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.

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

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.

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

ISBN 0-315-56041-X

Methodology and Tools for Distributed Debugging

Bao Minh Dang

A Thesis

in

The Department

of

Computer Science

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

December 1989

ABSTRACT

Methodology and Tools for Distributed Debugging

Bao Minh Dang

Errors in distributed programs can be classified into computational bugs and synchronization errors. The former are entirely local to a process whereas the latter occur at the interfaces between distributed processes. The pomset model is used in this thesis to describe the synchronization behaviour of distributed programs. Based on this model, a synchronization specification of a distributed program can be defined by the program designer. A synchronization error can be viewed as the failure of an observed distributed computation to match with the given specification. Debugging distributed programs can be done in two stages. First, any synchronization errors are located by having the debugger compare its observation of the application program against the given specification. Then other debugging facilities are used to examine internal states of component processes in order to locate the computation bug. An integrated debugging system should allow the two stages to be used interchangeably and interactively. In this thesis, debugging techniques such as breakpointing and tracing are studied in the context of distributed computing environments.

Based on this approach, a distributed debugger has been designed and implemented on a network of SUN workstations. The debugger supports automatic checking for synchronization errors, breakpointing, checkpointing and use of traces. Debugging can be done interactively, and an integrated sequential debugger provides access to internal variables and source codes of processes.

To my parents

and

to Trang

ACKNOWLEDGEMENT

First and foremost, I would like to thank my supervisors, Drs. T. Radhakrishnan and H.F. Li. Together, they have guided me through every step of the work leading to this thesis. This thesis would not have been completed without their invaluable assistance. They were always available to me whenever I needed them, offering me guidance, advice and encouragement. They have also been very patient with me even when I failed their expectations.

I am thankful for the financial support of the Fonds FCAR, of Concordia University and of my supervisors.

Chris Lea has been involved with the design and the implementation of the distributed debugger. He is bright and amiable, and I have enjoyed working with him. I believe that we have forged a lasting friendship. Dimitri Livas has helped me in verifying the correctness of the theorems and algorithms presented in this thesis, and has tutored me on the formalism which I have used for the project. Honna Segel and Chris Lea have done the daunting task of proofreading this thesis. I want to thank them all, along with other friends: Thomas Wieland, Jason Cheng and others, with whom I have shared the interest of our studies, the happiness and frustration of our progress, and the expectations of our future.

Last but not least, I would like to express my deepest gratitude to my parents who have encouraged and supported me at every stage of my education. I would also like to thank all my sisters, brothers and Trang for their love, patience and support.

Contents

List of Figures and Tables

List of Abbreviations

B/D	Blocked and Deadlock
CC	Channel Counter
CD	Central Debugger
CGS	Consistent Global State
DB	Database
DCS	Distributed Computing System
FIFO	First In First Out
GC	Global Clock
GT	Global Time
IPC	Inter Process Communication
LC	Logical Clock
LD	Local Debugger
Pomset	Partially Ordered Multiset
SBS	Synchronization Behaviour Specification
SDL	Synchronization Definition Language
ST	Space Time
TS	Timestamp

Chapter 1

Introduction

Distributed computing might have been a dream a decade ago, but not any more. Many factors have contributed to this trend. The demand for more computing power is ever growing and at the same time workstation type computing is becoming widespread within an organization. Local area networks connecting these machines are invariably present. Thus connectivity is not an issue. Not all of the workstations are busy at all times. As a result there is spare computing power available in a decentralized way.

The development of distributed computing has been slow, however. The distributed computing environment requires new solutions for the same problems that have been long solved in a centralized system. Determination of system state, fault-tolerance, scheduling, etc., have taken new meanings in distributed computing systems (DCS). Although a lot of research has been done in these areas and the nature of DCS has become better understood, software development for DCS has been slow. The lack of a programming language for distributed program development, and non-availability of a proven distributed operating system and proper software development tools are the main reasons for this slow growth. One of the tools that is probably most needed by any programmer in DCS is the distributed debugger.

In this thesis we develop an approach to debugging that is applicable to DCS. Based on this methodology, a design for a distributed debugger is presented.

1.1 Debugging and Testing

Before going into further discussion on debugging, we want to make the distinction between debugging and testing. Some have considered these operations to be the same [4], and it is true that many programmers carry on these two stages in program development at the same time.

According to Darlington, the main objective of testing is to derive a set of test data that "exercises" all aspects of a program [10]. As the result, faulty components of the program can be recognized. Therefore, testing is also seen as a process used to show the existence of bugs [11].

On the other hand, debugging is carried out only after a program has been detected to have bugs, either from testing or merely through regular use [32]. Therefore, in debugging, the programmer possesses the knowledge of the characteristics of the bug in terms of the *behaviour* which makes the program violate its specification. The objective of debugging is to locate the bug and carry out the necessary correction.

Usually, after a program is debugged, it requires further testing to make sure that no bug remains. If more bugs are found, more debugging is needed. The iteration is repeated many times until no more bugs are found, and the program is released for its intended application. Nevertheless, the two phases must be kept distinct: testing determines the existence of bugs while debugging locates the bugs and removes them.

1.2 Sequential Debugging Techniques.

Debugging techniques for sequential programs have been developed constantly since the first computer program was written. They range from adding extra statements to a program that print messages to the console screen, to sophisticated debuggers that allow the user to modify the contents of the processor's registers for interactive debugging. Basically, the process of debugging sequential programs can be viewed as one in which the user tries to locate the position of a bug in his program where the program's *state* starts failing to match the specification, that is the programmer's expectation.

Formally, each program code corresponds to a sequence of atomic actions, where each action is a statement or instruction. There is no clear definition of an atomic action, it depends on the debugger implementation. A program has bugs if one or more segments of the above sequence are incorrectly defined. For instance, the segment of computation has a fault because it manipulates a data structure incorrectly; or it just simply has an extra instruction that had been carelessly added by the programmer.

Debugging sequential programs can be done in two ways. The program's behaviour is verified by examining the sequence of *events* that occur when the program is executed. This can tell where the program, considered as a sequence of actions, starts deviating from expectation. The program's state can also be examined, if the bug is known to corrupt data structures or produce invalid data. Although an undesirable state cannot tell what causes the bug, it gives a lead to the programmer which assists in locating the bug.

Thus existing debugging techniques are based on examination of the program's behaviour and the program's state. The tools that provide the user the necessary facilities are: trace tools and controlled-execution tools.

Traces. A program trace is the record of the history of its execution. The simplest form of trace is output statements that indicate the execution of a program's segment. A detailed trace can reveal the absence of a necessary event; the presence of an extra, erroneous event; or incorrect ordering of the events. Debuggers can also provide a trace in which a partial picture of the program's state is recorded whenever a specified condition occurs. Thus the user can examine whether the effect of the events on the program's state is as expected.

Controlled-Execution. Since the whole picture of the program's state at any point during its execution can be enormous in volume, a controlled-execution debugging technique may be used in which the program execution is suspended during its execution at some point so as to allow the user to examine the state. Because all events in a sequential program are totally-ordered and the state is preserved, the duration of the suspension will not affect the future state of the program.

Usually, the user specifies a suspension point, called breakpoint, in correspondance with an event. The program will be suspended when that event occurs. Beside allowing the user to examine the program state, breakpoints may also give the user a general view of the program behaviour, for example, not reaching a breakpoint indicates that the corresponding program segment is not executed.

Another form of controlled-execution is stepping, in which the program is suspended after every step. This will give the user a step-by-step progress of the

computation. However, because the program progresses very slowly, stepping is usually limited to a small segment in the program.

The program state can also be monitored using watchpoints. Each watchpoint is associated with an invariant in the program. The program's execution can be suspended when such an invariant is violated. This technique is more useful in program testing; in debugging, it helps to narrow down the area where a bug can be found.

1.3 Related Work in Distributed Debugging

Garcia-Molina identifies the key difficulties in distributed debugging in [16]: There are multiple threads of control in the distributed program execution and there is random delay in inter-process communication. They propose that testing and debugging be done together in a bottom-up manner, where component processes are assembled to be run as one distributed program. The assembled program is then run so that its traces are collected for examination afterward. The main disadvantage of this approach is that it relies on the user's ability to identify the trouble spots in the program using the traces of the distributed program. A more effective use of the program trace is to use it in replay, as suggested by Curtis [9] and Miller [28]. During the replay, the program is not running. Instead, data from the trace files is used to recreate inter-process communication activity of the program. The replay is done at a slow pace, allowing the user the time to observe the behaviour of the program. With the availability of workstations with high quality graphic support, an

animation of this activity can be created, presenting a graphical image of the program's behaviour [6, 7, 20].

A debugging environment is suggested by Schiffenbauer [30] and Joyce [22] where the user has control of the inter-process communication. A central controller decides on the occurrence of every communication event in the program. Through this controller, the user can interactively determine the order of occurrences of events. Hence different program behaviours can be composed and examined. These tools, though, are more suggestive of a program testing facility than of a debugging facility.

The *Behaviourial Abstraction* approach is described by Bates in [5], where the user is allowed to define abstractions of events. When the program is debugged, a *Recognizer* is used to match the program's actual behaviour with the defined abstractions, providing the means for automatic detection of "safety violations" [23] in the program. One limitation with this approach is that the complexity of recognizing a mismatch can be quite large. The second limitation is that the event abstraction of Bates cannot efficiently describe partial order relations between different event abstractions.

Bates and Baiardi [4] are among the first to suggest debugging of distributed programs based on program behaviour. Baiardi's top-down approach is opposed to that of Garcia-Molina [16]. Testing and debugging are combined such that the behaviour of the whole program is first checked to identify erroneous components. The behavioural specifications of these components are given in greater detail, and the check is then repeated to identify erroneous sub-components. This process continues until the actual component causing the error is identified. A *Behaviour Specification* of the components is provided to the debugger for the purpose of checking. The

problem with this method is that the component that displays faulty behaviour might not be the one that causes it; for instance, it can receive invalid data generated by another component. Approaches similar to Baiardi's are found in [13, 18, 21].

Reproducibility is usually difficult in distributed debugging due to nondeterminism. Since the nondeterministic choice in one execution is not always the same as in another, a program may not follow the same course in a subsequent execution. A method of *Instant Replay* is proposed by Leblanc in [26] which makes use of the trace information from processes to guarantee reproduction of an execution. With this method, the program can be run in slow-motion, allowing the user to examine it without affecting the path of execution. *Instant Replay* can be very helpful if it is complemented with automatic checking of the observation against the specified behaviour of the processes. Reproducibility is also discussed in [13] where the patterns of interleaving of concurrent events and the non-deterministic choices are recorded for each process and used in later execution, forcing the program to produce the same synchronization behaviour in every execution.

Bei [6], Cameron [7] and Goldszmidt [17] suggest the use of an assertion checker in the debugger. In this technique, predicates are inserted in the program, and are validated by the debugger during execution. Thus the state of the program is deemed to be verified. The detection of bugs, using this method, relies on the user's choice of predicates. Many iterations of execution may also be needed before the location of bugs is determined.

Cheng presents a different use of program traces in [8]. Instead of letting the user of the debugger examine the program history by viewing the trace, they use the trace as input to a knowledge-based verification unit. This unit has been previously

supplied with the specification of the program. The trace is matched against this specification to detect errors. Since the specification is given in terms of the inter-process communication (IPC) activities of an individual process, the verification can be done for each process individually and concurrently.

What we find lacking in these works is a proper modeling of the distributed programs. The models that have been used do not fully describe the nature of a DCS and its properties. In addition, these debugging methods only go as far as identifying the synchronization errors. They do not provide the means to locate the actual programming bug which is the source of that error. Also lacking is a well-defined support for breakpoints. The works described above do not provide breakpoints in a distributed environment [4, 5]. They simply allow arbitrary breakpoints at which the program's state may be of little use to the user.

The approaches to debugging discussed above are also limited to functional aspects of the program, or "blackbox analysis". The program functionality, as reflected through by its behaviour, is checked against its specification in order to locate the synchronization error. However, we believe that another stage of debugging, the "whitebox analysis", is necessary. During this stage, the internal structure of the program is examined, with proper debugging tools, in order to locate the programming error which is the cause of the erroneous behaviour. In other words, an integrated hierarchical debugger is needed in order to provide an efficient debugging environment for a DCS.

1.4 Our approach

Our approach is based on the observation that a program is considered to have bugs when it exhibits abnormal behaviour that can be observed from the outside world. The abnormal behaviour is recognized when the program generates erroneous output or follows an undesirable course of action. If the program is sequential, the debugger is applied directly to locate the bug, which is then removed by the programmer. In a distributed program with many processes, a similar approach can be followed by posting an observer for every process. If the observer knows the expected behaviour of that process, it can detect the faulty behaviour of a process. In practice, the observer is a "checking program" which takes into account the specification of that process.

In principle, we follow Baiardi for debugging by checking the program's behaviour against the specification. However, we limit our observation to the program's behaviour at the process's interface, or the interactions of a process with other processes. This is similar to the semantics in [8]. Moreover, finding the process which has faulty behaviour does not mean that the bug has been found. Additional debugging tools are needed for locating the bug. Hence they are provided as an integral part of our debugging system.

Chapter 2

Distributed Computing and Debugging

Models are used in system analysis and problem solving for the sake of abstraction. A model is chosen in such a way that the components and properties of the system which are relevant to the analysis are emphasized and insignificant details are removed [3]. Thus the model becomes a simplified and generalizing view of the problem. We use the pomset (partially ordered multiset) theory for modeling distributed programs. The pomset model is chosen because of its ability to describe truly concurrent behaviours [29]. Before presenting the formalism of the pomset model, we will state the assumptions of the system on which the model will be applied. Some terminologies are also clarified.

Assumption: The following is assumed for a distributed program:

1. A program consists of a number of communicating processes, where communication is done via message-passing on peer-to-peer, unidirectional channels.
2. Primitives for message-passing are send and receive, where send is non-blocking and receive is blocked on an empty channel.
3. Channels have infinite buffers.
4. Messages are delivered in FIFO manner, and subjected to a variable, positive and finite delay.
5. Processing within a process is strictly sequential. □

The second assumption implies that a receive event is always causally dependent on a send event. The fourth assumption implies that the communication is reliable.

The Pomset model

Definition 2.1 A pomset p is a quadruple $[V, \Sigma, \Gamma, \mu]$ where:

V = set of process events

Σ = set of process actions

$\Gamma \subseteq V \times V$ = partial order, which is an irreflexive and transitive relation, defined on V which expresses:

1. the expected temporal precedence among events in V, where p describes a program's specified behaviour;

or

2. the observed temporal precedence among events in V, where p describes the program's observed behaviour at run-time; in this case, p is also called the *computation* of the program.

If $(e_1, e_2) \in \Gamma$ then the representation $e_1 \rightarrow e_2$ means e_1 *happens before* e_2.

μ = $V \rightarrow \Sigma$ is a many to one mapping. □

Each event $v \in V$ can be seen as an instance or an occurrence of an action a defined in Σ, where $\mu(v) = a$, and multiple occurrences of the same action are possible. Depending on the level of abstraction, such an action can represent a complex operation, or just a single machine instruction.

If an event e_1 is a *send* event, and e_2 is the *receive* event which receives the message sent by e_1, they are called *matching events*, and denoted:

$$e_1 = s(e_2), \quad \text{and} \quad e_2 = r(e_1).$$

Send and *receive* events are called *communication events.*

The algebra of pomsets is discussed in detail in [29]; we present only those parts that are relevant to our study:

Conjunction. Given pomsets $p_1 = [V_1, \Sigma_1, \Gamma_1, \mu_1]$ and $p_2 = [V_2, \Sigma_2, \Gamma_2, \mu_2]$, $p_1 \| p_2$ (conjunction pomsets) is a pomset $p = [V, \Sigma, \Gamma, \mu]$ where $V = V_1 \cup V_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Gamma = \Gamma_1 \cup \Gamma_2$ and $\mu = \mu_1 \cup \mu_2$. There is no causal dependency between events in p_1 and those in p_2, because there is no relation between $v_1 \in V_1$ and $v_2 \in V_2$. This construction is called *concurrency* in [29]. We use the term *conjunction* to remove the possible implication that p_1 and p_2 are required to occur simultaneously. Furthermore, our definition does not require V_1 and V_2 be disjoint as in [29].

Concatenation. The concatenation of p_2 to p_1 described as $p_1;p_2$ (or p_1p_2) is a pomset $p = [V, \Sigma, \Gamma, \mu]$ where $V = V_1 \cup V_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Gamma = \Gamma_1 \cup \Gamma_2 \cup (V_1 \times V_2)$, $\mu = \mu_1 \cup \mu_2$. $(V_1 \times V_2)$ is required to ensure that every pair (v_1, v_2), $v_1 \in V_1$ and $v_2 \in V_2$, is in Γ, i.e. $v_1 \rightarrow v_2$.

Prefix. A pomset p' is a prefix of p ($p' \leq_\pi p$), if p' is obtained from p by deleting a subset of events from p, provided that if e_1 is deleted and $e_1 \rightarrow e_2$ then e_2 is also deleted.

Augmentation. A pomset p' is an augment of p ($p \leq_\alpha p'$) if they are identical pomsets in all respects except that Γ' is a superset of Γ.

Projection. A pomset $p' = [V', \Sigma', \Gamma', \mu']$ is a projection of pomset $p = [V, \Sigma, \Gamma, \mu]$ onto a set A of actions, denoted as $p' = proj(p, A)$, if for every event $v \in V$ *such that* $\mu(v) \in A$, v is removed in V' along with all edges in Γ that associate with v. Formally:

$$p' = proj(p, A) \quad \Leftrightarrow \quad \begin{array}{l} (\ \Sigma' = \Sigma \cap A, \\ \quad V' = \{ v \mid v \in V \wedge \mu(v) \in \Sigma' \ \}, \\ \quad \Gamma' = (V' \times V') \cap \Gamma, \\ \quad \mu' = (V' \times \Sigma') \cap \mu \ ) \end{array}$$

□

Definition 2.2 A *distributed computation* is a pomset $p = [V, \Sigma, \Gamma, \mu]$ which takes the system from the initial state s_0 to the current state s_p, written as:

$$s_0 \, p \, s_p$$

□

Definition 2.3 A state s_r recorded during a computation p is a *consistent global state (CGS)* iff:

$$\exists r \leq_{\pi} p \, [\, s_0 \, r \, s_r \,]$$

□

2.1 Debugging Distributed Programs

Conventional debugging techniques cannot be applied directly to distributed programs because of the reasons discussed in chapter 1. Locating bugs in distributed programs is also difficult because of the 2-dimensional nature (space and time) of a DCS. In sequential programs, events are totally ordered, and from any mismatches in the observed behaviour, one can always trace back to the cause of that misbehaviour, which is located a certain number of events back. The same cannot be said for a

distributed program where the cause of the misbehaviour observed at some process could have originated from some faulty actions in another. Thus the effect of the error, or the bug, propagates in both time and space away from its original point until it surfaces and is noticed by the program's user.

An error in one process can cause another process to become erroneous if the processes communicate with each other. Either the erroneous process sends invalid data to the other, or it causes communication events to occur with incorrect timing, violating necessary coordination. If the computation of one process does not have any communication events then errors in that computation will never affect other processes. Thus communication events can be seen as having the property of being able to spread an error from one process to another. We can thus divide events in a distributed program into two groups: communication events and internal events, based on the effect they might have on the propagation of an error.

Communication events are also called synchronization events. Coordination of synchronization events is central to distributed programs. As stated earlier, a distributed program consists of a number of individual processes, each of which carries out its own computation and shares information with other processes through message-passing. The synchronization specification of a program is the required behaviour of each process with respect to the synchronization events. A pomset of these events, in which the necessary temporal behaviour of the events are specified, is used for this purpose.

Program failure can also be in the form of corrupted data. Synchronization events can still occur as expected but processes may be exchanging invalid data. However, the invalid data flow can be traced back using other means such as a

trace facility. Therefore, in our scope of debugging, we restrict the specification to concern only the relative ordering of synchronization events.

Every process can be viewed as a blackbox which carries out its own computation and has certain interactions with its surrounding environment. This behaviour, which is observable, should conform to that required by the program's specification. Failing this, the program is said to have bugs. The same view can be applied to each component of a distributed program. Each process can be seen as a blackbox whose sole connection to the outside world is through its IPC activities. These activities can be monitored by an observer, who is also provided with the process's synchronization requirements. Deviations of the observed behaviour from the specified behaviour can be detected. Using the pomset model, the synchronization errors, which cause erroneous oehaviour, will be formally defined.

Definition 2.4 A process synchronization specification is the set $P = \{ p_1, p_2, \ldots \}$ of pomsets describing a process's synchronization behaviour. Each $p_i \in P$ specifies a valid abstract behaviour of the process in terms of its synchronization events. □

In debugging a distributed program, the debugger maintains a partially ordered history of synchronization events that have occurred. A labeled partial order α which is isomorphic to the partial order history of events is employed to timestamp (label) the observation. Therefore, α is the equivalent of the actual behaviour of the process.

Definition 2.5 Given a specification P, an observation α is said to *agree with* $p_i \in P$ iff:

$$\exists \beta \leq_\pi p_i [\beta \leq_\alpha \alpha]$$ □

Synchronization errors can then be defined indirectly as follows:

Definition 2.6 An observed computation α is said to have synchronization errors with respect to a specification P iff:

$$\exists p_i \in P [\alpha \textit{ agrees with } p_i].$$ □

Intuitively, α *agrees with* p_i when every causal dependency among events in p_i is satisfied by the corresponding events in α. We say that α is a *valid observed behaviour* of P. Execution of a distributed program can result in different computations, each corresponding to one abstract behaviour p_i in the specification P. The resulting computation is the outcome of nondeterministic choice that is made at execution time. Some of these computations may be valid implementations of P while others may not. In debugging, we want to recreate the invalid computations in order to locate the synchronization error contained within those computations.

Detection of a synchronization error does not mean that the exact bug (or bugs) is located. The observed error can be the result of a bug that occurred in another process some time before, the effect of which might have propagated through different events and processes until it surfaced in the form of a synchronization error that is just detected. In such a case, the internal operations of processes need to be examined further to locate the real bug.

Therefore, our debugging process consists of two main stages. First, the synchronization error is located. Then the information and knowledge obtained from that is used to locate the faulty program codes which caused the violation.

2.2 Time and Clock

Time is a fundamental concept which we use to establish the order of events [24]. In debugging, time is important because the occurrence of an event "before" or "after" another can determine the validity of a computation. However, time can become an intriguing issue when the order of events is apparently different when perceived by different observers. This could happen when there is an arbitrary positive delay from the time an event occurs to the time an observer perceives that occurrence. To retain the actual ordering of events, a clock whose value is monotonically increasing is used, and each event is timestamped with the value the clock has when that event occurs.

In computer systems where a centralized control exists, there is a clock common to all subsystems. All events are timestamped with this clock value, resulting in a total ordering. On the other hand, a DCS has no common clock, and values of the local clocks by themselves are not sufficient if one wants to establish the temporal order of events at different nodes. To retain the temporal order, other schemes for timestamping have been proposed: *Logical Clock* (*LC*) by Lamport in [24] and *Global Clock* (*GC*) by Heping in [31].

