Monitoring Distributed Systems

by

Honna Segel

A Thesis in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April, 1993

ISBN 0-315-84651-8

Abstract

Monitoring Distributed Systems

Honna Segel

In debugging distributed programs a distinction is made between an observed error and the program fault, or bug, that caused the error. Testing reveals an error; debugging is the process of tracing the error through time and space to the bug that caused it.

A program is considered to be in error when some state of computation violates a safety requirement of the program. Expressing safety requirements in such a way that a computation can be monitored for safe behavior is thus a basic preliminary step in the testing-debugging cycle. Safety requirements are usually expressed as predicates. When a state of the computation violates such a safety predicate, that state can be said to be in error.

A predicate logic is proposed that permits the specification of relationships between distributed predicates. This increases the scope and precision of situation-specific conditions that can be specified and detected. It also permits the specification of safety primitives such as *P unless Q* using distributed predicates. Thus a distributed program can be directly monitored for satisfaction and violation of safety requirements.

Breakpoint conditions and predicates expressing safety may hold over a number of states of a program. A breakpoint state is meaningful if the causal relationships of events included in the breakpoint are unambiguous. At least two such states exist for each condition: the minimal and the maximal prefix of the computation at which the predicate holds. These states are specifiable as part of a breakpoint definition in the logic presented.

This work is dedicated to my parents, Stan and Ellie, and to those who went before and made it possible for the likes of me to follow.

Acknowledgments

I thank my advisor, Dr. Li, for his support, both academic and financial, and especially for his commitment. My thanks to Dr. Radhakrishnan, whose faith in me got me started on this work.

I thank my family: my parents, Stan and Ellie, my sister and brother Lisa and Jon. Their faith in me helped keep me going. I also thank my extended family: Judy Brown, who understood because she's been there; Sandra Henriksen, Andre Czernohorsky, Wilson Coneybeare, and Michael Thibault who also proof-read.

I thank my friends and colleagues for their discussions, advice, support, and sometimes commiseration: Adam Steele, Dimitri Livas, Marija Cubric and Dimitri Kourkopolous.

I thank my friends Emma Hancock, Claire Sergeant, Marco Rigotti, and Daniel Levesque, too, for their encouragement and good company.

My thanks to Peter Kaldis for his friendship and for the technical support that made it so much easier and pleasant to finish.

And especially my thanks to Jean Marjama, who somehow made it imaginable.

Table of Contents

List of Figures

Chapter 1 : Introduction

1.0 Problem Context

In debugging distributed programs a distinction can be made between an observed error and the program fault, or bug, that caused the error. The two are often widely separated; in sequential processes, they may be separated by time (steps of the computation as marked by a processor clock), and in distributed programs by both space (one processor to another) and time. Testing then reveals an error; debugging is the process of tracing the error through time and space to the bug that caused it.

Before the process of debugging a program can commence, it is necessary to discover an error. A program is considered to be in error when some state of computation violates a safety requirement of the program. Expressing safety requirements in such a way that a computation can be monitored for safe behavior is thus a basic preliminary step in the testing-debugging cycle. Safety requirements are usually expressed as predicates; these predicates define safe states and safe state transitions of programs. When a state of the computation violates such a safety predicate, that state can be said to be in error.

When a safety requirement is violated during a program execution, the user desires to examine the state of the program at which the violation occurred. This state will be called a breakpoint. The user may also wish to halt the computation at various breakpoints satisfying user-defined criteria.

Testing and debugging as described above are quite well understood in the area of sequential programs. Distributed programs present additional factors which complicate the task of testing and debugging.

The class of distributed systems considered here are those that are asynchronous and communicate via messages with finite, but unpredictable, delay. In this type of system, processes operate according to local clocks; a global version of time is not available. Synchronization occurs through message passing rather than through the use of shared memory.

An instantaneous global state of the system, in which all processes are halted at some real-time instant, is not possible in the absence of global clock and shared memory. However, it can be approximated using distributed snapshot algorithms [Chan85], which yield a global state that could have occurred. Algorithms implementing a virtual time mechanism [Mat89], permit each processor to construct a consistent view of system activity.

In sequential systems, a predicate describing a safety or breakpoint condition holds on a state or states of a single processor. States in sequential systems are totally ordered, so the interpretation of *predicate A holds before predicate B* is straightforward. However, in distributed systems predicates expressing safety or breakpoint conditions hold on global states of the system. As described above, global states are not immediately given and must be extracted. While events in distributed systems can be partially ordered using Lamport's happens-before relation [Lam78], global states cannot be partially ordered in this sense. Because of this lack of ordering, the interpretation of *predicate A holds before predicate B* cannot be interpreted as straightforwardly as in sequential systems.

In related work, specification and detection of a predicate on the states of multiple processors, such as *variable x at process A equals 2 AND variable y at process B equals 3* is fairly well-defined. One of the goals of this work is to propose a model of predicate behavior in distributed systems that gives a clear interpretation of the relationships *between* distributed predicates. A relationship between distributed predicates could be, for example, *predicate A holds before predicate B* where both *A* and *B* are distributed. Establishing relationships between distributed predicates permits the specification and detection of much more sophisticated conditions for the debugging of distributed programs.

Establishing relationships between distributed predicates will also serve to increase the expressiveness of a specification language so that safety requirements can be clearly stated. Fundamental operators expressing safety such as *P is invariant* or *henceforth P* can be expressed using distributed predicates on system state. Related work in testing and debugging has not clearly linked work in safety specification to specifications for the purposes of monitoring. Because this link has not been clearly established, monitoring for the purposes of testing has been limited to *ad hoc* specifications of safety properties. Another goal of this work is thus to achieve a model that permits direct specification of fundamental safety

operators; an operator is considered fundamental if some safety property cannot be expressed without it. Direct specification of safety properties permits monitoring of programs for satisfaction or violation of safety requirements, facilitating the detection of error in distributed program testing.

2.0 Organization of the thesis

In Chapter 2 an introduction is given to the problems associated with detecting conditions in a program. Related work is reviewed in light of the goals of this work. In Chapter 3 a naive logic for specification of distributed predicates is proposed. The problem of reachability between global states is examined and a logic to express relevant notions is proposed. The behavior and structure of distributed predicates, including unstable distributed predicates, is examined in Chapter 4 and the logic is extended to describe this structure. In Chapter 5 the relationships between distributed predicates are examined and the logic is extended to describe these relationships. Safety operators that can now be expressed using the extended specification logic are presented. In Chapter 6 an extended model is presented for specification of distributed predicates and their relationships, incorporating the increments in Chapters 3 through 5. A set of examples is given in Chapter 7 to demonstrate the application of the extended logic. In Chapter 8, the problem of detection is treated and algorithms for detection are given. Conclusions and suggestions for further work are presented in Chapter 9.

Chapter 2 : Background and Related Work

2.0 Introduction

The majority of the works reviewed in this chapter focus on the related problems of detecting distributed predicates and halting at distributed breakpoints for the purposes of debugging distributed programs. The different works have varying scope and efficiency, but conform to the following approach: firstly, a logic is proposed for the specification of distributed predicates or breakpoints; secondly, a detection algorithm is implemented; and finally, detection triggers a halting algorithm. These three subjects are introduced below. Two other related works on safety specification and non-atomicity are also introduced.

2.0.1 Logics for specification

The primitive for specification of predicates and breakpoints is the assertion on the local state of a process; for example $x = 3$, *process 1 has entered its critical section*, or *event y has occurred*. Predicates distributed across nodes, called distributed predicates, are built up using boolean and comparative operators between such primitives. The interpretation given to distributed predicates is that the predicate is considered to hold if local states of the processes on which the primitive elements of the predicate are defined can be composed using boolean operators to form a consistent global state of the system.

Some preliminary definitions follow, which are general enough to apply to most of the work covered in this chapter**[Fow90, Hab88, Mil88, Spez89, Wal91]**:

- Simple Predicates are predicates defined on the variables of a single process. For example, *x=true* is a simple predicate, where x is owned by a single process.
- Conjunctive/Disjunctive predicates are formed by 'or-ing' and/or 'and-ing' Simple Predicates.
- Linked Simple Predicates are formed by establishing some type of 'happens-before' or sequential ordering between the occurrences of Simple Predicates.
- Linked Conjunctive/Disjunctive Predicates are formed by establishing some type of sequence between the occurrences of Conjunctive/Disjunctive Predicates.

For simplicity, the above names will be used in most cases rather than switching vocabulary when describing the work of each author.

2.0.2 Detection

Work on detection has taken two approaches. In [Fow90, Hab88,Mil88], the condition to be detected was characterized as an atomic event. In [Spez89, Wal91] it was recognized that breakpoint condition may hold over a subset of the state space, and that this fact can be used to increase the efficiency and the precision of detection. For example, it can easily be determined if two local events have a causal relationship or if they are incomparable (neither happens before the other). For example, in Figure 1 event e1 precedes event e2. By contrast, let us assume that some algorithm is available to detect a breakpoint predicate defined as *variable x at process 1 is true AND variable y at process 2 is true*. Though e1 and e2 are ordered, there exist global states in which both hold. There are then possibly many global states at which this predicate holds.

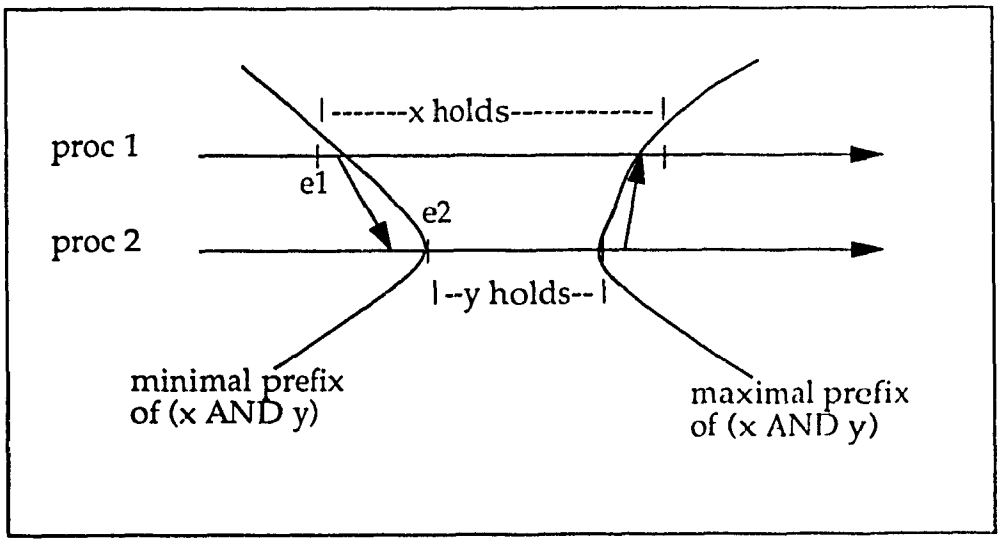

FIGURE 1: events and predicates as breakpoints

Two global states over which a predicate holds are well-defined in terms of the causal relationships in the computation. For example, the global state corresponding to the *minimal prefix* of the computation at which the predicate holds is analogous to a sequential breakpoint; everything that has occurred has a (possibly) causal effect on the breakpoint events. Similarly, the *maximal prefix* is the latest state at which the predicate holds, showing explicitly the events which

depend on the breakpoint events. The minimal and maximal prefixs of (*x AND y*) are shown in Figure 1.

2.0.3 Halting

The distributed snapshot algorithm [Chan85] yields an integrated view of system activity. For example the prefixs marked in Figure 1 are consistent views of this simple system, as no processes have received messages that have not yet been sent. The snapshot algorithm assumes that messages are well-ordered. It operates as follows: A coloring scheme is used to delineate the 'instant' of the snapshot - if an event occurs before the snapshot, it is defined as white, if after, it is defined as red. A process initiating a snapshot turns red, saves its current local state, and sends a warning message on every outgoing channel. On receipt of a warning message, a process stays red if it is already red; if it is white it turns red, records its current local state and sends a warning message on every outgoing channel. When all nodes in the system are red, the algorithm is complete. The local states that have been recorded are consistent (no messages have been received before they are sent because of the assumption of well-ordered message delivery) and form a consistent view of the distributed system state.

If a predicate is locally decidable - the value of the expression can be decided on the basis of the state of a single process - then the snapshot algorithm can capture a global state in which the predicate holds. The deciding process initiates the algorithm immediately after detecting the satisfaction of the predicate and before sending any application messages. However, if the predicate is not locally decidable, as in Figure 1, then distributed snapshot is not a good means for detecting and halting. If a process initiates a snapshot each time its local element holds - for example, at event e1 in Figure 1, then there can be no guarantee that the global state obtained by the snapshot algorithm will be a state at which the predicate holds, if, indeed, the predicate ever held.

Most of the works reviewed [Spez89, Mil88, Hab88] use some optimization of Chandy and Lamport's distributed snapshot algorithm to halt a distributed system in a consistent state once a locally decidable breakpoint condition or predicate has been detected.

2.0.4 Goals of this thesis

The specification and detection of breakpoints for use in debugging distributed programs is in itself a worthwhile goal; breakpoints are standard tools for debugging in sequential systems and should be available for use in distributed systems.

However, the goal of this thesis is motivated by the idea that before debugging can commence, some error must be detected. Error is defined with respect to some safety and progress requirements of a program. Since progress can in general only be detected at infinity, attention is restricted to safety requirements. The area of work that follows from this problem is that of explicitly connecting safety requirements with distributed predicates so that a computation can be monitored for the satisfaction of such predicates. **[Wal91]** addresses this question by using distributed predicates to express intuitive formulations of program requirements. **[Chan88]**, and others, present models to specify programs, and hence program safety, for concurrent systems. An operator fundamental to expressing safety properties is *P unless Q*, i.e., if some predicate P holds, it continues to hold unless some predicate Q holds. In the next chapters an interpretation of Chandy and Misra's model is given in terms of distributed predicates, permitting a computation to be explicitly monitored for satisfaction of safety requirements.

Stable properties of programs, such as termination, are relatively easy to detect using some variation on the distributed snapshot algorithm **[Hela90, Mis83]**. They can be detected by checking periodically to see if they have become true. Since the property is stable, it can be guaranteed to hold despite any gap between detection and halting. Unstable predicates however, are less straightforward to detect because of their transient nature. Lamport presents a model of non-atomic actions in concurrent systems in **[Lam86a, Lam86b]**; the concepts presented have strongly influenced the model developed in this thesis. The model developed here can be seen as an interpretation of Lamport's non-atomic actions for distributed systems, treating unstable predicates as non-atomic events. While fulfilling the goal of expressing safety properties using distributed predicates, this interpretation also provides a basis for modelling non-atomic behavior in distributed systems.

2.1 Miller and Choi, Haban and Weigel

Miller and Choi presented early work in this area, and so have had an influence on all the work presented here. In [Mil88] the goal was to provide the user with a breakpoint comparable to that obtainable in sequential systems - in other words, to halt at a state at which the predicate holds. Halting at a state in which the detected predicate holds is possible when the predicate is a Simple or Disjunctive Predicate, as the satisfaction of a predicate can be decided at a single process. As described in Section 2.0.3 , when a single process can determine if a predicate holds, a halting algorithm can be initiated immediately, guaranteeing a consistent global state in which the predicate holds. [Mil88] defines Simple and Disjunctive Predicates and Linked Simple Predicates as given in Section 2.0.1 . Algorithms based on the distributed snapshot algorithm are provided for halting in a state at which the predicate holds. Conjunctive predicates are defined, but no algorithms for their detection are provided.

The goal in [Hab88] was to define distributed predicates, detect and halt. The same types of predicates as are defined by [Mil88] are defined here, albeit in somewhat more detail. An interactive graphical editor is provided to aid in the construction of well-formed predicate expressions. For detection, the predicate is broken down over a binary tree, where the leaves of the tree are Simple Predicates and the root the value of the whole predicate expression. Detection algorithms are provided, as in [Mil88], as part of a complete debugger architecture. For non-locally decidable predicates such as conjunctive predicates, a halting algorithm is provided, but the authors recognize that yielding a state in which the predicate holds it cannot be guaranteed. To compensate for this, a local trace facility is provided that records the local states in which Simple Predicates held.

While providing solid basic work in debugging, [Hab88] does not attempt to address questions of monitoring for safety properties, nor is the question of halting at specific minimal or maximal states addressed.

2.2 Waldecker: detecting unstable predicates

Waldecker's goal was to provide a taxonomy of distributed predicates that would serve both to categorize and to construct distributed predicates, and to provide accurate detection methods that are reasonable in complexity. His specific focus

was unstable predicates, a class of predicates that has not been treated as thoroughly as stable predicates.

The taxonomy provided by Waldecker is based in an interleaving model of concurrency. The logic presented is quite comprehensive, but he provides detection algorithms for only a subset of predicates definable by the logic. While Waldecker permits the definition of all of the types of predicates defined above in Section 2.0.1 , he does not give a clear interpretation for the last two types (which can be interpreted many ways) and restricts the detection algorithms presented to the following types:

- Weak Conjunctive Predicate (WCP): a conjunctive predicate that holds in at least one interleaving of a program execution, written as $\exists\Diamond P$. (See Figure 2 below.)
- Strong Conjunctive Predicate (SCP): a conjunctive predicate that holds in all interleavings of a program execution, written as $\forall\Diamond P$. (See Figure 2 below.)
- Weak Linked Simple Predicate (Weak LSP): this holds if states in which Simple Predicates hold can be ordered using Lamport's 'happens-before', or are incomparable, i.e., all component predicates hold in some interleaving.
- Strong Linked Simple Predicate (Strong LSP): this holds if states in which Simple Predicates hold can be ordered using Lamport's 'happens-before' relation.
- Disjunctive Predicate (DP): this holds if there exists some interleaving in which one or more of the component Simple Predicates of the DP holds.

Detection algorithms are given for Weak Conjunctive Predicates, where the predicate is defined on two processes in the presence of other processes; for Strong Conjunctive Predicates where the predicate is defined on two processes in the presence of others; for Strong Linked Simple Predicates, where the predicate is defined on three or more processes. Disjunctive Predicates are detected using the algorithms for WCP.

While the initial logic presented in Waldecker's work is quite clear and comprehensive, only a subset of this logic is selected for detection and it does not provide any significant increment over any other work reviewed in this chapter. The detection algorithms are weak, as they are not generalizable to n processes for most types of predicates. Moreover, ambiguity exists both at the taxonomy level and in what exactly is detected: a WCP is indistinguishable from a Weak LSP, and a SCP will be detected using either the WCP or the SCP algorithm, without distinction. No attempt is made by Waldecker to interpret Linked Conjunctive

Predicates, so the goal of this thesis of defining relationships between distributed predicates is not met.

No halting algorithms are presented in Waldecker's work; the focus of the work is detecting satisfaction of a given predicate.

Waldecker does attempt to link safety requirements to distributed predicates. He notes that when searching for a predicate expressing an error condition, that is, a predicate specifying violation of some safety requirement, it is only necessary to search for a WCP. This is because it is only necessary to detect a single global state violating safety to assure that it is indeed violated; satisfaction of safety requires scanning the complete execution. However, WCP's need not necessary express safety violation. SCP's, he notes, seem more useful for expressing correctness. However, he does not make an explicit link between distributed predicates and a model for specifying safety. Given the lack of generality of the detection algorithms, extending the logic further may not be feasible.

While Waldecker does not address the notion of minimal and maximal prefixs at which a predicate may hold, such a notion is expressible using the interleaving model. Cooper and Marzullo, (next section), approach this idea and show other similarities to Waldecker's work due to their common use of an interleaving model of concurrency.

