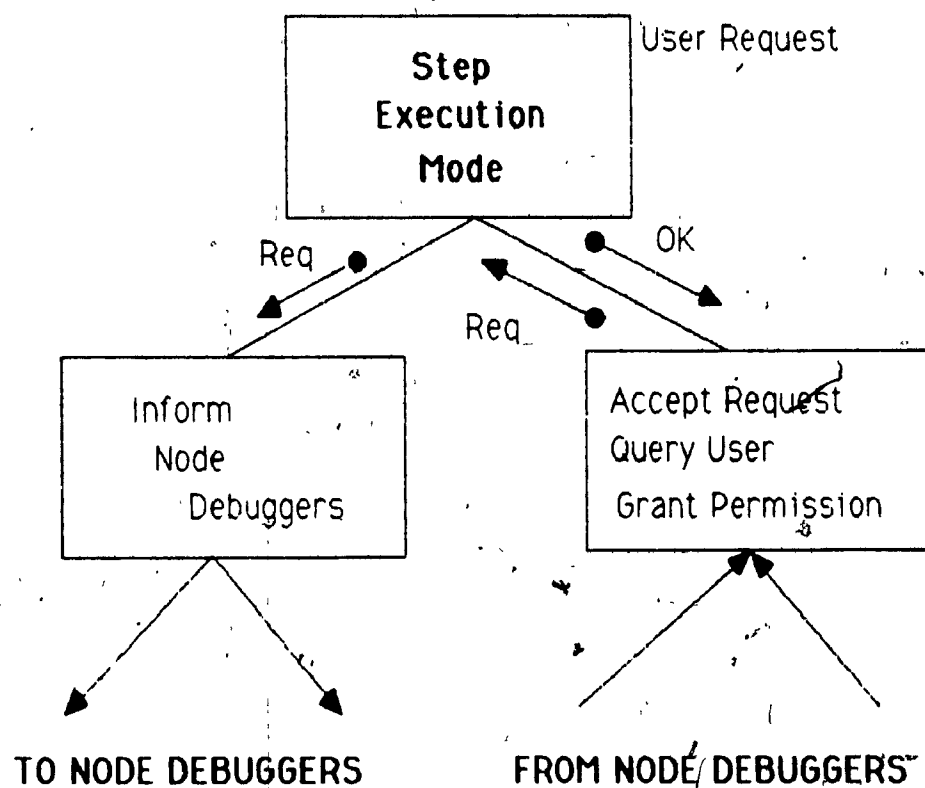
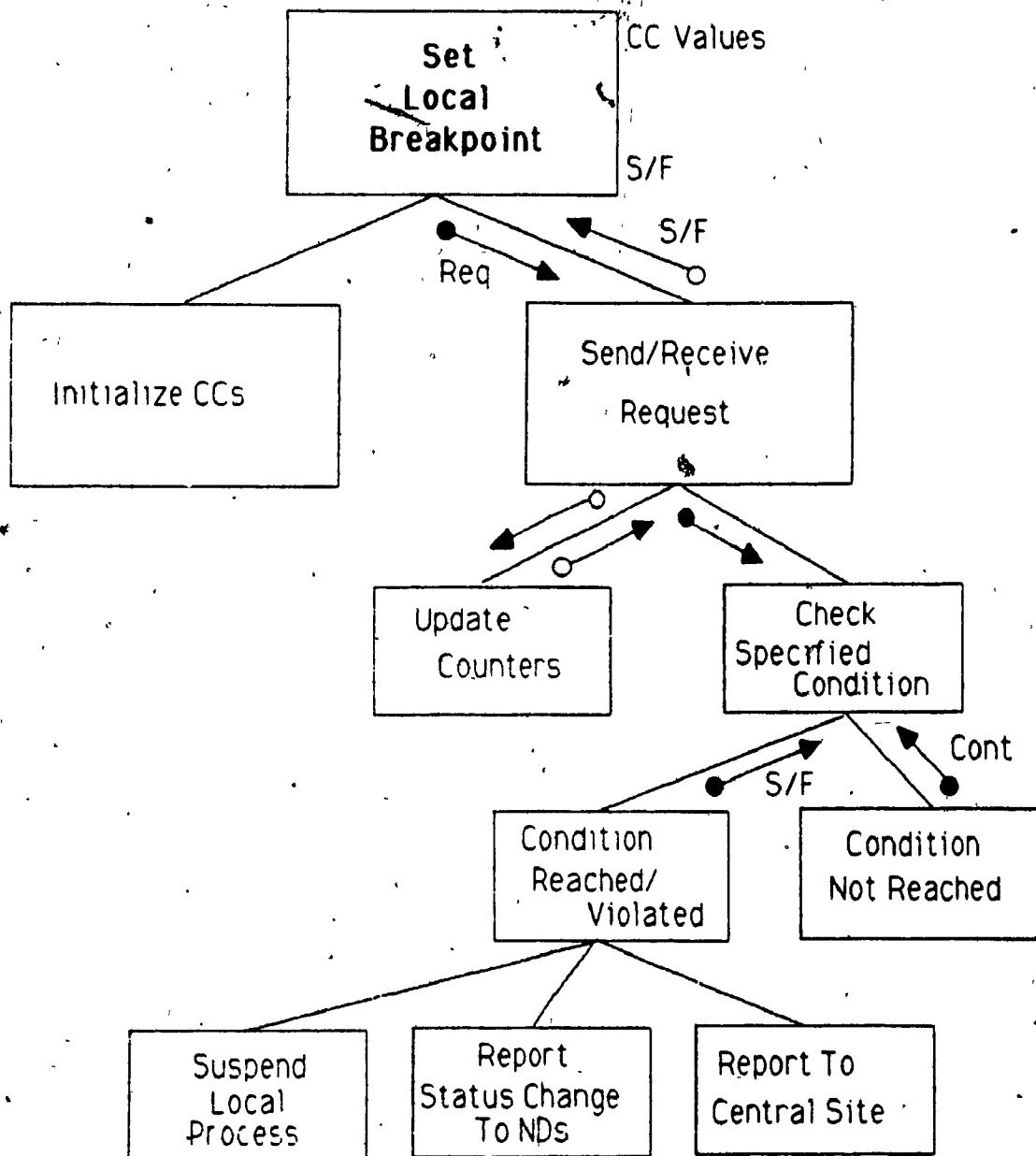


Fig. 5.5.3c: Structure chart of the master debugger (cont)



S/F: Success/Failure.