In *LC* scheme, a counter is associated with each process in a distributed program. The counter's value is the logical time for that process, and is initialized

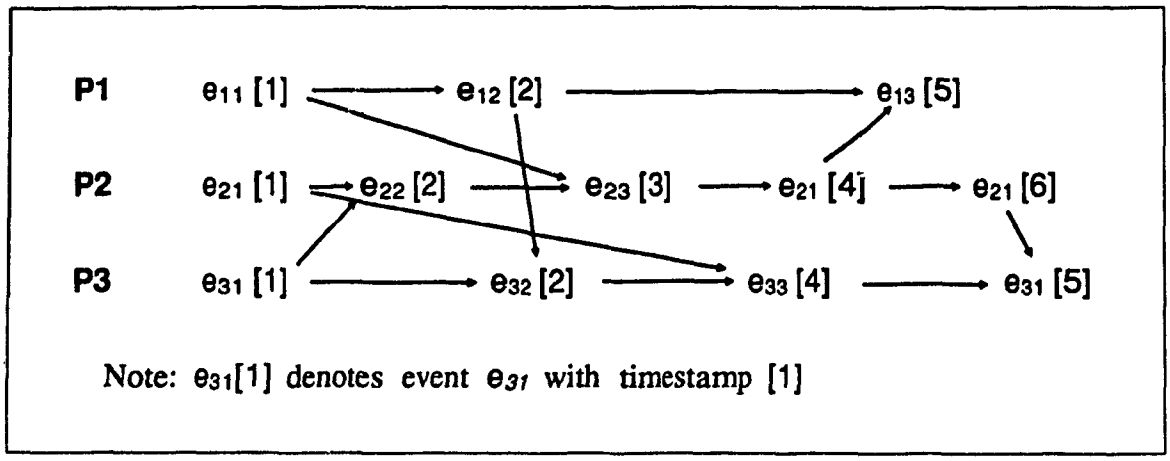


Figure 2.1 - Distributed Computation with timestamp in *LC*

at zero. When an event occurs, and if it is a *send* event or an internal event, the counter is incremented. The event is then timestamped with the counter value. For a *send* event, the message sent out is also timestamped with this value. For a *receive* event, it is handled a little differently. The counter value of the receiver is first compared with that of the timestamp of the received message. If the counter's value is less than that of the timestamp, it is replaced with the timestamp's value. Then the counter is incremented and its value is used to timestamp the *receive* event. Figure 2.1 illustrates events in a distributed computation with their timestamps in *LC*. If $LC(e_1)$ and $LC(e_2)$ are timestamps of events e_1 and e_2, respectively, then the following is always true:

$$e_1 \rightarrow e_2 \quad \Rightarrow \quad LC(e_1) < LC(e_2)$$

GC scheme also maintains counters for each process as their logical time. However, each local logical clock is a vector of counters of N elements, where N is the number of processes in the program. The algorithm for maintaining these

clocks is given in section 3.1.1. If $GC(e_1)$ and $GC(e_2)$ are timestamps of events e_1 and e_2, respectively, the following is true:

$$e_1 \rightarrow e_2 \quad \Leftrightarrow \quad GC(e_1) \ll GC(e_2)$$

where the relation « is defined as follows:

Definition 2.7 Given $GC_1 = [a_1, a_2, \ldots, a_n]$ and $GC_2 = [b_1, b_2, \ldots, b_n]$, then $GC_1 \ll GC_2$ iff

$$\forall i \in \{1 \ldots n\} [a_i \leq b_i] \wedge \exists j \in \{1 \ldots n\} [a_j < b_j].$$ □

Although *LC* is simpler to implement and has less overhead, events with these timestamps are mapped to a total order. Therefore truly concurrent events which have no temporal relationship between them cannot be identified. On the other hand, *GC* guarantees that the real partial order of a computation is preserved through the timestamp based on *GC*. Formally, a record of a computation in which events are timestamped with *GC* forms a labeled partial order which is isomorphic to the computation. Since such a record is needed for checking the synchronization error, *GC* is used in our debugging system.

2.3 Liveness Violation

The detection of an error in a program, based on the checking of temporal relationships between events, can only reveal one class of errors, which is that of synchronization errors. This approach cannot detect the non-occurrences of an event that is supposed to occur. This type of error will be called a "liveness violation". For instance, a deadlock in a system or a premature termination of a process constitutes

a liveness violation. While an observer can always tell the presence of safety violations in a process when the observed computation does not follow the specification, he is not able to say that a liveness violation exists just because an expected event has not been observed. Supporting facilities can be built to aid the user in detecting this kind of violation. Relevant parameters related to the characteristics of the application program and of the computer system which is used to run that program are supplied. Based on these parameters, potential liveness violations can be detected and the user could be informed of the possible errors. For example, a timeout period can be specified as the maximum time a process can be blocked on receiving a message. If the process is blocked longer than this period, the user is informed of the situation. In turn, the user can inspect the status of the process at the other end of the channel to know the cause of the timeout. In general, the user has the ultimate decision on whether the debugged program has violated the liveness requirements. However, in a future version of the distributed debugger, additional information can be added to the program's specification so that the debugger can detect liveness violations. Such a debugging facility is needed in particular for applications in real-time systems.

2.4 Probe effects

Probe-effect in debugging refers to the phenomenon in which the debugger has undesired effects on the debugged programs, altering the program's course of execution. Two typical scenarios may arise as the result of the probe-effect. In the first scenario, a program known to be erroneous starts to function as desired in the

debugging environment; the bug in the program seems to disappear under the influence of the debugger. In the second, the opposite occurs: program that works correctly under normal condition begins to exhibit erroneous behaviour in the debugging environment [14, 15].

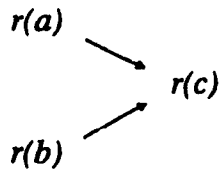
Probe-effect in the second scenario is easy to deal with, and can also be beneficial. It helps to reveal program errors that the programmer has yet to notice. However, the disappearance of bugs caused by the probe-effect, as in the first scenario, can introduce difficulty to debugging. With a program that executes correctly, one cannot get any clues that point to its defects.

One of the causes of the probe-effect is the influence of the debugger on the patterns of interleaving of concurrent events. Operations that are defined as concurrent can be implemented on a sequential system by interleaving their events. If some of the patterns of interleaving cause synchronization errors, and if the debugger prevents these patterns from taking place, then the probe-effect can be observed. The following example illustrates this situation.

Example 2.1 Consider a process P with two input ports: a and b and an output port c. The specification of P is:

$$r(a) \rightarrow r(b) \rightarrow s(c)$$

where $r(a)$ means receiving a message through port a and $s(c)$ means sending a message through port c. Suppose the implementation of P has $r(a)$ and $r(b)$ as concurrent events:



Thus during execution, two possible interleaving of *r(a)* and *r(b)* can be observed:

1. $r(a) \rightarrow r(b) \rightarrow s(c)$ (valid)

and

2. $r(b) \rightarrow r(a) \rightarrow s(c)$ (invalid)

If, under normal condition, the second computation is always chosen between the two as the program is executed, a synchronization error is observed. The probe-effect arises when the program is executed under a debugging environment in which, due to the debugger's influence, the first computation is always observed.

□

In a rather more subtle situation, the debugger may affect the outcome of nondeterministic choice made by the program. Recall that a program may consist of different possible computations, one of which will be realized during an execution. The selection process depends on both the implementation of the program and the environment in which the program is executed. If the debugger causes the correct computation to be selected, which would not be once the debugger is removed, then probe-effect is again observed. This is shown in the next example:

Example 2.2 Let $P = \{ p_1, p_2, \ldots \}$ be the specification of a program. $p_i, p_j \in P$ are two of the possible computations that the program can have. Assume

that the implementation of p_i is correct, and that of p_j has errors. If, under normal conditions, nondeterministic choice is always made in favor of p_j, we will observe errors on each execution of the program. However, if when the program is debugged, the debugger causes the implementation of p_i to be selected for the execution, the program will seem to have no error. □

It should be noted that it is possible for the correct computation to be selected under the normal condition, and no error would be observed during that execution. Likewise, the invalid computation can be chosen when the program is debugged, and the error is detected. All of this depends on the influence of both the computing and the debugging systems on the outcome of the interleaving patterns and on the nondeterministic choice.

Probe-effect is present in any debugging system, and one should understand its impacts in order to appropriately compensate for its effect. We do not attempt to address this issue here. However, as shown in the experiment done by Gait in [15], delay introduced by the debugger on synchronization events does mask some existing synchronization errors in a concurrent program. As this delay increases, it can completely mask out all synchronization errors. Therefore, in designing our debugger, we have attempted to minimize this additional debugging delay as much as possible.

Chapter 3

An Approach to Debugging based on Pomset.

As stated in chapter 2, our strategy of debugging a distributed program can be divided into two stages: locating synchronization errors, and locating programming bugs which cause these errors. A synchronization error is located by comparing the actual behaviour of the program against its specification. A programming bug is located by using debugging tools to examine the internal state and operation of component processes.

A typical debugging session can be as follows. First the specification of the program being debugged is supplied to the debugger. Then the program is executed under the debugging environment. The debugger monitors the IPC activity of component processes in the program and checks whether this activity follows the specification. A synchronization error is located when a mismatch occurs.

After the synchronization error is located, the execution is suspended. The user can then use debugging tools for breakpointing, checkpointing and tracing to examine the internal states of the processes to narrow down the search for the cause of the synchronization error.

In the following sections, the design of a debugger which is based on our strategy is described. The discussion includes the fundamental points that are necessary for achieving our design goal.

3.1 Basic needs

3.1.1 Global Time

As mentioned above, one of the main things we look for during a debugging session is the correctness of the temporal orders between events, as required in the problem specification. Therefore, events are to be labeled with timestamps so that their causality is preserved. We employ the *Global Clock* algorithm of Shang [31] for timestamping, which has properties that meet our requirement. This algorithm uses N-dimensional vectors called *Global Clocks*(*GC*), where N is the number of processes.

Each component of *GC* corresponds to a process. Each process has its own *GC*. When processes exchange messages, they also exchange the values of their *GC*. Each event will be stamped with the current value of the local *GC*, which is called *Global Time* (*GT*), the logical time indicating the moment the event occurs. If $TS(e_i)$ is the timestamp in *GT* of event e_i, the GC algorithm ensures that:

$$TS(e_1) \ll TS(e_2) \quad \Leftrightarrow \quad e_1 \rightarrow e_2$$

where the relation « is defined in definition 2.7.

Therefore, by comparing the value of their *GT*, one can always establish the causality of events. Moreover, events that have no causal relationship are also identified. When the *GC* are kept as part of event records in a trace file, execution history can be fully reconstructed.

Given a distributed program with N processes, the algorithm for managing the *GC* is described as follows:

Algorithm 3.1. *Global Clock Algorithm.*

A n-element vector of counters is associated with each process. Consider $GC_k = [c_1, c_2, \ldots, c_n]$ which is associated with process P_k.

Initialize: $\forall\ i \in \{\ 1 \ldots n\ \}\ [\ c_i = 0\ ]$.

For each occurrence of an event e,

Case 1: if e is a *send* event, then

$$c_k = c_k + 1,$$

$$TS(e) = GC_k$$

The new value of GC_k is used to timestamp the *send* event e as well as the message being sent.

Case 2: if e is a *receive* event, then the timestamp TS_m of the received message is retrieved first:

$$TS_m = [t_1, t_2, \ldots, t_n],$$

then GC_k is updated as follows:

$$C_k = C_k + 1,$$

$$\forall\ i \in \{\ 1 \ldots n\ \}\ [\ c_i = max(c_i, t_i)\ ],$$

$$TS(e) = GC_k$$

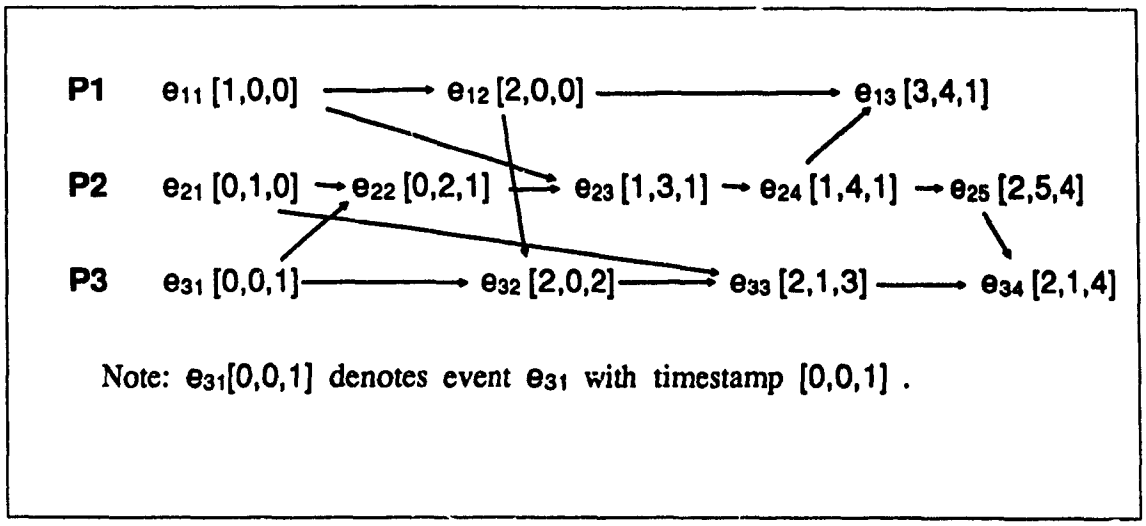


Figure 3.1 - Events in a distributed computation with timestamps in *GC*

The *receive* event e is then timestamped with the new value of GC_k. □

Example 3.1 Figure 3.1 shows a typical distributed computation where the events are timestamped with *GC* values. By comparing their timestamps, the causal relationship of the events can be established. For instance:

$$e_{23} \rightarrow e_{13}$$

since

$$[1,3,1] \ll [3,4,1]$$

where [1,3,1] and [3,4,1] are timestamps in *GC* of e_{23} and e_{13} respectively.

Meanwhile, there is no causal relationship between e_{23} and e_{12} since their timestamps, [1,3,1] and [2,0,0] respectively, are not comparable. □

The following are some properties of event timestamps using *GC*. e_{ij} denotes the j^{th} event in process P_i. For $e_{ij} \in P_i$, $TS(e_{ij})$ is the timestamp of e_{ij} in *GC* and

$TS(e_{ij})[k]$ is the k^{th} element in $TS(e_{ij})$.

Property P1 For $e_{kl} \in P_k$, $e_{kl} \rightarrow e_{ij}$ iff

$$TS(e_{kl})[k] \leq TS(e_{ij})[k], \quad \text{if } (k \neq i)$$

$$TS(e_{kl})[k] < TS(e_{ij})[k], \quad \text{if } (k = i)$$

that is, the value of the k^{th} element in the timestamp of event e_{ij} is always greater than or equal to those of the events that precede e_{ij}. □

Property P2 Define $prec(e_{ij}, k) = \{ e_{kl} \mid e_{kl} \in P_k \wedge e_{kl} \rightarrow e_{ij} \}$, then:

$$|prec(e_{ij}, k)| = TS(e_{ij})[k], \quad \text{if } k \neq i$$

and

$$|prec(e_{ij}, k)| = TS(e_{ij})[k] - 1, \quad \text{if } k = i$$

□

Properties **P1** and **P2** are proved in [27].

Definition 3.1 Given a pomset $p = [V_p, \Sigma_p, \Gamma_p, \mu_p]$ and $e_0 \in V_p$. $pref(p, e_0)$ is a pomset $q = [V_q, \Sigma_q, \Gamma_q, \mu_q]$ where:

$$V_q = \{ e \mid e \in V_p \wedge (e \rightarrow e_0 \vee e = e_0) \},$$

$$\Sigma_q = \{ a \mid e \in V_q \wedge \mu_p(e) = a \},$$

$$\Gamma_q = (V_q \times V_q) \cap \Gamma_p,$$

$$\mu_q = (V_q \times \Sigma_q) \cap \mu_p.$$

□

Lemma 3.1 Given a pomset p and $e_0 \in V_p$ is an event in p then:

$$q = pref(p, e_0) \quad \Rightarrow \quad q \leq_\pi p.$$

Proof Comes immediately from definition 3.1. □

In the following discussion, for a pomset $p = [V_p, \Sigma_p, \Gamma_p, \mu_q]$, by referring to a pomset q corresponding to a set of events $V_q \subseteq V_p$, we assume that $q = [V_q, \Sigma_q, \Gamma_q, \mu_q]$ where:

$$\Sigma_q = \{\ a \mid e \in V_q \wedge \mu_p(e) = a\ \},$$
$$\Gamma_q = (V_q \times V_q) \cap \Gamma_p,$$
$$\mu_q = (V_q \times \Sigma_q) \cap \mu_p.$$

□

Property P3 Given a computation W whose events are timestamped with GC. Define $U = CC(c_1, c_2, \ldots, c_n)$ as a pomset that consists of the first c_1 events in process P_1, the first c_2 events in process P_2, and so on. If:

$$\exists\ e_{ij} \in P_i\ [\ \forall\ k \in \{\ 1 \ldots n\ \}\ [\ c_k = TS(e_{ij})[k]\ ]\ ]$$

then $U = pref(W, e_{ij})$

Proof In the computation W, consider the event $e_{ij} \in W$ and the pomset $U = CC(c_1, c_2, \ldots, c_n)$ where $C_k = TS(e_{ij})[k]$, $k \in \{\ 1 \ldots n\ \}$.

$\forall\ e_{kl} \in W$:

$$\begin{aligned}
e_{kl} \in U \quad &\Leftrightarrow \quad l \le C_k \\
&\Leftrightarrow \quad TS(e_{kl})[k] \le TS(e_{ij})[k] && (\text{ algorithm 3.1 }) \\
&\Leftrightarrow \quad e_{kl} \rightarrow e_{ij} \vee e_{kl} = e_{ij} && (\text{ property P1 }) \\
&\Leftrightarrow \quad e_{kl} \in pref(W, e_{ij}) && (\text{ definition 3.1 }) \\
&\Leftrightarrow \quad U = pref(W, e_{ij})
\end{aligned}$$

□

Property P4 Given $e_{ij} \in P_i$ and $e_{kl} \in P_k$ being events in a computation W, the pomset $U = CC(c_1, c_2, \ldots, c_n)$ where

$$c_m = max(TS(e_{ij})[m], TS(e_{kl})[m]), \qquad m = 1 \ldots n$$

is a prefix of W: $U \le_\pi W$.

Proof

$\forall\ e_{rs} \in W$

$e_{rs} \in U \quad \Leftrightarrow \quad r \leq c_r$

$\Leftrightarrow \quad TS(e_{rs})[r] \leq TS(e_{ij})[r] \vee TS(e_{rs})[r] \leq TS(e_{kl})[r]$

(algorithm 3.1)

$\Leftrightarrow \quad e_{rs} \in pref(W, e_{ij}) \vee e_{rs} \in pref(W, e_{kl})$

(property P3)

$\forall\ e_{tu} \in W$:

$e_{tu} \rightarrow e_{rs} \quad \Rightarrow \quad e_{tu} \in pref(W, e_{ij}) \vee e_{tu} \in pref(W, e_{kl})$

$\Rightarrow \quad e_{rs} \in U$ (definition 3.1)

$\Rightarrow \quad U \leq_{\pi} W.$ □

Example 3.2

1. Consider event e_{13} of the computation showed in figure 3.1. For any event $e_{2j} \in P_2$,

$e_{2j} \rightarrow e_{13} \quad \Leftrightarrow \quad TS(e_{2j})[2] \leq TS(e_{13})[2]$ (= 4)

For example,

$e_{22} \rightarrow e_{13}$ since $TS(e_{22})[2] = 2$ (< 4)

$e_{25} \not\rightarrow e_{13}$ since $TS(e_{25})[2] = 5$ (> 4)

2. Since $TS(e_{13})[2] = 4$, there are four events in P_2 that *happen_before* e_{10}: e_{21}, e_{22}, e_{23} and e_{24}. Similarly, $TS(e_{13})[3] = 1$ means that there is one event in P_3 (e_{31}) that *happens_before* e_{13}.

3. $U = CC(1,2,3)$ is the pomset that consists of the first event in P_1 (e_{11}), the first two events in P_2 (e_{21}, e_{22}) and the first three events in P_3 (e_{31}, e_{32}, e_{33}) (figure 3.2).

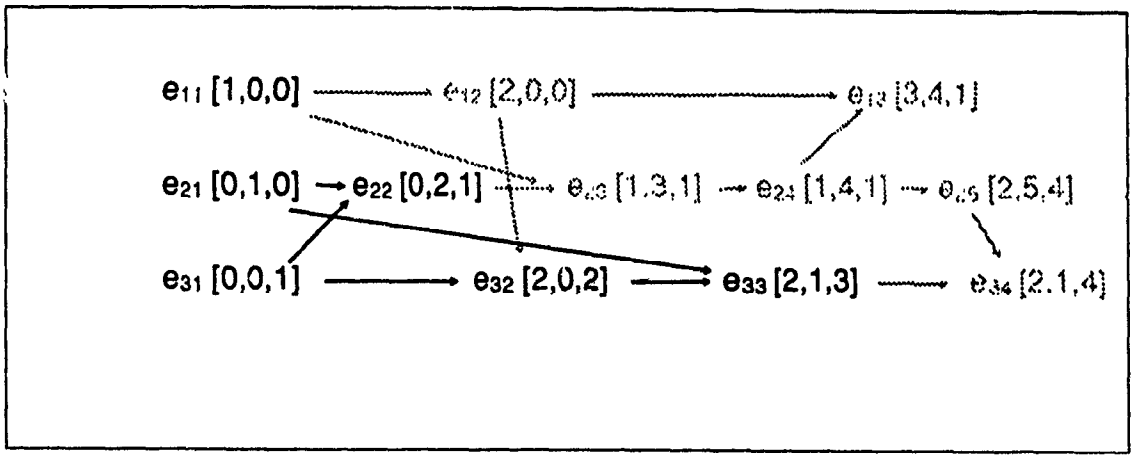


Figure 3.2 - Pomset $U = CC(1,2,3)$

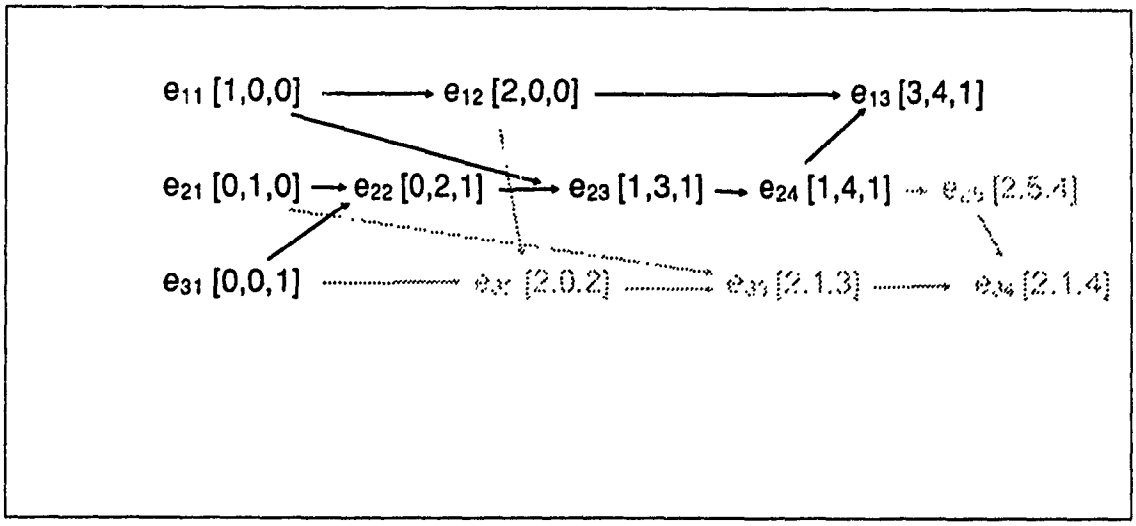


Figure 3.3 - Pomset $U = CC(3,4,1)$

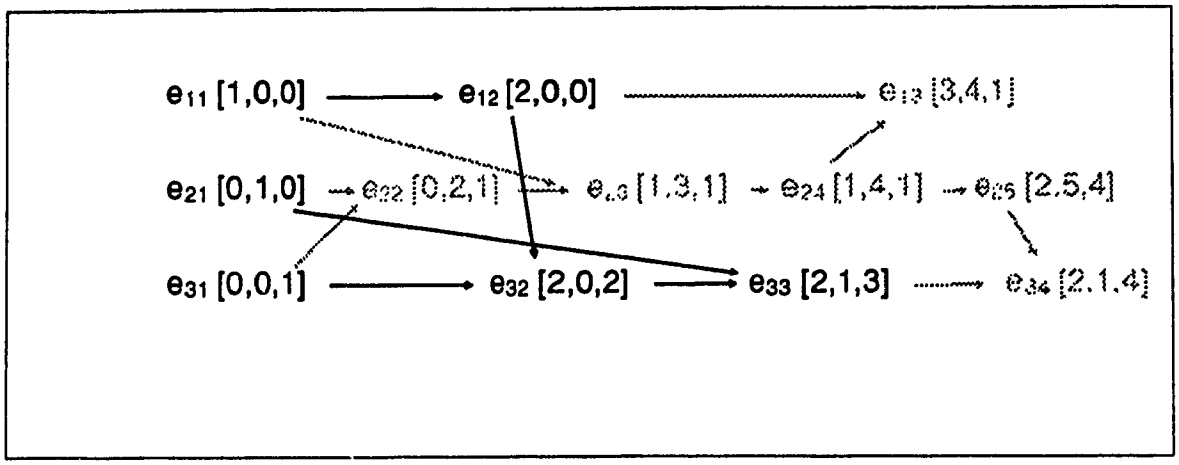


Figure 3.4 - Pomset U' = $CC(2,1,3)$

Now consider e_{13} whose timestamp is [3,4,1]: the pomset U = $CC(3,4,1)$ which consists of e_{11}, e_{12}, e_{13} (first three events in P_1), e_{21}, e_{22}, e_{23}, e_{24} (first four events in P_2), and e_{31} (first event in P_3) is a prefix of the current computation (figure 3.3).