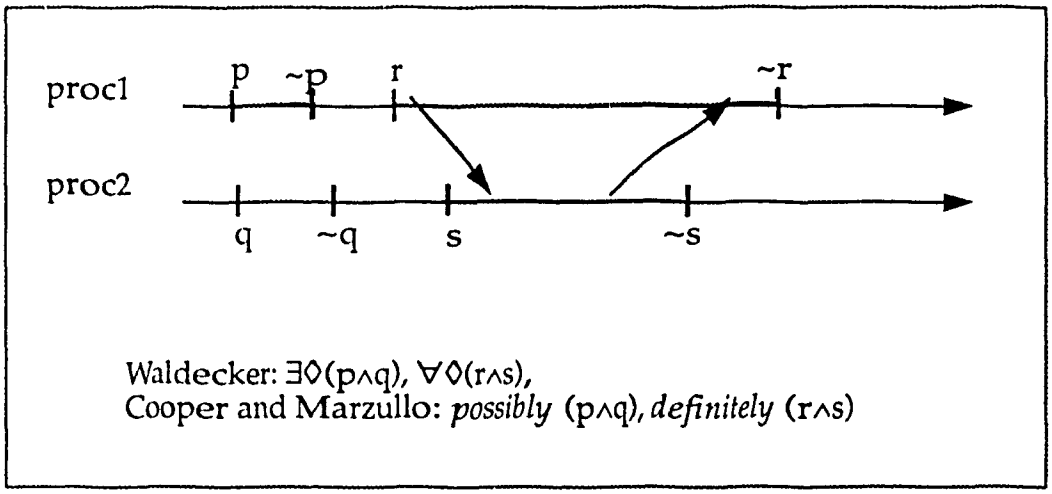

FIGURE 2: ∃◊ (*possibly*) and ∀◊ (*definitely*)

2.3 Cooper and Marzullo: detecting distributed predicates

Cooper and Marzullo, **[Coop91]**, focus on the detection of distributed predicates. They restrict the problem of halting to deciding whether the desired predicate holds in the current state.

They use the standard approach for a predicate logic as described above - boolean operators on boolean expressions constructed using local variables of the process. They use an interleaving model of concurrency, and make the following distinctions in terms of predicate detection: a predicate holds *possibly* if it held in at least one state of one interleaving of program execution. This corresponds to Waldecker's $\exists\Diamond P$ (WCP). A predicate holds *definitely* if it holds in at least one state of all interleavings of the program execution. This corresponds to Waldecker's $\forall\Diamond$ (SCP). A predicate holds *currently* if it holds in the current state of the program. Cooper and Marzullo do not say why the distinction between *possibly* and *definitely* is significant, as Waldecker attempted to do by linking them with violation and satisfaction of requirements.

However, some notion of minimal and maximal prefix is brought forward: the notion of a minimal and maximal global state that contains a particular event e_i is defined. The causality relationships of other events to e_i are somewhat obscured by use of the interleaving model, but this notion of minimal and maximal global states is used in the detection strategy rather than giving the user the minimal and maximal global states at which the predicate itself held.

Thus **[Coop91]** does not meet the goals of defining relationships between distributed predicates, expressing safety, and permitting specification and detection of minimal and maximal prefixs.

2.4 Fowler and Zwaenepol

The contribution of the work of Fowler and Zwaenepol, **[Fow90]**, is the identification of the notions of minimal prefix, called a Causal Distributed Breakpoint, of a computation with respect to a predicate. They note that, in sequential systems, halting at a breakpoint leaves the system in a state in which all events that have occurred have a causal effect on the event triggering the breakpoint. In their work, local predicates equivalent to the Simple Predicates

defined above are defined. Detection of satisfaction of a local predicate triggers a roll-back algorithm that restores the system to the global state corresponding to the minimal prefix of the computation at which the predicate was satisfied.

In sequential systems it is equally true that all events after a breakpoint depend on the event triggering the breakpoint. The corresponding notion of maximal prefix in distributed systems is not identified by Fowler and Zweanepol. In their work, no attempt is made to extend the notion of minimal prefix to more complex Conjunctive or Disjunctive predicates.

2.5 Spezialetti

The thesis of **[Spez89]** is that previous approaches to monitoring distributed systems have not made clear to the user the distinction between the state at which a predicate is detected and the state at which the system halts. A general approach is developed for the task of monitoring. This approach takes into account the characteristics of the predicate being monitored, with the goal of enabling the user to understand the basis on which detection occurs and the distinction between the state at which the predicate is detected and the state at which the system halts. Characteristics of predicates that are taken into account are the number of processes on which predicate is defined and whether the predicate is stable or unstable.

A logic is presented that permits specification of Simple Predicates, Conjunctive/ Disjunctive Predicates, Linked Simple Predicates, and Linked Conjunctive/ Disjunctive Predicates. A method called Simultaneous Regions is developed as a tool in detecting two or more Simple Predicates holding concurrently in some state of the system. Detection of Linked Simple Predicates is straightforward and proceeds more or less as in any of the works described above. No interpretation or detection methods are given for Linked Conjunctive/Disjunctive Predicates.

The method of distributed snapshot establishes a consistent image of system activity, and implies that the local states of which the global snapshot is composed occurred (or could have occurred) simultaneously for a global observer. Spezialetti presents an algorithm for detecting predicates using a version of the snapshot algorithm. However, this method requires the involvement of all processes in the system. This may be far more expenditure than is justified for the

detection of a predicate defined on a small subset of processes in the system. In **[Spez89]** the method of Simultaneous Regions is proposed as a tool for detection of this type of predicate.

The method of Simultaneous Regions is based on the observation that ordering of events in distributed systems is established through message passing. Sets of events at a process that fall in between two message deliveries can be considered to be local regions. Two Simple Predicates can be said to hold simultaneously if they both hold in some consistent global state, that is, if the regions in which they hold can be considered to be *simultaneous*. Local regions are numbered sequentially: each time the Simple Predicate value at a process changes, the local region number is incremented. When the local region number changes, a marker message containing the region number and the value of the Simple Predicate is sent to every other process on which the predicate to be detected is defined. These marker messages are considered to establish ordering among events. On receipt of such a marker message, the receiving process checks to see if the values of the Simple Predicates (its own, and that received in the marker message) satisfy the predicate. If two Simple Predicates hold in same-numbered regions, then they hold simultaneously. If a process's region number is less than the region number

received in the marker message, the receiving process sets its region number to the received region number.

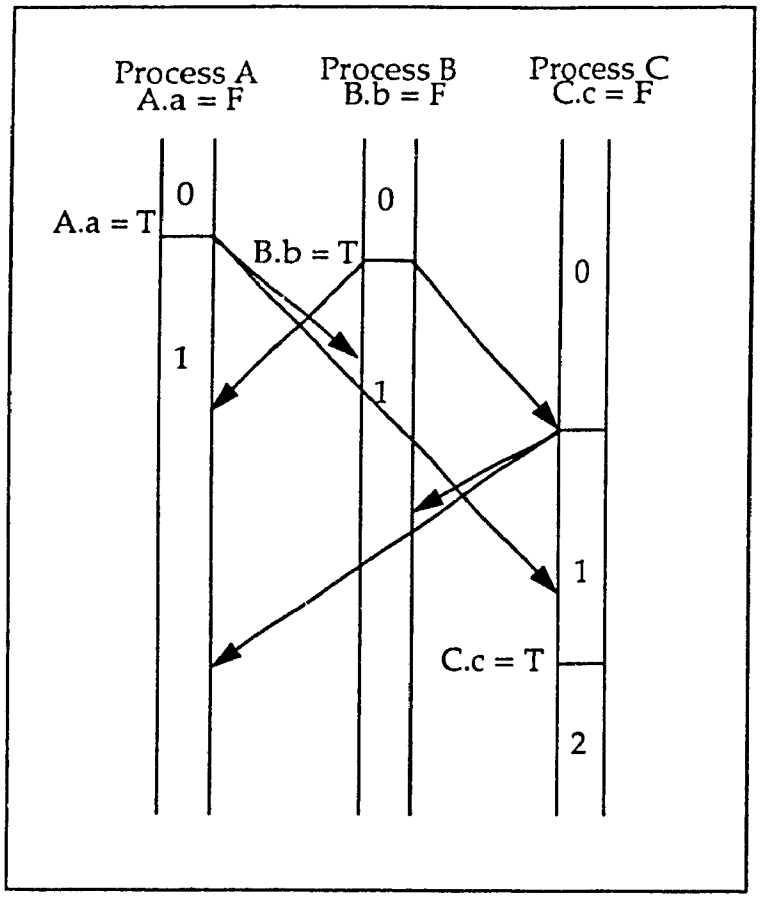

FIGURE 3: Simultaneous Regions

Figure 2 above is an example of monitoring for the predicate (A.a AND B.b AND C.c), where a, b, c are boolean variables. Initially all variables are false in region 0. When A.a and B.b become true, marker messages are sent to the other processes. Since process B is already in region 1 when it receives its marker message, it does not change its region number, and similarly for process A. Process C receives a marker message first from Process B, increments its region number and sends out its own marker messages. C.c becomes true and increments its region number to 2. Since a, b and c do not all hold in region 0 or in region 1, no instance of the predicate is detected.

Since all the messages shown in the above figure are marker messages, and not application messages, this conclusion does not make any sense. Clearly the a, b

and c all hold in some global state of the system. The flaw in this detection method is that it depends on ordering established by marker messages, and detection is therefore not reliable. Successful detection depends on the arbitrary time-delay in message delivery. Consider, for example, if exactly the same scenario is considered, but message arrival differs:

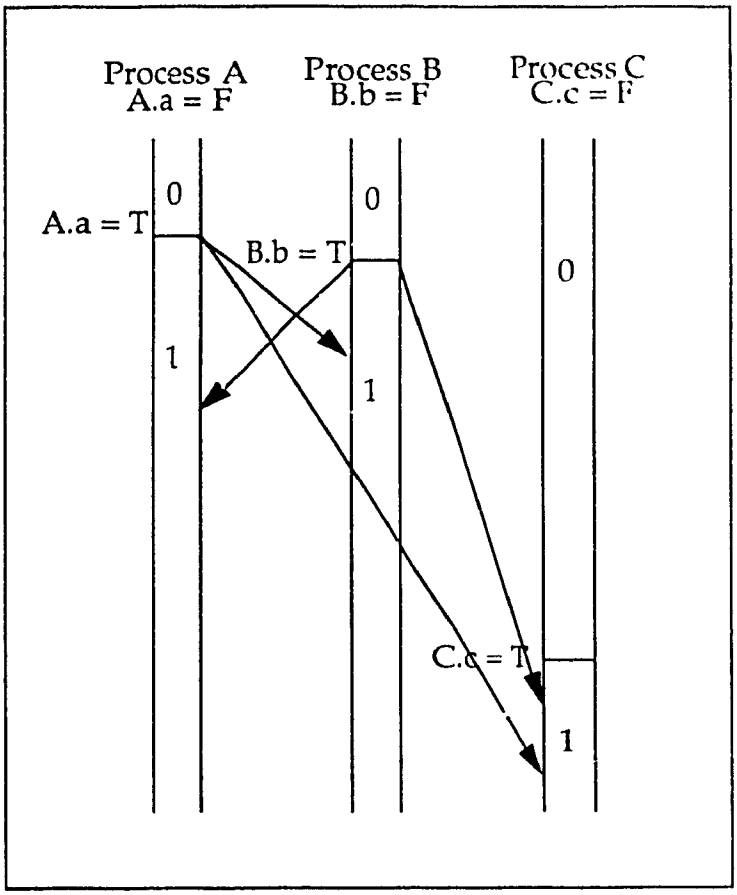

FIGURE 4: Simultaneous Regions Revisited

If messages from process A and B are delayed until after C.c has become true, then process C does not update its region number, being already in region 1, and detection does occur, as a, b and c all hold in region 1. Detection using the method of Simultaneous Regions is thus unreliable.

While flawed in its use of marker messages to establish ordering, the above technique provided a basis for some of the methods proposed here. Specifically, the idea of local regions in which a Simple Predicate holds is used as a basis for detection. This idea was also proposed in a similar form by Waldecker.

While [Spez89] does not present any means for 'Linking' distributed predicates, expressing safety using predicates, or for determining minimal and maximal prefixs at which a computation holds, her work does make some contribution in the area of algorithm design.

2.6 Chandy and Misra

Chandy and Misra propose a theory and notation for parallel program design in [Chan88]. Safety properties can be expressed using a fundamental expression, *P unless Q* (if P holds, then it continues to hold unless Q holds). *unless* can be used to construct other expressions such as *stable P* (once P holds it continues to hold, or P unless P) and *invariant P* (P holds throughout program execution, or P is stable and holds at the initial state of the program). Lamport shows in [Lam80] that enriching an ordinary temporal logic with an operator such as *unless* permits the specification of a more general class of properties.

However, neither [Chan88] nor [Lam80] gives an interpretation of these specification tools for the class of distributed systems considered here. The concepts proposed in this work are taken from the more abstract context of concurrent systems to the more specific context of asynchronous distributed systems. This interpretation makes it possible to express such safety properties in terms of distributed predicates. Systems can then be monitored for satisfaction or violation of such predicates for the purposes of testing and debugging or fault-tolerance.

2.7 Lamport

In [Lam86a, Lam86b] a formalism for specifying and reasoning about concurrent systems is presented. This formalism provides a basis for reasoning about non-atomic actions, which can be used to show what it means for one system to implement another - specifically, what it means to implement a system that provides atomicity using primitives that are inherently non-atomic.

An atomic operation is one whose execution is performed as an indivisible action. An operation is a class of actions, and an operation execution a particular instance of such an action. In a high-level language an action may appear to function atomically, yet at a lower or machine-language level must be implemented with

circuits that are inherently non-atomic. In [Lam86] the problem of implementing atomic operations to a shared register with more primitive, non-atomic operations is considered.

An operation execution is represented by a set of primitive actions or events, where A->B means that all the events of A precede all the events of B, and A--> B means that some event of A precedes some event of B. If s_A and f_A are the start and finish events of A, then in a global time model A->B means that A finishes before B starts, and A-->B means that A starts before B finishes. For all operation executions A and B in a set of events E, either A->B or B-->A. If A->B, then A-->B as well.

A Simple Predicate holds over a sequence of local states of a process. This sequence or region is identified with Lamport's notion of non-atomic operation; each region is marked by an initial and a final event, and regions can be ordered using an interpretation of '->' and '-->'. This notion of non-atomic action provides the basis for the interpretation of predicates as distributed, non-atomic actions, developed in this thesis.

2.8 Conclusion

While the body of work reviewed here includes contributions to the subject of detection of distributed predicates, some issues are left undefined or ambiguous. Issues still undefined or ambiguous are how to establish relationships between distributed predicates analogous to those that can be established between local events - for example, sequence and concurrency. Several works [Wal91, Spez89] permit the specification of 'causal' relationships between distributed predicates, but do not give an interpretation or adequate detection methods.

Detection has taken two approaches. In earlier work [Fow90, Hab88, Mil88], the condition to be detected was characterized as an atomic event. In later work [Wal91, Spez89] it was recognized that breakpoint condition may hold over a subset of the state space, and that this fact can be used to increase the efficiency and the precision of detection.

Most of the works reviewed [Spez89, Mil88, Hab88] use some optimization of Chandy and Lamport's distributed snapshot algorithm [Chan85] to halt a

distributed system in a consistent state once a breakpoint condition or predicate has been detected. The utility of applying a halting algorithm in the context of detection of unstable predicates is questionable. Either the state obtained by the halting algorithm corresponds to a state at which the distributed predicate held, or it does not: decreasing the temporal gap between detection and halting is not useful unless the gap is reduced to zero. Since the distributed program cannot spontaneously halt in a consistent state in which the breakpoint condition (predicate) first holds (except with the use of replay facilities, described below), the state obtained by a halting algorithm will be subsequent to the initial state at which the predicate held. For a large class of predicates, it cannot be guaranteed that the predicate will hold at the state obtained by a halting algorithm. Either the user wishes to know whether the predicate held pure and simple, or wishes to initiate some roll-back mechanisms if the predicate held. In either case, obtaining a consistent global state subsequent to the breakpoint state is, in general, irrelevant.

Since many distributed programs are inherently non-deterministic, the use of a replay mechanism **[LeBl87]** that permits the user to record and reproduce a particular execution of a program is an accepted method for debugging **[LeBl87, Kraw92]**. Employing replay permits the user to either utilize algorithms that will halt the computation exactly in a desired state at which the breakpoint condition holds, without any potential side-effects on the recorded computation, or permits the user to roll the computation back to a such a state.

None of the work reviewed permits the specification of relationships between distributed predicates as between Simple Predicates. Nor do any of the specification logics presented link explicitly with models for the specification of safety in concurrent systems, such as are presented by Chandy and Misra and Lamport. Lamport's work in non-atomicity will be taken as a basis for providing a more complete specification logic, which fulfills the requirement of being able to express safety properties in the form of distributed predicates.

Chapter 3 : Reachability

3.0 Introduction

Distributed predicates can be used to express the safety properties of programs. Distributed predicates can be monitored in the context of deterministic re-execution or in a real-time context for fault-tolerant applications.

However, distributed predicates do not behave as do predicates on local processes. Firstly, occurrences of a local predicate holding can be totally ordered at a process just as atomic events at a process are totally ordered. Distributed predicates are spatially distributed and hold over global states, which cannot be totally ordered. Secondly, change occurs *atomically* in local predicates. This means that an assignment to a local variable, which is considered to be an atomic operation occurring in no time, atomically changes the value of the predicate defined on the variable. Distributed predicates do not change value atomically; because they are distributed over multiple processes, distributed predicates may become true at several processes simultaneously, and false at several processes simultaneously. Thus a logic is required that will address the problems posed by distributed predicates.

A logic is proposed for distributed predicates for the purposes of interpreting specifications and performing monitoring. To interpret specifications, the logic must capture the relationships between distributed predicates and the mechanism of change of value of distributed predicates. To perform monitoring, the model proposed must capture these relationships in such a way that a detection strategy follows efficiently from the model.

3.1 Process Model

A process is an instance of a program execution, viewed as consisting of a sequence of events. An event is an atomic transition of the local state, considered to occur in no time (duration of zero). Events are thus atomic actions which occur at processes; events which are considered atomic are send events, receive events, and assignments to local variables of a process.

Processes are assumed to communicate by message-passing only, and it is assumed that message delay is finite, but unbounded. Messages are ordered such that if a process proc1 sends a message m1 at time t1 to a process proc2, and sends a message m2 at time t2>t1, then proc2 receives m1 before m2. It is not necessary to assume mechanisms for atomic broadcast or causal broadcast.[1]

Events at a process are totally ordered. Each send event has a corresponding receive event and the send and receive events of a particular message are sequenced as follows:

An event structure is a pair (E,<) where e is the set of events in a computation and '<' is an irreflexive partial order on E. For a given computation, e happens-before e' (e<e') iff

1. e and e' are events at the same process and e precedes e', or
2. e is the send event of message m1 and e' is the corresponding receive event of message m1.

If not (e<e') and not (e'<e) then e and e' are concurrent, written as (e | | e').

A consistent prefix π of an event set E is a finite subset

$$\pi \subseteq E: (e \in \pi) \wedge (e' < e \Rightarrow e' \in \pi) \qquad [1]$$

A consistent prefix is isomorphic to a consistent global state S. The global state of a system associated with a prefix π is the collection of local states of the processes immediately after the occurrence of all the events in π, and the set of messages sent but not yet received.