Fig. 5.5.4a: Structure chart of a node debugger

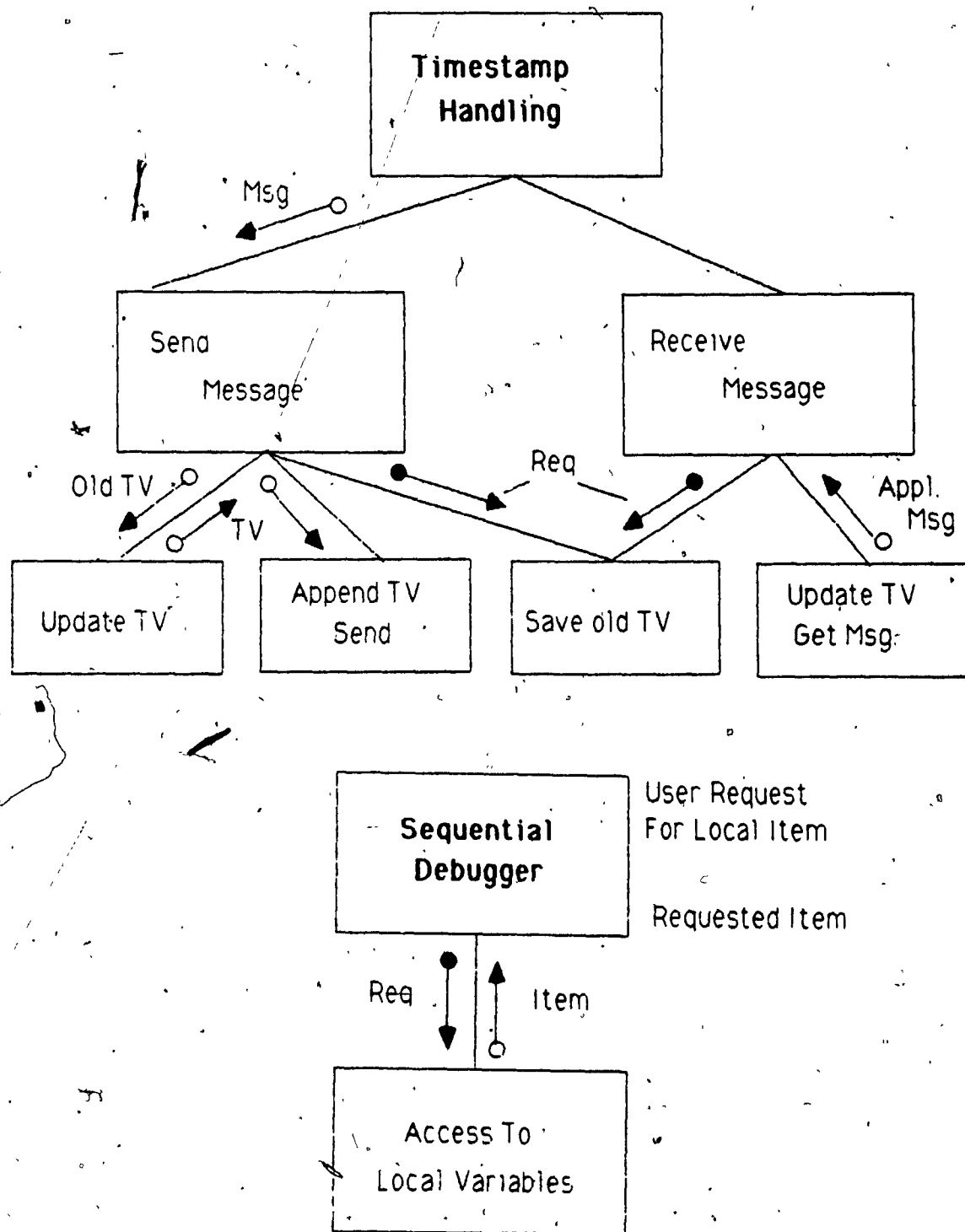


Fig. 5.5.4b: Structure chart of a node debugger (cont)

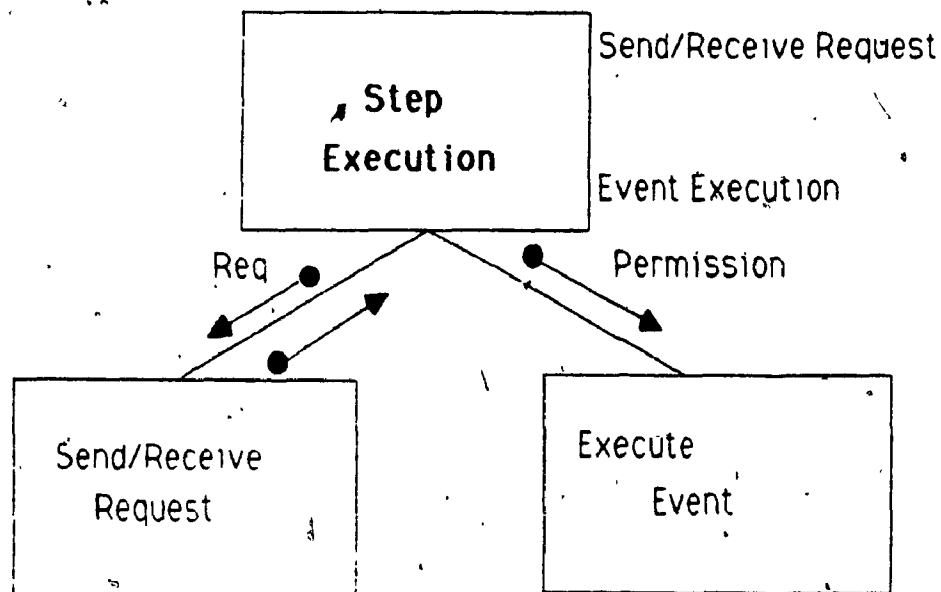


Fig. 5.5.4c: Structure chart of a node debugger[™] (cont)

5.6 Related Work.

In [4] and [5], P. Bates et al., describe a debugger for distributed systems, based on "behavioral abstraction". Their approach to debugging is similar to ours, in that it is based on a comparison of the actual system behavior against a model of its activity. However, their method of describing the model of system behavior is different. The programmer defines significant system behavioral models in the Event Definition Language, or EDL. EDL event definitions describe the type of events that may occur, and their attributes if they do occur. Lower level event definitions can be combined, through filtering and clustering, to provide definitions for higher level events. Once the system behavior has been described, the debugger monitors the system behavior, and compares it to the given description.

We feel that their model of a distributed computation has not been well defined. More precisely, EDL is a specification language that is close to the implementation, and is not useful for design specification. The usefulness of a model lies in the fact that it can be used by all system users, be they designers or implementors. In this aspect, we feel that EDL is not a comprehensive tool for system specification, and thus there is the problem of communication between the designer and the implementor. The proposed debugger will accept models of system behavior, will detect system misbehaviors by comparing it to actual behavior, and eventually suggest rudimentary bug remedies

to the user. Building the proposed debugger, however, may prove a difficult task, since a lot of sophisticated components must be incorporated, such as the behavior monitor and the event recognizer. The latter requires a sophisticated algorithm for the recognition of event occurrence, whose complexity can be unbounded. This algorithm cannot be easily distributed, and thus it can become the bottleneck of the system.

Baiardi et al. present their method for distributed debugging in [3]. Theirs is another approach based on the comparison of expected program behavior to actual behavior. The proposed debugger will accept the specification of a program and will be able to detect the presence of errors, by discovering discrepancies in system behavior. Hence, the debugger is also oriented towards automatic error detection. The debugger supports a specific concurrent language, called ECSP. While the proposed model is general, the specification language is tied to ECSP, limiting the debugger's usefulness.

Another potential problem stems from the operation of the debugger. Node debuggers trap each request for process communication made by their local process. They check whether the requested interaction conforms to process specification, and they also request permission from those processes, whose specification controls (i.e. contains) the particular interaction. The application process is allowed to proceed, only when permission is granted by the local debugger and all requested processes. We believe that this mode of execution will

impose an unacceptable overhead, and will confine to a high degree the autonomy of process interactions.