Similarly, U'= $CC(2,1,3)$, obtained by applying **P3** on $TS(e_{33})$, is also a prefix of the current computation (figure 3.4)

4. Consider $TS(e_{13})$ and $TS(e_{33})$ which are [3,4,1] and [2,1,3] respectively.
U = $CC(3,4,3)$ is the pomset formed by applying **P4** (figure 3.5). U is a prefix of the current computation. □

3.1.2 Global State

The program's global state at a point can reveal the correctness of the execution of that program up to that point. In debugging, when a program is in an undesirable state, we infer that a bug has already occurred. Thus one main practice in debugging

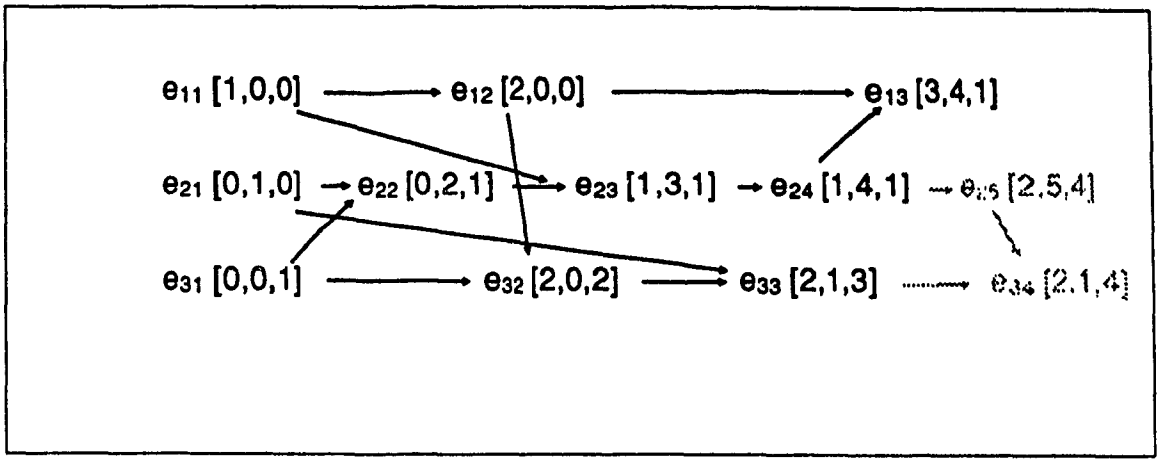


Figure 3.5 - Pomset $U = CC(3,4,3)$

is to examine the program's state from time to time and narrow down the "faulty region".

In a centralized system, state recording is done by suspending the execution and recording the program state information. *Instant snapshot*, as this technique is called, is impossible in DCS because of the lack of a common clock. The alternative is to obtain a *consistent global state* [33] since a *CGS* also has all the necessary details for determining the correctness of a computation.

We have two ways to obtain a *CGS*. One is to suspend the whole program at a breakpoint. With FIFO channels, the program is in a *CGS* when all processes are suspended, since it sees no receive event whose corresponding send event has not occurred. The requirement for the suspension, however, is that the program state at that point contains information that interests the user. Therefore uncontrolled suspension of the program is not very desirable. The breakpoint mechanism should

suspend the program at a point where its state can be verified. This is the objective of breakpoints, and will be discussed in section 3.3.2.

A *CGS* can also be obtained without having to suspend the program by applying the algorithm in [27] for *Spontaneous Global State Detection.* This algorithm can be used to obtain the *CGS* that is needed for deadlock detection.

3.1.3 Synchronization Behaviour Specification

One part of our strategy in distributed debugging is to have the debugger check the correctness of the program's synchronization behaviour. For this purpose, the debugger is provided with the *synchronization behaviour specification* (*SBS*) by the program's designer at the beginning of a debugging session. When the program is executed under the debugging environment, its actual synchronization behaviour is monitored and checked against the given *SBS* for possible discrepancies.

If an execution of the program is viewed as a pomset of events, the *SBS* is given as a set of pomsets. This set can, and in most case does, have an infinite number of elements. Each pomset element corresponds to a possible synchronization behaviour of the program. One obvious requirement for the *SBS* is that it must cover all possible behaviours of the programs. Therefore it is more feasible to use a behaviour generator to define the *SBS*. Through this generator, pomsets that compose the program's *SBS* are derived. Hence, the behaviour generator can also be called the pomset generator. Usually, the generator is designed in the form of a specification language. From the design of the application program, the user uses this language to describe the program's synchronization specification. A main requirement for this language is that it should effectively describe pomsets representing any computation.

3.1.4 Central and Local structures

Debugging is a process that relies heavily on human intervention. The debugger is only a tool which helps the user to examine a suspected area in the program, but it is the user's decision that an action should be taken in order to locate a bug. Therefore, control of the whole debugging environment needs to be centralized. Through a single user-interface, the user should be able to monitor the activity of the whole system and intervene at any time if necessary.

Certain aspects of debugging, such as deadlock detection or breakpointing, also need a global view. One approach is to assign a central site, which collects information from all other sites and composes a global picture from that.

However, for the sake of efficiency and minimizing the potential probe-effect, debugging operations should be distributed as much as possible. If an operation at one site does not depend on one at another site, it should be done independently at the local site. For instance, monitoring individual processes can be done independently at each local site.

Therefore, we choose a combination of central and local structures for our debugger. This follows the same philosophy as in [16]. The debugger has one central node, called the central debugger, which consists of the user-interface and other modules for global operations. Local debugger modules, one per node, receive commands from the central debugger. Figure 3.6 illustrates this organization.

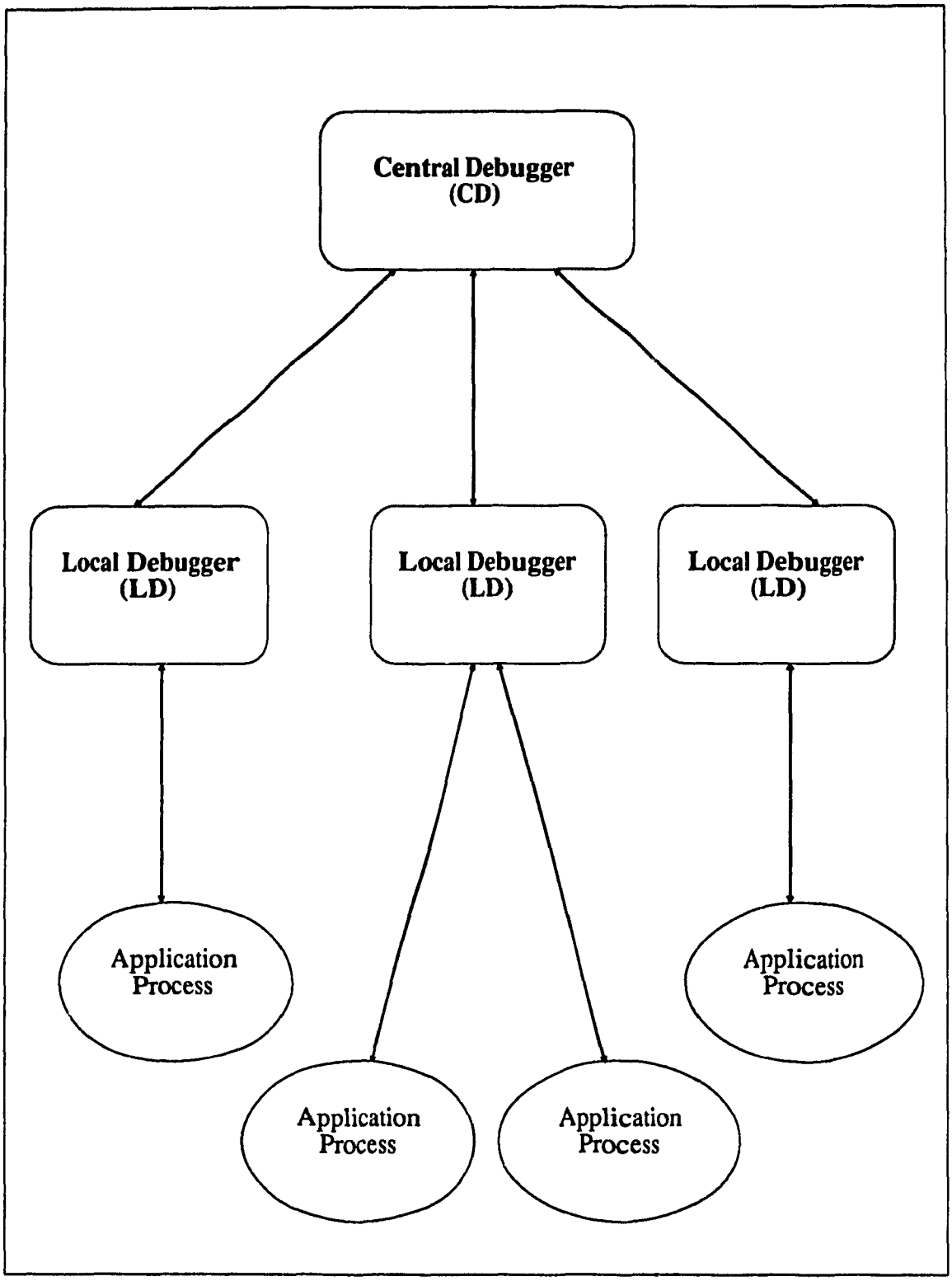


Figure 3.6 - General Structure of the Debugging System

3.2 Design Issues

3.2.1 Dependencies of debugger's modules

One design goal of the distributed debugger is to maximize its flexibility: the debugger should be able to work in different computing environments. It should also be able to support the debugging of programs written in different languages. When moving the debugger from one computing system to another, minimal changes should be required. For instance, the debugger needs process controllers which control the execution of the application processes. Obviously, these modules are system-dependent. When the debugger is used on different systems, appropriate versions of the process controllers are to be provided. However, the process controllers exchange information with the rest of the debugging system using data tokens of standard format. Thus changes are not required in the other modules, which are not system dependent.

To achieve this goal, the modules of our debugger address three classes of dependency: system, programming language, and specification language. Each module has its own operation and interacts with other modules through carefully designed interfaces.

Modules that are system-dependent contain operations which relate to the operating system and the underlying architecture. Programming-language dependent modules are those that work with the source codes of the debugged programs. There is also a need to have modules that are specification-language dependent. These modules must be modified when the specification-language is changed. Since different

people may have different ways of specifying designs, a fixed specification language should not be imposed on the users of the debugger.

3.2.2 Probes

To obtain internal information from processes, the debugger uses "probes". These probes are either inserted into the program codes, or attached to the communication ports or data areas. Through probes, internal states of a process can be extracted for examination by various debugging modules, and also by the user. Similar to probes that are used in electrical measurement systems, debugging probes may also alter the performance of the program. Compensation must be made to these probes in order to eliminate or reduce the so-called "probe-effect". As mentioned in section 2.4.3, this topic needs a detailed study of its own.

We divide probes into two types: static and dynamic. Static probes are inserted into the program code and remain there during the debugging process. Dynamic probes on the other hand can be set or removed by the user as desired during a single debugging session.

The static probes are mostly used to detect occurrences of synchronization events, since this information is always required during the debugging session. The Global Clocks that are attached to each process need this information to update their clock values. These probes send indicators of occurrences of every synchronization event to the *event processors*, which distribute them to other modules. For instance, the *tracker* needs this data as it monitors the behaviour of the program, and the breakpointing facility uses this information to determine whether a breakpoint has

been reached. Actually, there is a static probe for every action that is defined in the *SBS*.

Dynamic probes are used when the user needs to extract more information from a process. For instance, the user may want to monitor changes of a certain variable. A dynamic probe will be defined, which returns the new value of this variable every time it is updated. Dynamic probes are more useful once the synchronization error is located, and when the user is looking for the bug that caused it.

3.2.3 Event

An event is the occurrence of a specific action in a distributed program. Each action is characterized by the operation being carried out and the location where that operation takes place. All operations that are internal to one process are considered to take place at the same location. The communication ports of a process are considered to be different locations: the sending and receiving of messages through these ports are different actions from those internal to a process. Each port is used for only one type of message, hence sending of messages of different types are different actions.

Events are divided into two types: synchronization events and local events. Synchronization events are events occurring at communication ports. The actions of these events are defined in the *SBS*. In the actual computation, each event is associated with a label indicating its related actions. A set of counters are also maintained to keep track of the number of occurrences of each action. Through these identifiers,

the pomset representing the actual computation can be constructed and compared with those in the specification.

Local events are instances of actions done inside a process. Therefore, they have little importance when the debugger is being used to locate synchronization errors. Once synchronization errors are identified, and the user starts to look within a process, local events become significant. Through the use of dynamic probes, the user can view and record local events for investigation.

During the execution of a distributed program, event information can be collected in large volume. The user may reduce the volume of this information by setting *event filters.* An event filter is defined for an action. It removes all events of that action from the user's view. With event filters, the amount of pertinent information can be extracted to reflect only those activities of the program that interest the user at that time. An event filter is to be set up as a pair. If events of sending messages of one type are filtered out, so must be those of the corresponding receivings, and vice versa. Therefore, event filters can also be viewed as *channel filters.*

3.2.4 Checkpointing

Debugging is an iterative process: the user executes the program again and again, each time collecting more information about the program's behaviour, and narrowing down the suspected areas of the bug until the exact bug is located.

In a distributed environment, such re-execution can become very expensive. Distributed programs are usually complex, and it may take a lot of time to get to

the same point in the computation where the user is interested. Because of nondeterminism, the new execution might not even get to the same point. Therefore, it is useful to have a checkpointing mechanism, which allows the program to be re-executed from certain points during the computation instead of starting from the beginning. Checkpointing is also used in fault-tolerant computations for rollback and recovery.

To be able to roll back to a checkpoint, the program's state at that point must have been previously saved. When the program is rolled back to a checkpoint, the current state of the program is discarded, and the state saved at that checkpoint is loaded into the system. In a DCS, the saved state must be a *CGS*.

Obtaining the *CGS* of a distributed program during its execution is not simple. However, in our debugger, when the program is suspended at a breakpoint, its state is always a *CGS*. The user can create a checkpoint by saving this state. Therefore, in our debugging system, the user specifies the checkpoint by specifying a breakpoint, and then saving that state. Details on breakpoints are discussed in section 3.3.2.

The program's states that must be saved for the future roll-back also have to be clearly defined. However, the notion of what actually constitutes the program's state is ambiguous. Beside the processors' states, the program's state consists of many variables that are globally and locally defined, as well as port variables. Depending on each application program, what needs to be saved as a program's state at a checkpoint varies. Since the debugger is designed to be used for all kinds of programs, it must be able to identify what has to be saved in order for the program to be rolled back and restarted later. We propose a solution for this problem in section 4.4.

3.2.5 Database

While debugging a program, the user needs to observe and examine the program's states and behaviour as its execution proceeds. Because the execution happens too fast for a human to follow, *traces files* are used. A trace file has records of relevant state information, all in their proper order of occurrences, representing the execution history that the user can examine at a slower pace.

In a DCS environment, the volume of information in a trace file is much larger due to multiple threads of execution. Collection and maintenance of traces are better done using the distributed database system approach.

A distributed relational database is fragmented across processes. We assume one fragment of the database for each process in the debugged program. Details regarding a process are stored as a fragment at the local site of the process. The database actually consists of two tables. The first table contains information on synchronization events. The second contains only local events. The structure of the tables are:

- **Table 1:** *Timestamp* (Primary key)
  Action
  Counter
  Data

- **Table 2:**	*Timestamp*	(Primary key)
	Action	
	Counter	(Primary key)
	Data	

In table 1, *Timestamp* is in *GT*. *Action* is the activity done in the event. *Counter* shows the number of times the same action has occurred since the beginning of the execution. This is kept for the purpose of efficiency since one can always obtain the value of *counter* by scanning through the local fragment of the table. *Data* is relevant information related to the events, such as the contents of the messages.

The fields in table 2 are similar to those in table 1. Since in each process, the value of the GC is changed at the occurrence of an synchronization event, several local events may have the same TS, two fields – *Timestamp* and *Counter* – are used as the primary key in table 2. The *counter* counts the number of local events that has occurred since the immediately preceding synchronization event. In other words, the value of *counter* is reset every time a synchronization event occurs. Note that records on local events are collected through dynamic probes. As these probes are inserted and removed during the course of a debugging session, the set of local events varies accordingly.

This distributed database system also consists of local database managers, one for each site, which manage all database fragments at the local sites. A central database manager will coordinate all of these local managers and keep the whole database in a consistent state and offer a global, singular view of the database to the user.

Data in the database is updated when the program is executed in the debugging environment. These data are obtained through static and dynamic probes that are previously inserted into the program. During the execution, the "event processor" will collect this information and pass it to the local database manager, to store it in the right place. Local events are always kept in relation with their preceding synchronization events. Therefore, in operations on the database, we are concerned only with the synchronization events that are kept in table 1.

For any number of records in table 1 of the database, a *corresponding* pomset can always be formed based on information from these records. Each record corresponds to an event, whose action is specified in the *action* field of the records. The timestamps of the records define their partial order. For a database which has the full recording of a computation W, the *corresponding pomset* of its records is W.

The volume of data collected for a complete execution of a program can be enormous, and maintaining the database becomes impractical because of limitations of the system resource. To cope with this a *data window* can be defined. When the data window is employed, only records of a certain number of recent events are kept. As the execution proceeds, recent records are added to the database while the old ones are removed. Hence the size of the database will never exceed the window size, and the window can be viewed as a moving window. The central database manager will coordinate deletion of old records to keep the database in a consistent state. Consistency of the database with respect to the deletion of records is defined as follows:

Definition 3.2 Given a database of records from a distributed computation, its *consistency* is defined recursively as:

1. If no record is deleted from the database, the database is *consistent.*
2. Let p be the corresponding pomset of the database before a deletion, q be the corresponding pomset of records that are deleted, and p' be the corresponding pomset of the records that remain in the database after the deletion. p' is *consistent* iff p is *consistent* ***and*** $q \leq_{\pi} p$.

□

In practice, the user specifies the window size of the database at the beginning of a debugging session. This value of the window size is given to the local database managers. When the data of the program trace is collected, the local manager keeps track of the size of data fragments. When the size of a fragment exceeds the window size, the corresponding local manager will determine the cut-off point in the database fragment. All records preceding the cut-off point are deleted so that the size of the fragment can be reduced. The central database manager is also informed so that it can coordinate similar deletion in other database fragment in order to keep the whole database consistent. The following algorithm provides a method for reducing the size of the database and keeping its consistency.

Algorithm 3.2. *Database Size-reduction Algorithm.*

Input: - Database for a distributed computation of n processes: $P_1, P_2, \ldots, P_n$; $DB_1, DB_2, \ldots, DB_n$ are the database fragments corresponding to the processes.

- a certain event $e_{ij} \in P_i$, $i \in \{ 1 \ldots n \}$

Output: - the database in which a number of records are deleted.

Step 1: $\forall\ k \in \{\ 1...n\ \}\ [\ TS(e_{ij})[k]$ *is sent to* $DB_k\ ]$

Step 2: $\forall\ e_{kl} \in P_k\ [\ TS(e_{kl})[k] \leq TS(e_{ij})[k]$

$\Rightarrow$ *record for* e_{kl} *is deleted*]. □

Theorem 3.1 Let p be the pomset corresponding to the database DB; $e_{ij} \in P_i$ be an event in p; q be the pomset corresponding to the records deleted by applying algorithm 3.2 to DB; p' be the pomset corresponding to the resulting database DB' after the application of the algorithm. If p is consistent then:

1. $\forall\ e_{im} \in P_i\ [\ m \leq j \Rightarrow e_{in} \in q\ ]$
2. p' is consistent.