3.2 Predicates

A predicate is an assertion defined on the local variables of one or more processes. The processes on whose variables a predicate is defined are called *predicate*

1. To guarantee the possibility of consensus in asynchronous systems in the presence of stop-failure, messages must be ordered and the transmission mechanism be broadcast. Messages must be ordered in the following way: processor P_r receives message m1 before message m2 when P_1 sends m1 to P_r at real time t1, p2 sends m2 to P_r at real time t2, and t1<t2. It is sufficient to assume the slighter weaker condition of atomic broadcast, that either all sites receive m' before m or all sites receive m before m'. However, for the purposes of this work the existence of such facilities is not assumed; unordered delivery (with respect to the definition of ordering given above) is dealt with.

processes. No variables are shared, and a variable is said to be owned by the process at which it resides.

A logic for specification of distributed predicates is defined in this section and will be extended in Chapter 6. A *local* predicate is a predicate on the local variables of a sequential process. Local predicates can be specified and conjoined with logical operators (AND, OR, NOT) and comparative operators (<,>,=) to form *distributed* predicates. If two assertions are joined by a logical or comparative operator, satisfaction of the expression requires that both assertions hold in some single consistent global state of the system.

Well-formed predicates are constructed in the usual way, following this grammar:

An assertion is a boolean-valued predicate. This boolean-valued expression is constructed by naming a non-boolean valued variable (nbv), a comparative operator, and a number or a non-boolean variable; or by naming a boolean variable directly:

a = ((nbv | number) + (> | < | =) + ((nbv) | number) | bv

Examples:

- Variable n at process proc3 is greater than 2: (proc3.n>3)
- Variable x at process proc3 is less than variable y at process proc4: (proc3.x<-proc4.y)
- A boolean assertion uses assertions to build more complex boolean-valued expressions. It is constructed by naming an assertion, a logical operator, and another boolean assertion or assertion:

ba = (a | ba) + (AND | OR | NOT) + (a | ba)

Examples:

- Variable DONE at process proc1 is true and variable COUNT at process proc1 is greater than 10: ((proc1.DONE) AND (proc1.COUNT >10))
- Variable TOKEN at process proc1 is true and variable TOKEN at process proc2 is true: (proc1.TOKEN AND proc2.TOKEN).

If an assertion or boolean assertion names variables local to one process, it is a local predicate; if it names variables of more than one process, it is a distributed predicate.

Events at a process are totally ordered. The execution of an event e_x leads to a state S_x; the execution of e_{x+1} to a state S_{x+1} and so on. If a local predicate holds in a state S_x, then it is said to *hold at* the event e_x.

The local variables of a process are considered to change value atomically; the change occurs in no time. A local predicate thus changes value atomically, as the variables on which it is defined change value atomically. These changes are totally ordered.

An instance of a local predicate is a totally ordered sequence of events $e_0<..<e_f$ such that the predicate holds at e_0, e_f, and all events in between. Such instances of a local predicate P_i are totally ordered; all events in the xth instance of P_i (written as P_i^x) happen-before all events in the $(x+1)$th instance of P_i. An instance of a predicate is a totally ordered sequence of events; instances of local predicates are totally ordered:

$$P_i^x = \{\pi \subseteq E: e_0 < \ldots < e_f \wedge (\forall e \in P_i^x, \forall e' \in P_i^{x+1}: e < e')\}.$$

P_i^x is the xth instance of local predicate P_i.

3.3 Motivation for a Logic of Distributed Predicates

The predicate logic defined in **Section 3.2** can be used to specify breakpoint conditions; the predicate logic specifies some interesting event or condition and the computation is halted at a global state in which the predicate holds.

Using an assertional approach in the predicate logic is equivalent to the behavioral approach and presents some advantages. Predicates and events are analogous: the event *'x is assigned the value 2'* is equivalent to the predicate which asserts *'the event 'x has been assigned the value 2' has occurred'*. Predicates are appropriate to specification of more complex conditions and properties because they lend themselves to abstraction; a condition can be specified, rather than specifying all the possible behaviors satisfying the condition.

Breakpoints are an important tool in sequential debugging. They permit the user to examine program state at interesting places and to track the flow of control by using successive breakpoints. Some work has been done adapting the concept of breakpointing to the debugging of distributed programs **[Hab88, Mil88, Coop91]**.

However, this work has for the most part been limited to characterizing and detecting sequential breakpoints - local predicates as defined above **[Fow99, Mil88]**. There are two drawbacks to these works. First, the type of distributed predicates that can be specified or detected is limited **[Hab88, Spez89, Wal91]**: they do not consider the cases where there is more than one state corresponding to the specified breakpoint. Second, the causal relationships between distributed predicates is limited: for example, specification and detection of one distributed predicate causally affecting another is not successfully treated **[Hab88, Spez89, Wal91]** and no other possible relations between predicates are considered.

The difficulty in determining relationships between distributed predicates lies in the fact that distributed breakpoints cannot be easily ordered, as sequential breakpoints can, using Lamport's 'happens-before' relation **[Lam78]**. The relationship between two events e and e' can be characterized as sequential; depending on our knowledge of the events, we may decide that there is a causal relationship between e and e'. But predicates, unlike events, hold over global states; they are *spatially distributed*. Since strictly speaking it cannot be said that a state 'happens-before' another state, we are unable to decide if a predicate can have a causal affect on another predicate on this basis.

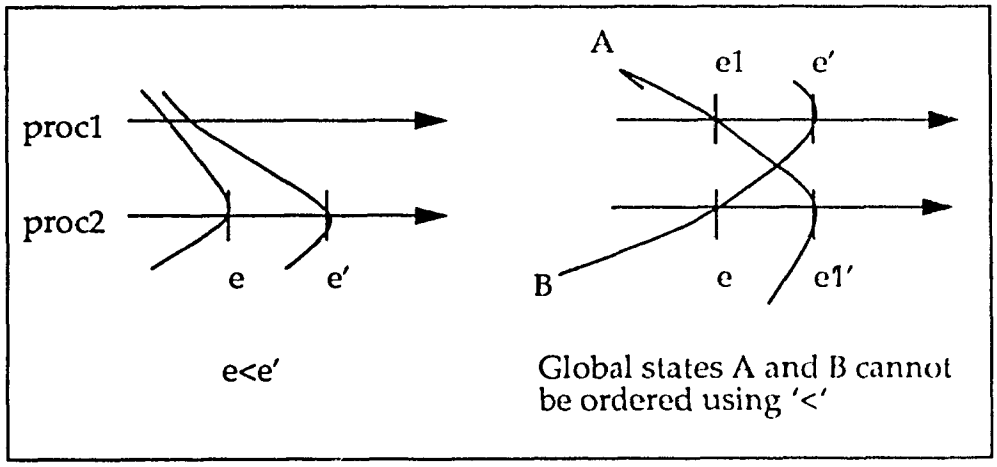

FIGURE 5: Ordering using 'happens-before' (<)

However, if a prefix of a computation, when augmented by some events, is identical to some other prefix of the computation, then we can conclude that the first prefix can causally affect the second. This idea is used in developing a model to characterize the relationships between the global states at which two predicates hold. This topic is covered under the heading 'Reachability' in Section 3.4.

Characterizing the relationship between two global states or prefixs is not in itself enough to decide if two predicates can affect each other. Predicates hold over many prefixs of the computation, that is, over some subset of the global state space. They do not, in general, hold only at one global state as drawn in Figure 5.

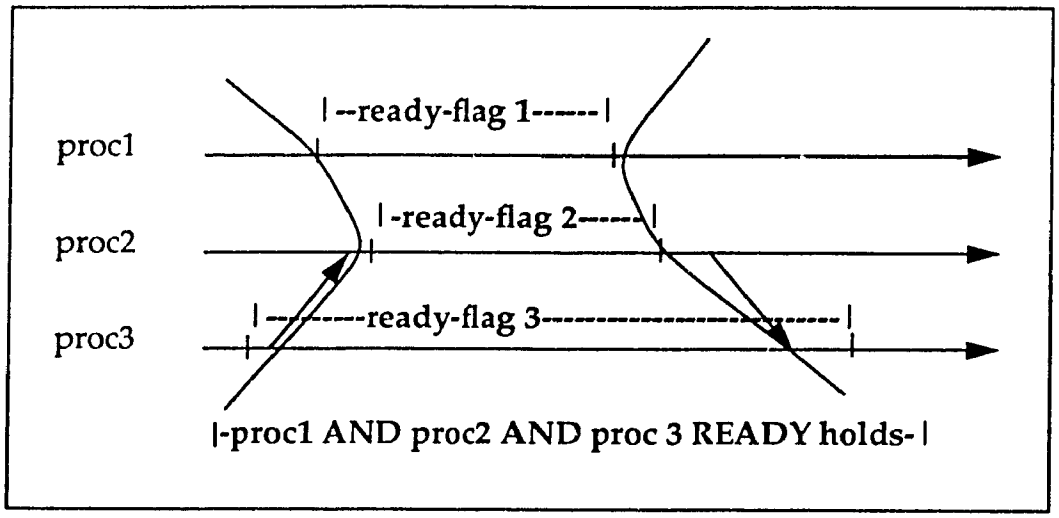

FIGURE 6: Predicates hold over a subset of the state space

To decide if one predicate can, causally or otherwise, affect another predicate, every prefix at which the two predicates hold must be taken into account.The concepts and notation developed in the Reachability section will be used to specify the relationships between distributed predicates in Chapter 5. Two problems will be addressed; spatial distribution of distributed predicates, and non-uniqueness of initial and final states at which a predicate holds.

Discussion of applications of the expanded model will focus on uses for breakpointing in debugging (Chapter 7, Section 7.1) and for safety specification, for the purposes of monitoring and debugging (Chapter 7, Section 3.0).

3.4 Reachability

3.4.1 The basic relation: reachability

Let A be a non-empty set of concurrent events at PA's predicate processes, at which PA holds. Then π_A is a consistent prefix of E found with respect to the set A:

$$\pi_A \subseteq E: (A \in \pi) \wedge (\forall e \in A: e' < e \Rightarrow e' \in \pi) \quad [2]$$

and Π_A is the set of all such π_A.

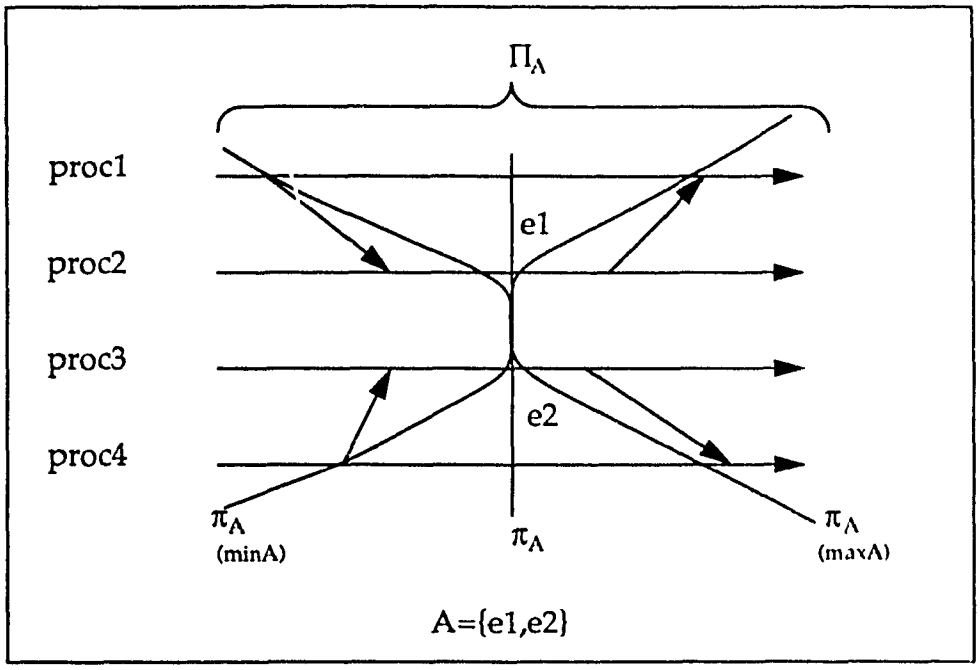

FIGURE 7: Π_A

Π_A is the set of prefixs π_A of the computation which give the global states at which events A have occurred and in which PA holds.

Recall that a local predicate is said to *hold at* an event e_x if the predicate holds in the associated state S_x. When a predicate holds in a global state, and that global state is a prefix of the computation found with respect to a set A, the predicate is said to *hold at* the events in A.

If a prefix of a computation, when augmented by some events, is identical to some other prefix of the computation ($\pi_1 \subseteq \pi_2$), then we can conclude that the first prefix can affect the second.

This observation leads to the idea of *reachability.* A predicate PA holds at a set of events A; and a predicate PB at a set of events B. We will say that PB is *reachable* from PA if some element of Π_A, when augmented by a (possibly empty) set of events, is identical to some element of Π_B.

There is some prefix π_A in Π_A which when augmented by some events e gives some prefix π_B in Π_B:

$$\Pi_A \sim> \Pi_B \text{ iff } \exists \pi_A \in \Pi_A, \exists \pi_B \in \Pi_B : \pi_A \subseteq \pi_B \quad [3]$$

which is equivalent to

$$\Pi_A \sim> \Pi_B \text{ iff } \forall a \in A, \forall b \in B : \neg(b < a) \quad [4]$$

Proof:

$$\Rightarrow: (\exists \pi_A \in \Pi_A, \exists \pi_B \in \Pi_B : \pi_A \subseteq \pi_B) \Rightarrow (\forall a \in A, \forall b \in B : \neg(b < a))$$

Assume that there exists some event *a* in A and some event *b* in B so that (b < a):

$$\exists a \in A, \exists b \in B : (b < a)$$

then for all π_A in Π_A, there is some *a* which is in π_A but which is not in any π_B in Π_B such that:

$$\forall \pi_A \in \Pi_A, \forall \pi_B \in \Pi_B : \neg(\pi_A \subseteq \pi_B)$$

which is contradictory. Therefore, if $\neg(\pi_A \subseteq \pi_B)$ then $\neg(b < a)$ holds for all events *a* in A and all events *b* in B:

$$(\forall a \in A, \forall b \in B : \neg(b < a)) \Rightarrow (\exists \pi_A \in \Pi_A, \exists \pi_B \in \Pi_B : \pi_A \subseteq \pi_B)$$

Assume that there does not exist any $(\pi_A \subseteq \pi_B)$; then:

$$\forall \pi_A \in \Pi_A, \forall \pi_B \in \Pi_B : \pi_B \subset \pi_A$$

then there must exist some *b* that happens-before some *a*:

$$\exists a \in A, \exists b \in B : (b < a)$$

But this is also contradictory. Therefore, there must exist some π_A in Π_A and some π_B in Π_B such that $(\pi_A \subseteq \pi_B)$.

■

3.4.2 Other relationships, using reachability

By determining if Π_A and Π_B are each reachable from the other (mutually reachable) the possible relationships between Π_A and Π_B are defined:

Is Π_B reachable from Π_A? Is Π_A reachable from Π_B?

1. Yes, yes; Π_B is reachable from Π_A, and Π_A is reachable from Π_B.
2. No, no; Π_B is not reachable from Π_A, and Π_A is not reachable from Π_B.
3. Yes, no;Π_B is reachable from Π_A, but Π_A is not reachable from Π_B.
4. No, yes; Π_B is not reachable from Π_A, and Π_A is reachable from Π_B.

1. Π_B is reachable from Π_A, and Π_A is reachable from Π_B: when all events in A are concurrent with all events in B, then there is some prefix π_A and some prefix π_B which are identical:

$$\Pi_A \| \Pi_B \text{ iff } \Pi_A \sim> \Pi_B \wedge \Pi_B \sim> \Pi_A \quad [5]$$

which is equivalent to

$$\Pi_A \| \Pi_B \text{ iff } \forall a \in A, \forall b \in B: (a \| b) \quad [6]$$

The above follows directly from the definition of '~>'. Since $\Pi_A \sim> \Pi_B$ holds, then $\neg(b < a)$; and since $\Pi_B \sim> \Pi_A$ holds, then $\neg(a < b)$; therefore $a \| b$ for all events a in A and all events b in B.

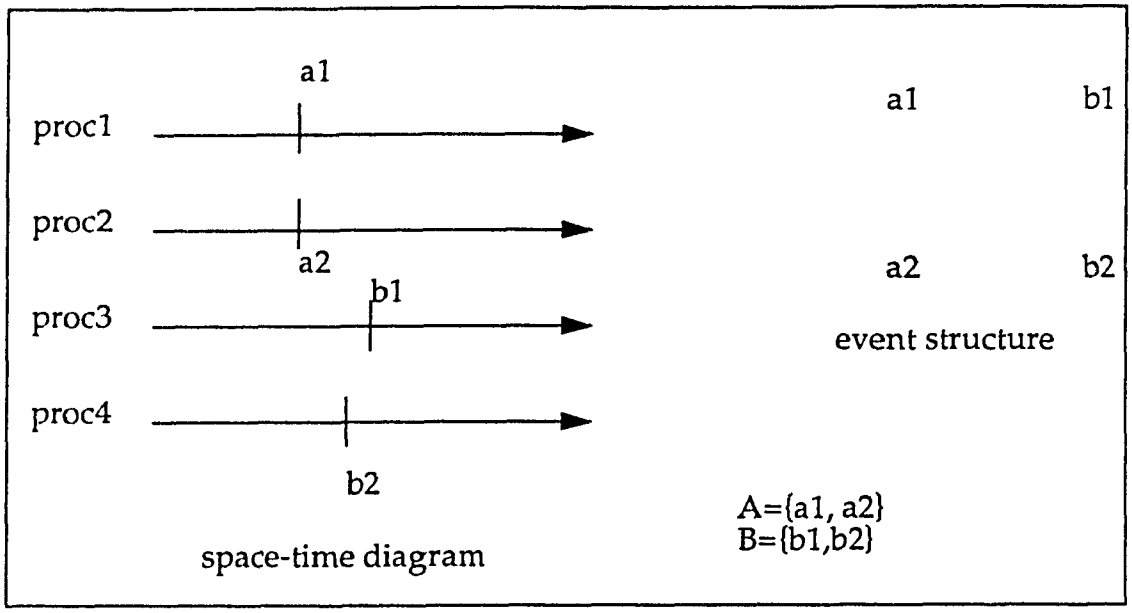

FIGURE 8: $\Pi_A||\Pi_B$

Let A and B be non-empty sets of concurrent events:

|| is symmetric and reflexive:

if $\Pi_A||\ \Pi_B$then $\Pi_B||\ \Pi_A$

$\forall A \subseteq E,\ \Pi_A||\ \Pi_A$

2. If Π_B is not reachable from Π_A, nor Π_A from Π_B, then

$$\Pi_A X \Pi_B \text{ iff } \neg(\Pi_A \sim> \Pi_B) \wedge \neg(\Pi_B \sim> \Pi_A) \qquad [7]$$

which is equivalent to

$$\Pi_A X \Pi_B \text{ iff } \exists a, a' \in A, \exists b, b' \in B:\ (a < b) \wedge (b' < a') \qquad [8]$$

and follows directly from the definition of '~>'.