M. Garcia et al. [11], present another approach to concurrent debugging, using Petri nets to describe process interaction. The debugger monitors system events and attempts to recognize specified behaviors. When a predefined behavior is recognized, it responds by executing certain commands that have been dictated by the user. The EDL [4] is used for the description of expected behaviors. Petri nets are used for an off-line visual representation of recorded process interaction. The debugger is implemented on a system that supports an encapsulation mechanism, called "object". An "object" comprises a resource and its appropriate operations. Events related to the access of an "object" are trapped by the debugger, and a log file is updated. Once execution is suspended a Petri net player is called to analyze the computation. The debugger was implemented for a set of concurrent processes on a single node. As the authors observe, problems of naming, ordering of events, and time stamping must be answered, before the method can be extended to distributed systems.

Bruegge and Hibbard [6], propose a high-level debugging mechanism, called "Generalized Path Expressions". Path expressions were originally intended for the synchronization of parallel processes. Their purpose is to restrict the set of possible behaviors of concurrently executing programs, to those that are appropriate. Although the authors discuss their

suggestions in the framework of centralized systems, they also propose the use of path expressions for distributed debugging. The main idea is to regard the application program and the debugger, as two concurrent activities that synchronize at places specified by the path expression. A lot of their ideas have been adopted in [11].

Ed. T. Smith describes tools for message-based communicating processes, in [23]. Events of interest for his debugger are sending and receiving of messages. The event detection language is simpler than EDL, or Generalized Path Expressions, and consists of boolean expressions, called triggers. However, his work is intended for a multi processing programming environment, which is radically different from ours.

Finally, there are quite a few descriptions of debugging facilities, that have been implemented for existing experimental distributed systems, such as [7], [12], [14] and [15]. They discuss a lot of implementation issues, but as the described systems are tied to the particular programming environments, they do not address any of the underlying problems of distributed debugging.

CHAPTER 6

DEBUGGING TWO DISTRIBUTED ALGORITHMS

In this chapter, we present a few examples of distributed debugging. Emphasis has been placed in providing exact process specifications, and showing how most of the synchronization errors generate faults that can be perceived as violations of those specifications. The hypothetical errors might seem trivial, or ad hoc; however, we feel that they do serve the purpose of highlighting the debugging method and the use of the suggested tools. The examples are presented, for the sake of simplicity, in the trivial case of a three-process DCS. They can be easily extended for systems with a larger number of processes. The two algorithms that we discuss are a global state detection algorithm and a network message-routing algorithm. We do not claim, that they cover the whole range of possible distributed algorithms. Nevertheless, we believe that they are typical examples of a broad class of distributed algorithms.

6.1 The VLR Global State Detection Algorithm.

This algorithm [16], provides the global state of a distributed system, even when the communication subsystem does not guarantee ideal behavior. The VLR algorithm is not a user application; it offers its services to other user applications as a utility of a distributed programming environment. It is implemented by the global state kernel, a software layer between the application and the communication subsystem layers. For our purpose, we will ignore the implied application program and consider the VLR as the debuggee.

First, we informally present the VLR algorithm. The ensuing discussion, focuses in the synchronization aspects of a VLR process, rather than any internal actions that are dictated by the algorithm. A more formal discussion of the VLR algorithm can be also found in [24]. VLR was developed for a distributed system expressed by the familiar model, of a finite number of identical processes, interconnected with logical point-to-point channels. In this case, each process is a global state kernel. When a process wants to obtain the system state, it initiates a global state recording. We refer to this process as the initiator.

The initiator sends a special marker message to all its output channels, to inform other processes about a new global state recording. Any process that receives a marker message on one of its channels, and decides that a new state recording is

required, will propagate the same message, through its own output channels, to other processes. However, no marker messages are sent to the initiator, or to the process, if any, from which it was first informed about that particular recording. Finally, the process will forward to the initiator a message containing the recorded state.

In a fully connected three-process DCS, the behavior of any of the three processes P_1 , P_2 or P_3 , is produced by the following pomset generators:

$$\begin{aligned} B_1 &: r_i \rightarrow r_j \rightarrow s_i^s \\ B_2 &: r_i \rightarrow s_j \rightarrow (s_i^s \parallel r_j^*)^1 \\ B_3 &: r_i \rightarrow (s_j^s \parallel r_j) \\ B_4 &: (s_i \parallel s_j) \rightarrow (r_i^s \parallel r_j^s) \end{aligned}$$

where $i, j \in \{1, 2, 3\}$ and $i \neq j$. The subscript in a send (receive) event indicates the receiver (sender) process. In other words, it associates an event with a particular port, at which it is observed. The superscript represents an event label. There are two types of event labels. Events marked with an "s" are associated with the sending (or receiving) of state conveying messages. Unmarked events are associated with marker messages. As we have already mentioned, event labels correspond to message colors, and they can be identified by the debugger. Hence, the

¹ The symbol "*" denotes that the marked event will be observed either once or not at all.

debugger maintains two color sensitive counters for each channel.

The synchronization behavior of a VLR process can be expressed as the concurrent execution of pomsets produced by the first three pomset generators, preceded, or followed, by the execution of B_4 . Symbolically, we have:

$$\pi \delta (B_1^+ \parallel B_2^+ \parallel B_3^+ ; B_4 \mid B_4 ; B_1^+ \parallel B_2^+ \parallel B_3^+)$$

where π symbolizes the prefix closure, that makes a process abortable by permitting it to get only part of the way through its computation. The symbol δ indicates local linearization, that enforces serialization of colocated events. This is an essential precondition for a sequential process, where events at the same channel-end (port) are necessarily linearly ordered. Notice, however, that local linearization does not contain any information as to which of the possible pomsets each event comes from. Finally, the "+" symbol, as a superscript, means indefinite concurrence.

Let us imagine the following error scenario for P_2 's implementation: when P_2 learns of a new state recording, it does not inform other processes about the new initiation of the algorithm. Let us further assume, that the programmer who tests the implementation of the algorithm, observes the following misbehavior: the algorithm is initiated once by process P_2 , and it terminates correctly. Following that, a second initiation by process P_1 , fails to terminate, because process P_3 never responds

with a state message. Although, the sequence of initiations may seem fairly simple, we should keep in mind that, this is observed during the testing phase, when it is easy to control execution parameters.

We will examine the possible debugging procedure in the case of two systems: the familiar fully-connected DCS, Fig. 6.1.1a and a partially-connected DCS, Fig. 6.1.1b. Let us first look into the simpler case of the partially connected DCS. The programmer's first reaction would be to suspect process P_3 . The particular process interconnection modifies the possible behaviors. Actually, the only possible pomsets are the following:

$$\begin{aligned} P_1 &: (r_2 \parallel r_3) ; s_2^S ; s_2 | ; (r_2^S - | r_3^S) \\ P_2 &: (s_1 \parallel s_3) ; (r_1^S \parallel r_3^S) ; r_1 ; s_3 | ; s_1^S \\ P_3 &: r_2 ; s_1 ; s_2^S ; r_2 | ; s_1^S \end{aligned}$$

The first suspicion is that P_3 does not execute, for some reason, its last event, s_1^S . Hence, the programmer decides to suspend execution and examine the state of P_3 , and the sequence of event occurrence, just before the execution of P_3 's last send event. For this reason, he performs process composition and chooses to stop the system after the prefix that consists of the events to the left of the symbol " $|$ ", has been observed. If we translate this prefix into channel counter values, we get:

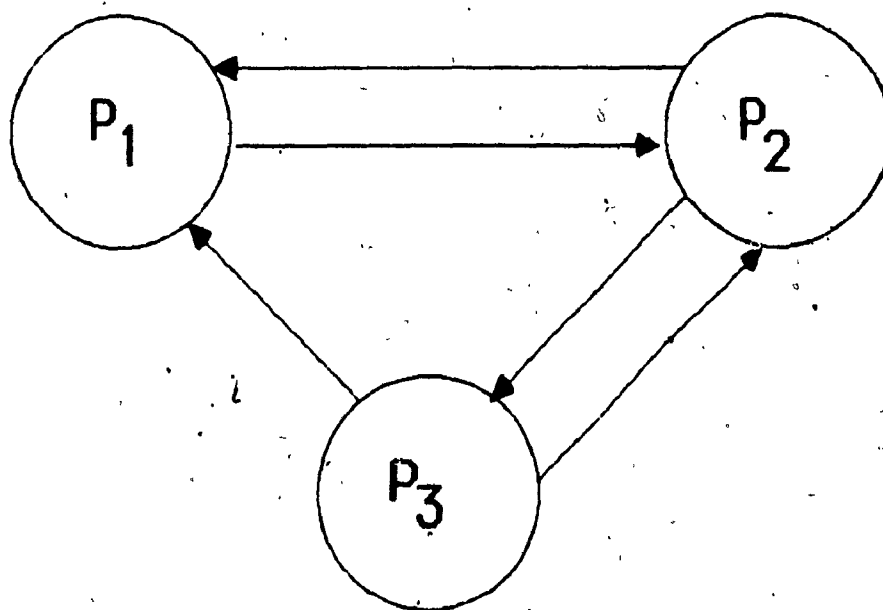
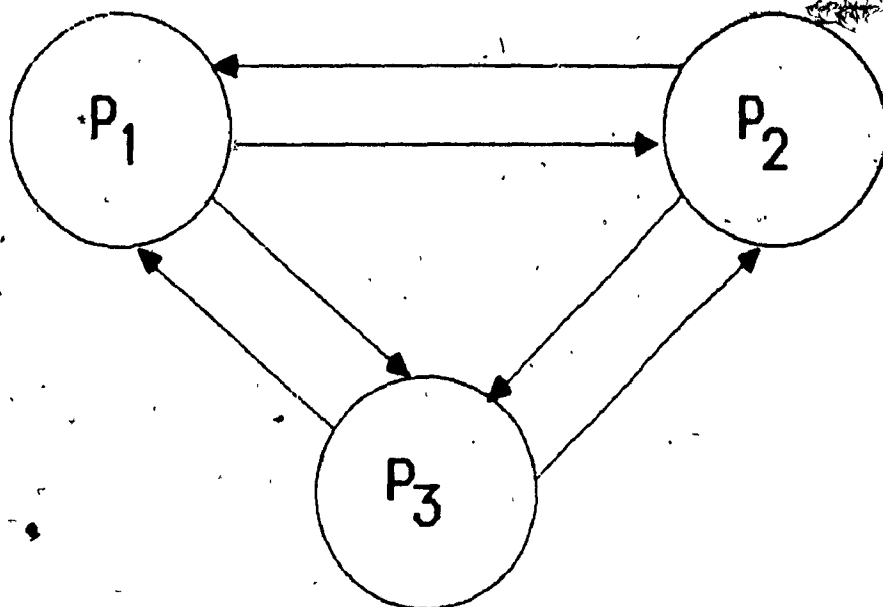


Fig. 6.1.1: Fully and partially connected DCSs

$$P_1 : CC_{12} = 1$$

$$P_2 : CC_{23} = 2 \text{ AND } CC_{21}^S = 0$$

$$P_3 : CC_{23} = 2 \text{ AND } CC_{31}^S = 0$$

Process P_1 will eventually execute the specified prefix and it will be suspended. Process P_2 , however, will skip event s_3 , and it will execute event s_1^S . This will be noted as a violation of the specified cut, and P_2 will be blocked. Eventually, process P_3 will also be blocked, since both its input channels have become "dead", and consequently it can never reach the specified cut. At this point, the programmer is informed of the failure to reach the specified breakpoint. Naturally, he requests construction of the ST diagram of the execution up to this point. Upon examination of the ST diagram, the faulty behavior of process P_2 is revealed, since it exhibits the sequence:

$$P_2 : \dots r_1 ; s_1^S \dots$$

from which s_3 is missing. The programmer has now the first indication of a bug in P_2 . Subsequent tests will confirm his suspicions and will direct him to a closer examination of P_2 . The ensuing examination should reveal the bug in P_2 's code.

Let us now look in the case of the fully-connected DCS. The same bug, under the same test case, would cause an observable misbehavior only in the case that a marker message from process P_1 fails to reach P_3 . Otherwise, even though process P_2 does not

inform P_3 of the latest initiation by P_1 , P_3 will eventually find out directly from P_1 , and it will respond by sending back its state message.

Let's assume that process P_3 never receives the message from P_1 . In this case, the programmer observes again that the algorithm does not terminate, because P_3 never sends its state to P_1 , the second initiator. The possible behaviors for the three processes are more complicated now:

$$P_1 : (B_1 \mid B_2 \mid B_3) ; B_4$$

$$P_2 : B_4 ; (B_1 \mid B_2 \mid B_3)$$

$$P_3 : (B_1 \mid B_2 \mid B_3) \quad (\text{not complete})$$

where " \mid " denotes choice. The first suspicion is that P_3 fails to respond to a state recording request. In other words, P_3 never executes the s_1^S event. The programmer decides to suspend the process just before the execution of that event. Since he does not know in advance which behavior each process will exhibit, he should now define a class of pomset/prefixes, rather than a single prefix. The specified prefixes for each process correspond to the following channel counter values:

$$P_1 : CC_{12}^S = 1 \text{ AND } CC_{12} = 1 \text{ AND } (CC_{13} = 1 \text{ OR } CC_{13} = 2)$$

$$P_2 : CC_{21}^S = 1$$

$$P_3 : CC_{32}^S = 1 \text{ AND } (CC_{23} = 2 \text{ OR } CC_{32} = 1)$$

According to the specified counter values, P_1 is stopped after it has initiated the algorithm, P_2 after it has responded to P_1 's initiation, and P_3 just before it sends its state message. Eventually, both P_1 and P_2 pass the specified prefix, while P_3 fails since it never hears from any of the other two processes. By looking at the ST diagram that is constructed the programmer realizes that P_3 never received a message informing it of the initiation of the algorithm by P_1 . He can also discover the misbehavior of P_2 , which exhibits the familiar sequence:

$$P_2 : \dots r_1 ; s_1^s \dots$$

a violation of B_2 . The programmer now, turns his attention to process P_2 . He has isolated the error that was responsible for the observed fault.

Incidentally, we can display how elusive a synchronization bug can be, by looking at Fig. 6.1.2b. Here, process P_2 receives a message from P_3 , before responding with a state message to the initiator P_1 . Although, the two ST diagrams in Fig. 6.1.2a and 6.1.2b differ only in the interchange of two events, there is no violation in the specification of P_2 , in the second diagram. This demonstrates how non determinism can mask synchronization errors, and why determining the exact event sequence is crucial for debugging. It also shows that a bug should, somehow, manifest itself before any correcting action is taken.