Proof

1. $\forall\ e_{im} \in P_i$:

$m \leq j \Rightarrow$ $TS(e_{im})[i] \leq TS(e_{ij})[i]$ (algorithm 3.1)

$\Rightarrow$ *record for* e_{im} *is deleted* (step 2, algorithm 3.2)

$\Rightarrow$ $e_{im} \in q$

2. $\forall\ e_{kl} \in q$:

$TS(e_{kl})[k] \leq TS(e_{ij})[k]$ $\Leftrightarrow$ $e_{kl} \rightarrow e_{ij} \vee e_{kl} = e_{ij}$ (property P1)

$\Leftrightarrow$ $e_{kl} \in pref(p, e_{ij})$ (definition 3.1)

$\Rightarrow$ $q \leq_\pi p$ (lemma 3.1)

$\Rightarrow$ p' *is consistent.* (definition 3.2)

□

Thus every time the size of a database fragment has to be reduced, the local database manager for that fragment determines the record corresponding to the cut-off point. The timestamp of the record is then sent to the central manager to initiate the size-reduction. The central manager in turns distributes the values of the timestamp's

DB_1

Event	Timestamp
e_{11}	1,0,0
e_{12}	2,0,0
e_{13}	3,1,0
e_{14}	4,1,0
e_{15}	5,1,0
e_{16}	6,5,1
e_{17}	7,5,1

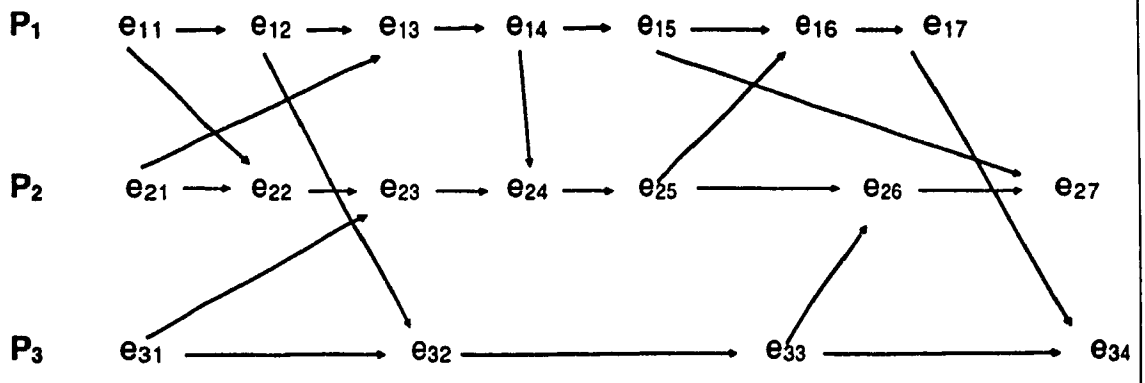
DB_2

Event	Timestamp
e_{21}	0,1,0
e_{22}	1,2,0
e_{23}	1,3,1
e_{24}	4,4,1
e_{25}	4,5,1
e_{26}	4,6,3
e_{27}	5,7,3

DB_3

Event	Timestamp
e_{31}	0,0,1
e_{32}	2,0,2
e_{33}	2,0,3
e_{34}	7,5,4

a) Records in database



b) Distributed computation corresponding to records in (a)

Figure 3.7 - Database before *size reduction*

components to other local managers according to algorithm 3.2. Upon receiving these values, and based on algorithm 3.2, the local managers can determine which records to be deleted in response to the size-reduction so that the database can still be consistent.

Example 3.3 The tables in figure 3.7a show a sample of the database, which contains the recording for the computation in figure 3.7b. Each table corresponds to one process, and contains records of events and their timestamps.

Assume that the size of DB_1 has exceeded the window size and e_{14} and all records preceding it must be deleted to reduce the size. While the records are removed, DB_1 also sends $TS(e_{14})[2]$ and $TS(e_{14})[3]$, which are 1 and 0, to DB_2 and DB_3 respectively. Upon receiving this value, DB_2 will delete all events e_{2j}, where $TS(e_{2j})[2] \leq 1$ from its database. Thus e_{21} is deleted from P_2. It easy to see that no record is deleted from P_3. The tables after the size-reduction and the ST-graph constructed from the data in these tables are showed in figures 3.8a and 3.8b.

Assume that P_2 also has to reduce its size and must delete all records up to and including e_{24}. $TS(e_{24})[1]$ and $TS(e_{24})[3]$ (which are 4 and 1) are sent to DB_1 and DB_3, respectively. Since $TS(e_{31})[3] = 1$, it is deleted from P_3. In P_1, there is no e_{1j} where $TS(e_{1j})[1] \leq 4$, none is deleted. The tables and their corresponding ST-graph become as shown in figures 3.9a and 3.9b. □

Sometimes, a complete history of a process has to be preserved, such as for use in an artificial environment to debug other processes. In that case, instead of discarding the past records completely, they can be archived onto a backup file for later use.

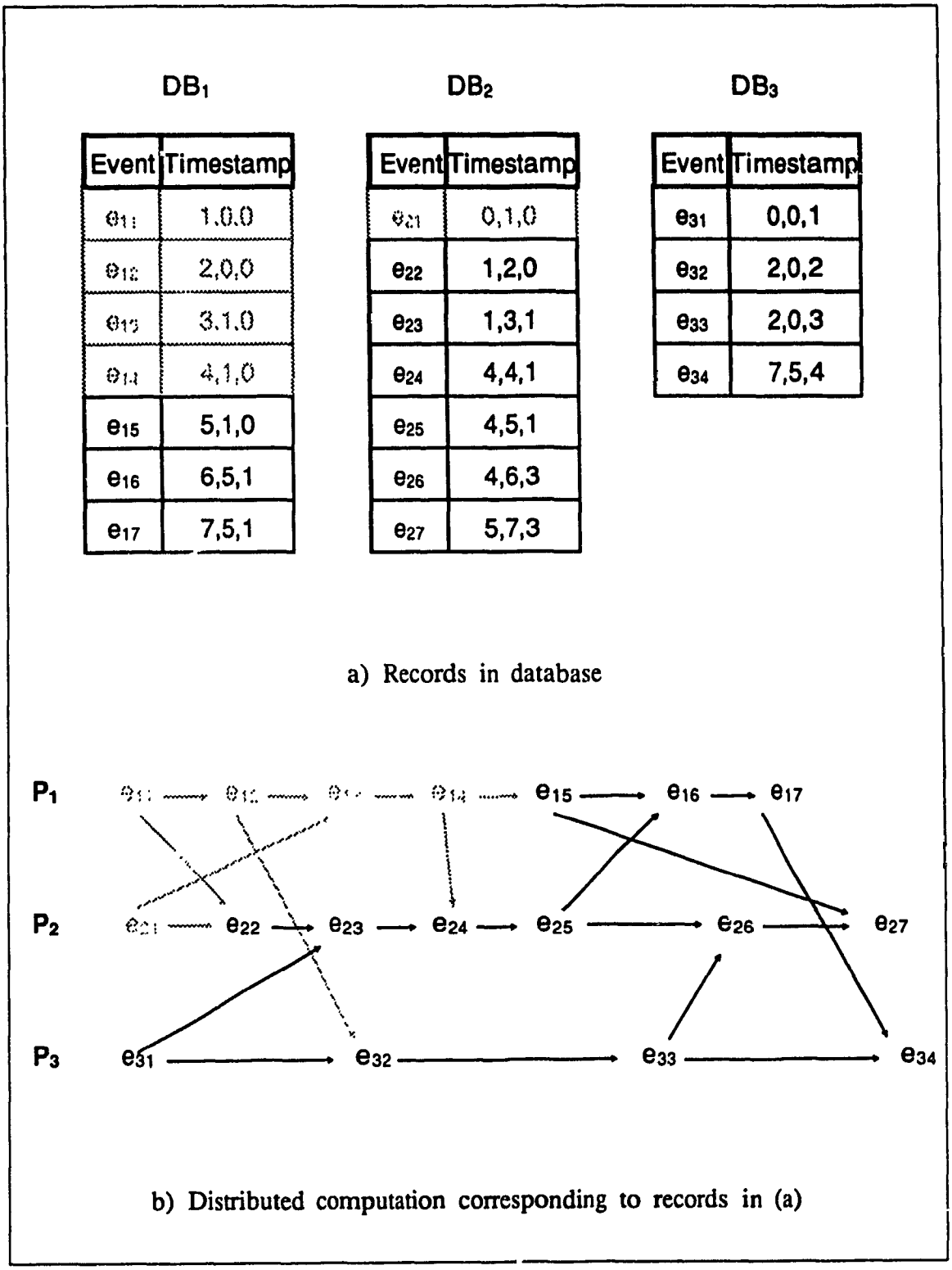
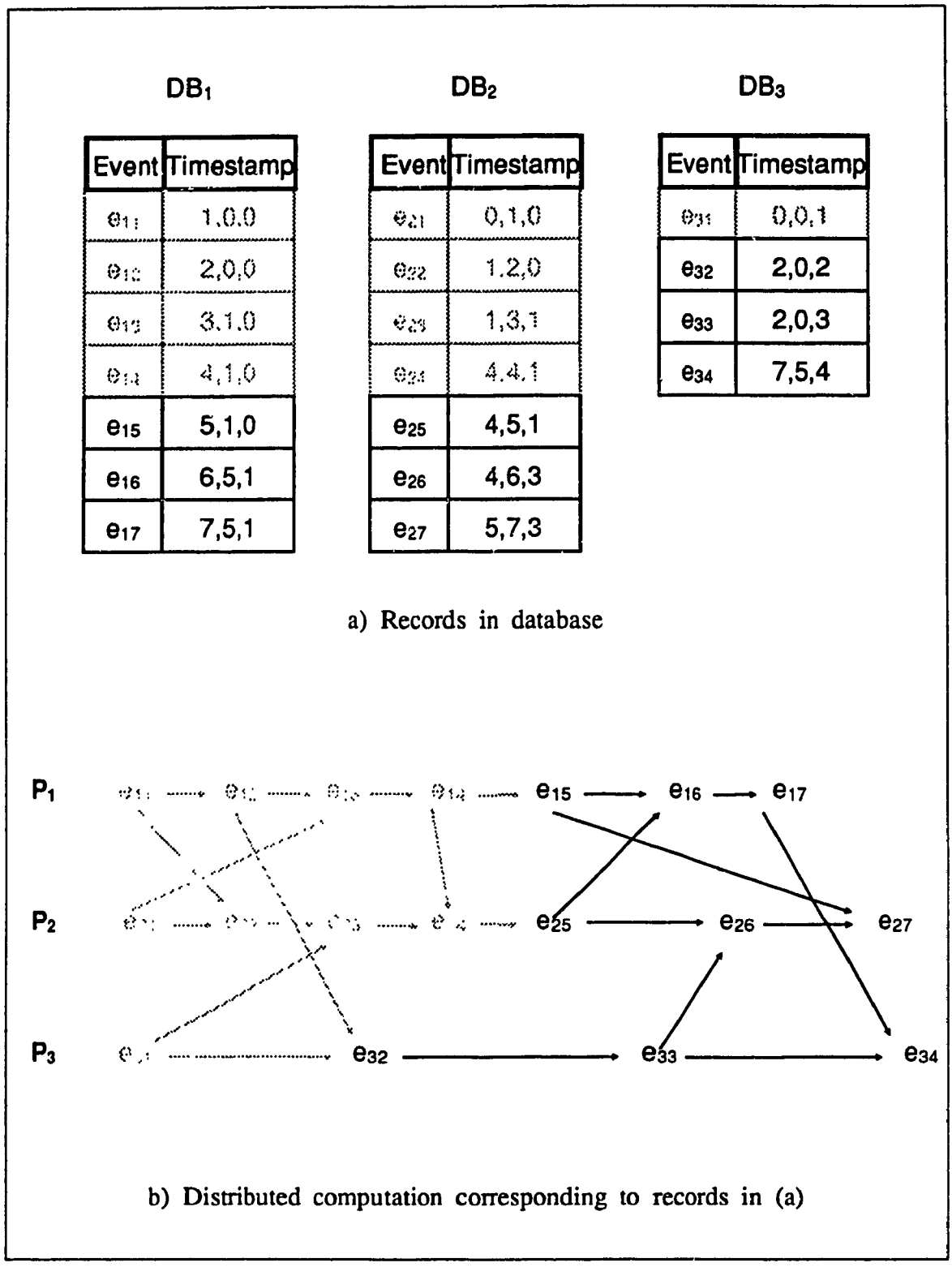


Figure 3.8 - Database after 1st *size reduction* initiated by DB_1



a) Records in database

b) Distributed computation corresponding to records in (a)

Figure 3.9 - Database after 2nd *size reduction* initiated by DB_2

Queries can be directed to the database by the user or by other debugging utilities, when the program is not running. Queries are made to the central manager, which analyses them and sends secondary queries to appropriate local managers. The user examines the program's behaviour and its states at various points during the computation through such queries. Utilities can also send queries to the central manager to retrieve information relevant to their operations. For instance, a query requesting all event data sorted in their timestamp is made when the ST-graph is built for figure 3.7.

The trace data collected and stored in the form of a distributed database can be used for several purposes. First of all, the user can examine the history of an execution of the debugged program. In another application, the user may decide to run only one but not all distributed processes but can still have the effect of them running. The trace information in the database is used to simulate the messages sent to the selected process. Genuine data in these messages can also be replaced by synthesized data before the messages are delivered to the receiving process. Thus the user will be able to examine the behaviour of the process under different sets of input data.

3.3 Design Choices

3.3.1 SBS and Compiler

We needed a language to describe the *SBS* which we call the *specification definition language* (*SDL*). For our prototype distributed debugger, we have proposed a simple *SDL* that is described below. This language lacks certain features such as allowing the use of predicates in the specification. There are also certain pomsets it cannot define. Nevertheless, it allows us to experiment with our prototype.

The basic components of the *SDL* are *actions* and *operators*. Each instance of an action represents an event. Operators define the causal relationship between events.

In the scope of our *SDL*, an action is characterized by the operation being carried out and its locality, i.e., the process in which the action occurs. Since each process is viewed as strictly sequential, we have the following restrictions on events:

1. All events of the same locality are serialized, and
2. No two events of the same action can occur simultaneously.

These restrictions pose some limits on how a process can be specified. On the other hand, without the restrictions, some pomsets cannot be defined using this language.

The four operators on pomsets that we use are defined in [29]. We repeat the definitions here, and then define their application when used on *sets of pomsets.*

Given two pomsets, $p = [V, \Sigma, \Gamma, \mu]$, $q = [V', \Sigma', \Gamma', \mu']$, then:

Concatenation

$$p;q = [V \cup V', \Sigma \cup \Sigma' \cup (V \times V'), \Gamma \cup \Gamma', \mu \cup \mu']$$

We usually write $p;q$ as pq for short.

Conjunction

$$p \,\|\, q = [V \cup V', \Sigma \cup \Sigma', \Gamma \cup \Gamma', \mu \cup \mu']$$

that is, there is no ordering requirement between members of V with those of V'.

The two following operators produce a set of pomsets instead of a new pomset.

Disjunction

$$p\%q = \{ p, q \}$$

If a process is described to have a behaviour of $p\%q$, it is to have either p or q as its computation for every execution, but not both.

Kleene's star

$$p^* = \{ \varepsilon, p, pp, ppp, \ldots \}$$

We extend the above operators on sets of pomsets as follows:

$$P;Q = PQ = \{ pq \mid p \in P \wedge q \in Q \}$$

$$P \,\|\, Q = \{ p \,\|\, q \mid p \in P \wedge q \in Q \}$$

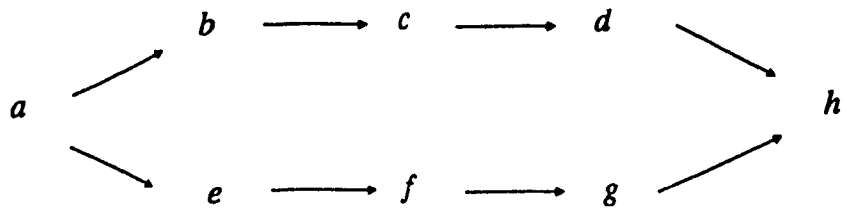
$$P\%Q = P \cup Q$$

$$P^* = \{ p^* \mid p \in P \}$$

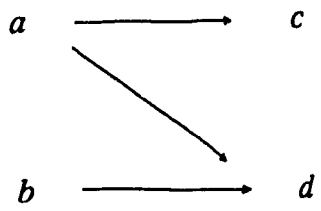
where P and Q are sets of pomsets.

Example 3.4

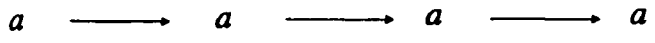
1. a[(bcd) ‖ (efg)]h is the pomset



2. [a(c ‖ d)] ‖ (bd) is the pomset

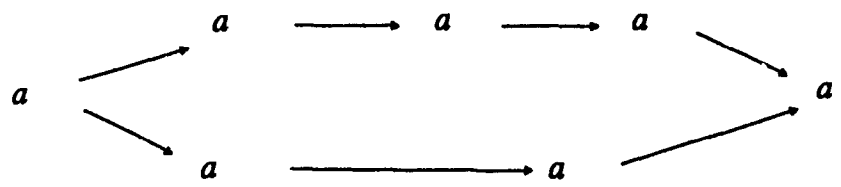


3. a[a ‖ (aa)]a is the pomset



which can also be defined simply as *(aaaa)*.

Because of our restrictions on the definition of actions and events, we cannot have, for instance, the following pomset:



However, we will be able to define pomsets such as the one in example 3.4(2).

□

A compiler will translate the *SBS* of the debugged program written in *SDL* into a data structure that represents the *SBS* internally for the debugger. In addition, the compiler will also check for semantic correctness in the specification. For example, it can detect if an exchange of messages between two processes is attempted without a channel connecting them.

The use of a compiler for handling *SBS* allows flexibility. Different people can have their own versions of *SDL* and they can de designed to suit their own needs. Only the compiler module will have to be changed in the debugger. The compiler can also be designed to carry out more extensive checks on the program semantics as well as verification of certain design rules required by the organization.

3.3.2 Breakpoint

The use of breakpoints in debugging allows the user to examine the program's state on-line. When the execution reaches the breakpoint, it is suspended. The user can then examine the program as long as is necessary. Furthermore, the advantage of breakpoints is that the execution can be resumed. The user can continue to debug without having to restart the program from the beginning. By examining the program's state and its behaviour history up to the breakpoint, the user may be able to assert the correctness of the preceding computation, and may gain more clues about the location of the errors.

In conventional debugging, the breakpoint is defined in the "implementation space", that is the user defines a section in the code as a breakpoint. As the program is about to execute that section, it is suspended. As discussed before (section 3.3.1), defining breakpoints in this manner for a distributed program is not practical.

Our notion of breakpoints in distributed debugging is based on the program's behaviour. To be more precise, it is based on what is expected to be observed in the program's behaviour. In other words, the breakpoints are defined in the "specification space".

In contrast with conventional breakpoints, breakpoints based on behaviour are imprecise. In conventional breakpoints, the execution is always suspended just before the code at a breakpoint is executed. In controlling breakpoints defined in the specification space, the specified behaviour must be observed before the execution is suspended. When the program's execution is suspended, its history is supposed to

possess the specified behaviour corresponding to the breakpoint specification. Of course we assume that the program's computation contains no error up to that point.

In our design, the user chooses a breakpoint based on the program's specification. Given a program specification $P = \{ p_1, \ldots, p_n \}$, a breakpoint can be specified as a pomset p_{bp}, $p_{bp} \leq_\pi p_i$, $p_i \in P$. That is, p_{bp} is a prefix of a pomset in the specification.

During the execution, the debugger will compare the observed computation α of the program with the specified breakpoint p_{bp}. Execution is suspended when the following condition is true:

$$\exists \alpha \exists \beta [p_{bp} \leq_\alpha \beta \leq_\pi \alpha]$$

In other words, α contains a prefix which *agrees with* the breakpoint p_{bp}. It is possible that for the breakpoint p_{bp}, $p_{bp} \leq_\pi p_i$, there exists $p_j \in P$, $p_j \neq p_i$, such that $p_{bp} \leq_\pi p_j$. The breakpoint is reached, i.e., the execution is suspended, if the current computation is the implementation of any of these $p_j \in P$.

A defined breakpoint p_{bp} may never be reached. The actual computation does not have the behaviour that is expected. That means, given the observed computation α:

$$\nexists \beta [\beta \leq_\pi \alpha \wedge p_{bp} \leq_\alpha \beta]$$

One reason for a specified breakpoint not being reached is that, due to nondeterminism in the execution, the pomset that the breakpoint is based on is not chosen for the computation. For instance, consider a breakpoint p_{bp} defined as:

$$p_{bp} \leq_\pi p_i, p_i \in P.$$

If the observed computation α *agrees with* $p_j \in P$, but does not with $p_i \in P$:

$$\nexists \beta \leq_\pi p_i [\beta \leq_\alpha \alpha]$$

then the program will not be suspended. Neither will the breakpoint be reached if the computation contains synchronization errors, in which case α does not *agree with* any $p_i \in P$. However, the user may have to do further investigation, using other debugging tools, in order to determine the exact reason which caused the breakpoint not to be reached.

The breakpoint might not be reached if the computation fails the program's liveness requirement. For instance, the execution terminates prematurely, or enters a deadlock. In this case, the reason for the breakpoint not to be reached is quite apparent to the user.

The user may also want to specify the breakpoint based on a few, but not all, actions in the computation. For example, the breakpoint is specified based on the expected behaviour of the program with respect to only events of action a and b in the program. In such case, the defined breakpoint p_{bp} is a prefix of the projection of p_i onto the set of the desired actions. Formally, given the set A of actions that interest the user, then a breakpoint can be defined as:

$$p_{bp} \leq_\pi q,\ q = proj(p_i, A),\ p_i \in P.$$

Obviously, two different computations, α_1 and α_2, may reach the same breakpoint p_{bp} which is defined in this manner, as long as the following condition is true:

$$\exists\, \beta\, [\, \beta \leq_\pi proj(\alpha_1, A) \wedge \beta \leq_\pi proj(\alpha_2, A) \wedge p_{bp} \leq_\alpha \beta\,]$$

The breakpoint defined in this manner corresponds to different behaviours, all of which possess common behaviour with respect to a number of actions. The program is suspended whenever this common behaviour is observed.

The breakpoint specification is simply the description of the pomset p_{bp}. The syntax for the description, however, should be simple for the user to use in order to specify the chosen pomset. On the other hand, the debugging system should also be able to use the specification and match it with the actual behaviour. We use predicates which are based on event counters for this purpose. Each predicate has two parts: one is used to specify the required temporal order of events in the pomset, the other to specify the prefix corresponding to the breakpoint. Here are a few examples on the breakpoint specification, whose detail semantics are being worked out at the time of this writing by Lea [25]:

Example 3.5 Given the pomset:

$$(a \rightarrow b \rightarrow c)^*$$

meaning that the sequence *abc* is to repeat indefinitely. If a breakpoint is to be reached after three iterations of this sequence, it is specified as:

$$(\ C(c) \leq C(b) \leq C(a)\ ) \qquad \text{(part 1)}$$

$$\wedge \quad (\ C(a) = 3 \wedge C(b) = 3 \wedge C(c) = 3\ ) \qquad \text{(part 2)}$$

where $C(a)$ denotes the number of occurrences of action a.

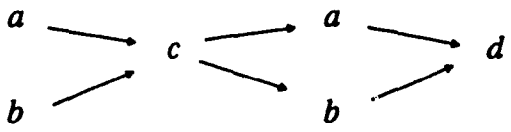
Part 1 of the breakpoint specification is used by the debugger to determine whether the observation matches the desired pomset, i.e., $(\ a \rightarrow b \rightarrow c\ )^*$. Part 2 is used together with part 1 to determine the prefix of the pomset that corresponds to the breakpoint, which is:

$$a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \rightarrow c$$

□

Example 3.6

For the prefix:



the corresponding breakpoint can be specified as:

$$(C(a) = 2 \wedge C(b) = 2 \wedge C(c) = 1 \wedge C(d) = 1)$$
$$\wedge \quad (C(c) \leq 1 \wedge C(a) \leq C(c) + 1 \wedge C(b) \leq C(c) + 1$$
$$\wedge\ 2(C(c) + C(d)) \leq C(a) + C(b))$$

□

When the execution is suspended at a breakpoint, the user can examine the current program state as well as the recent computation. Or he can step through a local process without generating a synchronization event. He can add probes or remove the existing probes. His ultimate aim is to locate the bug.

3.3.3 Trace

Trace is used to preserve the history of a program. As the program reaches a "milestone" during the execution, a record is entered into a file called the *trace file*. Thus the path of computation through which the program has gone can be reconstructed using a trace file. Depending on the granularity of the user's needs, the milestones can be the procedures that have been called, the conditional branches that have been taken, or even every statement in that program.