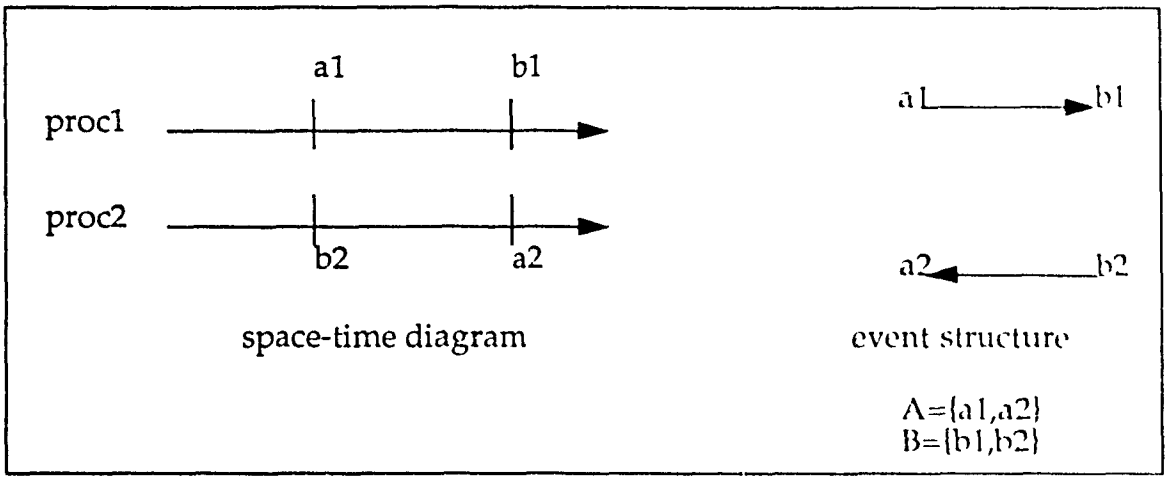

FIGURE 9: $\Pi_A X \Pi_B$

Let A and B be non-empty sets of concurrent events:

X is symmetric:

if $\Pi_A X\ \Pi_B$ then $\Pi_B X\ \Pi_A$

3. If Π_B is reachable from Π_A, but Π_A is not reachable from Π_B, then

$$\Pi_A \models\Rightarrow \Pi_B \text{ iff } (\Pi_A \sim> \Pi_B) \wedge \neg(\Pi_B \sim> \Pi_A) \qquad [9]$$

which is equivalent to

$$\Pi_A \models\Rightarrow \Pi_B \text{ iff } (\exists a \in A, \exists b \in B: a < b) \wedge (\forall a \in A, \forall b \in B: \neg(b < a)) \qquad [10]$$

following again directly from the definition of '~>'.

That is, some prefix π_A, when augmented by some events, is identical to some prefix π_B; and there is no prefix of π_B which can be augmented to any prefix π_A; and there is no π_A which is identical to a π_B.

If $\Pi_A \models\Rightarrow \Pi_B$ then it can be said that PA can causally affect PB in a sense analogous to 'event *a* can causally affect an event *b*'.

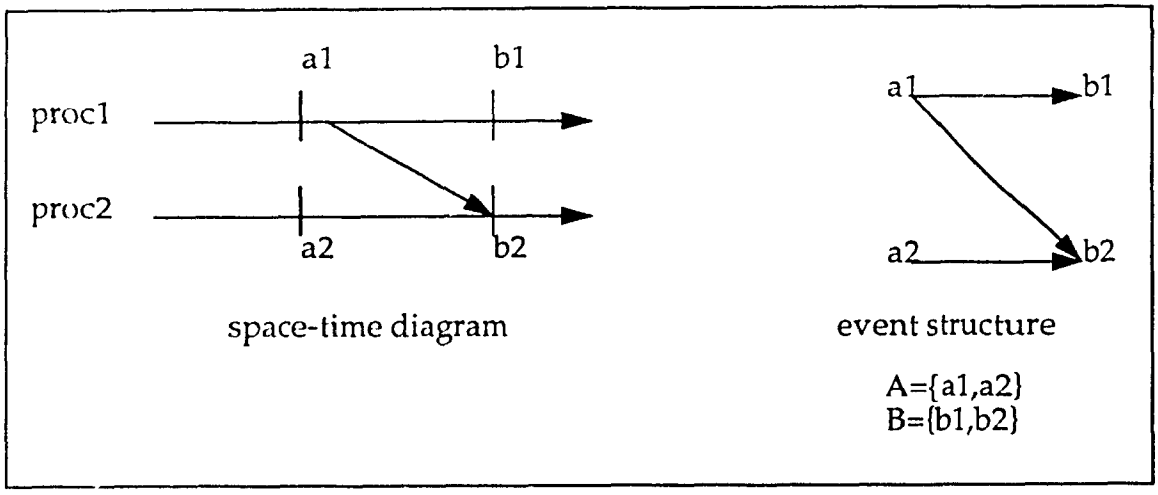

FIGURE 10: $\Pi_A \mid\Rightarrow \Pi_B$

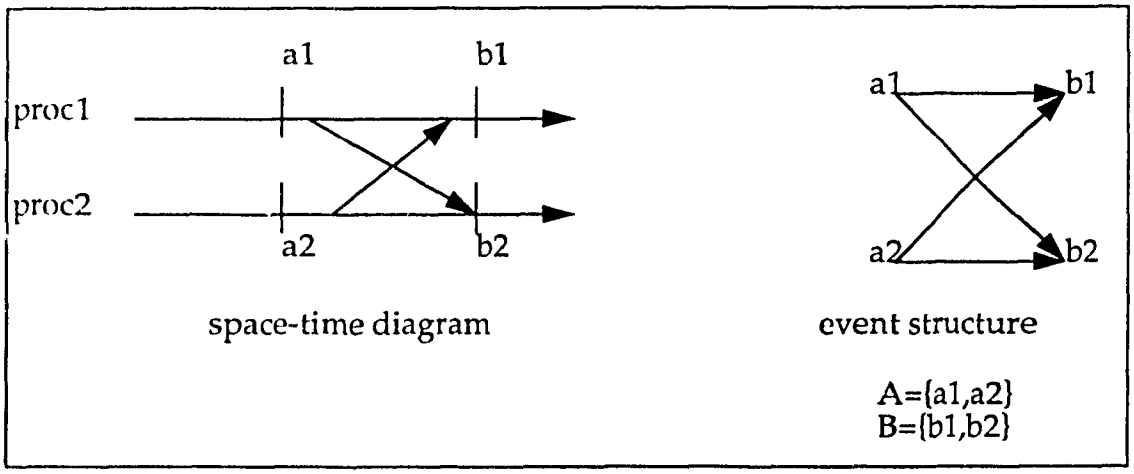

FIGURE 11: $\Pi_A \mid\Rightarrow \Pi_B$

Let A and B be non-empty sets of concurrent events:

$\mid\Rightarrow$is irreflexive:

if $\Pi_A \mid\Rightarrow \Pi_B$ then $\sim(\Pi_B \mid\Rightarrow \Pi_A)$

4. Case 4 ($\Pi_B \mid\Rightarrow \Pi_A$) is analogous to case 3.

Subcases of each of ($\|$, $\mid\Rightarrow$,X) can be defined where utility warrants. Two such cases are defined:

1a) Special case of $\Pi_A \| \Pi_B$:

$\Pi_A \angle \Pi_B$ iff : $A \subseteq B$ [11]

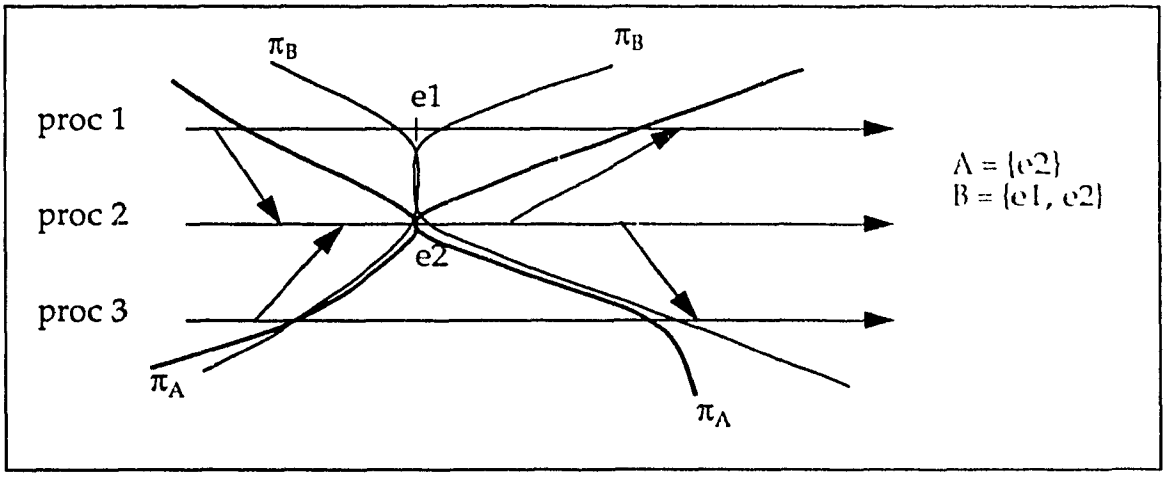

FIGURE 12: $\Pi_A \angle \Pi_B$

$\Pi_A \angle \Pi_B$ is reflexive and transitive: if A, B and C are non-empty sets of concurrent events then

$\forall (A \subseteq E)\ \Pi_A \angle \Pi_A$

if $\Pi_A \angle \Pi_B$ and $\Pi_B \angle \Pi_C$ then $\Pi_A \angle \Pi_C$

3a) Special case of $(\Pi_A \mid\Rightarrow \Pi_B)$, where all events in A precede all events in B:

$(\Pi_A \mid\rightarrow \Pi_B)$ iff $\forall a \in A, \forall b \in B: (a < b)$ [12]

depicted in Figure 11.

$\mid\rightarrow$ is irreflexive and transitive: if A and B are non-empty sets of concurrent events then

if $\Pi_A \mid\rightarrow \Pi_B$ then $\sim(\Pi_B \mid\rightarrow \Pi_A)$;

if $\Pi_A \mid\rightarrow \Pi_B$ and $\Pi_B \mid\rightarrow \Pi_C$ then $\Pi_A \mid\rightarrow \Pi_C$

3.4.3 Properties and lemmas on reachability

Lemma 1: Exactly one of ($|\Rightarrow$, $\|$, X) holds between any two non-empty sets of concurrent events.

Let A and B be non-empty sets of concurrent events. Then Lemma 1 says

$$(\Pi_A |\Rightarrow \Pi_B) \otimes (\Pi_B |\Rightarrow| \Pi_A) \otimes (\Pi_A \| \Pi_B) \otimes (\Pi_A X \Pi_B)$$

Given two non-empty sets of concurrent events, one of the following relationships must hold:

	$\exists a \in A, \exists b \in B: (a < b)$	$\exists a \in A, \exists b \in B: (b < a)$	Relation
1	yes	yes	$\Pi_A X \Pi_B$
2	yes	no	$\Pi_A \|\Rightarrow \Pi_B$
3	no	yes	$\Pi_B \|\Rightarrow\| \Pi_A$
4	no	no	$\Pi_A \| \Pi_B$

TABLE 1:Reachability Relations

Therefore at least one, and exactly one, of the above relations must hold.

1. Some element in A happens-before some element in B; some element in B happens-before some element in A.
2. Some element in A happens-before some element in B; no element in B happens-before any element in A.
3. Some element in B happens-before some element in A; no element in A happens-before any element in B.
4. All elements in A are concurrent with all elements in B; no element in A happens-before any element in B, and no element in B happens-before any element in A.

■

Let A, B, C be non-empty sets of concurrent events:

1. $|\rightarrow$ is irreflexive and transitive:
   if $\Pi_A |\rightarrow \Pi_B$ then $\sim(\Pi_B |\rightarrow \Pi_A)$
   if $\Pi_A |\rightarrow \Pi_B$ and $\Pi_B |\rightarrow \Pi_C$ then $\Pi_A |\rightarrow \Pi_C$
2. $\|$ and X are symmetric:
   if $\Pi_A \| \Pi_B$ then $\Pi_B \| \Pi_A$

if $\Pi_A X\ \Pi_B$ then $\Pi_B X\ \Pi_A$

3. $\|$ and $\angle$ are reflexive
   $\forall A \subseteq E,\ \Pi_A \| \Pi_A$
4. $\angle$ is transitive:
   if $\Pi_A \angle \Pi_B$ and $\Pi_B \angle \Pi_C$ then $\Pi_A \angle \Pi_C$
5. $(\Pi_A\ R\ \Pi_B)$ is treated as a predicate which is either true or false; thus it cannot, in general, be treated as a function that yields a set of prefixs.
   Where **R** is any of ($|\Rightarrow$, $\|$, X)
   $(\Pi_A\ R\ \Pi_B)\ R\ (\Pi_C\ R\ \Pi_D)$
   is undefined.
6. As shown in Lemma 1:
   if $\Pi_A |\!\rightarrow \Pi_B$ then $\Pi_A |\!\Rightarrow \Pi_B$;
   if $\Pi_A \angle \Pi_B$ then $\Pi_A \| \Pi_B$;
   i f $\Pi_A |\!\Rightarrow \Pi_B$ or $\Pi_A \| \Pi_B$ then $\Pi_A \sim\!>\! | \Pi_B$;
   and either $\Pi_A \sim\!>\! | \Pi_B$ or $\Pi_A X \Pi_B$.

3.4.4 Summary

A non-empty set of concurrent events A at a predicate PA's predicate processes is chosen. The set of consistent prefixs of the computation found with respect to the set A is found; this set is called Π_A and PA holds at every prefix π_A in Π_A. A set Π_B is found in the same way.

If no event in B happens-before any event in A, then Π_B is reachable from Π_A ($\Pi_A \sim\!>\! \Pi_B$). If all events in B are concurrent with all events in A, then ($\Pi_A \| \Pi_B$); Π_B is reachable from Π_A, and Π_A is reachable from Π_B. If some event in A happens-before some event in B, and no event in B happens-before any event in A, then ($\Pi_A |\!\Rightarrow \Pi_B$), and Π_B is reachable from Π_A. If some event in A happen-before some events in B, and some event in B happens-before some event in A, then ($\Pi_A X\ \Pi_B$) and neither of Π_A nor Π_B is reachable from the other. Either Π_A is reachable from Π_B, Π_B is reachable from Π_A, or ($\Pi_A X\ \Pi_B$) must hold between any two prefixs (global states) of a computation.

Reachability between global states at which predicates hold is thus determined by examining the causal relationships between sets of concurrent events at which predicates hold.

The reachability semantics presented in this section represents an increment over [Hab88, Mil88, Spez89, Wal91] because causality and other relationships between distributed predicates are defined.

These relationships, (~>, C, | |) are useful in themselves as is demonstrated by the examples given with each definition. They are also useful as building blocks. They will be used in the next chapter in reasoning about how distributed predicates occur and particularly in defining the relationships between distributed predicates.

Chapter 4 : Distributed Predicates

4.0 Distributed Predicates

The concept of reachability between global states at which predicates hold has been introduced along with its notation. But predicates in general may hold over a subset of the global state space, not just over single global states, and the concept of reachability alone is insufficient to address the question of whether one predicate affects another. The spatial distribution that occurs in distributed predicates is addressed by the concept of reachabilty and non-reachabililty; when dealing with predicates that hold over more than one global state, the non-uniqueness of initial and final states at which a predicate holds must also be addressed. The concept of minimal and maximal prefix will be used to address this non-uniqueness.

A given predicate may hold several times over the course of a computation. One occurrence of a predicate holding must be characterized and distinguished from other occurrences before the problem of how two predicates can affect each other can be addressed.

In Section 4.1, minimal and maximal χ are introduced. These are interesting in themselves for use in breakpointing, as will be discussed in Chapter 7, and will be used to specify a notion of initial and final states of an occurrence of a predicate. In Section 4.2, a single instance of a predicate is characterized with respect to some state of the computation. The initial and final states of the instance are shown to be possibly non-unique. Some lemmas on the relationships between initial and final states are given in Section 4.2.2. An example is given in Section 4.2.3 for deciding which sets of events belong to an instance of a predicate. The initial definition of an instance was made with respect to some arbitrary 'initial' state of the computation. To capture all instances of the predicate the minimal sufficient initial states are found in Section 4.2.4. In Section 4.3 the notion of corresponding initial and final states is presented. This will be used in the definition of simultaneity in Chapter 5. Finally, a summary is given in Section 4.4.

4.1 Minimal and Maximal χ

Two important subsets of Π_A are the minimal and maximal χ of the computation found with respect to the events in A.

Let A be a non-empty set of concurrent events.

The *minimal* prefix of the computation with respect to A is:

$$\min_A \subseteq E: (A \subseteq \min_A) \wedge (\forall e \in A: e' < e \Rightarrow e' \in \min_A) \quad [13]$$

and is unique for a given set A. Minimal prefix is shown in Figure 13 and Figure 14.

The *maximal* prefix of the computation with respect to A is:

$$\max_A \subseteq E: (A \subseteq \max_A) \wedge (\forall e \in A: \neg(e < e') \Rightarrow e' \in \min_A) \quad [14]$$

and is also unique for a given set A. Maximal prefix is shown in Figure 13 and Figure 14.

Lemma 2: $\min_A$ and $\max_A$ are unique for a given subset A of an event set E:

By contradiction:

Assume the existence of a π_1, π_2 such that both π_1 and π_2 are minimal χ of A; then $\neg(\pi_1 \subseteq \pi_2)$ and $\neg(\pi_2 \subseteq \pi_1)$.

Then there must exist some $e_1 \in \pi_1$ and some $e_2 \in \pi_2$ such that $e_2 \notin \pi_1$ and $e_1 \notin \pi_2$.

But according to the definition of the set A

if $e_1 \in A$, then $e_1 \in \pi_2$; and

if $e_1 \notin A$, then for some $e' \in A$, $e_1 < e'$, since $e' \in \pi_2$, then $e_1 \in \pi_2$.

Similarly,

if $e_2 \in A$, then $e_2 \in \pi_1$; and

if $e_2 \notin A$, then for some $e' \in A$, $e_2 < e'$, since $e' \in \pi_1$, then $e_2 \in \pi_1$.

Thus $\neg(\exists e \in E: (e \in \pi_1 \wedge \neg(e \in \pi_2)) \vee (e \in \pi_2 \wedge \neg(e \in \pi_1))$

$\therefore ((\pi_1 \subseteq \pi_2) \wedge (\pi_2 \subseteq \pi_1))$

so $\pi_1 \equiv \pi_2$ and the minimal prefix of A is unique.

The proof of the uniqueness of the maximal prefix of A is analogous.

n

Lemma 3: min_A is a subset of max_A.

Proof: For all e, e' in min_A either

1. e is an element of A: then
   e is an element of min_A and
   e is an element of max_A

or

2. if e'<e then
   e is an element of min_A and
   $\neg(e < e')$ so e is an element of max_A.
   Since there is no event which is an element of min_A which is not also an element of max_A, min_A is a subset of max_A.

n

Having defined min_A and max_A, a proof of a property of the reachability relation $\angle$ can be given. This property will be used in defining simultaneity between two predicates in Chapter 5.

Recall that $\Pi_A \angle \Pi_B$ iff : $A \subseteq B$.

Lemma 4: if $A \subseteq B$ then PA holds at all π_B in Π_B.

max_A, min_A

If $A \subseteq B$, then $min_A \subseteq min_B$ and $max_B \subseteq max_A$:

By contradiction:

$min_A \subseteq min_B$: Assume that there exists an a in min_A such that a is not an element of min_B. Then there must exist a b in B such that $a \| b$ or $b<a$. If $a \| b$ then a is an element of B (because a element of A implies that a is an element of B, because $A \subseteq B$), so a is also an element of min_B. If $b<a$ then there exists some a' in A, (and therefore also in B, because $A \subseteq B$), such that $a<a'$. This contradicts the definition of min, so this cannot occur. Therefore there cannot exist an a such that a is in min_A and not in min_B, so $min_A \subseteq min_B$.