According to a second imaginable error scenario, when

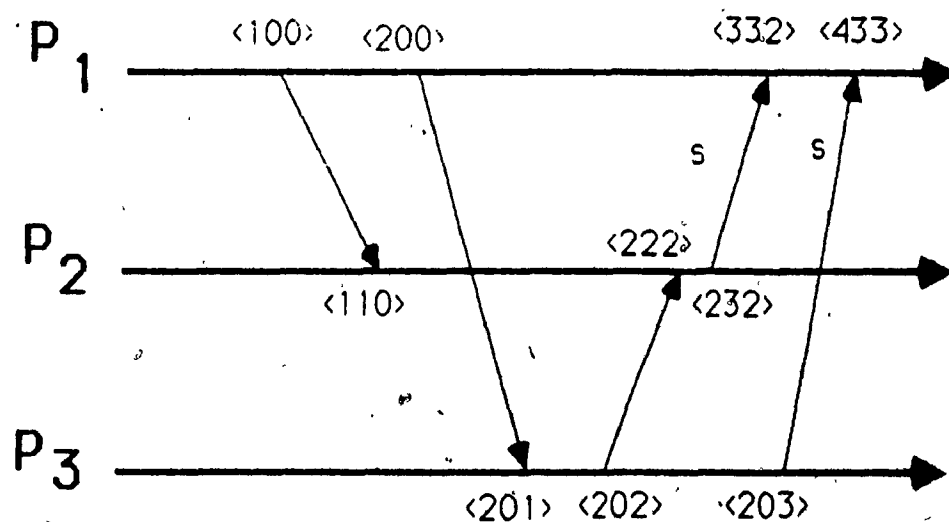
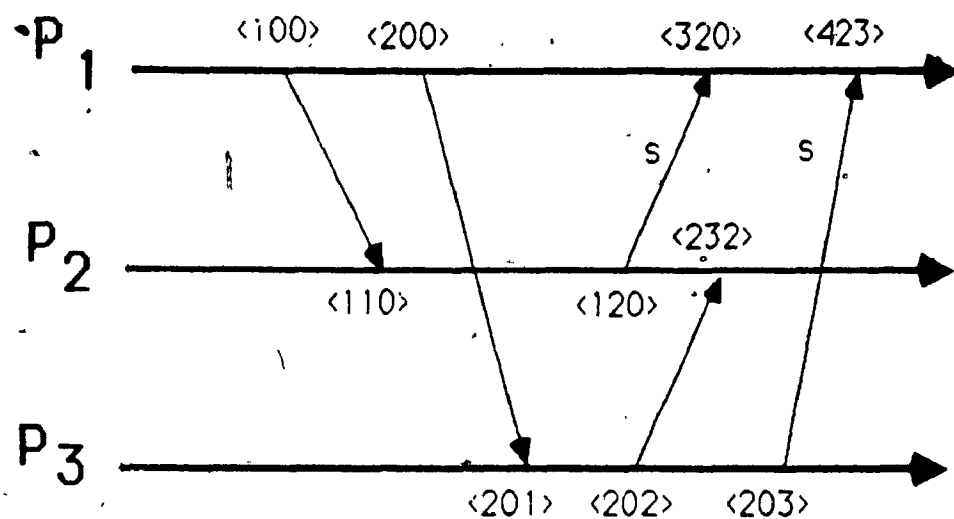


Fig. 6.1.2. Invalid and valid ST diagrams for the VLR algorithm (scenario 1)

process P_3 is informed about a state recording by a process other than the initiator, it insists on sending a marker message back to that process. This is an elusive bug, because it does not cause any observable failures. Actually, this error can even go unnoticed, masked by inherent system nondeterminism, due to variable channel delays. The receiving process has no means of knowing, whether the sender was informed about the recording by its own marker message, or by the initiator's marker message, in which case the extra message would be justified. This is an interesting example of the subtleties of distributed programming and the sometimes puzzling problems of distributed debugging.

Since there are no observable misbehaviors, the programmer has no reasons of suspecting the existence of an error. The only debugging method that could possibly help in discovering a bug of this nature, is to define a general breakpoint, stop the system and examine the ST diagram. It can be rightly argued that in this case the error is discovered through testing, rather than debugging. A general breakpoint is usually based on a particular global invariant that the values of the counters should conform to. In the VLR algorithm, it is easy to deduce that the number of the state messages that have been sent, should be equal, after the completion of a particular initiation of the algorithm, to the number of the messages that have been received. This observation leads to the following prefix: each system process is suspended after it has executed the events that complete one, or more initiations of the algorithm, no matter who the initiator

was. The prefix is defined in every process, by two possible sets of channel counter values:

$$CC^S_{O/ANY} = N \quad \text{OR} \quad CC^S_{I/ALL} = N$$

where O stands for Output and I for Input. N is any integer, depending on how many initiations of the algorithm we want to observe. The condition specifies that a process should be suspended, when any output channel counter, for state messages, records the sending of N such message. This set of counter values corresponds to the process having responded to N state recording requests. The values of the other counters are "don't care". The alternative, is to suspend a process when the state message counters, for all its input messages, have the value of N. This clearly signifies that this process was the initiator, for N times. Again the values of the rest of the counters are "don't care". We should point out, that one of the two alternative prefixes should always be observed, at one point or another during program execution. Consequently in this case, failure of the system to reach the specified prefix would indicate faulty behavior.

Once the defined prefix is observed, and system processes have been suspended, the programmer commands the debugger to construct and display the ST diagram, using the recorded message timestamps. Based on this the user can compare actual process behavior to process specifications. Assume that the ST diagram

that is constructed is the one shown in Fig. 6.1.3. From this, we can extract P_3 's behavior, as a Totally Ordered Multiset (tomset) of events. In order for the derived tomset to be a valid process behavior, it should correspond to a linearization of a possible process behavior. As we can see, the recorded behavior for P_3 is:

$$r_2 \rightarrow r_1 \rightarrow s_2 \rightarrow s_1^s$$

which, if we replace i for 2 and j for 1 , becomes:

$$r_i \rightarrow r_j \rightarrow s_i \rightarrow s_j^s$$

The underlined event is a clear violation of B_3 , which very specifically focuses the programmer's attention to a part of P_3 's sequential code.

In both error scenarios, a judicious use of the channel counters, along with the debugger's capability to reconstruct the actual ST diagram, give the programmer enough evidence about the location of the errors.