In our debugger, traces of all synchronization events are recorded in the database. Each record has a timestamp so that the proper order of the events can be fully reconstructed. Optionally, the user can define the local events to be recorded. The local events are also timestamped with the current Global Clock value.

The ST-graph is used to represent the history of the program. In its graphical form, the ST-graph gives a visual picture of the temporal order of events in a computation. We define the ST-graph formally below.

Definition 3.3 A *directed labeled graph* q is a quadruple (V, Σ, E, μ) where:

V = set of vertices

Σ = set of labels

μ = $V \rightarrow \Sigma$ is a many-to-one mapping

$E \subseteq V \times V$ is a non-transitive, irreflexive and asymmetric relation. □

From now on we will use the name *directed labeled graph* to refer to the isomorphism class $[V, \Sigma, E, \mu]$ of such a graph.

Definition 3.4 $g(p)$ is an one-to-one and onto function from the set P of pomsets to the set G of directed labeled graphs (where P and G are considered universal sets) which is defined as:

$\forall\ p \in P$:

$$g(p) = q \quad \Leftrightarrow \quad \Big(V_q = V_p,\ \Sigma_q = \Sigma_p,\ \mu_q = \mu_p,\ E_q = \{\ (v_1, v_2)\ \big|\ (v_1, v_2) \in \Gamma_p \wedge \exists\, v_3\ [\ (v_1, v_3), (v_3, v_2) \in \Gamma_p\ ]\ \}\ \Big)$$ □

From definition 3.4, we also have:

$$p = g^{-1}(q) \quad \Leftrightarrow \quad \Big(v_p = v_q,\ \Sigma_p = \Sigma_q,\ \mu_p = \mu_q,\ \Gamma_p = \{\ (v_1, v_2) \mid (v_1, v_2) \in E_q \vee (\exists\, v_3 \in V_q\ [\ (v_1, v_3), (v_3, v_2) \in E_q\ ])\ \}\ \Big)$$

For a distributed computation W, $g(W)$ is called the ST-graph of W. Each pair $(v_1, v_2) \in E$ is called a *process edge* if v_1 and v_2 belong to the same process, otherwise it is called a *channel edge*. There should only be a *channel edge* between a matching pair of *send* and *receive* events if the graph is used to represent a distributed computation. In debugging, to prevent having an ST-graph which has *channel edges* between non-matching events, the corresponding pomsets should be maintained so that whenever a *receive* event is removed from the pomset, its matching *send* is also removed.

An ST-graph can always be constructed from a *consistent database* (definition 3.2) which keeps the partial recording of the distributed computation (section 3.2.5). The following algorithm shows how the ST-graph can be constructed:

Algorithm 3.3 *ST-graph Construction Algorithm.*

Input: Database fragments DB_1, DB_2, ..., DB_n corresponding to processes P_1, P_2, ..., P_n. Each record in the database has information on the event type (*send* or *receive*) and the event timestamp in GC.

Output: An ST-graph $q = [V_q, \Sigma_q, \Gamma_q, \mu_q]$.

Step 1: V_q, Σ_q and μ_q can be obtained from data stored in the records.

Step 2: For every $e_{ij}, e_{ik} \in P_i$:

$$(e_{ij}, e_{ik}) \in E_q \quad \Leftrightarrow \quad TS(e_{ij})[i] + 1 = TS(e_{ik})[i].$$

These pairs form the process edges of the graph.

Step 3: For every *send* event e_{ij}, form the set

$$R(e_{ij}) = \{\ e_{kl} \mid (e_{kl} \in P_k) \wedge (e_{kl} \textit{ is a receive event}) \wedge (\ TS(e_{kl})[i] = TS(e_{ij})[i]\ )\ \}.$$

If

$$\exists\ e_{rs} \in R(e_{ij})\ [\ \exists\ e_{kl} \in R(e_{ij})\ [\ TS(e_{kl}) \ll TS(e_{rs})\ ]\ ]$$

then $(e_{ij},\ e_{rs}) \in E_q$.

These pairs forms the channel edges of the graph. □

Theorem 3.2 If p is the corresponding pomset of records in a consistent database, and q is the ST-graph obtained from applying algorithm 3.3 to the database then

$$q = g(p)$$

Proof

a) $\forall\ e_{ij},\ e_{ik} \in P_i$

$(e_{ij},\ e_{ik}) \in Eq$

$\Leftrightarrow$ $(\ e_{ij} \rightarrow e_{ik}\ ) \wedge (\ \exists\ e_{il} \in P_i\ [\ e_{ij} \rightarrow e_{il} \wedge e_{il} \rightarrow e_{ik}\ ]\ )$ (definition 3.4)

$\Leftrightarrow$ $(\ TS(e_{ij})[i] < TS(e_{ik})[i]\ ) \wedge (\ \exists\ e_{il} \in P_i\ [\ TS(e_{ij})[i] < TS(e_{il}[i]) \wedge TS(e_{il}[i]) < TS(e_{ik}[i])\ ]\ )$ (property P1)

$\Leftrightarrow$ $TS(e_{ij})[i] + 1 = TS(e_{ik})[i].$ (algorithm 3.1)

b) $\forall\ e_{ij} \in P_i,\ e_{rs} \in P_r,\ r \neq k$:

$(e_{ij},\ e_{rs}) \in Eq$

$\Leftrightarrow$ (e_{ij} *is a send event* $\wedge$ e_{rs} *is a receive event*

$\wedge\ \exists\ e_{kl}\ [\ e_{ij} \rightarrow e_{kl} \wedge e_{kl} \rightarrow e_{rs}\ ]$) (definition 3.4)

$\Leftrightarrow$ (e_{ij} *is a send event* $\wedge$ e_{rs} *is a receive event*

$\wedge\ e_{rs} \in R(e_{ij}) \wedge \exists\ e_{kl} \in R(e_{ij})\ [\ e_{kl} \rightarrow e_{rs}\ ]$)

(property P1 and step 3 of algorithm 3.3)

$\Leftrightarrow$ (e_{ij} *is a send event* $\wedge$ e_{rs} *is a receive event*

$\wedge\ e_{rs} \in R(e_{ij}) \wedge \exists\ e_{kl} \in R(e_{ij})\ [\ TS(e_{kl}) \ll TS(e_{rs})\ ]$)

(algorithm 3.1)

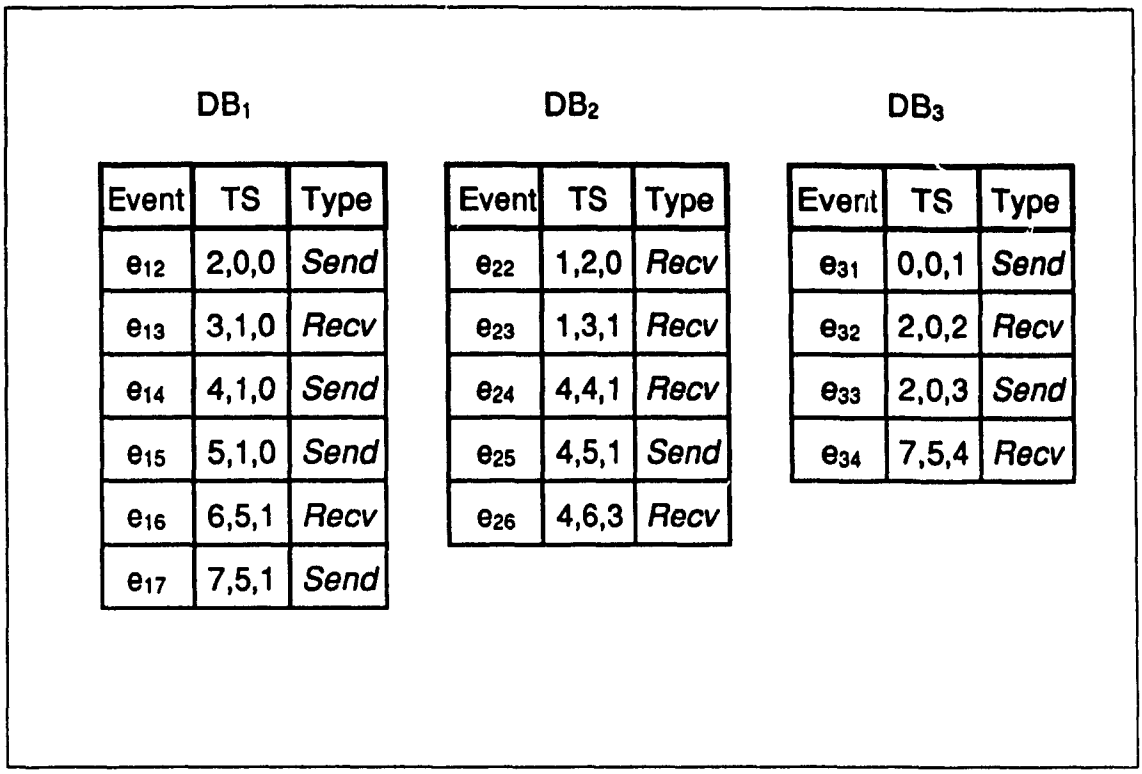
□

Example 3.7 Let us use algorithm 3.3 to construct the ST-graph using the data given is figure 3.10.

a. *Process edges* for events in DB_i are obtained by sorting the timestamp of the events on the i^{th} element ($TS(e_{ij})[i]$). Figure 3.11 shows these edges.

b. *Channel edges* for event e_{25} can be established by first forming the set $R(e_{25})$: $R(e_{25}) = \{\ e_{16},\ e_{34}\ \}$ (because $TS(e_{16}) = TS(e_{34}) = 5$). Since $TS(e_{16}) = [6,5,1]$ and $TS(e_{34}) = [7,5,4]$, $TS(e_{16}) \in TS(e_{34})$. Therefore there is a *channel edge* from e_{25} to e_{16}, and $e_{16} = r(e_{25})$.

c. Consider the *send* event e_{15}, $R(e_{15}) = \emptyset$. We conclude that $r(e_{15})$ has not been recorded.



DB_1

Event	TS	Type
e_{12}	2,0,0	*Send*
e_{13}	3,1,0	*Recv*
e_{14}	4,1,0	*Send*
e_{15}	5,1,0	*Send*
e_{16}	6,5,1	*Recv*
e_{17}	7,5,1	*Send*

DB_2

Event	TS	Type
e_{22}	1,2,0	*Recv*
e_{23}	1,3,1	*Recv*
e_{24}	4,4,1	*Recv*
e_{25}	4,5,1	*Send*
e_{26}	4,6,3	*Recv*

DB_3

Event	TS	Type
e_{31}	0,0,1	*Send*
e_{32}	2,0,2	*Recv*
e_{33}	2,0,3	*Send*
e_{34}	7,5,4	*Recv*

Figure 3.10 - Database used for example 3.7

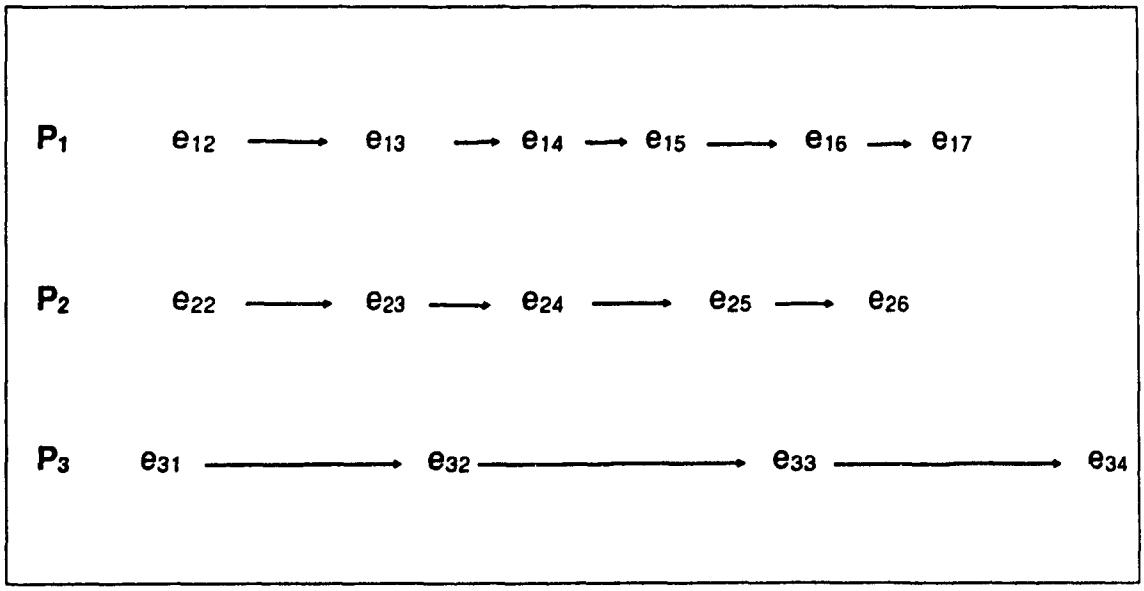


Figure 3.11 - *Process edges* in the ST-graph of example 3.7

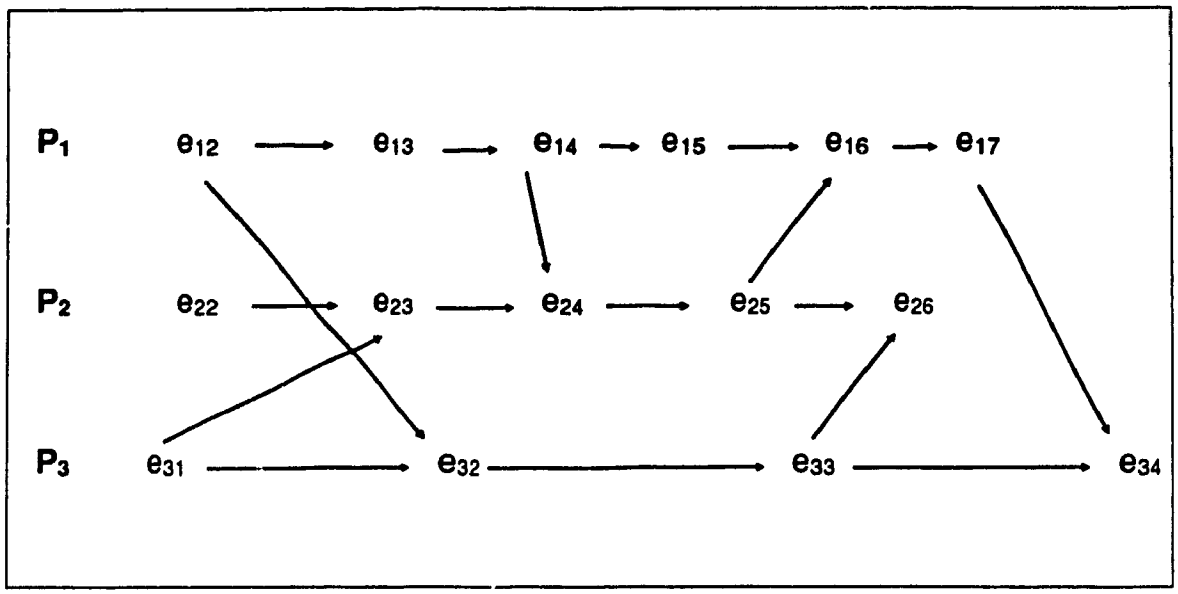


Figure 3.12 - The ST-graph constructed from data in figure 3.10

The full ST-graph for the database is shown in figure 3.12. Notice that the receive events e_{23} and e_{33} have no *matching send.* Their *matching* events have been deleted from the database. □

After the ST-graph is constructed from the synchronization events, the local events can be added to the graph, the *TS* and *counter* field in the database is used to determine their positions in the graph.

An ST-graph can be very large when drawn out. A sliding window can be used to display only a section of it. When this window is moved to another section, that part of the graph will be shown to the user.

Conceptually, the use of the sliding window on a ST-graph $g = (V, \Sigma, E, \mu)$ is equivalent to dividing V into three disjoint partitions, V_l, V_m and V_r. The window

is moved when elements are transferred from one partition to another. The following requirements must always be observed for the window to be correct:

1. V_l, V_m and V_r are always pair-wise disjoint;
2. $\forall\ v_1, v_2 \in V\ [\ v_2 \in V_l \wedge (v_1, v_2) \in E\ ] \quad \Rightarrow \quad v_1 \in V_l$;
3. $\forall\ v_1, v_2 \in V\ [\ v_2 \in V_m \wedge (v_1, v_2) \in E\ ] \quad \Rightarrow \quad v_1 \in (\ V_l \cup V_m\ )$

The last two requirements ensure that for any two elements that belong to two different partitions, the edge between them, if there is one, always goes from V_l to V_m or from either V_l or V_m to V_r.

To move the window without violating the above requirements, the following rules can be followed:

a. No element can be transferred directly from V_l to V_r or vice versa;

b. For any $v \in V_m$, v can be transferred to V_l iff $pred(v) \subseteq V_l$;

c. For any $v \in V_l$, v can be transferred to V_m iff $succ(v) \subseteq (\ V_m \cup V_r\ )$

where

$pred(v) = \{\ v' \mid v' \in V \wedge (v', v) \in E\ \}$, and

$succ(v) = \{\ v' \mid v' \in V \wedge (v, v') \in E\ \}$.

Similarly,

d. For any $v \in V_r$, v can be transferred to V_m iff $pred(v) \subseteq (\ V_l \cup V_m\ )$; and

e. For any $v \in V_m$, v can be transferred to V_r iff $succ(v) \subseteq V_r$.

An example of the ST-graph and the sliding window is show in figure 3.13.

At each vertex in the graph, the user can extract more information related to the associated event by querying the database. Queries sent to the database use the timestamps and, in case of a local event, the local event *counter* as keys to search for related records.

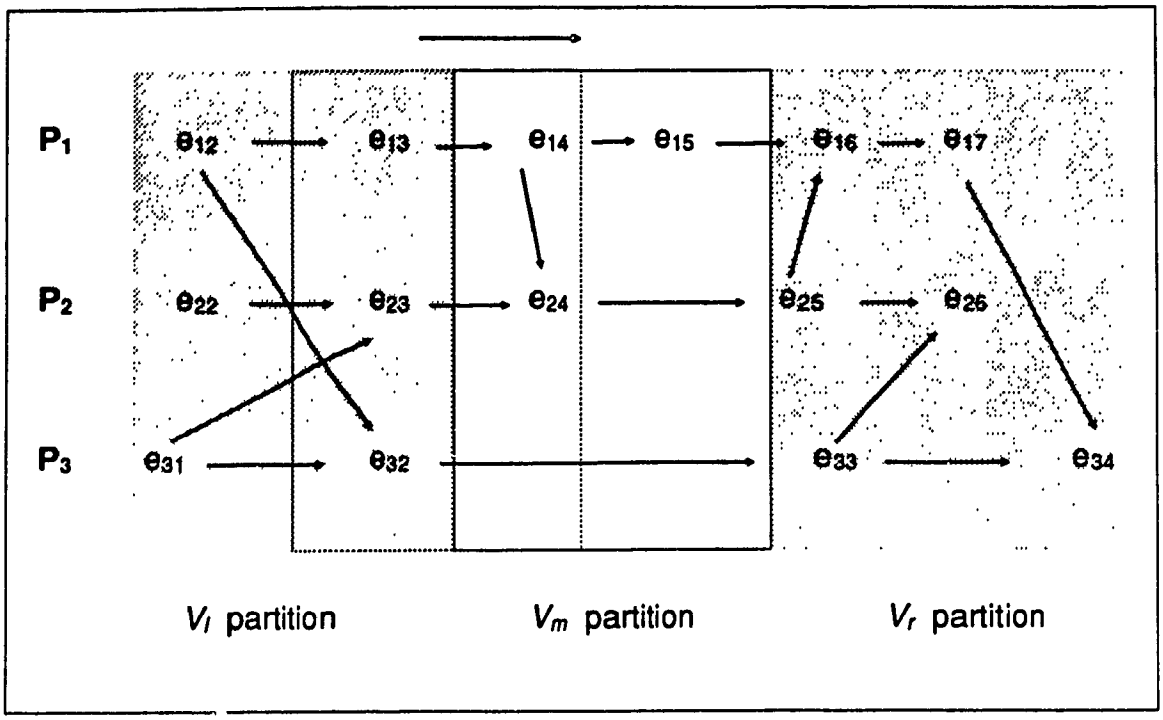
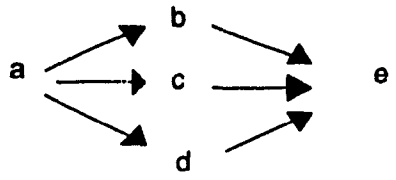


Figure 3.13 - ST-graph and the sliding window

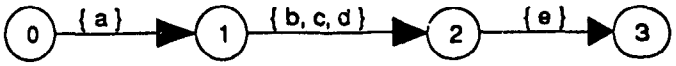
3.3.4 Tracking

Tracking is the operation in which the observed behaviour of a program is compared on-line against the specification of that program. Thus tracking is the main operation in the first stage of distributed debugging. The program is set to run while the debugger monitors the program behaviour in terms of the communication activities. Having been provided with the *SBS*, the debugger will suspend the program whenever it detects an occurrence of an event that does not follow the *SBS*. The user is notified of the location of the violation.

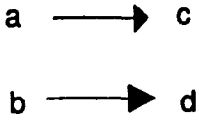
The method of checking the actual behaviour of the program against its specification has been proposed by Bates in [5] and also used in [4, 18]. Bates uses a *Shuffle Automaton* for the purpose of tracking. In the *Shuffle Automaton*, the tokens that cause state transitions are defined as sets of concurrent events. For example, the *Shuffle Automaton* for the following sequence of events:



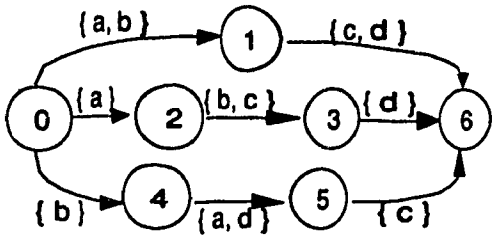
looks like:



A tracking facility using a *Shuffle Automaton* can recognize any interleaving of the concurrent events b, c and d. However, a *Shuffle Automaton* may face a combinatorial explosion if there are partial orders between some of the concurrent events. For instance, the *Shuffle Automaton* representing the following sequence of events:



is



Baiardi [4] and Gordon [18] use an assertion checker instead. Their approach reduces the complexity of tracking, but it could not cover all possible errors the program may have.

By using the pomset model, we can reduce the complexity faced by the *Shuffle Automaton* approach, and still keep track of all possible moves of the program. The tracking procedure is done basically in the following steps:

- From the *SBS*, the set of pomsets $R(\alpha)$, which represents all possible computation of the program is built,
- From this set, the set of actions that can occur, called *next_event*, is formed.
- If the action of the next event is not in this set, a violation is assumed,
- If the next event is valid, $R(\alpha)$ is updated, and a new *next_event* set is built.

This method does not try to construct all possible combinations of events that a serialized concurrent computation can have. In fact, the set *next_event* represents

all concurrent actions, and the actual order of their occurrence is irrelevant. Therefore, the complexity that is faced by methods based on *FSM* is no longer present.

Tracking using the pomset approach can effectively handle concurrency and partial orders. The drawback in the pomset approach is in the way nondeterminism is described. Every computation which corresponds to an outcome of a nondeterministic choice must be represented by a pomset. Thus a set of pomsets is used to specify the program behaviour. Since an observation can *agree with* more than one of these pomsets in the specification, more than one *next_event* set is needed for the purpose of tracking.