$max_B \subseteq max_A$: Assume that there exists a b in max_B such that b is not an element of max_A. Then there must exist an a in A such that $a<b$, by definition of max. But if $a<b$, then there must be some b' in B such that $b'<b$, because all a's are also b's since $A \subseteq B$. But this contradicts the definition of max, so it cannot occur. Therefore there cannot exist a b such that b is in max_B and not in max_A; therefore $max_B \subseteq max_A$.

Therefore $\pi_B \in \Pi_B \Rightarrow \pi_B \in \Pi_A$, by the above and Lemma 4. Since PA holds at all π_A in Π_A, PA holds at all π_B in Π_B.

4.2 Instances of a Predicate

An unstable predicate may hold multiple times throughout the course of a computation. Each occurrence of a predicate holding, or an instance of the predicate, can be distinguished from every other. This is straightforward when the predicate in question is a local predicate, as such instances are totally ordered as discussed in Chapter 3. Distinguishing instances of distributed predicates is more complex because such instances cannot be totally ordered.

4.2.1 Initial and final states of an instance

Let PA be a predicate, and let A be a set of concurrent events at which PA holds. Let $\boldsymbol{A}$ be the set of such sets A. Then

$$\boldsymbol{A} = \{ A \mid \text{PA holds at } \Pi_A \wedge \neg(\exists A', A'' : \text{PA holds at } \Pi_{A'} \wedge \neg(\text{PA holds at } \Pi_{A''}) \wedge \Pi_{A'} \models\!\!\Rightarrow \Pi_{A''} \wedge \Pi_{A''} \models\!\!\Rightarrow \Pi_A) \} \quad [15]$$

and $\boldsymbol{A}$ represents an instance of PA.

Then $\boldsymbol{A}_0$ is a subset of $\boldsymbol{A}$:

$$A \in \boldsymbol{A}_0 \text{ iff } \neg(\exists A' \in \boldsymbol{A}:\ \Pi_{A'} \models\!\Rightarrow \Pi_A) \qquad [16]$$

and $\boldsymbol{A}_f$ is a subset of $\boldsymbol{A}$:

$$A \in \boldsymbol{A}_f \text{ iff } \neg(\exists A' \in \boldsymbol{A}:\ \Pi_A \models\!\Rightarrow \Pi_{A'}) \qquad [17]$$

$\boldsymbol{A}_0$ is the set of sets of concurrent events A at which PA initially holds; $\boldsymbol{A}_f$ is the set of maximal (latest) sets of concurrent events at which PA holds.

If $|\boldsymbol{A}_0| > 1$, then the initial state of this instance of PA is not unique; there is more than one minimal prefix.

It will be useful to designate the set of minimal χ ($\min_A$) of each element A in $\boldsymbol{A}_0$:

$$\boldsymbol{min}_A = \pi \subseteq E:\ (A \in \boldsymbol{A}_0 \Rightarrow A \subseteq \pi) \wedge (\forall e \in A:\ e' < e \Rightarrow e' \in \pi) \qquad [18]$$

and similarly to designate the set of maximal χ ($\max_A$) of each element A in $\boldsymbol{A}_f$:

$$\boldsymbol{max}_A = \pi \subseteq E:\ (A \in \boldsymbol{A}_f \Rightarrow A \subseteq \pi) \wedge (\forall e \in A:\ \neg(e < e') \Rightarrow e' \in \pi) \qquad [19]$$

$\boldsymbol{A}_0$ and $\boldsymbol{A}_f$ may have more than one element when the predicate PA has disjunctive elements.

Example:

- PA = (proc1.x OR proc2.y)
  Let $\boldsymbol{A}_0 = \{\{x_0\},\{y_0\}\}$.
  Let $\boldsymbol{A}_f = \{\{x_f\},\{y_f\}\}$.

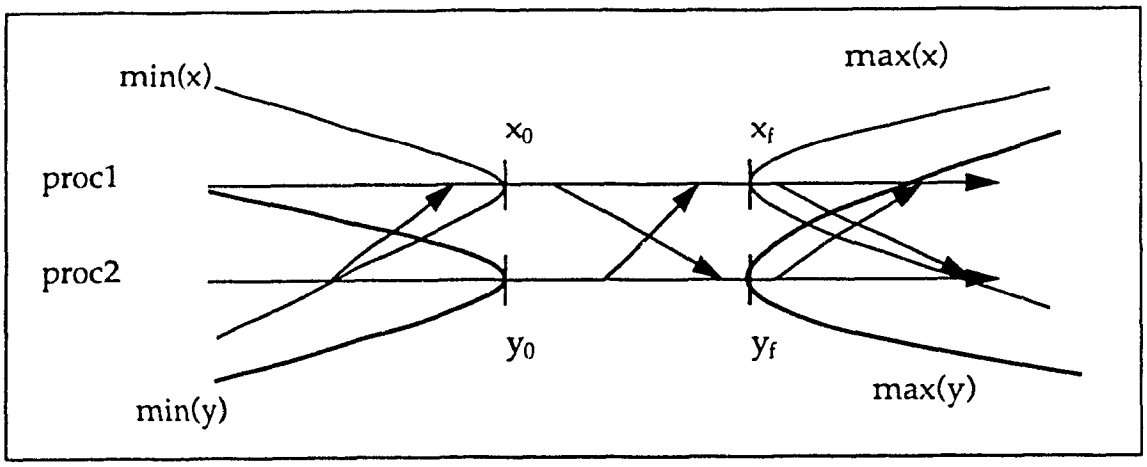

FIGURE 13: Non-uniqueness of min_A and max_A

If the events at which x and y first hold occur concurrently, then PA has not one, but two minimal χ, namely, min(x) and min(y).

- PA = (proc1.x AND proc2.y)

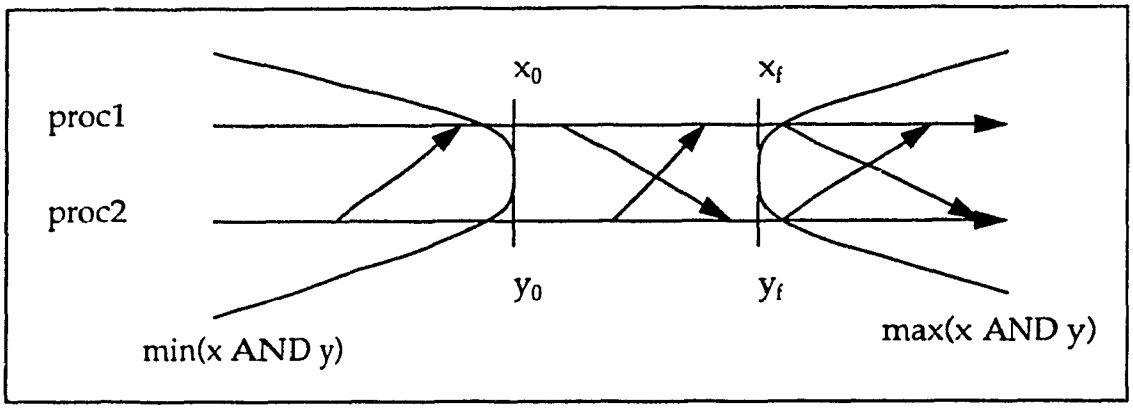

FIGURE 14: Uniqueness of min_A and max_A

The minimal prefix of any set of concurrent events is unique. The minimal prefix is found with respect to each set of sets of concurrent events, as in Figure 13; if for two such minimal χ π_1 and π_2, neither is a subset of the other ($\neg(\pi_1 \subseteq \pi_2)$ and $\neg(\pi_2 \subseteq \pi_1)$), then the minimal prefix is not unique, and similarly for the maximal prefix.

If the minimal or maximal prefix is not unique, then for a given occurrence of a predicate holding there is more than one initial or final state at which it holds. This can be thought of as an action that is distributed in space and is initiated at

several points; the instants at which initiations occur are not totally-orderable. For example, consider a wave receding from a beach. The action of recession is distributed in space along the beach, and each molecule of water that changes direction is a point at which the action of receding begins.

While not totally-orderable, there are certain relationships that obtain among the initial and final states, and also between them. These are set out in the next section.

4.2.2 Initial and final states: some lemmas

The following Lemmas set out the relationships of initial and final states. These lemmas will be used in Section 4.3 in showing the relationships between instances of a predicate, and thus in distinguishing instances of a predicate.

Lemma 5: No initial state of an instance of a predicate precedes ($|\Rightarrow$) any other initial state of that instance of the predicate; no final state of an instance of a predicate precedes any other final state of an instance of that predicate.

$$\forall A, A' \in \boldsymbol{A}_0: (\Pi_A X \Pi_{A'}) \vee (\Pi_A \| \Pi_{A'}) \quad [20]$$

$$\forall A, A' \in \boldsymbol{A}_f: (\Pi_A X \Pi_{A'}) \vee (\Pi_A \| \Pi_{A'}) \quad [21]$$

Let A and A′ be elements of $\boldsymbol{A}_0$.

One of ($|\Rightarrow$,||,X) holds between any two non-empty sets of concurrent events (Lemma 1, Chapter 3). But $\neg(\Pi_A |\Rightarrow \Pi_{A'})$ and $\neg(\Pi_{A'} |\Rightarrow \Pi_A)$ (definition of $\boldsymbol{A}_0$). Therefore one of (||,X) must hold between A and A′, and similarly for $\boldsymbol{A}_f$.

n

Lemma 6: No final state of an instance of a predicate precedes ($|\Rightarrow$)any initial state of that instance of the predicate.

$$\forall A \in \boldsymbol{A}_0, \forall A' \in \boldsymbol{A}_f: \neg(\Pi_{A'} |\Rightarrow \Pi_A) \quad [22]$$

From the definition of $\boldsymbol{A}_f$, $\neg(\exists A, A': \Pi_A |\Rightarrow \Pi_{A'})$.

■

Lemma 7: All final states of an instance of a predicate are reachable from some initial state of that instance of the predicate.

$$\forall A' \in \boldsymbol{A}_f, \exists A \in \boldsymbol{A}_0 : \Pi_A \sim> \Pi_{A'} \quad [23]$$

Each element of $\boldsymbol{A}_f$ is reachable from some element of $\boldsymbol{A}_0$. It is possible for $\boldsymbol{A}_0$ to be identical to $\boldsymbol{A}_f$, if there is just one set of concurrent events at which PA holds. It is also possible for the set $\boldsymbol{A}_f$ to be empty if the predicate PA is stable.

By contradiction: assume some A' in $\boldsymbol{A}_f$ such that there does not exist an A in $\boldsymbol{A}_0$ that reaches A' ($\sim(\Pi_A \sim> \Pi_{A'})$). A' cannot precede A ($\sim(\Pi_{A'} \sim> \Pi_A)$), by Lemma 5, so either ($\Pi_A X \Pi_{A'}$) by Lemma 1 or A does not exist. If ($\Pi_A X \Pi_{A'}$) then there must exist some A'' in $\boldsymbol{A}_0$ such that ($\Pi_{A''} \sim> \Pi_{A'}$); otherwise A' would be a member of $\boldsymbol{A}_0$. If A does not exist, then A' cannot exist either; by definition it would be a member of $\boldsymbol{A}_0$. Therefore the contradiction cannot hold and each element of $\boldsymbol{A}_f$ is reachable from some element of $\boldsymbol{A}_0$.

n

4.2.3 Example: determining membership in *A*

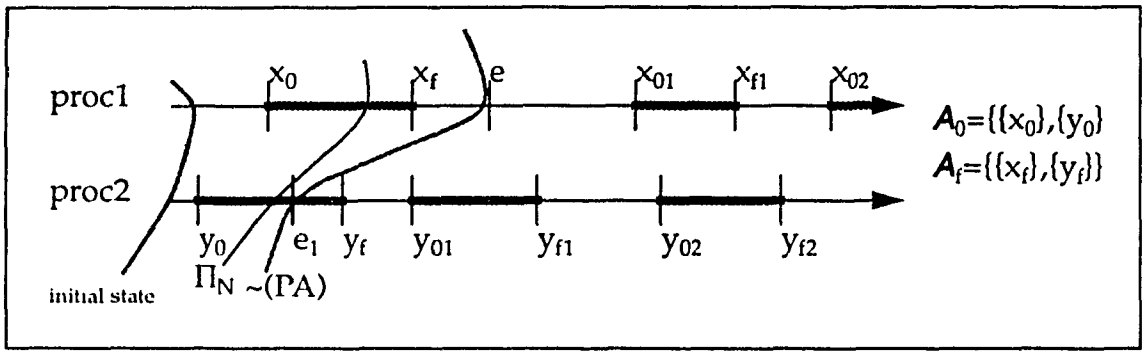

FIGURE 15: Determining the set *A*

From some initial state, *A* is determined. In Figure 15, the value of the variable x toggles between true and false at process 1; at process 2 the value of y toggles as well. The two processes communicate infrequently. Local predicates x and y hold respectively at x_0 and x_f, y_0 and y_f, and at all events in between; the time lines are shaded where the local predicates holds.

If (PA = P1.x OR P2.y), then from the initial state marked $\{x_0\},\{x_f\},\{y_0\},\{y_f\}$ are all elements of *A*, but $\{x_{01}\}$ is not because then $\Pi_{x0} \models\Rightarrow \Pi_N$ and $\Pi_N \models\Rightarrow \Pi_{x01}$. Since ~(PA) holds at Π_N, this is proscribed by the definition of *A*; therefore $\{x_{01}\}$ is not an element of *A*.

If (PA = P1.x AND P2.y), then from the initial state marked $\{x_0,y_0\}$ and $\{y_0,x_1\}$ are elements of ***A***. $\{x_{01}, y_1\}$ is not an element of ***A*** because then $\Pi_{\{x0,y0\}} \models\Rightarrow \Pi_N$ and $\Pi_N \models\Rightarrow \Pi_{\{x01, y1\}}$. Since ~(PA) holds at Π_N, this is proscribed by the definition of ***A***. therefore $\{x_{01}\}$ is not an element of ***A***.

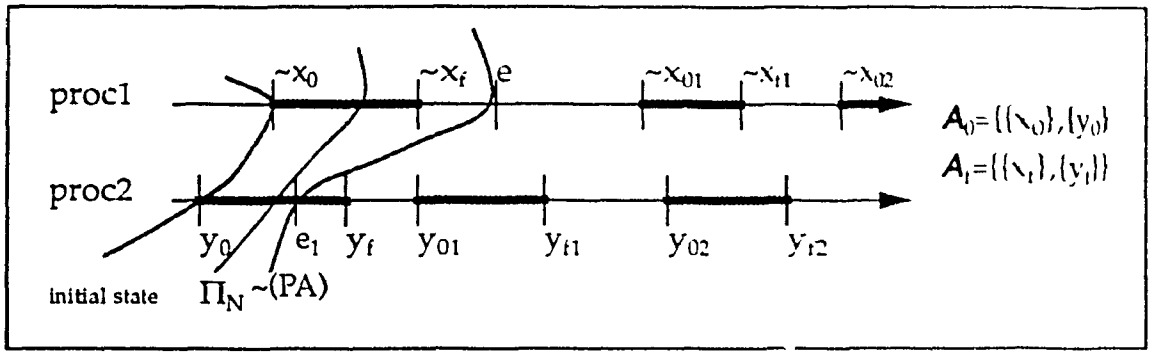

FIGURE 16: Determining the set ***A***- P1.x> P2.y

Assume that the values of x and y toggle between 0 and 1. Let x=0 at $(\sim x_0)$, $(\sim x_1)$ and the events in between. Then if (PA = P1.x> P2.y), then $\{x_0,y_0\}$ and $\{y_0,x_1\}$ are elements of ***A***. $\{x_{01}, y_1\}$ is not an element of ***A*** because then $\Pi_{\{x0,y0\}} \models\Rightarrow \Pi_N$ and $\Pi_N \models\Rightarrow \Pi_{\{x01, y1\}}$. Since ~(PA) holds at Π_N, this relationship with Π_N is proscribed by the definition of ***A***; therefore $\{x_{01}\}$ is not an element of ***A***.

4.2.4 Determining the set of initial states

In Figure 15, the set $\{x_0, y_{02}\}$ is not a member of the set ***A***, given the initial state marked in the example. Yet clearly $\{x_0, y_{02}\}$ is a state at which PA holds. To ensure that each instance of PA is recognizable, the relevant initial states must be used.

A set of states to be used as initial states as assumed in the definition of the set ***A***; this set will be called I and is developed below. Every minimal prefix of the computation at which PA holds is reachable from a member of this set of initial states.

Let Pred be the set of local predicates with which the predicate PA is defined.

Let $\overline{\text{Pred}}$ be the set of negations of local predicates in Pred, i.e., if

PA = ((proc1.x = 3 AND proc2.y = 4) OR proc3.z = 5) then

$Pred = \{ proc1.x = 3, proc2.y = 4, proc3.z = 5 \}$ and

$\overline{Pred} = \{ \neg(proc1.x = 3), (\neg(proc2.y = 4)), (\neg(proc3.z = 5)) \}$

Let $\overline{Pred}$ be ordered such that each local predicate is given a subscript, and let i, j be subscripts to the n predicates in $\overline{Pred}$:

$\overline{Pred} = \{ P_1, P_j, \ldots, P_n \}$

Let x be a label of the instances of predicates in $\overline{Pred}$. Recall that instances of local predicates are totally ordered: an instance of a local predicate is a totally ordered set of events, and such instances are totally ordered:

$$P_i^x = \{ \pi \subseteq E: e_0 < \ldots < e_f \wedge (\forall e \in P_i^x, \forall e' \in P_i^{x+1}: e < e') \}.$$

Thus P_i^x is the xth instance of local predicate P_i.

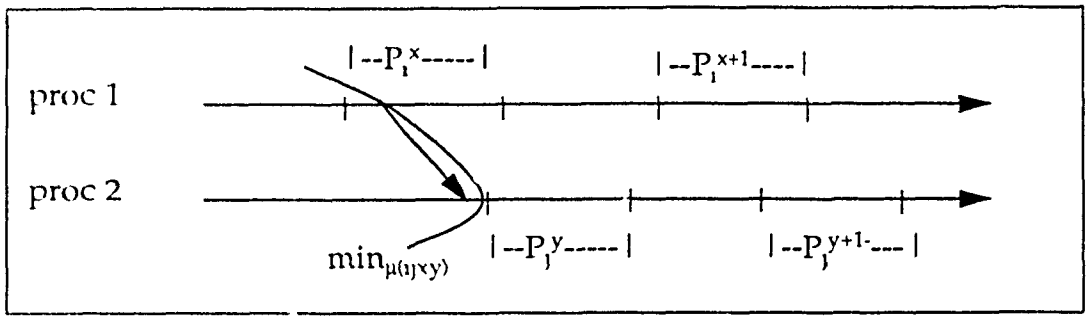

FIGURE 17: (P_i^x AND P_j^y)

Let μ be the set of the earliest elements of P_i^x and P_j^y that are concurrent:

$$\mu(i, j, x, y) = \{ e_1 \in P_i^x, e_2 \in P_j^y \mid (e_1 \| e_2) \wedge \neg\exists e'_2 \in P_j^y: (e'_2 < e_2) \wedge \neg\exists e'_1 \in P_i^x: (e'_1 < e_1) \}$$

That is, e_1 from P_i^x and e_2 from P_j^y such that e_1 and e_2 are concurrent, and there is no other event in P_i^x that happens-before e_1, and no event in P_j^y that happens before e_2.