6.2 A Message Routing Algorithm.

This algorithm [1], maintains message-routing tables for a computer network in which communication links can fail and be repaired. Below, we discuss very briefly the basic ideas of the

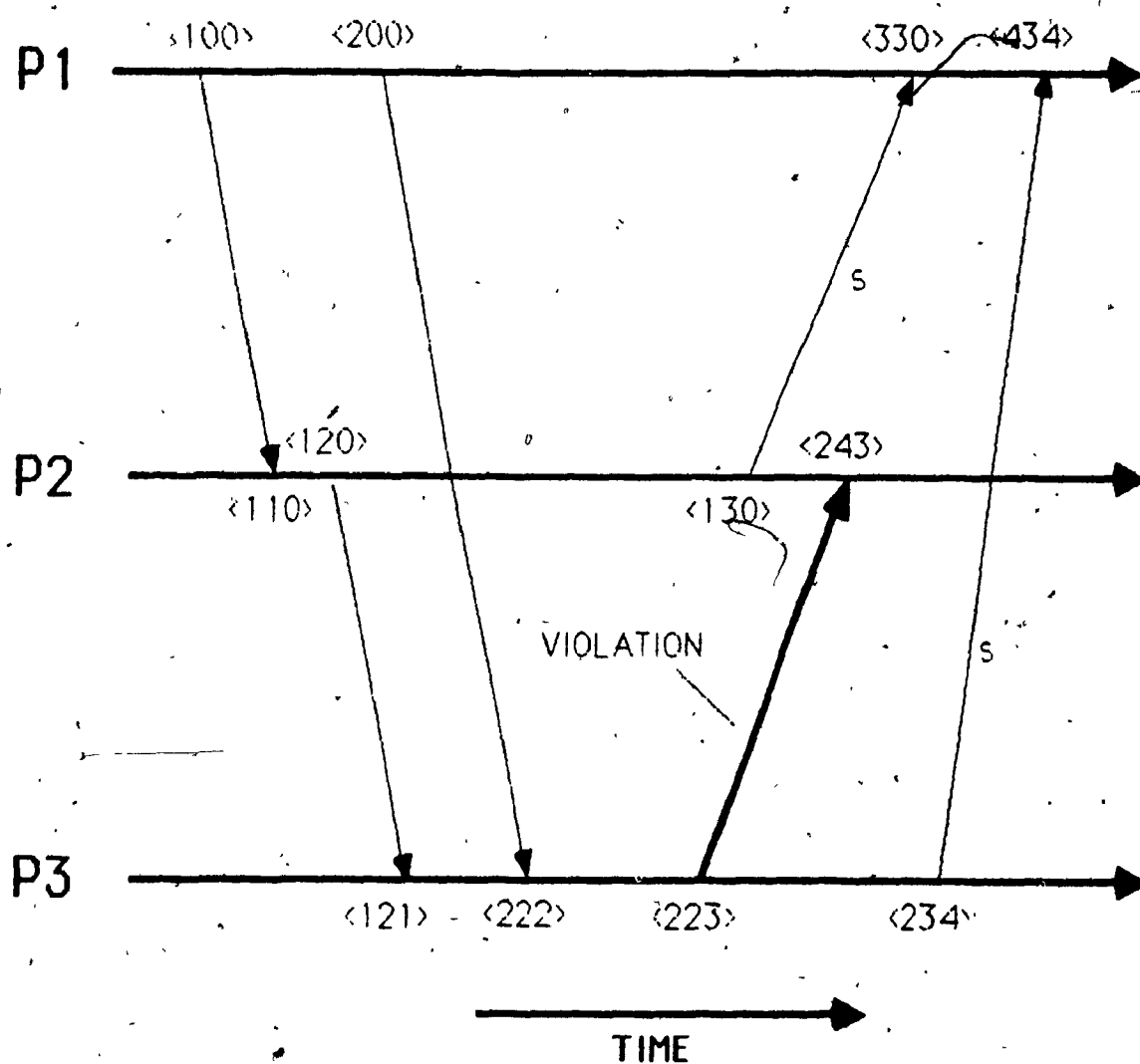


Fig. 6.1.3 Reconstructed ST diagram for the
VLR algorithm (scenario 2)

algorithm, before moving into the characterization of a process by its synchronization behavior. The optimal routing of messages in a network of computers, can be achieved if each computer has the following information:

Its distance to every other computer.

Who its neighbors are.

The distance from each of its neighbors to every other computer.

Assuming a computer knows who its neighbors are, the remaining problem is that of computing the distances. An additional implication is that communication links in the network can fail and be repaired. A computer is notified when a communication line to one of its neighbors, fails or is repaired. Hence, every computer must, at all times, maintain updated distance tables, since failure or repair of the communication line to a node, can affect its minimum distance from other nodes.

In the fully distributed version of the algorithm, each process independently recalculates its distance tables, every time it receives information of a line repair, line failure, or more recent values of distance tables, from any of its neighbors. Distance tables, whose values are changed during a new calculation are exported to all the neighbors of the process.

The main disadvantage of the algorithm is that a process does not know when the algorithm has terminated and its distance tables are correct. As a final remark we should notice that, at the beginning, it is assumed that all communication lines have failed. When the system is started, it must generate the appropriate notifications for the working lines.

In this algorithm, there are messages of three different colors: d-messages, that contain new values for distance tables; f-messages, that inform a computer of a line failure; and finally, r-messages, that inform of a line repair. As a result, there is an equal number of event labels, and color sensitive message counters for each channel. Following the symbolism we used in the previous section, the pomset generators that produce all possible process behaviors, in an N-process system, are:

$$\begin{aligned}
 B_1: r_i^d &\rightarrow (s_1^d \parallel s_2^d \parallel \dots \parallel s_k^d) \\
 B_2: r_i^f &\rightarrow (s_1^d \parallel s_2^d \parallel \dots \parallel s_k^d) \\
 B_3: r_i^r &\rightarrow s_i^d \rightarrow (s_1^d \parallel s_2^d \parallel \dots \parallel s_k^d)
 \end{aligned}$$

where k is the number of a process' neighbors, at any time. This implies that when the process is informed of the failure of a communication line to another node, it is not considering this node as a neighbor anymore, until it is notified of the repair of that line. This is the case in B_2 . The symbol " \dots " denotes that the marked sequence of events might be either executed or not, depending on whether the received distance tables resulted in the

computation of new values, that the process should export. The complete specifications of any process can be defined as the concurrent interleaving of the above three possible sequences.

Symbolically:

$$B: \pi \delta [(s_1^r \parallel \dots \parallel s_k^r) ; (B_1^+ \parallel B_2^+ \parallel B_3^+)]$$

where the process sends repair messages to all its neighbors, at the initialization stage.

At this point, we should make a general observation. We feel that the presented algorithm is a typical example of a broad category of similar algorithms, where a specific sequence of events is triggered in a process, upon the receipt of a particular message. The above given process specifications can accommodate different implementations. For example, we can impose the restriction that a process fully responds to a message, before receiving any new messages. This would result in clearly demarcated repetitions of the three possible sequences of events. Such a restriction corresponds to a communication protocol based on channel polling, performed only at specific points in the process' code (i.e. after it has fully processed a received message). On the other hand, process communication can be realized with interrupt-driven receive events. Then a process can receive a message as soon as it arrives, regardless of when the message will be consumed, message consumption now being an internal process event. This "asynchronous" message reception

leads to a process behavior, consisting of interleaved event sequences. Both behaviors are accounted for in the given specifications.

The possible behaviors of a process for a three-process system are:

$$B_1: r_i^d \rightarrow (s_i^d \parallel s_j^d)$$

$$B_2: r_i^f \rightarrow s_j^d$$

$$B_3: r_i^r \rightarrow s_i^d \rightarrow s_j^d$$

where $i, j \in \{1, 2, 3\}$ and $i \neq j$. The process specification is:

$$B: \pi \Delta [(s_i^r \parallel s_j^r) ; (B_1^+ \parallel B_2^+ \parallel B_3^+)]$$

If we want our specification to contain the spontaneous notifications of line failures and repairs, we should include:

$$B_4: s_i^f \text{ and } B_5: s_i^r$$

in B.