Tracking is effective in locating synchronization errors. However, it does present additional overhead to debugging operations. The application process cannot be allowed to execute more than one event ahead when the tracking module is checking the validity of the last event. As mentioned in section 2.4, the delay imposed on the application process due to tracking can be a source of probe-effect. Therefore, the use of tracking should be discouraged when probe-effect is suspected to be present in the debugging system.

3.3.5 Event Colors and Logical Channels

During a computation, processes can, and usually do, exchange different kinds of messages. However, for practical reasons, these messages are sent on the same channel that connects two processes. Internal computations of a process will decipher the message data to know the type of the received message.

While monitoring the program, the debugger can only perceive the actions that are done when messages are exchanged. That is, they can only detect that the action is for sending or for receiving a message. For the purpose of checking against specification, this kind of information is apparently not enough. Hence, more detail should be obtained from the communication ports of the processes. On the other hand, the debugger should not go too far to examine the contents of the messages, as that will cause too much interference to the normal execution of the program.

Our solution is to assign colours to synchronization actions [19]. These actions are divided into groups, each of them will have a distinct colour. When an event occurs, it will have the colour of the related action. The monitors of the debugger are built to be colour sensitive. They recognize the colour of the events, hence the events can be classified into groups without requiring the debugger to look into the message contents.

In assigning colour to the synchronization events, we also extend the idea by making the messages have the same colour as their corresponding *sends* and *receives*. In effect, logical channels can be formed by assuming that a channel can carry only messages of one colour. Thus two processes may have more than one logical channel connecting them, as viewed by the debugger, instead of only one physical channel as implemented.

With events grouped together by colours, colour filters can be installed. The filters give a projected view of the program behaviour by removing certain types of actions from the whole picture. Formally, given a program behaviour $p = [V, \Sigma, \Gamma, \mu]$, and a set F of actions to be filtered, then the resulting behaviour after filtering is the pomset $p' = proj(p, \Sigma - F)$. Thus the user can screen out irrelevant

actions when debugging a program, keeping only those that are deemed significant to the computation. Figure 3.14 illustrates this effect. Figure 3.14a shows a distributed computation whose events have two colours, *dark* (e.g., e_{11} and e_{25}) and *shade* (e.g., e_{12} and e_{33}). If events of *shade* colour are filtered out then the computation will be perceived as shown in figure 3.14b.

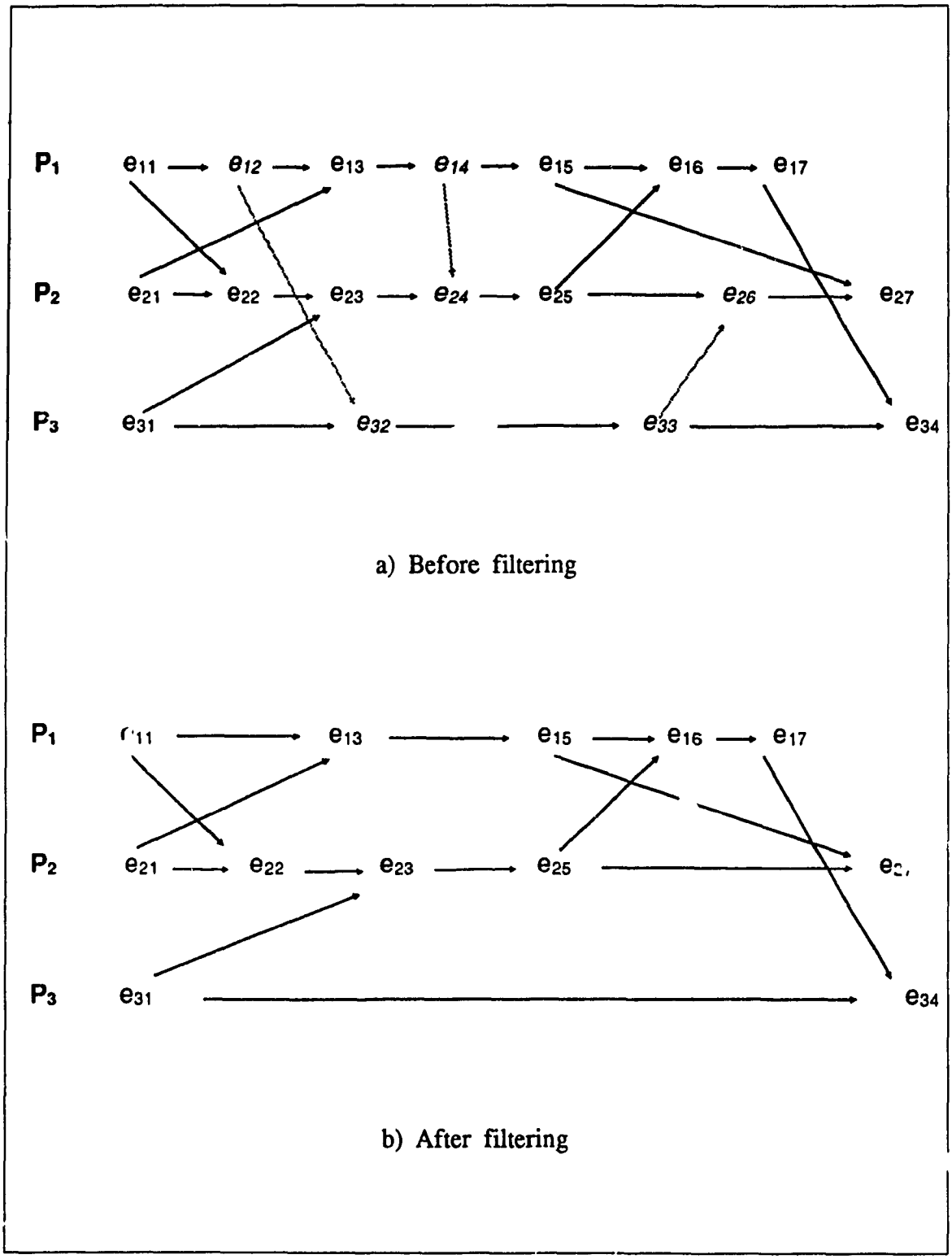


Figure 3.14 - A computation with coloured events being filtered.

Chapter 4

Implementation

A prototype for the distributed debugger using the design discussed in chapter 3 has been implemented and tested. The programming language used for the implementation is C. The test programs for the debugger were also written in C. The DCS that is used for this implementation consists of four SUN workstations, running under SUN's UNIX. In the following sections, we will describe the structure of the debugger, operation of the modules, and other major details of the implementation.

4.1 System Structure

The debugger is divided into two parts: the *central debugger* (*CD*) and the *local debugger* (*LD*). Each part of the debugger is further divided into several modules for handling different tasks. The list of the modules and their functionalities is shown in table 4.1 for the *CD* and table 4.2 for the *LD*. In the current implementation, the *CD* and *LDs* are treated as single processes running on their own sites. However, the modular construction of this software allows for extensions to support multiple processes in each site.

Table 4.1 - Central Debugger modules

Module	System Dependent	Programming Language Dependent	Specification Language Dependent
Transformer		X	
SBS Compile			X
Breakpoint Coordinator			
B/D Detector			
Database Central Mgr.			
User Interface	X		
CD/LD Interface	X		

Table 4.2 - Local Debugger modules

Module	System Dependent	Programming Language Dependent	Specification Language Dependent
Process Controller	X		
Event Processor			
Tracker			X
Breakpoint Recognizer			
Database Local Mgr.	X		

4.1.1 Central Debugger

There is a single *CD* for the whole debugging system. The *CD* consists of coordinator modules which keep the operations in the *LDs* in synchronization. Figure 4.1 shows the general structure of the *CD*.

The Transformer is a preprocessor which converts the source program into a "debuggable" version. Abstractly viewed, the transformer inserts static probes into the program. These probes allow the debugger to detect occurrences of synchronization events. This is achieved by adding additional codes to the program. When these added codes are executed, they will send information to the debugger.

With proper annotation in the source codes by the programmer, colours can be added to synchronization events. Thus more logical channels can be assigned to the same physical channel, where each logical channel carries only messages of the same colour (see 3.3.5).

By the use of the transformer, the debugger can be adapted for use with different programming languages. In fact, a debugger can have several transformers, one for each programming language that the debugger is intended for.

SBS Compiler reads the *SBS* written in *SDL* and generates the necessary data structures which define the program's components and their synchronization. Two of these structures are the *process table* and the *action table*. The *process table* is the list of all processes in the program. This list has the process names as assigned by the user, along with their site locations, as well as the logical channels inter-connecting them. The *action table* contains, for each action, its identifier, the assigned colour, and the type of action (e.g., *send* or *receive*). The *CD* will distribute these tables to

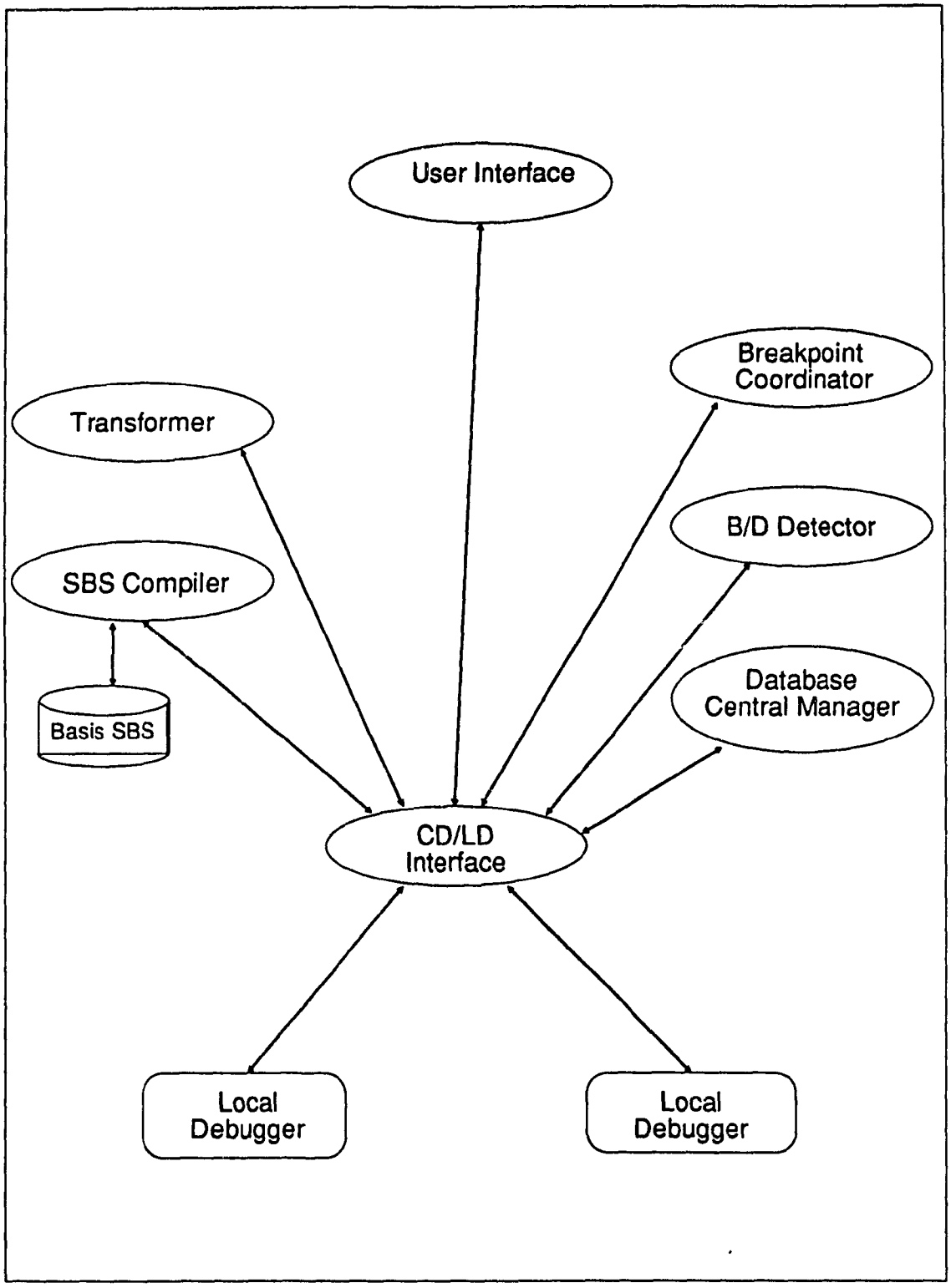


Figure 4.1 - Central Debugger Structure

all *LDs*; information in them is necessary for identifying various components and events in the program.

The compiler also generates the internal representation of the *SBS*. This data structure is described in detail in section 4.2.2. While generating this data structure, the compiler also does some semantic checking, such as whether an action can occur in a certain process.

The *SBS* structure that is built from *SDL* is called the *basis SBS*. This is the program specification in its most detailed form. Prior to an execution of the debugged program, the compiler module will read the basis *SBS* and generate another version called the *working SBS*. In the *working SBS*, actions with colours that are to be filtered will be removed. The working *SBS* is then distributed to the *LDs*. Furthermore, an *SBS* for a composite process can be generated from the basis *SBS*. This feature is not included in this implementation.

Breakpoint Coordinator receives breakpoint definitions from the *User Interface*. The condition for a breakpoint is analysed and the predicates and action counters are identified. The counter values are sent to *Breakpoint Controllers* at the *LDs*; the module ensures that a counter value of an action and predicates involving that action are only sent to the node where that action is allowed to occur. For example, if the *counter* part of the definition for a breakpoint is :

$$a_1 = x_1 \textbf{ and } a_2 = x_2 \textbf{ and } a_3 = x_3 \ldots$$

where a_i is an action defined for process P_i, then the *coordinator* will send $[a_1 = x_1]$ to the *LD* which controls P_1, $[a_2 = x_2]$ to the *LD* which controls P_2, and so on.

The local *Breakpoint Controller* will signal the *BP coordinator* whenever a local action counter reaches the specified value. The *coordinator* will evaluate its list of BPs, and suspend the execution if a breakpoint is reached, i.e., all the conditions of that breakpoint are satisfied.

Blocked and Deadlock (B/D) Detector is used to detect the condition where an application process is blocked indefinitely on a dead channel because the process at the other end of the channel has terminated prematurely. The detector also recognizes deadlock condition in which a cycle of processes are all waiting for messages to be sent from one another.

For the purpose of B/D detection, a predefined *timeout* value can be set for every awaited receive event. If a process is blocked on an empty channel longer than this *timeout* period, its *LD* will send a signal to the B/D detector. The detector will obtain the execution state of the process at the other end of the channel. If it finds out that the channel is "dead", i.e., the process at the other end of the channel has terminated prematurely, the whole program will be suspended, and the user will be informed.

If the other process is also blocked on another empty channel, the detector repeats the inquiry. A directed graph is built and updated at every inquiry. Each vertex of the graph corresponds to a process and is timestamped with the process's local *GC* value. If a cyclic path is detected in this graph, the detector gets one more sample of local *GC* values of all processes in the cycle. A deadlock is presumed to have occurred if the current *GC* values are the same as those in the timestamps. In this case as well, the execution will be suspended and the user will be informed of the situation.

The B/D detector is implemented as an additional tool to help the user detect liveness violations in the program. The effectiveness of this module relies on appropriate setting of the *timeout* value, the selection of which in turns depends on the user's knowledge and experience with the system. The debugger and its detection module, however, cannot determine anything about the state of a process which is alive but does not exhibit any activity for a long period. Such processes may be either in an undesirable infinite loop, or busy doing a very complex internal computation. If this inactiveness of a process causes some other processes to "*time-out*", the B/D detector will only inform the user. The latter should decide on how to handle the situation.

Database Central Manager can also be called the *query processor*, since it handles only queries; the collecting and updating of data are done locally by the *database local managers* in the *LDs*. A query is analysed and broken down into subqueries by the *CD* before they are sent to *LDs*. Using the *process table* and *action table*, the central manager can decide on the destination for each subquery. For instance, if a query is for events belonging to only one process, it will be sent to the *LD* that handles that process.

User Interface. The user controls the debugger through this module. Among the sub-modules in the *User Interface* are the *command processor* and the *ST-graph generator*. The ST-graph can be generated in different styles, using text or graphic representation. It can also be set to grow in "slow-motion", giving a picture of how the computation proceeds. If graphic capabilities are available, the ST-graph can also be used to generate an animation of a computation.

The Interface between *CD* and *LDs*. All instructions and data going between the *CD* and *LDs* must pass through an interface module called the *Central-Local*

Interface (*CLI*). The *CLI* operates in two modes: *idle* and *running*. It is in *idle* mode when the program is not in execution. In this mode, the *CLI* reacts on commands entered through the *User Interface*. In other words, it is invoked directly or indirectly by user commands. By contrast, in *running* mode, the *CLI* is the server for all other modules at the *CD* as well as the *LDs*. It receives requests, instructions or data from others, and passing them to the proper destination.

4.1.2 Local Debugger

The *LD* controls all application processes at the local node. Therefore there is only one copy of a specific module in a *LD*. The exception is the *Process Controller*, which requires one copy for every application process. The structure of the *LD* is shown in figure 4.2.

Process Controller. This is the only module that has access to the "internal space" of an application process. Through the process controller, dynamic probes can be set and removed during the debugging process. It can also return any part of the process's state as requested. Any breakpoints that are defined in the implementation space will be set through this module. In addition, the module is used to launch the application process at the start of an execution, and to detect whether the process has terminated.

Event Processor. All static probes send their signal and information to this module, which in turn will distribute the information to other modules that need them, such as the tracker or the database manager. The module also manages the *Global Clock*, and the timestamping of the events as they occur.

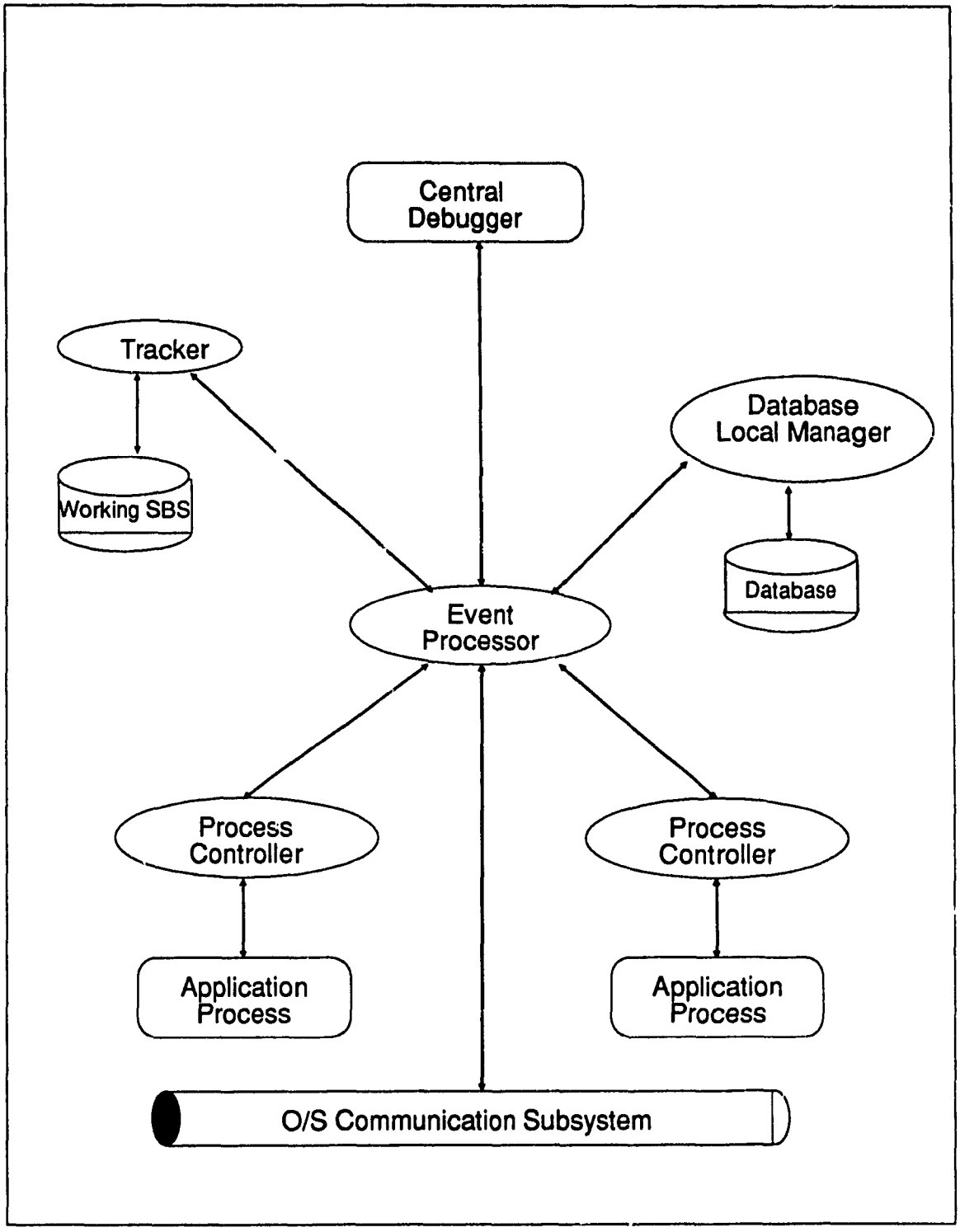


Figure 4.2 - Local Debugger Organization

In this implementation, the *event processor* is also used as the central server of the *LD*. It relays signals from other *LD's* modules to the *CD*, as well as passing requests and commands sent from the *CD* to the *LD*'s modules.

Tracker. Just before the debugged program is executed (under the debugger's control), the tracker receives the copies of the working *SBS* (from the *CD*) that are pertinent to application processes at the local nodes. From this *SBS*, it derives the set of events that can occur and sends to the event processor. When an event occurs, the event processor passes the related information to the tracker, which updates its structures and derives the new list of possible next events.

Because the process for tracking is quite complex compared to the operations of other modules, the tracker is implemented as a separate process. Thus after passing the list of next events to the event processor, the tracker can carry out further operation on its structures, preparing for the next step of tracking.

Breakpoint Recognizer keeps a list of action counters and predicates, updating them from tokens sent by the event processor. There is also another list of specified counter values related to breakpoints. These values are sent from the breakpoint coordinator. When an event occurs, the list of action counters are updated, and then the list of predicates is reevaluated. Any predicates that are no longer valid will be flagged; the current computation does not *agree with* the pomset specified with those predicates. When a counter reaches the specified value, it generates a signal to the coordinator, which will update its master list of breakpoints.

Database Local Manager. During the execution, the database local manager collects data on events from the event processor and inserts them into the database,

updating the indexes accordingly. Using these indexes, it can answer queries sent by the database central manager.

When the size of the database file exceeds a preset limit (see 3.2.5), the module will determine the event at which all preceding events will be deleted. The timestamp of this event is sent to local managers at other sites through the central manager. The local managers will delete old records at their sites accordingly, so that the remaining data will be consistent.

4.2 Main Data Structure

Process Table and Action Table

The process and action tables contain the basic information about the program to be debugged. Figure 4.3 shows an example of the process table of a distributed program that consists of three processes. Their communication channels are shown in figure 4.4. Each entry in the process table consists of the process ID, its location in the system, and all channels connected to it. The user can use this ID to identify the process in various user commands. The channels are logical channels and each can carry messages of one colour. Hence, a channel is identified by its direction, the connecting process and the colour assigned to that channel. The action table has the names assigned to every actions, along with the type of the action as well as the assigned colour (figure 4.5).