Let min_μ be the minimal prefix found with respect to the set of events μ:

$$min_{\mu(i,j,x,y)} = \{\pi \subseteq E | \mu(i,j,x,y) \subseteq \pi \wedge \forall e \in lb(i,j,x,y): (e' < e) \Rightarrow e' \in \pi\}$$

and let χ be the set of desired initial states:

$$I = \{Q | \forall i \forall j \forall x \forall y : Q = min_{\mu(i,j,x,y)}\} \cup$$

{initial state of the computation}

Claims:

1. Every Q in is a consistent prefix.

   μ gives a set of concurrent events, and every Q is a minimal prefix with respect to this set of events: $min_{\mu(i,j,x,y)}$. Since the minimal prefix is by definition a consistent prefix, each Q so found is consistent.

2. χ contains the minimal prefix of each instance of each predicate in $\overline{Pred}$, i.e., for each of $P_i, P_j..., P_n$.

   For each P_i in $\overline{Pred}$, the minimal prefix of P_i^x is found where

- j=0 and y=0 (minimal prefix of P_i^1)
- j=y and y=x (for every subsequent P_i^x): where $P_i^x = P_j^y$, the set of concurrent events found is simply the event(s) at which P_i holds for the xth time.

3. χ contains the minimal prefix of each instance of all predicates in $\overline{Pred}$, i.e., for each of (P_i AND P_j AND..AND P_n).

   Where j<>i, μ finds the earliest concurrent events in each instance x and y of P_i^x and P_j^y.

4. I does not contain any other χ.

   Where j=i, the minimal χ of each P_i are found; where j<>i, the minimal χ of P_i and P_j and..and P_n are found. Since either j=i or j<>i these are all the χ found.

5. Every prefix of the computation at which PA holds is reachable from a prefix in χ.

   Let $q = \mu_{(i,j,x,y)}$. Assume that there is some A_f which is not reachable from any q:

   $\exists A \in A_f, \forall Q \in I: \neg(\Pi_q \sim> \Pi_A)$

   Then either $(\Pi_A \Rightarrow \Pi_q)$ or $(\Pi_A X \Pi_q)$, by Lemma 1.

If $(\Pi_A \models\Rightarrow \Pi_q)$ then PA held at the initial state of the computation; but if this is the case then Π_A must be reachable from the Q that is the initial state of the computation, so $(\Pi_A \models\Rightarrow \Pi_q)$ cannot hold.

If $(\Pi_A X \Pi_q)$ then this PA represents a transition from some ~PA. But if this is the case, then there is some Q that is a minimal prefix of ~PA, from which PA is reachable, so $(\Pi_A X \Pi_q)$ cannot hold.

Since neither $(\Pi_A \models\Rightarrow \Pi_q)$ nor $(\Pi_A X \Pi_q)$ can hold,

$\forall A \in \boldsymbol{A}_f, \exists Q \in I: (\Pi_q \sim> \Pi_A)$

4.3 Corresponding initial and final states

As was shown above, an instance of a predicate can have multiple initial and final states because the minimal and maximal χ are not necessarily unique. However, these initial and final states are related; as shown above in Lemma 5, no final state (maximal prefix) precedes an initial state (minimal prefix), and each final state is preceded by an initial state. In this section minimal and maximal χ of predicates are identified so that an initial state can be matched with its corresponding final state. This clarifies how predicates change value and will be used in the definition of simultaneity in Chapter 5 as well as in other proofs.

Let Pred be the set of predicates with which PA is defined: i.e., if

PA =((proc1.x AND proc2.y) OR proc3.z))

then Pred = {(proc1.x AND proc2.y), (proc3.z)}.

Let Pred be ordered such that each predicate is given a subscript; let i,j,k be these subscripts to the predicates in Pred:

Pred $=\{P_i, P_j, .., P_n\}$

Then for any element P_j in Pred, there are corresponding elements in $\boldsymbol{A}_0$ and $\boldsymbol{A}_f$, or if P_j is stable, $\boldsymbol{A}_0$ only. For example, if proc3.z holds and is unstable then there must be an element in $\boldsymbol{A}_0$ and an element in $\boldsymbol{A}_f$ corresponding to the minimal and maximal χ at which proc3.z holds.

$\forall P_i \in \text{Pred} \quad ((\exists A_i \in \boldsymbol{A}_0, \exists A'_i \in \boldsymbol{A}_f: ((\Pi_{Ai} \sim> \Pi_{A'i}) \vee A'_i = \{\varnothing\})) \vee \boldsymbol{A}_0 = \{\varnothing\})$

If proc3.z is stable, then there will be an element in $\boldsymbol{A}_0$ and an empty element in $\boldsymbol{A}_f$, corresponding to the minimal state at which proc3.z holds; there is no maximal state, so the $\boldsymbol{A}_f$ element is empty. (If PA consists of (P_j AND P_k), where P_j is stable and P_k is unstable, then the final state of P_k will determine the final states of PA.)

If $\boldsymbol{A}_0$ is empty, then PA never holds.

Let the minimal element of an instance of P_i be called A_{i0}; let the maximal element of an instance of P_i be called A_{if}; then these two elements will be *corresponding* for each instance of P_i.

Lemma 8: Let C be any non-empty set of concurrent events. Then if Π_C is reachable from some Π_{Ai} in $\boldsymbol{A}_0$, and some $\Pi_{A'i}$ in $\boldsymbol{A}_f$ is reachable from Π_C, then C is an element of $\boldsymbol{A}$ and PA holds at C.

$$\forall (C \subseteq E) \forall A_i \in \boldsymbol{A}_0 \forall A'_i i \in \boldsymbol{A}_f: \ ((\Pi_{Ai} \sim> \Pi_C \wedge \Pi_C \sim> \Pi_{A'i}) \Rightarrow (C \in \boldsymbol{A})) \qquad [24]$$

Assume the existence of Π_C as defined above, but assume that $\neg(C \in \boldsymbol{A})$. However, the definition of $\boldsymbol{A}$ states that there cannot be a Π_C such that $\Pi_{Ax} \sim> \Pi_C$ and $\Pi_C \sim> \Pi_{A'x}$ and not $(C \in \boldsymbol{A})$. If (~PA) holds at Π_C then either $\boldsymbol{A}_f$ is not an element of $\boldsymbol{A}_f$ or $\boldsymbol{A}_0$ is not an element of $\boldsymbol{A}_0$, or $\boldsymbol{A}$ includes two instances of PA; any of which contradicts the assumptions. Therefore PA must hold at Π_C and $(C \in \boldsymbol{A})$.

n

4.4 Summary

An instance of a predicate may have several initial and final states. Just as the relationships of two local predicates can be determined by considering their unique initial and final states, the relationship of one instance of a distributed predicate to another can be determined by considering all initial and final states.

An instance of a predicate PA is represented by a set of events $\boldsymbol{A}$. $\boldsymbol{A}_0$ is the set of sets of concurrent events at which PA initially holds; $\boldsymbol{A}_f$ is the set of final sets of concurrent events at which PA holds.

The minimal prefix of a single set of concurrent events A is $\min_A$, and is unique for a given set A. The maximal prefix of A is $\max_A$, and is also unique for A. The

set of initial states of an instance of PA is given by the set of minimal χ found with respect to the set $\boldsymbol{A}_0$, and the final states by the set of maximal χ found with respect to the set $\boldsymbol{A}_f$.

$\boldsymbol{A}$ is found with respect to some arbitrary 'initial' state of the computation; to find all instances the minimal necessary set of I is found. Using the definition of $\boldsymbol{A}$ with each state in I finds all instances of PA in the computation at minimal expense.

Certain reachability relationships hold between the initial and final states of any instance of a predicate; these are described in Lemma 5, Lemma 6, and Lemma 7. Each predicate P_i in a distributed predicate PA has *corresponding* initial and final states. This relation will be used in proofs and in defining simultaneity of predicates in the next chapter.

Chapter 5 : Relationships between distributed predicates

5.0 Non-atomic actions and distributed predicates

Lamport developed the notion of manipulating non-atomic events so that inherently non-atomic operations can be used to construct atomic operations, such as the construction of an atomic register. Lamport specifies a concept of simultaneity among non-atomic actions **[Lam86a]**. However, this notion is insufficient for the specification of simultaneity among distributed predicates.

Non-atomic actions as specified by Lamport are not explicitly spatially distributed; the initial and final points of such an action are assumed to be unique. Thus initial and final points may have the usual relationships among events; one precedes the other, or neither precedes the other and they are concurrent. Simultaneity among Lamport's non-atomic actions is given by 'not all of one action precedes all of another'.

Distributed predicates, by contrast, are spatially distributed and do not necessarily have unique initial and final points. Because of this non-uniqueness, the possible relationships between the final state of one distributed predicate and the initial state of another is not simply one of precedence or concurrency as it is with non-atomic actions; rather, reachability semantics are appropriate and the relationship is one of ($\|, \not\Rightarrow, X$). The notion of simultaneity among distributed predicates is given by 'the final event(s) of each distributed predicate do not precede the initial event(s) of another'. The contribution of this section is to specify a notion of simultaneity among distributed predicates, as well as specifying the other relationships that may hold between distributed predicates.

Because distributed predicates are spatially distributed and do not necessarily have unique initial and final points, change in value is also spatially distributed. This mechanism of change is captured by the logic that follows.

In Section 5.1 a brief example is given pulling together many of the concepts that have been presented. A brief discussion of the problem of characterizing and detecting simultaneity follows in Section 5.2. The primitive concept that one predicate may affect another is presented in Section 5.3. Simultaneity and other

possible relationships between instance of predicates, based on this primitive notion of one instance affecting another, are also presented along with some brief examples and figures. Proofs and some lemmas on the relationships between the initial and final states of two predicates are presented in Section 5.4. Safety operators based on the relations developed in this chapter are constructed in Section 5.5, and finally a summary is given in Section 5.6.

5.1 Example: initial and final states in distributed predicates

To pull together many of the concepts that have been presented, a brief example is given.

Let x, y and z be boolean variables of processes proc1, proc2, and proc3 respectively.

Let a predicate PA =((proc1.x AND proc2.y) OR proc3.z)).

Let the events (marked in the figure below) at which PA first holds be (x_0, y_0) and (z_0); that is, x_0 is the event x:= true, y_0 the event y:= true, z_0 the event z:= true.

Let the maximal events at which PA holds be x_f, y_f, and z_f.

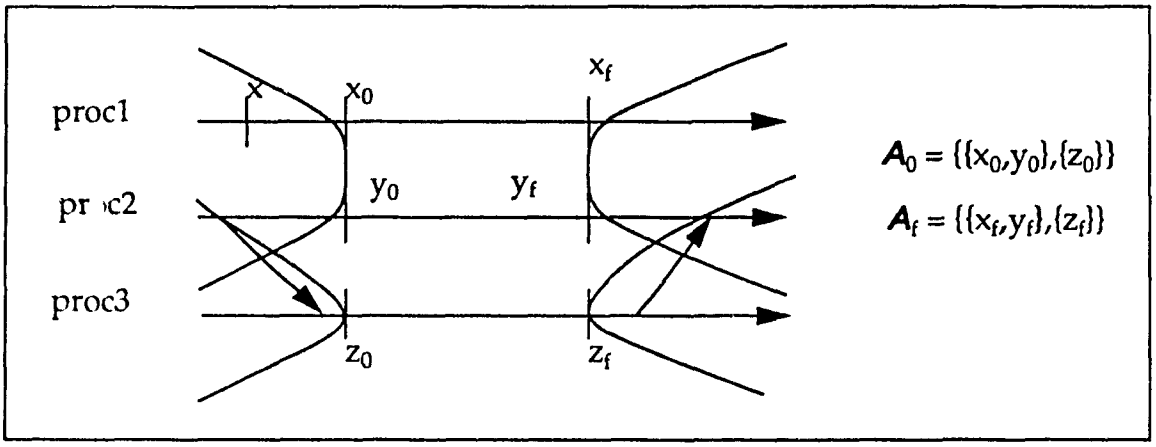

FIGURE 18: A_0 and A_f

Then $\boldsymbol{A}_0 = \{\{x_0, y_0\},\{z_0\}\}$ and $\boldsymbol{A}_f = \{\{x_f,y_f\},\{z_f\}\}$.

PA holds at every $\min_A$ in $\mathbf{min}_{A0}$, at every $\max_A$ in $\mathbf{max}_{Af}$; it holds at any prefix H_A which is reachable from a minimal prefix of PA (some element of $\mathbf{min}_{A0}$), and from which a maximal prefix of PA (some element of $\mathbf{max}_{Af}$) is reachable.

Thus where a predicate holds is defined using $\boldsymbol{A}_0$, $\boldsymbol{A}_f$, and the reachability relation.

$\boldsymbol{A}_0$ and $\boldsymbol{A}_f$ will be used to determine whether two conditions, as expressed by predicates, can affect each other.

5.2 Simultaneity and the state explosion problem

Two predicates are said to hold simultaneously if there is any global state in which they hold. Detecting simultaneity using local states at which a predicate holds can easily lead to a state explosion problem because all states in the execution have to be checked. Using the minimal and maximal states at which predicates holds, or rather using the events at which the states are found, permits the detection of simultaneity without examining every global state. The detection of simultaneity using this approach is thus feasible and relatively efficient.

Simultaneity is one possible relationship between predicates; there are other possible relationships that can be expressed and detected using this approach. These will be discussed in the next section. Defining the possible relationships between two predicates makes it possible to specify safety operators. This is discussed in Section 5.5.

5.3 'Affects'

It was shown in Chapter 3 that the relationship between any two global states falls into two classes; one or both states is reachable from the other, or neither state is reachable from the other. Analogously, the relationship between two predicates also falls into two classes; at least one predicate affects the other, or neither predicate can affect the other. A predicate PA is said to *affect* a predicate PB iff

1. there is some global state in which both PA and PB hold

or

2. there is some sequential or causal relationship between PA and PB; some prefix at which PB holds is a proper subset of some prefix at which PA holds.

PA affects PB:

PA 'starts before, or at least at the same time as', PB finishes **[Lam86a]**; some state at which PB holds is reachable from some state at which PA holds.

$$PA \text{-->} PB \text{ iff } \exists A \in \boldsymbol{A}_0, \exists B \in \boldsymbol{B}_f\colon \Pi_A \sim\!> \Pi_B \qquad [25]$$

The operators that define the relationships between two predicates are based on the relationships between the minimal and maximal states at which the prefaces hold. The same operators are used throughout; where the operands are predicates, the operator indicates 'affect'; where the operand is a prefix, the operator indicates 'reachability'.

Does PA affect PB? Does PB affect PA?

1. Yes, yes; PB affects PA, and PA affects PB.
2. No, no; PB does not affect PA, and PA does not affect PB.
3. Yes, no; PB affects PA, but PA does not affect PB.
4. No, yes; PB does not affect PA, and PA affects PB.

1. PA affects PB, and PB affects PA:
   $PA \text{<-->} PB \text{ iff } (PA \text{-->} PB) \wedge (PB \text{-->} PA)$

- some state in which PB holds is reachable from a state in which PA holds;
- some state in which PA holds is reachable from a state in which PB holds.

Consider where a predicate PA = ((proc4 AND proc3) OR (proc2 AND proc1), and a predicate PB = (proc3 AND proc2). If the following situation occurs, then PA<-->PB:

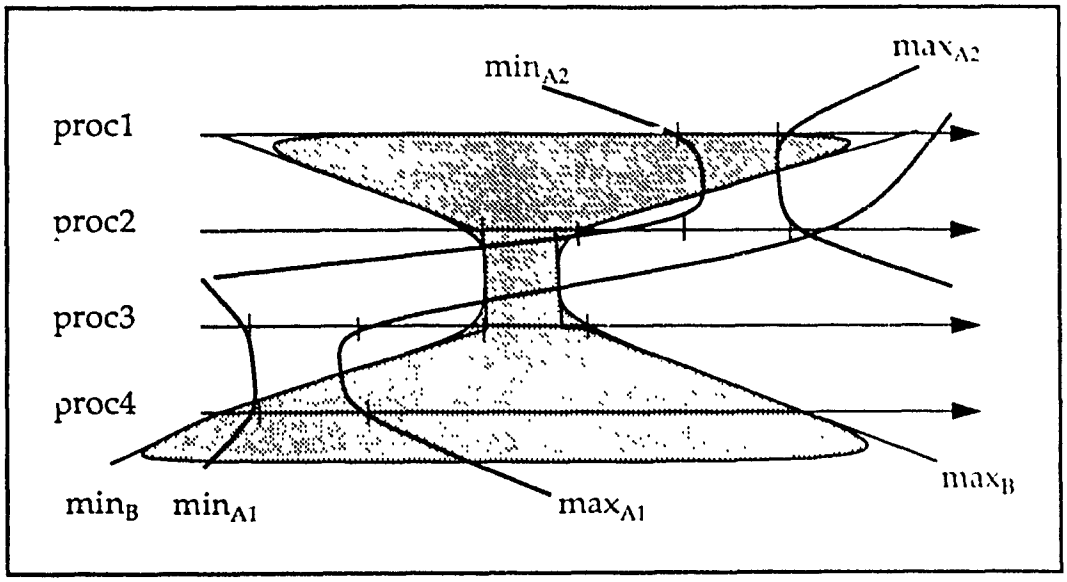

FIGURE 19: PA<-->PB

Notice that there is no global state in which both PA and PB hold in the above example, although each predicate affects the other.

2. PA does not affect PB, and PB does not affect PA. This means that

- no state in which PB holds is reachable from a state in which PA holds
- no state in which PA holds is reachable from a state in which PB holds

PA X PB iff $\neg$(PA-->PB) $\wedge \neg$(PB-->PA) [26]

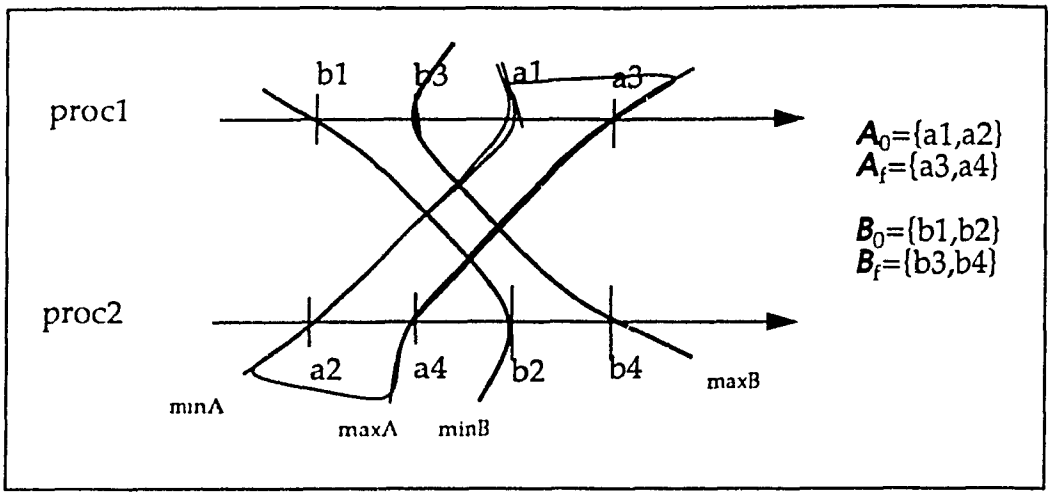

FIGURE 20: PA X PB

3. PA affects PB, but PB does not affect PA:

$PA \models\Rightarrow PB$ iff $PA \dashrightarrow PB \wedge \neg(PB \dashrightarrow PA)$ [27]

This operator indicates that:

- there does not exist any global state in which both PA and PB hold;
- some state in which PB holds is reachable from some state in which PA holds;
- no state in which PA finishes is reachable from a state in which PB starts.