Let us imagine that process P_2 contains a bug in its implementation: when informed of a line repair, it does not send its distance tables to the recently connected neighbor. Since the termination criteria of the algorithm are not known, we assume that the programmer constructs a test case, where the algorithm terminates after one exchange of d-messages. This can

be achieved by choosing suitable input values for the distance tables. Hence, each process is expected to execute the following:

$$(s_i^r \parallel s_j^r) ; [(r_i^r \parallel r_j^r) ; (s_i^d \parallel s_j^d)]$$

The programmer, however, notices that the algorithm does not terminate after the initial exchange of d-messages; rather additional exchanges take place. After verifying once more the program input, to make sure that the algorithm should have terminated, he suspects after checking the specifications, that one of the processes did not respond with the proper number of d-messages. The reason for this could be, either that a process did not receive the first two r-messages, or that it did not respond properly. Therefore, the programmer must proceed cautiously, in order to isolate the bug, by ruling out one of the two possible hypotheses.

First, he sets a breakpoint to make sure that each process did receive the initial two r-messages:

$$CC^r_{I/ALL} = 1$$

This cut, indicates that each process should be suspended after the receipt of one r-message in every input channel. Since, each process receives an r-message all processes will be stopped. The programmer can now discard the first hypothesis, that a process

did not receive an r-message. Consequently, he must verify that each process sends out the correct number of d-messages, that is two, in every output channel. To check this, he sets another breakpoint, with the following expression for the channel counters of each process:

$$CC^d_{0/ALL} = 2$$

That is, every process should be stopped after it has sent the two anticipated d-messages in every output channel. Processes P_1 and P_3 will eventually reach this cut. Process P_2 never reaches the cut, because it sends only one d-message for each r-message it received initially. After enough time P_2 will also be stopped. The ST diagram that can be constructed, Fig 6.2.1, shows a liveness violation on behalf of P_2 .

Again the important steps in the error location process were the carefully provided specifications, and the exact use of the channel counters.

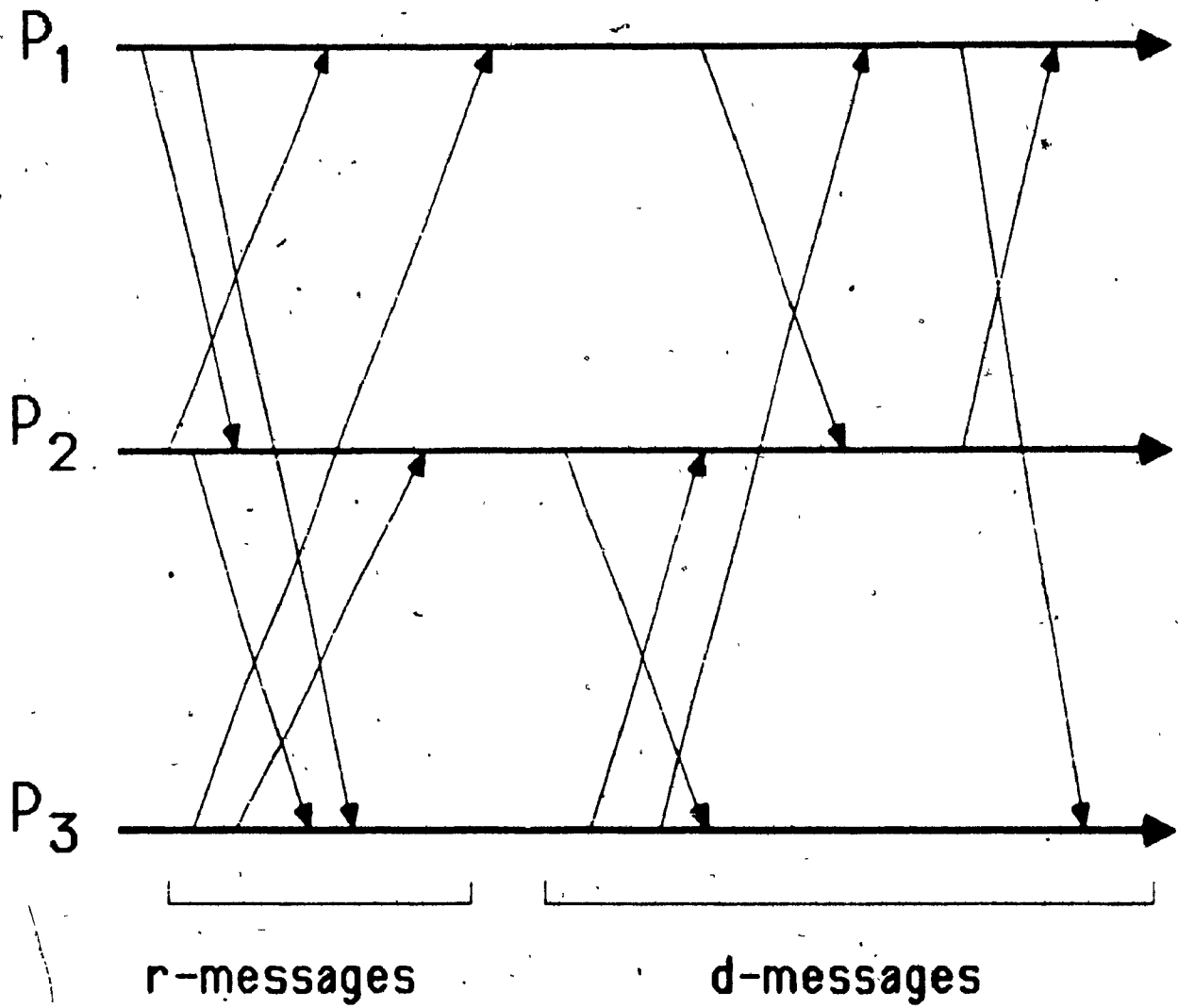


Fig. 6.2.1: Reconstructed ST diagram for the message routing algorithm

CHAPTER 7

CONCLUSIONS

A lot of attention is lately being paid to distributed processing. The development of distributed architectures, and languages, has accentuated the need for an integrated set of tools, to support the development of reliable software in such environments. Development systems, and their impact on the process of software development in distributed programming environments, have become an area of intense research. This thesis describes work done on two development tools, for a distributed system viewed as a collection of message-based, communicating processes. The described systems were a Rollback and Recovery Kernel and a distributed debugger.

In chapter 2, we discussed the problem of rollback in distributed systems. The stimulus for that discussion was our work on the design and implementation of the RRK. The resulting RRK can be easily linked up to application programs, and still perform quite efficiently. Throughout the implementation, apart from attesting once more the lack of experience in distributed programming, we realized that a more formal approach should be

followed in designing and implementing distributed software systems.

Our work on the implementation of the RRRK, served as the impetus for studying the problem of distributed debugging. We ascertained that existing debugging methods and tools are inadequate and inefficient for distributed programming. We disagree with some researchers, who have expressed the opinion that distributed debugging is an extension of its sequential namesake, and hence, only an implementation problem. We feel that traditional debugging methods fall short in DCSs, due to new problems related to true concurrency, increased complexity and absence of total control.

The new element in distributed programs is the concept of synchronization. Therefore, we shifted our attention from computational errors to synchronization errors, even though we do not underestimate the importance of traditional debugging tools, as an aid in debugging the sequential part of distributed programs.