The locations of the processes are only logical, where the nodes are identified with numbers 1, 2, 3, ... When the debugger is launched, the corresponding numbers

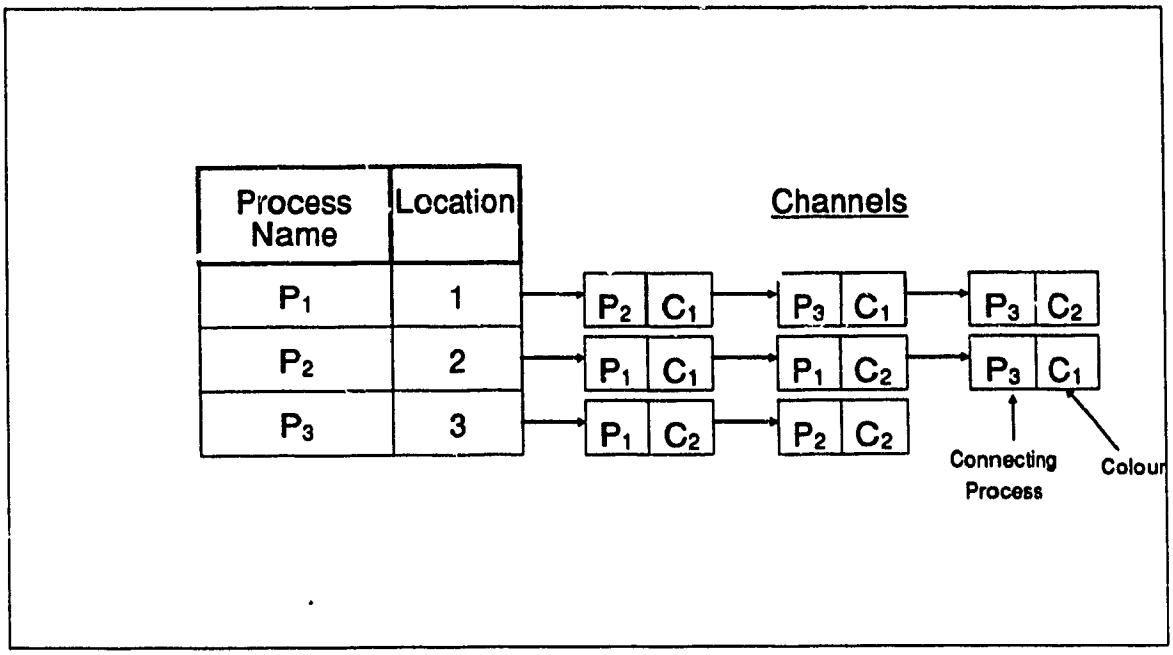


Figure 4.3 - Process Table

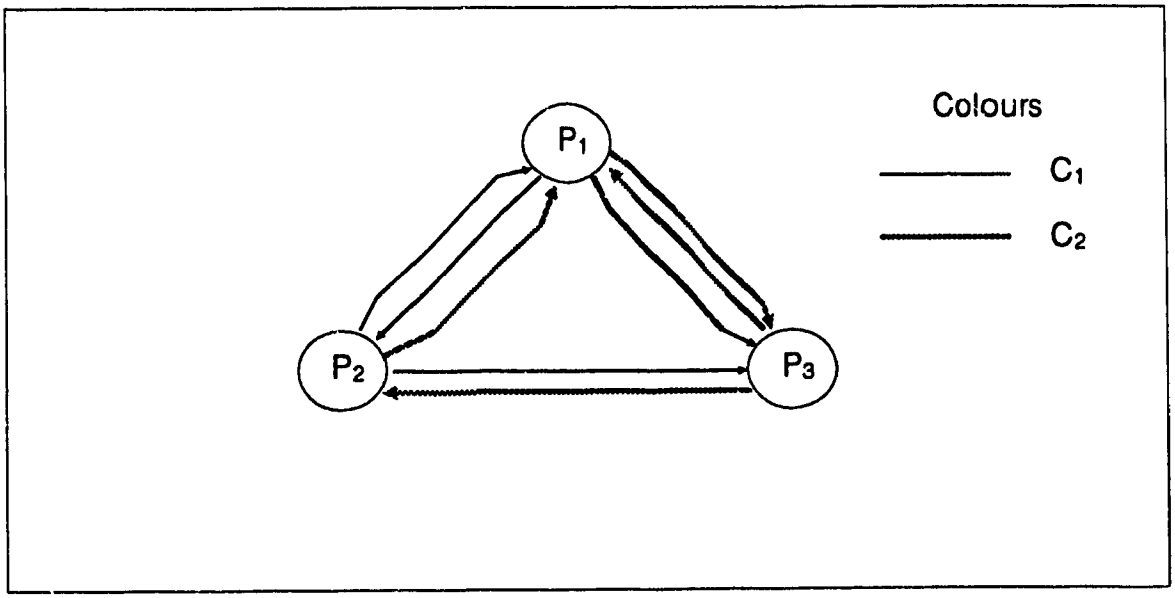
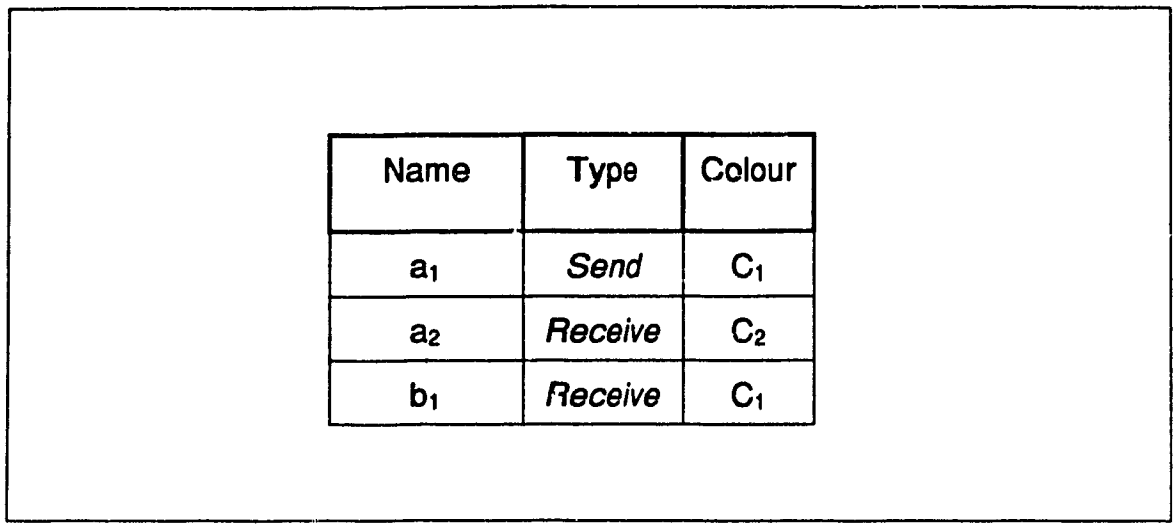


Figure 4.4 - Processes and channels corresponding to data in figure 4.3



Name	Type	Colour
a_1	*Send*	C_1
a_2	*Receive*	C_2
b_1	*Receive*	C_1

Figure 4.5 - Action Table

numbers can be assigned to each physical node by the user. The debugger will locate and load the executable codes of the process accordingly.

The process table and the action table are used to identify the processes and the actions, as well as the validity of the many operation related to them. Therefore, these tables are not changed during a debugging session. When modification is made to these tables, the program must be reloaded, which is equivalent to starting a new debugging session.

SBS Structure

The internal structure of the *SBS* is a cyclic directed graph, where a cyclic path represents a sequence of actions that repeatedly occur. This structure is best explained by an example: Consider an *SDL* statement for the *SBS*: *ab*c[(d%e)* || *(g%h)]*i*

where *a, b, c* ... are actions. The graph in figure 4.6 represents the *SBS* structure that is generated by the compiler for the above statement. Because *a,b,c* occur in a sequence, there is an edge going from node 1 to node 2, and another from node 2 to node 3. Since $b^* = \{ \varepsilon, b, bb, \ldots \}$, there is also an edge from node 1 to node 3, as well as an edge from node 2 to itself. Where there is more than one edge coming out from a node, it signifies the point where the common prefix of two or more pomsets has terminated. Thus the graph section containing nodes 1, 2 and 3 represents the pomsets: *ac, abc, abbc,* ...

The pomsets containing conjunction actions need special nodes, as illustrated in the graph section between nodes 4 and 11. Actions in different branches of this section can occur in any order. However, node 5 signifies a choice between the two actions *d* and *e*; a similar case is in node 8. Although the actions in this section can repeat indefinitely, as illustrated by the edge going from node 11 to node 4, the actions in all conjunction branches must be completed before the next sequence can take place. The use of node 11 ensures that requirement.

The tracker follows the execution by keeping a list of different pomset prefixes; the current computation has satisfied all these prefixes. Each pomset prefix has a distinct list of actions that can be its next event. These actions form the *next-event* set. For example, if *a* in figure 4.6 has occurred, the list will look as in figure 4.7a. If *c* is the action of the next event, the first entry is deleted from the list, the list is updated (figure 4.7b), and expanded (figure 4.7c) to reflect the new prefixes that correspond to the computation. The entry has sublists if a conjunction is encountered. Its *next-event* set is the union of the *next-event* sets of its sublists. Such an entry is allowed to go to the next node when all of its sublists have reached their corresponding end-points (e.g., node 11).

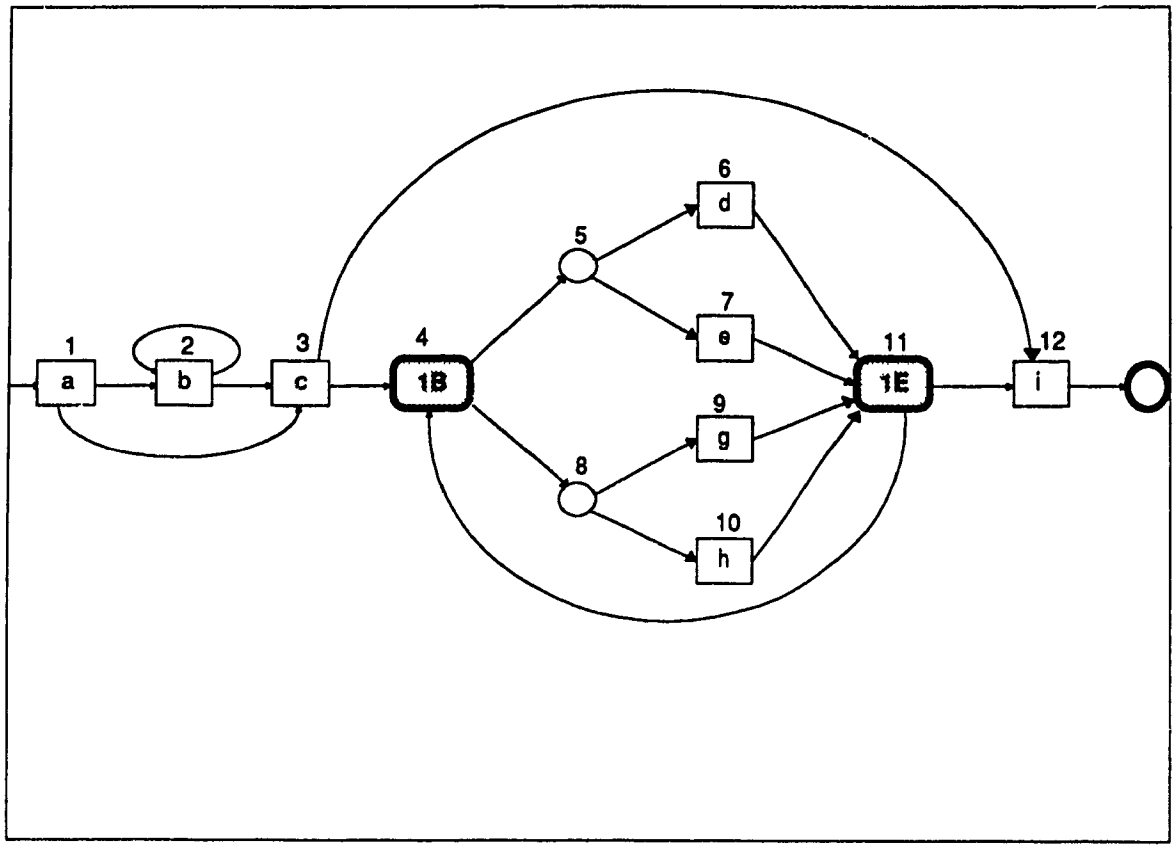


Figure 4.6 - *SBS* Structure

4.3 The Sequential Debugger

The *Process Controller* (4.1.2) needs the ability to start and terminate a process as well as to access the process's state space. We achieve this by employing a sequential debugger. The sequential debugger is able to load a process, run it and suspend it. It can also access any of the local variables of the process under debugging. With a source code debugger, these operations can be done with the source programming language.

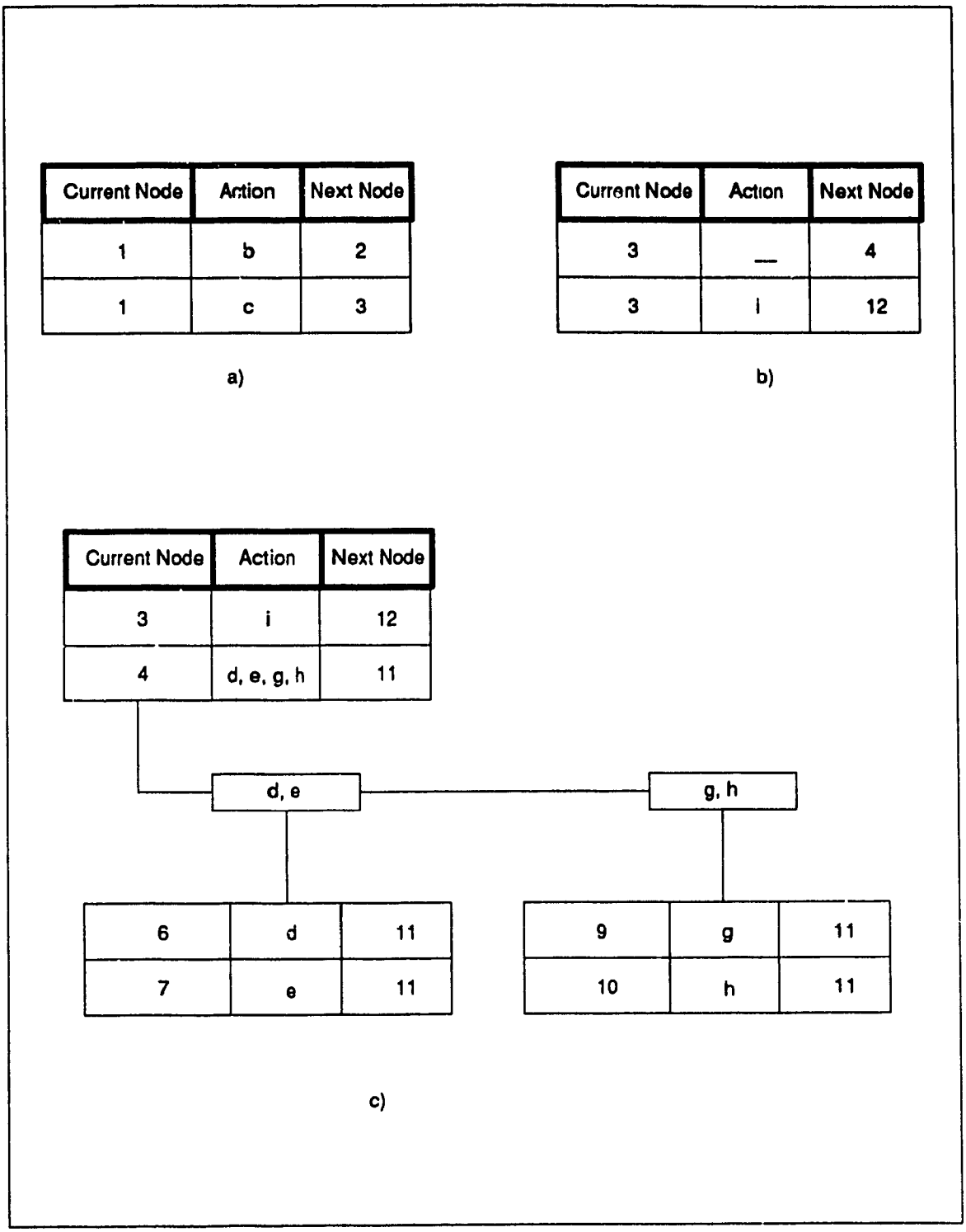


Figure 4.7 - Manipulation on *SBS* structure

The dynamic probes, which signal the occurrence of an event internal to a process, can be made using the watchpoint facility of the sequential debugger. This allows the user to exercise wide control over the process. When the process is suspended at a breakpoint, the user can also "step" through statements, or set up conventional breakpoints, as long as the effective computation does not involve any synchronization events.

In order to use the sequential debugger, we have created a three-way connection between the event processor, the sequential debugger and the application process, as in figure 4.8. While the behaviour of the process as it appears on the surface is monitored directly by the event processor, the sequential debugger will provide the necessary access inside the process.

4.4 Creating Checkpoints

To implement checkpoints in our debugging system, we take advantage of the *fork()* system call of UNIX [2]. When a checkpoint is to be created, all application processes are caused to spawn identical child processes. Meanwhile, the *LDs* also save all pertinent information needed for rolling-back, such as the action counters, the *Global Time* and in-transit messages. The child processes are kept dormant and their IDs are saved along with other relevant information. A checkpoint ID is given. When the program is rolled-back, the saved information is restored. All current application processes are killed, and the child processes are awaken from their sleep. Thus the complete program's state can be saved and fully restored.

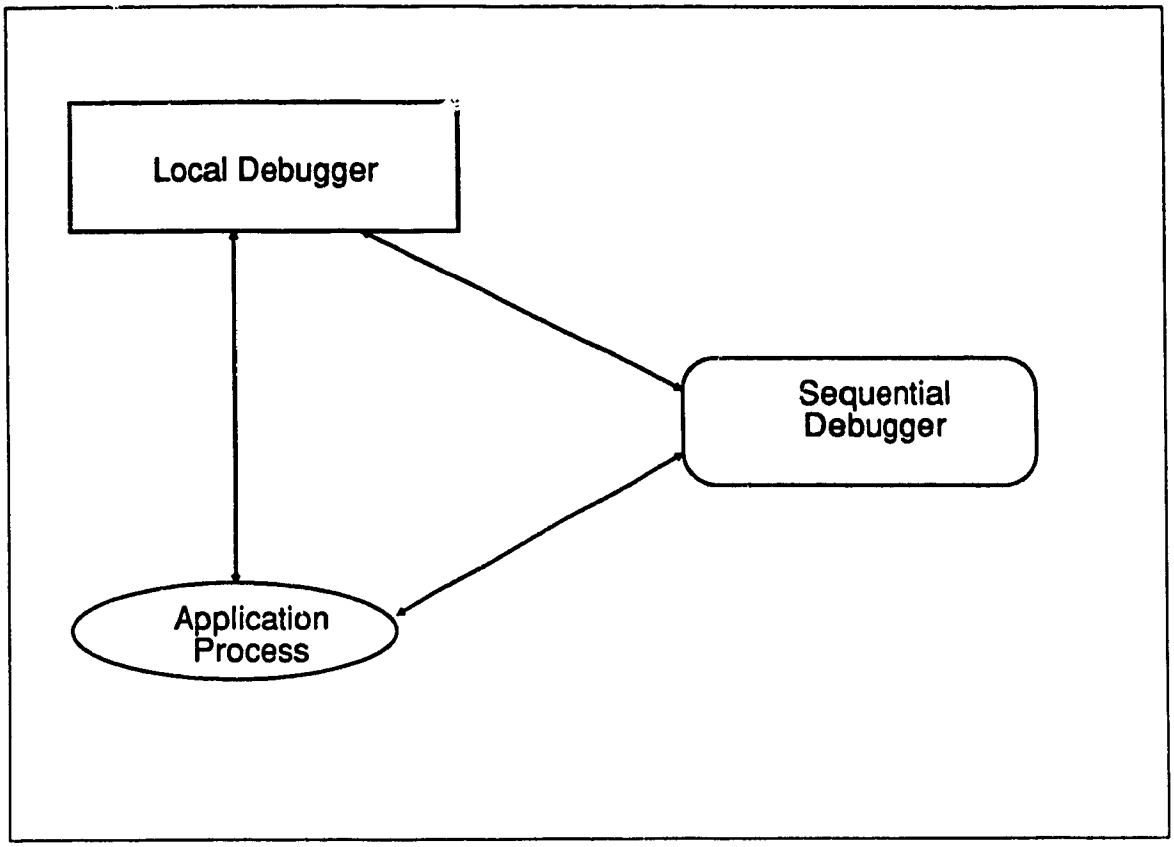


Figure 4.8 - Sequential Debugger in a Local Debugger

The drawback of this approach is that it is very expensive in terms of the space needed to save the whole program's code and data space. The compromise is in limiting the number of checkpoints the user can keep at any point. In our present implementation, only one checkpoint can be kept. Improvement on checkpointing based on this approach can be made if the debugger is implemented in the operating kernel, where it has access to the *page table* of the paging system. Multiple checkpoints can be made without having to save the whole process space each time; only pages of memory which are changed since the last checkpoint need to be saved [12].

4.5 Constraints and Limitations

With the *Global Clock* scheme, it is necessary that the debugger be able to append and extract the timestamp on the messages. However, implementation in the user space does not allow it to do so. Our solution to this problem is to build an additional layer between the application process and the communication subsystem. The transformer will reroute all calls to the communication subsystem to corresponding routines through this layer. The drawback of this approach is that in certain operating system calls, there are features that are disabled. We cannot collect information about them. However, this approach allows us to capture the whole information on an application message, which may be saved in the database and can be useful in playback. The use of an additional communication layer is also favourable in instant replay, where the occurrence of synchronization events must be under control. Furthermore, synthesized messages can be inserted into the channel through this layer when an artificial environment is created.

The additional layer is not necessary if the communication subsystem can be modified. Since it is installed in the kernel, it can be made to provide all the information and features described above at minimum overhead. However, we could not do it due to practical problems.

The structure of our testing facility also poses some potential problems. While all the SUN workstations have their own processing power, they all share the same mass storage. This creates a bottleneck in certain operations, especially for the database. The operating system makes all messages go through this mass storage device before being delivered to the destination. This in effect serializes all messages.

One major constraint is the lack of a sequential debugger that can be interfaced to our system. The sequential debugger *dbx* is the powerful source-code debugger on the UNIX system [1]. It offers almost every feature that a sequential debugger can have. However, the interface to this debugger is intended for a human user, and is not highly compatible to another software process. If this interface is modified such that *dbx* can exchange information with the event processor using the same data tokens, then we can exploit the full capability of this facility to our advantage.

There are many limitations in the current implementation of the distributed debugger. One of them is the lack of a well-designed specification language to specify the synchronization behaviour specification. Although we use the pomset model to describe distributed programs, our *SDL* is still based on regular grammar. Therefore we cannot use *SDL* to effectively describe some application programs.