The figures below illustrate two possible configurations of $PA \models\Rightarrow PB$:

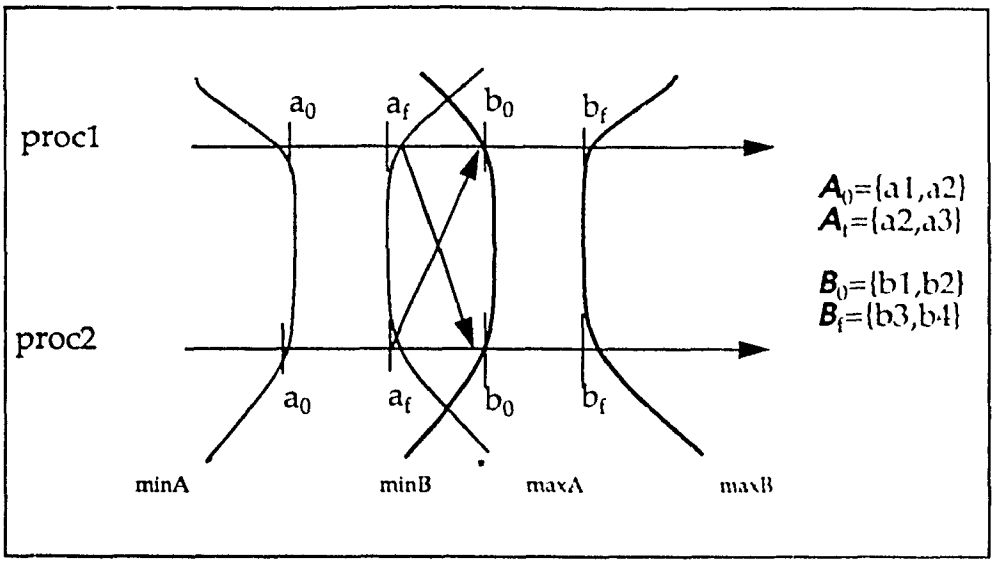

FIGURE 21: PA |⇒PB (PA |→PB)

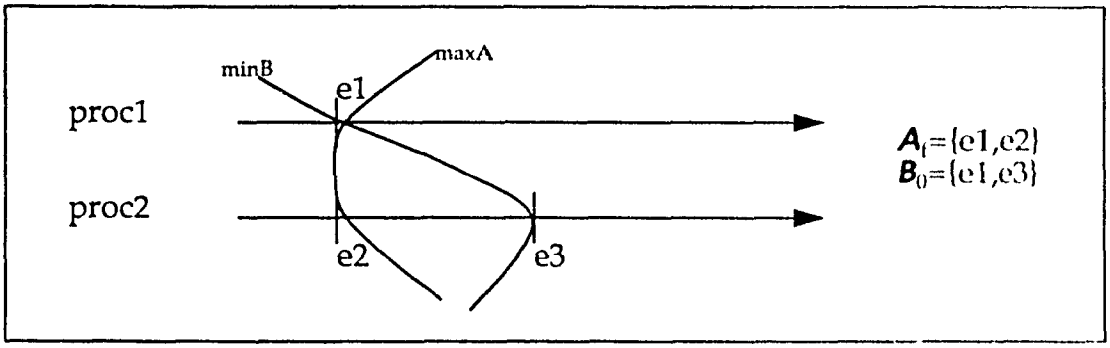

FIGURE 22: PA |⇒PB

Subcases of each of (<-->, |⇒,X) can be defined where utility warrants. Two such cases are defined:

1a) Subset of PA<-->PB: PA sim PB

PA sim PB iff $\exists A_i \in \boldsymbol{A}_0, \exists B'_j \in \boldsymbol{B}_f, \exists B_j \in \boldsymbol{B}_0, \exists A'_i \in \boldsymbol{A}_f$: $(\Pi_{A_i} \sim> \Pi_{B'_j}) \wedge (\Pi_{B_j} \sim> \Pi_{A'_i})$

There is some state in which both PA and PB hold. This could be expressed as $\exists A \in \boldsymbol{A}, \exists B \in \boldsymbol{B}$: $(\Pi_A \| \Pi_A)$, but looking for this would lead to a state explosion problem: Consider the simple case of P and Q, local predicates that hold in local instances P_x and P_y. Then a brute-force method of determining if there exists some global state in which both P and Q hold is to compare each event in P_x with each event in P_y. If two events are incomparable, then there exists a global state in which both P and Q hold. The cost of this is $|P_x| \cdot |Q_y|$. By contrast, using $\boldsymbol{A}_0$ and $\boldsymbol{A}_f$ just two comparisons are required to determine if P AND Q holds. Generalizing to *k* instances of P and *m* instances of Q, the cost of the brute force method is $(|P_x| \cdot |Q_y|)^{km}$. Using $\boldsymbol{A}_0$ and $\boldsymbol{A}_f$ costs only $2km$.

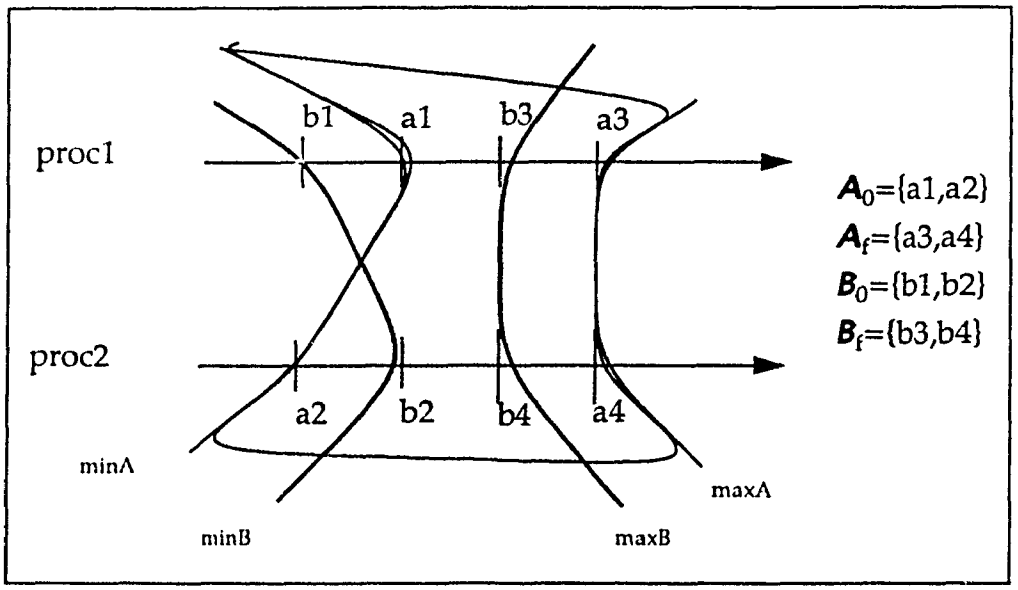

FIGURE 23: PA sim PB

3a) Special case of PA $|\Rightarrow$PB, where all events in A precede all events in B:

$$PA \mid\!\rightarrow PB \text{ iff } \forall A \in A_f, \forall B \in B_0 : \Pi_A \mid\!\rightarrow \Pi_B \qquad [28]$$

depicted in Figure 21.

$\mid\!\rightarrow$ is irreflexive and transitive: if A and B are non-empty sets of concurrent events then

if $\Pi_A \mid\!\rightarrow \Pi_B$ then $\sim(\Pi_B \mid\!\rightarrow \Pi_A)$;

if $\Pi_A \mid\!\!\rightarrow \Pi_B$ and $\Pi_B \mid\!\!\rightarrow \Pi_C$ then $\Pi_A \mid\!\!\rightarrow \Pi_C$

Let PA, PB, PC be predicates:

1. $\mid\!\!\rightarrow$ is irreflexive and transitive:
   if PA $\mid\!\!\rightarrow$PB then ~(PB $\mid\!\!\rightarrow$PA)
   if PA $\mid\!\!\rightarrow$PB and PB $\mid\!\!\rightarrow$PC then PA $\mid\!\!\rightarrow$PC
2. <-->, sim and X are symmetric:
   if PA sim PB then PB sim PA
   if PA X PB then PB X PA
   if PA<-->PB then PB<-->PA
3. <--> and sim are reflexive
   PA <-->PA
   PA sim PA
4. (PA **R** PB) is treated as a predicate which is either true or false; thus it cannot in general be treated as a function that yields a predicate.
   Where **R** is any of (<-->, X, $\mid\!\!\Rightarrow$)
   (PA **R** PB) **R** (PC **R** PD)
   is undefined.
5. If PA sim PB then PA<-->PB.
   If PA $\mid\!\!\rightarrow$PB then PA $\mid\!\!\Rightarrow$PB.

Examples using these operators are given in Chapter 7.

5.4 Initial and final states of two predicates: some lemmas

To help show that the definitions of (<-->, $\mid\!\!\Rightarrow$,X) support what is claimed for them, recall the definitions of *corresponding* elements of $\boldsymbol{A}_0$ and $\boldsymbol{A}_f$.

Lemma 9:

This lemma will be divided into two parts; Part Two is just the contrapositive of Part One.

Part One:

$$\forall B \in \boldsymbol{B}, \forall A_i \in \boldsymbol{A}_0, \forall A'_i \in \boldsymbol{A}_f\colon\ ((\Pi_B \sim\!> \Pi_{Ai}) \Rightarrow (\Pi_B \sim\!> \Pi_{A'i})) \wedge ((\Pi_{A'i} \sim\!> \Pi_B) \Rightarrow (\Pi_{Ai} \sim\!> \Pi_B)) \qquad [29]$$

Part One of the Lemma says that

-if any state in which PB holds can reach an initial state *i* of PA, then it can also reach the corresponding final state *i* of PA, if it exists;

-if a final state *i* of PA can reach any state in which PB holds, then the corresponding initial state *i* of PA can reach it too.

Part Two:

$$\forall B \in \boldsymbol{B}, \forall A_i \in \boldsymbol{A}_0, \forall A'_i \in \boldsymbol{A}_f: (\neg(\Pi_B \sim> \Pi_{A'i}) \Rightarrow \neg(\Pi_B \sim> \Pi_{Ai})) \wedge (\neg(\Pi_{Ai} \sim> \Pi_B) \Rightarrow \neg(\Pi_{A'i} \sim> \Pi_B)) \quad [30]$$

Part Two of the Lemma says that

-if a state in which PB holds cannot reach a final state *i* of PA, then it cannot reach the corresponding initial state *i* of PA either;

-if an initial state *i* of PA cannot reach a state in which PB holds, then the corresponding final state *i* cannot reach this state either.

PA <--> PB:

Claim: some state in which PA holds is reachable from some state in which PB holds, and some state in which PB holds is reachable from some state in which PA holds.

This follows directly from the definition of '<–>'.

PA sim PB:

Claim: if PA sim PB then there exists some state in which both PA and PB hold.

PA sim PB iff $\exists A_i \in \boldsymbol{A}_0, \exists B'_j \in \boldsymbol{B}_f, \exists B_j \in \boldsymbol{B}_0, \exists A'_i \in \boldsymbol{A}_f: (\Pi_{Ai} \sim> \Pi_{B'j}) \wedge (\Pi_{Bj} \sim> \Pi_{A'i})$

It is given that $(\Pi_{Ai} \sim> \Pi_{B'j}) \wedge (\Pi_{Bj} \sim> \Pi_{A'i})$. Either $(\Pi_{Ai} \sim> \Pi_{Bj})$ or $\neg(\Pi_{Ai} \sim> \Pi_{Bj})$.

If $(\Pi_{Ai} \sim> \Pi_{Bj})$ and $(\Pi_{Bj} \sim> \Pi_{A'i})$, then PA must hold at Π_{Bj} by Lemma 8, and so both PA and PB hold at Π_B.

If $\neg(\Pi_{Ai} \sim> \Pi_{Bj})$, then either $(\Pi_{Bj} \models\Rightarrow \Pi_{A1})$ or $(\Pi_{Bj} X \Pi_{A1})$. If $(\Pi_{Bj} \models\Rightarrow \Pi_{A1})$, then $(\Pi_{Bj} \sim> \Pi_{Ai}) \wedge (\Pi_{Ai} \sim> \Pi_{B'j})$, so PB must hold at Π_{Ai} by Lemma 8 and so both PA and PB hold at Π_{Ai}. If $(\Pi_{Bj} X \Pi_{Ai})$, then there must exist a some set of concurrent events G such that $(\Pi_{Ai} \sim> \Pi_G) \wedge (\Pi_G \sim> \Pi_{A'1})$ and $(\Pi_{B1} \sim> \Pi_G) \wedge (\Pi_G \sim> \Pi_{B'1})$

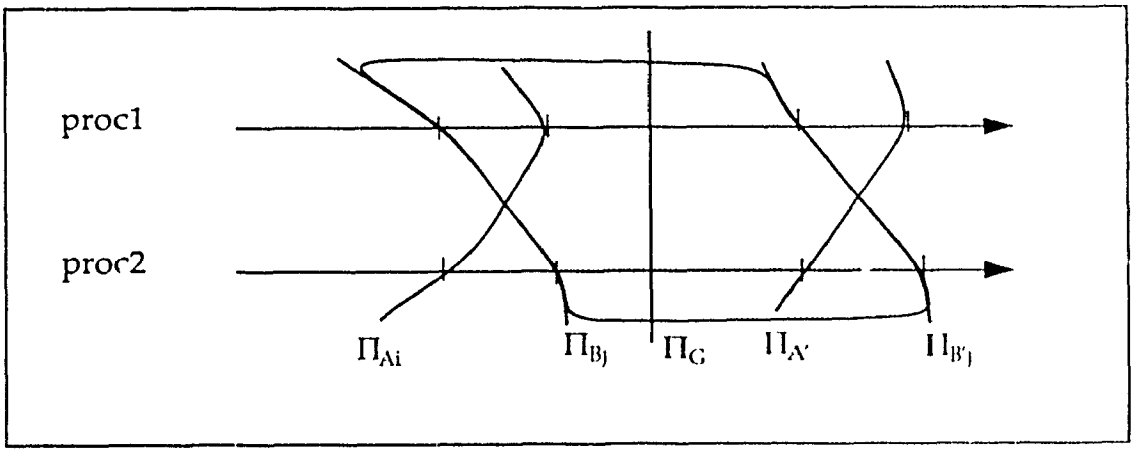

FIGURE 24: $(\Pi_{Bj} X \Pi_{A1})$

Since $(\Pi_A \sim> \Pi_{A'})$, then some $\pi_{A'}$ is reachable from the minimal prefix of A, by definition of '~>' and Lemma 3:

$\Pi_A \sim> \Pi_B$ iff $\exists \pi_A \in \Pi_A, \exists \pi_B \in \Pi_B : \pi_A \subseteq \pi_B$ and $min_A \subseteq max_{A'}$.

Therefore min_A is a subset of some $\pi_{A'}$ in $\Pi_{A'}$.

Since $(\Pi_B \sim> \Pi_{B'})$, then min_B is a subset of some $\pi_{B'}$ by similar reasoning.

Let $\pi_G \subseteq min_A \cup min_B$. Then π_G is a subset of $\pi_{A'}$ because both min_A and min_B are subsets of $\pi_{A'}$. Then $min_A \subseteq \pi_G \subseteq \pi_{A'}$: and both PA and PB hold at π_G by Lemma 8. This completes the proof.

n

PA $\models\Rightarrow$ PB iff $(\exists A \in \boldsymbol{A}_0, \exists B' \in \boldsymbol{B}_f: (\Pi_A \sim> \Pi_{B'})) \wedge$

$\neg(\exists B \in \boldsymbol{B}_0, \exists A \in \boldsymbol{A}_f: (\Pi_B \sim> \Pi_{A'}))$

Claim 1: There is no prefix at which both PA and PB hold, for a given ***A*** and ***B***

Claim 2: No prefix at which PA holds is reachable from a prefix at which PB holds, for a given ***A*** and ***B***.

The second part of the definition says that
$\forall B \in \boldsymbol{B}_0, \forall A \in \boldsymbol{A}_f$: $(\Pi_A \mid\!\Rightarrow \Pi_B) \vee (\Pi_A X \Pi_B)$, since $\neg(\Pi_B \sim\!> \Pi_A)$ and at least one of $(\mid\!\Rightarrow, X, \mid\mid)$ must hold by Lemma 1.

Let us choose a P_j in PA and a P_k in PB. Then for all j and all k,

$$\forall P_j, \forall P_k, \forall A \in \boldsymbol{A}_0, \forall B' \in \boldsymbol{B}_f, \forall B \in \boldsymbol{B}_0, \forall A' \in \boldsymbol{A}_f:$$
$$((\Pi_{jA'} \mid\!\Rightarrow \Pi_{kB}) \Rightarrow (\Pi_{jA} \mid\!\Rightarrow \Pi_{kB})) \wedge$$
$$((\Pi_{jA'} X \Pi_{kB}) \Rightarrow ((\Pi_{jA} \mid\!\Rightarrow \Pi_{kB}) \vee (\Pi_{jA} X \Pi_{kB})))$$

Therefore, $\forall B \in \boldsymbol{B}_0, \forall A \in \boldsymbol{A}_0, \forall A' \in \boldsymbol{A}_f$: $\neg(\Pi_B \sim\!> \Pi_A) \wedge \neg(\Pi_B \sim\!> \Pi_A)$ from the above and from Lemma 1.

Therefore no prefix at which PA holds is reachable from a prefix at which PB holds, satisfying Claim 1.

Since no prefix at which PA holds is reachable from a prefix at which PB holds, there cannot be an A in ***A***, and a B in ***B*** such that $\Pi_A \parallel \Pi_B$. Thus if $(\Pi_A \mid\!\Rightarrow \Pi_B)$ then there is no prefix at which both PA and PB hold, satisfying Claim 2.

$$PAXPB \text{ iff } \neg(\exists A \in \boldsymbol{A}_0, \exists B' \in \boldsymbol{B}_f: (\Pi_A \sim\!> \Pi_{B'})) \wedge$$
$$\neg(\exists B \in \boldsymbol{B}_0, \exists A' \in \boldsymbol{A}_f: (\Pi_B \sim\!> \Pi_{A'}))$$

Claim 1: There is no prefix at which both PA and PB hold, for a given ***A*** and ***B***.

Claim 2: No prefix at which PA holds is reachable from a prefix at which PB holds, and no prefix at which PB holds is reachable from a prefix at which PA holds, for a given ***A*** and ***B***.

Let P_j be an instance of PA; let P_k be an instance of PB. Then

$\forall P_j, \forall P_k, \forall A \in \boldsymbol{A}_0, \forall B' \in \boldsymbol{B}_f, \forall B \in \boldsymbol{B}_0, \forall A' \in \boldsymbol{A}_f$:
$(\neg(\Pi_{Bk} \sim\!> \Pi_{jA'}) \Rightarrow \neg(\Pi_{Bk} \sim\!> \Pi_{jA}))$ by Lemma 9, so there is no element of $\boldsymbol{B}$ and no element of $\boldsymbol{A}$ such that $(\pi_B \sim\!> \pi_A)$;

$(\neg(\Pi_{B'k} \sim\!> \Pi_{Aj}) \Rightarrow \neg(\Pi_{B'k} \sim\!> \Pi_{A'j}))$ by Lemma 9, so there is no element of $\boldsymbol{A}$ and no element of $\boldsymbol{B}$ such that $(\pi_A \sim\!> \pi_B)$;

which satisfies Claim 2; and since Claim 2 holds, there can be no $(\pi_A \| \pi_B)$, satisfying Claim 1.

Lemma 10: Exactly one of (<—>, X, |⇒) hold between any two predicates.

If PA |⇒PB, then PA <—> PB does not hold by Claim 1 for PA |⇒PB; and PA X PB does not hold by Claim 2 of PA X PB.

If PA <—> PB, then PA |⇒PB does not hold by Claim 1 for PA |⇒PB. PA X PB does not hold by Claim 1 for PA X PB.