We presented (chapter 3) two models for distributed computations: the ST model and the Pomset model. The particulars of each model determine their use: as a pictorial tool of "what actually happened" for the ST diagram; as a specification tool of "what may happen" for the Pomset model. We advocate the use of the Pomset model, as an appropriate medium for expressing process synchronization requirements. We feel that its basic notions of behavior as partially ordered sequence of events, and process as

a set of alternative behaviors, can prove a feasible and viable approach in the specification of distributed systems.

Following the discussion of the two models, we gave an overview of the difficulties involved in distributed debugging, and we described the method we adopted to overcome some of those problems. Our approach is based on a comparison between "expected behavior" (i.e. design) and "actual behavior" (i.e. implementation). The main argument for this premise, is that synchronization errors manifest as violations in pomset specifications. Synchronization specifications provide a common ground for this comparison. Their use allows for a "clear cut" of what information is needed in debugging, and what is not. In other words, the fundamental elements in our debugging method are synchronization requirements specifications, and their comparison against actual behavior. We also considered the probe effect, which is apt to be a side effect of the execution of concurrent programs through a debugger. We attempted a characterization of probe effect, without suggesting any solutions.

The end product of our discussion on distributed debugging methods, was the high level specification of a distributed debugger. We described the tools that the debugger should provide, and gave examples on their use. The main tools are message timestamps and channel counters. The former are indispensable for the reconstruction of the exact sequence of event occurrence, without losing any knowledge of event dependencies. We proved that their use guarantees the

construction of a unique ST diagram. Channel counters help the programmer set distributed breakpoints, and verify the validity of execution, by confirming the execution of valid pomset prefixes. We showed that pomset prefixes can be expressed in terms of a set of channel counter values, and that color sensitive counters are useful for the description of particular pomset cuts. Judicious use of these tools, along with the programmer's knowledge of the symptoms of the error, should help him isolate, and eventually locate, synchronization bugs.

7.1 Suggestions for Future Work.

As we have already stated, our work was carried out in the general framework of the development of a distributed programming environment. A distributed debugger is an essential development tool of such an environment. Our work was the first step towards that direction. We raised some issues and proposed solutions. The main venue for the continuation of the work described in this thesis, is the refinement of the debugger specifications and its implementation.

More precisely, we feel that a lot of attention should be given to the integration of the debugger with the other elements of the environment: OS, communication subsystem and language constructs. Incidentally, we believe that the development of sophisticated languages is of vital importance in the area of

distributed programming, and it is closely related to the development of debugging tools.

An important issue in the operation of distributed debuggers is their transparency and the phenomenon of Probe Effect (PE). We feel that deeper analysis of the PE is required. Its effects on the debugger's transparency should be analyzed, as it may be desirable that measures be taken to ensure the debugger's invisibility.

It is questionable whether the debugger should possess the capability of rollback, since the sequence of event occurrence can always be reconstructed. It is not clear to us what the additional advantages of such a feature might be. In the case of an affirmative answer, however, the careful integration of the debugger and the RRK software should be considered.

The incorporation of the debugger with traditional sequential debuggers is another problem that must be addressed. The debugging facility must offer most of the features of a standard debugger, for sequential process debugging. A related issue, is the processing of the large volume of information, that will invariably be produced by the debugger. Questions of storing results, manipulating data, and discarding obsolete information, should be answered before the debugger implementation becomes feasible. Presentability to the user is a practical aspect of the same problem.

Considering systems with dynamic architecture would provide another possible research issue. In our work we have dealt with

systems of static topology, that is known at the beginning of the execution. Dynamic process generation and termination will bring about debugger reconfiguration problems.

We believe, that further work in distributed debuggers should proceed along with related work in other areas of the distributed processing domain: architectures, operating systems, languages. Above all, additional experience with distributed programming is required. After all, more experience in this field can only motivate further work for debuggers and other development tools.

REFERENCES

1. M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery, F.B. Schneider: "Distributed Systems - Methods and Tools for Specification - An Advanced Course", Springer-Verlag LNCS 190, 1985.
2. T. Anderson, P.A. Lee: "Fault Tolerance - Principles and Practice", Prentice-Hall International, 1981.
3. F. Baiardi, N. De Francesco, G. Vaglini: "Development of a Debugger for a Concurrent Language", IEEE TSE, Vol. SE-12, No 4, April 1986.
4. P.C. Bates, J.C. Wielden: "EDL: A Basis for Distributed System Debugging Tools", Proceedings of the Fifteenth Hawaii International Conference on System Sciences, 1982, pp. 86-93.
5. P.C. Bates, J.C. Wielden, V.R. Lesser: "A Debugging Tool for Distributed Systems" 311 - 315.
6. B. Bruegge, P. Hibbard: "Generalized Path Expressions: A High-Level Debugging Mechanism", Proceedings of the ACM

SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugging, March 1983, pp. 34 - 44.

7. R. Cooper: "Pilgrim: A Debugger for Distributed Systems", Seventh International Conference on Distributed Computing Systems, September 1987.
8. R.H. Dunn: "Software Defect Removal", McGraw-Hill, 1984.
9. P.D. Ezhilchelvan, S.K. Shrivastava: "A Characterization of Faults in Systems", Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp. 215 - 222.
10. J.Gait: "A Probe Effect in Concurrent Programs", Software Practice & Experience, Vol. 16(3), March 1986.
11. M.E. Garcia, W.J. Berman: "An Approach to Concurrent Debugging", Fifth International Conference on Distributed Computing Systems, May 1985, pp. 507 - 514.
12. H. Garcia-Molina, F. Germano Jr., W. Kohler: "Debugging a Distributed Computing System", IEEE TSE, Vol. SE-10, No. 2, March 1984.
13. M. Gardner: "The Fantastic Combinations of John Conway's New

Solitaire Game Life", Scientific American, October 1970, pp. 120 - 123.

14. P.K. Harter Jr., D.M. Heimbigner, R. King: "IDD: An Interactive Distributed Debugger", Fifth International Conference on Distributed Computing Systems, May 1985, pp. 507 - 514.
15. S.H. Jones, R.H. Barkan, L.D. Wittie: "Bugnet: A Real Time Distributed System", Sixth Symposium on Reliability in Distributed Software and Database Systems, March 1987, pp. 56 - 65.
16. H.F. Li, T. Radhakrishnan and K. Venkatesh: "Global State Detection in Non-FIFO Networks", Proceedings of the Seventh Conference on Distributed Computing Systems 1987.
17. J.Martin, C. McClure: "Structured Techniques for Computing", Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1985.
18. C.J. Passier: "Experimental Evaluation of Distributed Rollback and Recovery Algorithms", Master's Thesis, Concordia University, Montreal, 1988.
19. V.R. Pratt: "On the Composition of Processes", Proceedings of the Ninth Annual ACM Symposium on Principles of

Programming Languages, January 1982, pp. 213 - 223.

20. V.R. Pratt: "The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial", Proceedings CMU/SERC Workshop on Analysis of Concurrency, Springer-Verlag LNCS 197, 1984.
21. V.R. Pratt: "Modelling Concurrency with Partial Orders", International Journal of Parallel Programming, Vol. 15, No. 1.
22. R.S. Pressman: "Software Engineering: A Practitioner's Approach", McGraw-Hill, 1982.
23. E.D. Smith: "Debugging Techniques for Communicating, Loosely Coupled Processes", Ph.D. Thesis, University of Rochester, Rochester NY, 1981.
24. K. Venkatesh: "Global States of Distributed Systems: Classification and Applications", Ph.D. Thesis, Concordia University, Montreal, 1988.