In addition, our implementation does not contain a transformer module. The design calls for a preprocessor for source programs which can insert debugging codes in them. However, this has not been implemented due to time limitation. For testing purposes, the transformation process has been done by hand.

A database manager which can handle updates and retrievals on a large database is also needed. Currently, the traces are kept in a sequential file, without any indexing. Retrieval therefore is done through sequential search. A database system which can handle SQL queries is desirable.

4.6 Module Integration

The implementation has been done through the following stages:

Stage 1: The modules were built individually. After each module was built, its functionality was tested. We had to write test programs to provide the test environment and the input required for that module.

Stage 2: After the modules had been tested, we started putting them together. First, we tested the interfaces between the *CD* and the *LDs* for proper coordination. Then the *user interface* and the *event processor* were added. With a small test program, we tested the launch of an application in the debugging environment. Then breakpointing, deadlock detection and tracking and checkpointing modules were added. Testing was done at every step.

Stage 3: The whole system was put together. We could start running the debugger, using a small and simple distributed program for testing. At this stage, we used the debugger itself for debugging. For example, the traces were collected to determine how far the debugger had worked before it failed. Then breakpoints were set to suspend the application program, which in effect would also suspend the activity of the debugger. With a few user commands installed for the purposes of debugging, we examined the state of the debugger. This procedure was repeated until the module which had caused the problem was identified. The sequential debugger was then used to locate the bugs in those modules.

Chapter 5

Summary and suggestion for future work

The lack of a well-defined methodology in distributed debugging has prompted the study presented in this thesis. Although there are existing debugging facilities for distributed programs, no formal model is used to formalize the problem for which these facilities are intended. The notion of error is left to the user's discretion.

We believe that synchronization plays an important role in the correctness of a distributed program. Therefore, synchronization between processes in a distributed program is our main focus. To describe the characteristics of the program synchronization behaviour, we advocate the use of the pomset model. Based on this model, behaviour of the program with respect to activity at the interfaces of its component processes can be specified. The specification is given in terms of the order of occurrences of events at the interfaces. From this abstract model, we are able to define the synchronization specification of a process formally. A definition of synchronization error with respect to a given synchronization specification is also stated. With this definition of synchronization errors, we narrow the problems in distributed debugging to that of finding an effective method for locating and removing the error from the debugged program.

Based on our analysis, we propose a debugging strategy consisting of two stages. First, the synchronization error is located. Second, the programming bug which caused the error is located and subsequently removed. The first stage is a black-box analysis, where the behaviour of the component processes is observed and compared with a supplied synchronization specification. In the second stage, white-box analysis is carried out in which the internal states of component processes are scrutinized so that the programming bug can be found. Debugging facilities supporting these two stages can be used alternatively during a typical debugging session. In other words, an integrated debugging system which provides analysis at both black-box and white-box levels is called for.

With the debugging methodology as suggested, and with the guidelines for an effective distributed debugger given above, we propose a design for such a debugger. Chapter 3 is the detailed discussion of this design. Among the basic notions that are used in the design are those of a *Consistent Global State* in a distributed system, and a *Synchronization Behaviour Specification.* We also employ the labeling scheme called *Global Clock.* Timestamps with this logical clock value on events retain true partial order between the events. Using the timestamps, we are able to reconstruct the pomset corresponding to the actual behaviour of the program, which is needed for our analysis.

At the conceptual level of the design, the debugger consists of two parts: *Central Debugger* and *Local Debuggers*, each of which has different modules to handle different operations in the debugging system. The modules are designed so that they belong to at most one of the three dependencies: system, programming language and specification language. Thus, only a limited number of modules need to be changed when the debugger is used in a different environment.

We also present an abstract notion of the collection of information from the execution of the debugged program. This information is seen as being obtained by means of probes that are inserted into the program. The actual implementation of these probes is therefore not restricted to any particular method. The probes are the only connections between the debugger and the debugged program. Hence the influence of the debugger on the execution of the program can be analysed at these probing points. As the volume of information collected through the probes can be enormous, we suggest the use of a database for maintaining and manipulating this information.

As for the actual design, we provide a specification language (*SDL*) to describe the *SBS*. Using its set of production rules which is part of the language, *SDL* provides a means for specifying the set of pomsets representing all possible behaviours of the debugged program. However, the grammar employed by this language is similar to those of regular expression. Therefore, *SDL* renders a certain degree of ambiguity and complexity to other operations that must use the *SBS*, such as the tracker and the breakpoint controller.

Applying the design, we have implemented a prototype version of a distributed debugger. The operations that were actually implemented are: *SBS* compiler, tracking module, breakpoint controller, blocked and deadlock detector, checkpointing mechanism and the probes. However, the breakpoint module has not yet provided full support since a semantics for breakpoint specifications has not been finalized. Also, we have not provided support for dynamic probes. The database and related facilities are not available at the current stage of implementation.

As one of our design goals is to provide an integrated debugging system for black-box and white-box analysis, this objective has not been met. The program

behaviour is monitored and checked against the specification by the tracker, while the integration of the sequential debugger allows internal states of component processes to be accessible to the debugger. However, for tracing back from the location of synchronization error to the programming bug, the debugger offers limited support to the user. Without the database that manages the trace information, the user can only use the integrated sequential debugger to trace to the source of the synchronization error.

By limiting each module in our debugging system to at most one of the three defined dependencies, we have maximized the portability of our implementation. Minimum modification is needed in order to make the debugger workable in different environments. Furthermore, the areas where changes might be necessary are clearly defined based on these dependencies.

We believe that our strategy for distributed debugging is constructive and productive in terms of locating the cause of errors in distributed programs. The definition of program behaviour and synchronization error are based on a well-defined abstract model. Therefore analysis can be done conceptually without restriction to any particular system in application. However, our study, as a continuation of what has been done by [19], is still in a early stage. Further studies are needed, some of which are in the areas discussed below.

First of all, a pomset language with a well-structure semantics is much needed. The pomset language will generate pomsets representing the program's behaviour. It should be simple for the user to use, and also descriptive and unambiguous for the debugger to interpret.

Support for process composition is a desirable feature of a distributed debugger that is not currently supported in our design. A composite process can be created by grouping together several component processes. The behaviour of the composite process is described by the interactions of the component processes within the group with those outside. Interactions between processes of the same group are deemed to be invisible. When a composite process is created, the *SBS* corresponding to that new process should also be generated accordingly and automatically. One should focus, among other things, on the problems that may arise when processes in different nodes are to be combined.

Further studies on the application of the database are also required. The information that is maintained in the database is potentially very useful to the analysis of a debugged program. For example, *Instant Replay* allows the execution to be done in slow motion, and relies on trace information to preserve the original ordering of events. Thus the user can scrutinize the debugged program on-line, while still being able to maintain the same synchronization behaviour of the program as that of a recent execution.

In the scope of our study, we have excluded the errors that are violations to the liveness requirements of the program. To make our debugger support the handling this type of error, the pomset semantics have to be augmented with checkable liveness requirements. A utility which performs the verification of the observation against these requirements must also be added to the debugger. The findings on this particular topic can enable the expansion of the debugger to make it applicable to real-time systems.

Finally, there have been few extensive study on probe-effect [14, 15]. We have known that one of the sources of the probe-effect is delays in communication caused by the debugger. However, we do not know about other possible sources of the probe-effect. Until there is a clear understanding of the characteristics of probe-effects, it is not possible to decide on compensation methods either to prevent or to reduce this undesirable effect in debugging systems.

Distributed debugging is still in the early stages of development. Far more studies have to be done before one can expect to have a debugger for use with distributed programs that is as effective as those being used on sequential programs. Nevertheless, we believe that this thesis has provided a structured approach to the problem of distributed debugging. Further enhancement can be made to provide a complete debugging environment that actually meets the needs of those who design and implement distributed programs.

REFERENCES

1. "DBX(1)," in *Commands Reference Manual*, Part No. 800-1295-04, Sun Microsystems, Inc., Mountain View, CA, September 1986.

2. "FORK(2)," in *UNIX Interface Reference Manual*, Part No. 800-1303-04, Sun Microsystems, Inc., Mountain View, CA. September 1986.

3. Alford, M.W., J.P Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery, and F.B. Schneider, "Distributed Systems -- Methods and tools for specification – An advanced course," Lecture notes in Computer Science 190, Springer-Verlag, 1985.

4. Baiardi, Fabrizio, Nicoletta De Francesco, and Gigliola Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Trans. on Software Eng.*, vol. SE-12, no. 4, pp. 547-553, April 1986.

5. Bates, P.C., "Debugging Programs in a Distributed System Environment," Ph.D. Thesis, University of Massachusetts, Massachusetts, February 1986.

6. Bei, James N.W. and Eric G. Manning, "CGDS: A Graphical Debugger for a Distributed System," in *Proc. Congress 85, Canadian Information Processing Society*, pp. 166-175, Montréal, Québec, June 1985.

7. Cameron, E. Jane, "The IC System for Debugging Parallel Programs via Interactive Monitoring and Control," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 261-270, University of Wisconsin, May 1988.

8. Cheng, Wan-Hong S. and Virgil E. Wallentine, "DEBL: A Knowledge-Based Language for Specifying and Debugging Distributed Programs," *Communications of the ACM*, vol. 32, no. 9, pp. 1079-1084, September 1989.

9. Curtis, R. and L. Wittie, "BugNet: A Debugging System for Parallel Programming Environments," in *Proc. 3th International Conference on Distributed Computing Systems*, pp. 394-399, Fort Lauderdale, Fla, October 1982.

10. Darlington, J.L., "The Role of Verification and Testing in Software Development," *Microcomputers: Developments in Industry, Business and Education*, pp. 81-89, Euromicro, North-Holland, 1983.

11. Dijkstra, E.W., "Notes on structured programming," in Dahl, O.J., E.W. Dijkstra and C.A.R. Hoare, *Structured Programming*, Academic Press, New York, 1976.

12. Feldman, Stuart I. and Channing B. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 112-123, University of Wisconsin, May 1988.

13. Fidge, C.J., "Partial Orders for Parallel Debugging," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel & Distributed Debugging*, pp. 183-194, University of Wisconsin, May 1988.

14. Gait, Jason, "A Debugger for Concurrent Programs," *Software – Practice and Experience*, vol. 15, no. 6, pp. 539-554, June 1985.

15. Gait, Jason, "A Probe Effect in Concurrent Programs," *Software – Practice and Experience*, vol. 16, no. 3, pp. 225-233, March 1986.

16. Garcia-Molina, Hector, Frank Germano, and Walter H. Kohler, "Debugging a Distributed Computing System," *IEEE Trans. Software Eng.*, vol. SE-10, no. 2, pp. 210-219, March 1984.

17. Goldszmidt, German S., Shmuel Katz, and Shaula Yemini, "Interactive Blackbox Debugging for Concurrent Languages," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 271-282, University of Wisconsin, May 1988.

18. Gordon, Aaron J. and Raphael A. Finkel, "Handling Timing Errors in Distributed Programs," *IEEE Trans. Software Eng.*, vol. SE-14, no. 10, pp. 1525-1535, October 1988.

19. Hamamtzoglou, Ioakim, "A Model and A Methodology for Distributed Debugging," Master's Thesis, Department of Computer Science, Concordia University, Montréal, Québec, July 1986.

20. Hough, Alfred A. and Janice E. Cuny, "Initial Experiences with a Pattern-Oriented Parallel Debugger," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 195-205, University of Wisconsin, May 1988.

21. Hseush, Wenwey and Gail E. Kaiser, "Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 236-247, University of Wisconsin, May 1988.

22. Joyce, J., G. Lomow, K. Slind, and B. Unger, "Monitoring distributed systems," *ACM Trans. Computer Systems*, vol. 5, no. 2, pp. 121-150, May 1987.

23. Lamport, Leslie, "A Simple Approach to Specifying Concurrent Systems," DEC System Research Center Report 15, December 1986.

24. Lamport, Leslie, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

25. Lea, Christopher, from discussions.

26. LeBlanc, Thomas J. and John M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Computer*, vol. C-36, no. 4, pp. 471-481, April 1987.

27. Li, H.F. and D. Livas, "Spontaneous Global State Detection Using Global Time," in *Proc. of the 1989 International Symposium on Computer Architecture & Digital Signal Processing*, pp. 444-449, Hong Kong, October, 1989.

28. Miller, Barton P. and Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 141-150, University of Wisconsin, May 1988.

29. Pratt, Vaughan R., "Modeling Concurrency with Partial Orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33-71, February 1986.

30. Schiffenbauer, R.D., "Interactive debugging in a Distributed Computational Environment," Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1981.

31. Heping, Shang, "Consistent Global State: Algorithms and an Application in Distributed Garbage Collection," Master's Thesis, Department of Computer Science, Concordia University, Montréal, Québec, August 1988.

32. Tassel, D. Van, *Program Style, Design, Efficiency, Debugging and Testing*, Prenctice-Hall Inc., Englewood Cliffs, New Jersey, 1974.

33. Venkatesh, Krishnarao, "Global States of Distributed Systems: Classification and Applications," Ph.D. Thesis, Department of Computer Science, Concordia University, Montréal, Québec, January 1988.

34. Wittie, Larry D., "Debugging Distributed C Programs by Real-Time Replay," in *Proc. ACM SIGPLAN & SIGOPS Workshop on Parallel and Distributed Debugging*, pp. 57-67, University of Wisconsin, May 1988.

Appendix A

A BNF Definition for *SDL*

<prog_sbs> ::= <prog_def><sbs_def>

<prog_def> ::= **process** <procdef>
color <coldef>
channel <chnldef>
event <eventdef>

<procdef> ::= <procdesc>; | <procdesc>, <procdef>

<procdesc> ::= <pid>:<node_id>

<coldef> ::= <col_id>; | <col_id>, <coldef>

<chnldef> ::= (<src_pid>, <dst_pid>, <colspec>); |
(<src_pid>, <dst_pid>, <colspec>), <chnldef>

<colspec> ::= <col_id> | <col_id>, <colspec>

<eventdef> ::= <eid> (<type>, <col_id>); |
<eid> (<type>, <col_id>), <eventdef>

<sbs_def> ::= <pid> { <statements> } |
<pid> { <statements> } <sbs_def>

<statements> ::= <onestatement>. | <onestatement>. <statements>

<onestatement> ::= <expr_id> := <choice_expr>

<choice_expr> ::= <seq_expr> | <seq_expr>%<choice_expr>

<seq_expr> ::= <conj_expr> | <conj_expr>;<seq_expr>

<conj_expr> ::= <star_expr> | <star_expr>|| <conj_expr>

<star_expr> ::= <comp_expr> | <comp_expr>*

<comp_expr> ::= <expr_id> | <eid>(<pid>) | [<choice_expr>]

<pid> ::= <id>

<node_id> ::= <id>

<col_id> ::= <id>

<src_pid> ::= <id>

<dst_pid> ::= <id>

<eid> ::= <id>

<type> ::= **send** | **receive**

<expr_id> ::= <id>

<id> ::= <letter><rest_of_id>

<rest_of_id> ::= <letter><rest_of_id> | <digit><rest_of_id>

<letter> ::= A | B | C | D | E | F | G | H | I |
J | K | L | M | N | O | P | Q | R |

S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i |
j | k | l | m | n | o | p | q | r |
s | t | u | v | w | x | y | z | _

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

Example A.1 Consider a distributed program consisting of four processes: P_1, P_2, P_3 and P_4. The informal specified operation of the program is:

> After receiving a message from P_1, depending on the contents of that message and on its own computation, P_2 will send a message to either P_3 or P_4. However, P_2 should never send two messages in a row to P_3 without interleaving them with one to P_4. Also, messages between P_1 and P_2 are of different types than those between P_2 and P_3 or P_4.

Assume that the program is distributed over a *DCS* having three nodes: *A*, *B* and *C*. Processes P_1 and P_2 are located at node *A* while P_3 is located at *B* and P_4 at *C*. The *SBS* for the above program can be given as follows:

```
/* Program definition part */
process  P1:A,  P2:A,  P3:B,  P4:C;
colour   C1,  C2;
/* C1 is the colour assigned to messages
between P1 and P2, C2 is assigned to
those between P2 and P3 or P4 */
```

```
channel    (P1,  P2,  C1),  (P2,  P3,  C2),
           (P2,  P4,  C2);

event      snd1(send,  C1),  rcv1(receive,  C1),
           snd2(send,  C2),  rcv2(receive,  C2);


/*  Process  SBS  definition  part  */

P1  {
           S1  :=  snd1(P2)*.
}

P2  {
           S1  :=  rcv1(P1);  snd2(P3).
           S2  :=  rcv1(P1);  snd2(P4).
           S3  :=  [  S1;  S2;  [S2]*  ]*.
}

P3  {
           S1  :=  rcv2(P2)*.
}

P4  {
           S1  :=  rcv2(P2)*.
}
```

Note: for a process *SBS*, as many statements as necessary may be given. However, only the last statement is effective: it is used as the *SBS* for that process. If the last statement contains the names of other statements, these names are replaced by the corresponding specification. For example, S_3 in P_2 will be expanded into:

```
S3  :=    [    rcv1(P1);  snd2(P2);
               rcv1(P1);  snd2(P4);
               [  rcv1(P1);  snd2(P4)   ]*
          ]*.
```

Appendix B

Distributed Debugger - User's Manual

B.1 Starting the Debugger

The debugger is started by entering the following shell command at the site where the *CD* is located:

<debugger name> node-list

<debugger name> has not been finalized at the moment. *node-list* is the list of Internet names of machines where the *LDs* are located. The *LDs* are assigned numbers 1, 2, 3, ..., corresponding to their order in *node-list*. The prompt '$' is displayed when the debugger is ready to accept user commands.

B.2 User Commands

Name: bp_clear, bp_disable, bp_enable

Usage:

bp_clear [breakpoint names...]

bp_disable [breakpoint names...]

bp_enable [breakpoint names...]

removes (clears), disables or enables breakpoints whose names are given.

Name: bp_list

Usage: bp_list -cds [breakpoint names...]

lists breakpoints; if breakpoint names are given then lists only those breakpoints.

-c: gives the list of counters in the breakpoint definitions and their specified values

-d: gives the full definition of the breakpoint(s)

-s: gives the status of the flags defined for the breakpoint(s).

The default is -s.

Name: bp_set

Usage: bp_set -cs [-f file | definition]

sets a breakpoint based on channel counter values. The actions to be taken when the breakpoint is reached are specified using the flags:

-c: make a checkpoint (allowing the program to be able to roll-back later; will automatically set -s flag.

-s: suspend the program

If [-f file] is specified, the breakpoint definition is read from *file*; otherwise it is given in *definition.*

A breakpoint definition has the following format:

breakpoint-name := breakpoint-condition

breakpoint-name is an identifier assigned to the defined breakpoint. *breakpoint-condition* is a predicate of channel counter values:

counter-value [[|][&] counter-value ...]

where | is logical OR and & is logical AND. Each *counter-value* has a format of:

[local-pid, remote-pid, event-name, value],

where *event-name* is the name of the action whose occurrences are counted for the breakpoint: the *counter-value* is reached when the number of occurrences of the action is greater than or equal to *value*. *local-pid* is the process-id of the process where the action occurs. *remote-pid* is the process-id of the process at the other end of the channel.

Example B.1 A typical breakpoint definition would be:

```
bp1 :=   [p1, p2, snd1, 3] & [p2, p3, snd2, 6] &
         ([p3, p2, rcv2, 6] | [p4, p2, rcv2, 10])
```

Note that the operator & has higher precedence than |, and that brackets can be used to override the precedence.

Name: dbx

Usage: dbx pid dbx-command

executes a *dbx* command on process *pid.* This is similar to executing *dbx-command* under *dbx* when the process is debugged alone using *dbx.*

Name: getsbs

Usage: getsbs sbs-file

reads the program's *SBS* from *sbs-file.* This command should be executed at the beginning of every debugging session, and it can be used only once per debugging session.

Name: help

Usage: help

gives a list of user commands with a brief description of every command.

Name: prsbs

Usage: prsbs [-d] [-[s|w] [process-ids ...]]

prints the *SBS.*

-d: prints the definition part only

-s: prints the basis *SBS* (default)

-w: prints the working *SBS*

If no *process-id* is given, the *SBS* for all processes will be printed. Otherwise, only the *SBS* of the indicated process is printed.

Name: quit

Usage: quit

ends a debugging session.

Name: roll_back

Usage: roll_back

roll back to a previously created checkpoint (see *bp_set*). The program would be in the suspended state, and the *resume* command can be used to resume execution.

Name: set, reset filter

Usage: set filter colours...

reset filter colours...

removes or sets colour filters.

Name: set, reset help

Usage: set help

reset help

similar to the *help* command, prints the list of all set and reset commands.

Name: set procdefn

Usage: set procdefn process-id exec-filename [args]

specifies how a process can be executed. *exec-filename* is the name of the executable file which contains the code for the process *process-id. args* are command line arguments needed for starting up the process.

Name: set trace

Usage: set trace [on | off]

turns trace collection on or off.

Name: sh

Usage: sh [shell-command]

executes *shell-command.* If no command is given, the UNIX shell whose name is in the SHELL environment variable is invoked. If SHELL is not defined, then */bin/csh* is used by default.

Name: start, resume

Usage: start

resume

starts the execution of the debugged program from the beginning or resumes the execution after it has been suspended (either by reaching a breakpoint or by a user command).

Name: suspend

Usage: suspend

suspends the execution. The exact point where each process is actually suspended is undetermined.

B.3 A Debugging Session

For the distributed program whose *SBS* is given in example A.1, following are some typical steps of a debugging session:

1. The code for every process is recompiled and then linked with the debugging-support library so that the program can be debugged. This is to be done by the *transformer.*

2. The debugger is started.

3. The program's *SBS* is given to the debugger. If the program's *SBS* is stored in the file *prog.sbs*, then the user command will be:

   getsbs prog.sbs

4. The name of the executable file for each process, along with the required command line arguments, is specified with the *set procdefn* command. For example:

   set procdefn P1 App1 arg1 arg2
   set procdefn P2 App2 arg1
   ...

   where *App1* and *App2* are the names of the executable file for processes P_1 and P_2 respectively.

5. The program is executed by *start*. If a synchronization error is suspected, the execution is suspended and the user is informed by a message similar to:

```
Synchronization error suspected in p2:
[rcv1(P1); snd2(P3); rcv1(P1); <snd2(P4)>;
[rcv1(P1); snd2(P4)]*]*
```

Since *snd2(P4)* is enclosed in angle brackets, it means that *snd2(P4)* is the only event that is expected to occur in P_2 but another event has occurred instead.

6. Breakpoints can be set and the program can be restarted using the *start* command. For example, if the following breakpoint command is entered:

bp_set -s bp1 := [P1, P2, snd1, 1] & [P2, P1, rcv1, 1]
& [P2, P3, snd2, 1] & [P3, P2, rcv2, 1]

the program would be suspended after one message has been sent from P_1 and received by P_2 and another has been sent from *P2* and received by P_3. When the program is suspended after a breakpoint is reached, internal variables of processes can be examined using the *dbx* command.

7. A checkpoint can be specified given the above program behaviour (one *send* from P_1 to P_2 and from P_2 to P_3 and the *receives* corresponding to those *sends*). Using the following *bp_set* command:

bp_set -c bp1 := [P1, P2, snd1, 1] & [P2, P1, rcv1, 1] &
[P2, P3, snd2, 1] & [P3, P2, rcv2, 1]

then when the program is suspended because the breakpoint *bp1* is reached, the program state is saved. Later, if the *roll_back* command can be used to restart the execution of the program from this checkpoint.