If PA X PB, then PA <—> PB does not hold by Claim 1 and PA |⇒PB hold by Claim 2 for PA X PB.

If PA<—>PB does not hold, then neither of PA sim PB and PA |→PB hold. (By properties of sim and |→.)

5.5 'Unless', and other safety operators

The contribution of this section is that an interpretation of Chandy's *unless* **[Cha88]** is given in terms of distributed predicates. Chandy's *unless* is defined in the statement space; there is no control flow of underlying events. In addition, the statement *s* in Chandy's definition is required to be unique. By contrast, the *unless* specified here is defined in the event space. Because a distributed predicate may be spatially distributed, the change in value of a predicate is not necessarily traceable to a unique event. The concept of *unless* has been adapted to accommodate the characteristics of non-uniqueness and spatial distribution in distributed predicates.

We define 'unless' as follows:

if a predicate PA holds at Π_A, then after the execution of any event e, either PA continues to hold or PB holds. Let PA(Π_A) mean that predicate PA holds at PA:

$$PA(\Pi_A) \text{ implies } (PA \vee PB)(\Pi_A \cup e)$$

where e is any e in E as defined in Chapter 3.

The transitions of interest are thus those from PA to ~PA. The definition constructed ensures that the initial states of ~PA are considered only when they are preceded by an instance of PA; the definition also ensures that all such initial states of ~PA are states at which PB holds. Thus, any transition from PA to ~PA which satisfies the definition satisfies 'unless' as defined above.

~PA may hold at the initial state of the computation; the initial state does not represent a transition from PA. To exclude the initial-state possibility, the definition requires that all initial states of ~PA be preceded by some final state of PA. This requirement guarantees that all initial states of ~PA considered represent transitions from PA and provides the first part of the definition.

Recall that if PA$\angle$PB, then PB holds PA holds at PB (Lemma 2).

Recall that if $(\Pi_{Bi} \mid\!\rightarrow \Pi_A \wedge \Pi_A \mid\!\rightarrow \Pi_{B'i})$ then PB holds at PA (Lemma 8).

Either PB 'overlaps' the initial state(s) of ~PA, or the initial or final states of PB are a subset of ~PA. Thus PB can be guaranteed to hold at all the initial states of ~PA if both of the above conditions are met. This provides the second and third parts of the definition:

$$\{\forall A \in \neg \boldsymbol{A}_0, \exists B \in \boldsymbol{B}_f, \exists B' \in \boldsymbol{B}_0, \exists A' \in A_f:$$
$$(\Pi_{A'} \mid\!\Rightarrow \Pi_A) \wedge$$
$$((\Pi_B \angle \Pi_A) \vee (\Pi_{B'} \angle \Pi_A) \vee$$
$$(\Pi_B \mid\!\rightarrow \Pi_A \wedge \Pi_A \mid\!\rightarrow \Pi_{B'}))$$

1. $(\Pi_{A'} \mid\!\Rightarrow \Pi_A)$ guarantees that every minimal state of ~PA represents a transition from PA;

2. $(\Pi_B \angle \Pi_A) \vee (\Pi_{B'} \angle \Pi_A)$ covers the case where the initial or final states of PB are a subset of the initial states of ~PA;
3. $(\Pi_B \mid\!\!\rightarrow \Pi_A \wedge \Pi_A \mid\!\!\rightarrow \Pi_{B'})$ covers the case in which every initial state of ~PA is reachable from every initial state of PB and can reach some final state of PB.

Stable and invariant properties of programs are expressed in terms of 'unless'.

Liveness properties can also be expressed in terms of 'unless'. For non-terminating programs, violation of liveness is only detectable at infinity. For terminating programs, termination before satisfaction of liveness can be classed as a safety violation. In either case, the satisfaction of progress requirements can be detected and reported. Liveness properties can be specified by using a predicate stating the termination condition; if a predicate expresses liveness, then it should occur before termination:

liveness predicate ~> termination predicate.

Several other operators can immediately be constructed using 'unless':

A predicate is *stable* if, once it holds, it continues to hold.

stable PA: PA unless PA

An *invariant* holds throughout the computation; it holds at the initial state of the computation and is stable.

invariant PA: (initial condition implies PA) and stable PA

PA as long as PB: (~PA unless (PB AND PA)) AND ((PB AND PA) unless ~PA)

That is, PA holds if and only if PB also holds; PB may hold regardless of the value of PA.

Examples using these safety operators are given in Chapter 7.

5.6 Summary

An instance of a predicate *affects* another instance of a predicate PB if it 'starts before, or at least at the same time as' PB finishes. In terms of reachability semantics, *affects* means that some state at which PB holds is reachable from a

state at which PA holds. If both predicates affect the other, then PA<--> PB. Simultaneity, PA sim PB, is a special case of PA<-->PB. If neither predicate affects the other, then PAXPB. If PA affects PB, but PB does not affect PA, then PA $|\Rightarrow$PB. Simultaneity between distributed predicates with non-unique initial and final states is defined using the *corresponding* initial and final states at which a predicate holds. This reduces the complexity of determining of PA sim PB to $2km$, for k events in each of m elements if $\boldsymbol{A}_0$, for two given instances of PA and PB.

These operators are used to construct *unless* and other safety operators *stable, invariant*, and *as long as*.

Chapter 6 : Expanded Predicate Logic

An assertion yields a boolean-valued predicate. A boolean-valued predicate is constructed by naming a non-boolean valued variable (nbv), a comparative operator, and a number or a non-boolean variable; or it names a boolean variable directly:

a = ((nbv) + (> | < | =) + ((nbv) | number) | bv

A boolean assertion uses assertions to build more complex boolean-valued expressions. It is constructed by naming an assertion, a logical operator, and another boolean assertion or assertion:

ba = (a | ba) + (AND | OR | NOT) + (a | ba)

Let ba be a boolean assertion as defined above. Then another boolean assertion BA can be defined using the new operators:

BA = (ba + (—> | <—> |⇒ | sim | X | unless | as long as | stable) | ba)

Chapter 7 : Applications and examples

7.0 Breakpoints

One of the fundamental facilities required in interactive debugging is the capability to set and detect breakpoints. Breakpoint conditions may be expressed as predicates on the system state. The user wants to examine system state where a breakpoint condition is satisfied. To satisfy this desire the system should halt in some consistent state in which the predicate holds. Let this predicate be called the breakpoint predicate, and let the state at which the system halts be called the breakpoint.

7.0.1 Minimal and Maximal Breakpoints for Debugging

In sequential systems, all events happen either before or after a breakpoint. Events are totally ordered, and only events that occurred earlier in time can affect events that occurred later in time; thus, at the breakpoint, only events that can have had a causal effect on the predicate have occurred. This is not necessarily the case for distributed systems, as events are only partially rather than totally ordered. Previous work in the area of breakpoints in distributed systems, with the exception of **[Fow90]**, detects a distributed breakpoint at *any* global state at which the predicate holds. Clearly this is unsatisfactory, as the breakpoint found should make explicit rather than obscure the causal relationships of events.

To use the notation developed earlier, if the desired breakpoint is some state at which PA holds, then the breakpoint should be explicitly some minimal or

maximal prefix of A, rather than some arbitrary prefix of A, or even some prefix of an element of A_0.

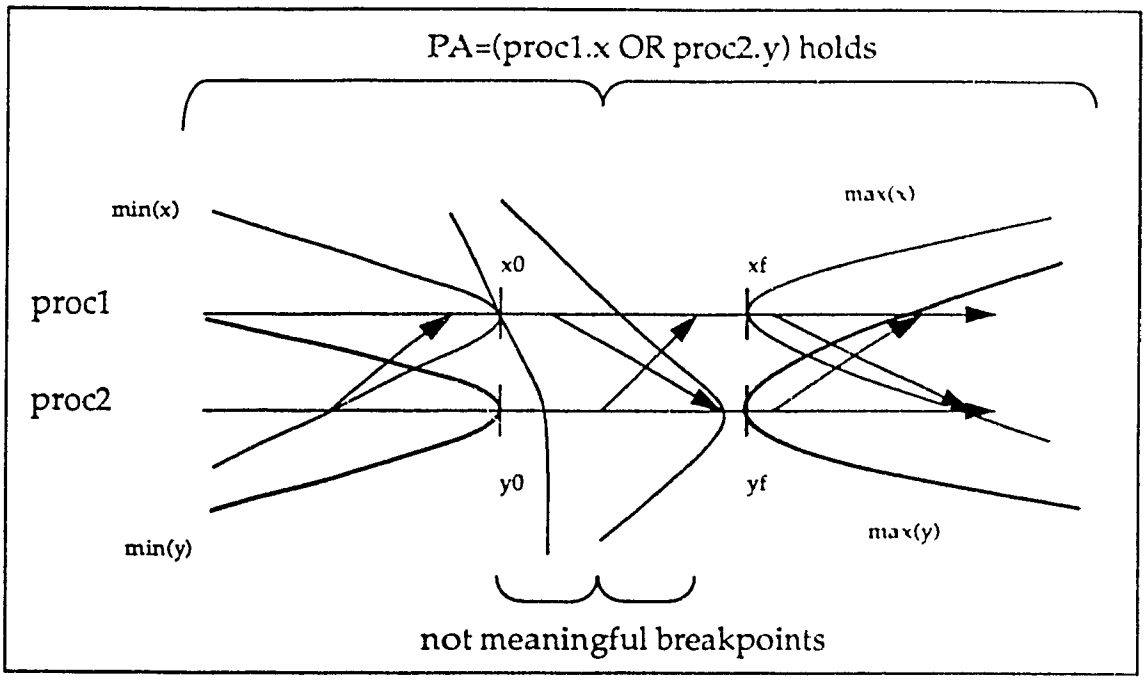

FIGURE 25: Meaningful breakpoints

Since PA holds over many global states, an additional requirement is added; a breakpoint is *meaningful* if the prefix found includes or excludes events in such a way that the causal relationship between the events and the value of the breakpoint predicate are unambiguous.

A minimal prefix satisfying the breakpoint predicate will be called the causal breakpoint; a maximal prefix satisfying the breakpoint predicate will be called the maximal breakpoint.

A causal breakpoint corresponds to a sequential breakpoint in that all events included in the breakpoint can have had a causal effect on the state of the computation at the breakpoint. The maximal breakpoint is found as far along as the computation can proceed while the breakpoint predicate still holds; all states of the computation after the maximal breakpoint will be affected by events included in the maximal breakpoint. The events included in any global states that occur in between these two breakpoints have an undefined relationship to the predicate; events included in these global states may or may not be causally

related to the state of the computation at the breakpoint. Thus, these two breakpoints provide global states that are well-defined with respect to the causal relationships of events inherent in distributed systems.

7.0.2 Breakpoint Predicates

Predicates that are intended to designate breakpoints and those are intended for monitoring differ only in how they are treated during detection. With the latter, the desired result is determination of satisfaction or violation of the predicate; with the former, the desired result is halting at a particular global state at which the predicate is satisfied.

A breakpoint predicate is a predicate qualified by a statement as to where the breakpoint (halt state) should occur if the predicate is satisfied. Let predicates that involve only conjunction or disjunction of local predicates be called ba-type predicates; recall that

a = ((non-boolean variable | number) + (> | < | =) + ((non-boolean variable) | number) | boolean variable

ba = (a | ba) + (AND | OR | NOT) + (a | ba). For this type of predicate

let min_p(ba) be any minimal prefix of a ba-type predicate,

let maxp(ba) be any maximal prefix of a ba-type predicate,

Then a breakpoint predicate is

bp = min_p(ba) | max_p(ba)

Recall from Chapter 6 that predicates using the relationships between distributed predicates or using safety operators are called BA-type predicates:

BA = (ba + (—> | <—> |⇒ | sim | X | unless | as long as | stable) | ba).

For these more complex BA-type predicates the user has the choice of several possible interesting points at which to halt the computation. If bpp is a breakpoint predicate of a BA-type predicate, then a possible semantics is

bpp(PA sim PB) = ba(PA AND PB)

bpp(PA X PB) = ba(PA) | ba(PB)

bpp(PA |⇒PB) = min(PB)

bpp(PA unless PB) = min_p(PB) | max_p(PA)

bpp(PA as long as PB) = bp(PA)

7.1 Examples

It is difficult to give convincing examples for breakpointing; their use is always particular to the program, system, and bug being examined. Examples giving programs with built-in errors seem contrived, and real-life examples are too complex to serve as general examples. Here some examples will be given that show the use of breakpoints in the investigation of program behavior. Breakpoint predicates specifying violations of a program requirement can also be used to find the minimal or maximal states of the computation at which a program is in error with respect to that particular requirement.

7.1.1 Using 'ba' type predicates

A control system consists of two control processes (P1, P2) each of which controls a physical component, and four dedicated processes (i_1, i_2, i_3, i_4) monitoring conditions that affect those components. P1 and P2 receive input and calculate the proper positioning of their component as a function of this input. Processes P1 and P2 both receive inputs i_1 and i_2; P1 also receives i_3 and P2 receives i_4. Using this input, P1 calculates a function fn_1, and P2 calculates a function fn_2. The relation between fn_1 and fn_2 is given by $fn_3(fn_1, fn_2)$, where $k<fn_3$. Where $k<fn_3<m$ an alert condition is signaled. To examine how the processes behave under alert conditions, a breakpoint predicate is set where $fn_3>k$. Detecting the minimal breakpoint of this predicate will capture the values at the input processes that resulted in $fn_3>k$; this is turn will allow an analysis of which input components or combination of components gave rise to these conditions.

```
P1::repeat
        receive(i1,i2,i3)
        x:= fn1(i1,i2,i3)
    until(false)
```

P2::**repeat**

receive(i_1,i_2,i_4)

y:= fn_2(i_1,i_2,i_4)

until(false)

Let the predicate PA = min(fn_3(x, y) >*k*) be the breakpoint predicate. Then input processes i_1..i_4 will break immediately after sending input values to P1 and P2. Thus the input processes and P1 and P2 are left in the state whose result was fn_3>*k*.

7.1.2 Discovering dependencies in resource-sharing

Deadlock is the mutual blocking of processes; every member of a set of two or more processes is waiting for an event to occur that can only be made to occur by another member of the same set. For example, if each of three processes $proc_1$, $proc_2$, and $proc_3$ needs simultaneous and exclusive access to some two out of a set of three resources r_1, r_2, r_3, there is a deadlock when each process has seized one resource and is waiting for another process to release the second resource needed by that process.

A process $proc_i$ is said to be dependent on a process $proc_j$ if $proc_j$ holds a resource that $proc_i$ is waiting for. These dependencies can be shown in a graph-theoretic model: the processes are represented by the nodes of directed graph, and an arc from the node representing $proc_i$ to that representing $proc_j$ shows that $proc_i$ is waiting for an event that must be brought about by P_j **[Ray88]**.

This type of dependency graph will show deadlocks at any instant. Minimal and maximal prefixs can also be used to show these dependencies. Let $proc_i$ be halted in a state in which it holds all its required resources. Then the minimal prefix of this local state shows the set of processes on which $proc_i$ depends both directly and transitively. The maximal prefix found with respect to this local state shows the set of processes which depend on $proc_i$. Let

PI = ($proc_2$ holds all needed resources).

Then the breakpoint predicates sought will be

PI_1 = min_p($proc_2$ holds all needed resources) and

$PI_2 = max_p$(proc$_2$ holds all needed resources).

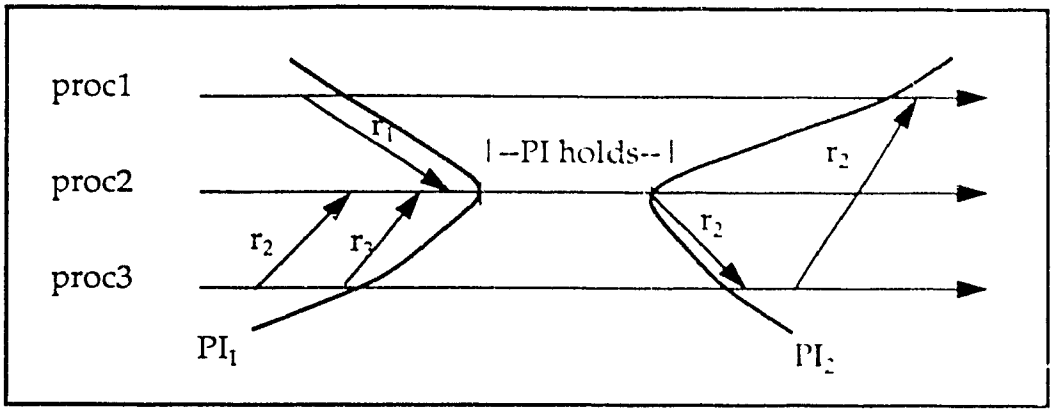

FIGURE 26: Resource dependencies

The dependencies observed in the above computation are proc$_2$ depends on proc$_1$ and proc$_3$; proc$_3$ depends on proc$_2$; proc$_1$ depends on proc$_3$.

7.1.3 Using |⇒: Atomic Broadcast

Protocols for atomic broadcast are particularly important because of their role in enabling distributed consensus: any system that can implement ordered atomic broadcast can also achieve distributed consensus.

An atomic broadcast mechanism ensures that for process groups G1 and G2, if a process p broadcasts a message m1 to G1 and a process q broadcasts a message m2 to G2, then all processes in the intersection of G1 and G2 deliver the messages in the same order. Moreover, if the broadcast of m1 'happens-before' the broadcast of message m2 , then all processes in the intersection of G1 and G2 deliver m1 before delivering m2. (*Delivery* of a message is distinct from *receipt* of that message. Delivery is easily implemented with a message buffer that holds messages and orders them until it is safe to deliver them to the process)

Let us assume that such a protocol works correctly. However, if the broadcasts constitute requests for resources, then consistent delivery of m1 before m2 may put process q at disadvantage in the competition for resources. To check that both safe orderings are eventually chosen, the following two breakpoints are set.

Let the predicate PA=(proc1.abcast AND proc2.abcast) indicate that process 1 and process 2 broadcast messages m1 and m2 concurrently.

Let PB=proci.deliver_m1, and let PC=proci.deliver_m2 indicate that for some process i, message m1 and m2 respectively are delivered.

PD = ((PA |⇒PB) AND (PB |⇒PC)) and

PE = ((PA |⇒PC) AND (PC |⇒PB))

If one of the PD and PE is never satisfied, then the local state of the broadcasting processes can be examined to clarify the decision procedure used. Setting the breakpoint to the minimal prefix of PC, for PD, and to the minimal prefix of PB for PE, the broadcasting processes will be left in a just-post-broadcast state for analysis.

7.2 Safety specification and checking

In debugging, breakpoints have shown themselves to be an invaluable tool in gaining information about the program behavior. But before any debugging can proceed, an error must be detected. Testing programs to discover errors requires a characterization of error. Simply put, a distributed program is in error if there exists some state of an execution or some state transition that violates a safety requirement of the program. The first phase of program testing is thus that of safety conformance checking. When a safety requirement is violated, an error has been detected and debugging can proceed.

Performing safety conformance checking requires tools to specify and detect safety properties of programs. Safety properties of programs are expressed as *stable*, *invariant*, or *unless* **[Chan88]**. Some temporal properties of programs cannot be expressed without 'unless' or an equivalent operator **[Lam80]**. Since stability and invariance are defined in terms of 'unless', 'unless' is a fundamental concept for expressing safety.

During an initial checking phase a program may be monitored for predicates expressing safety requirements. If such a predicate is satisfied during an execution run, then the property has been checked for the subset of the state space corresponding to that run. It can be concluded that the property is not wrong for the whole state space. If such predicate is violated during an execution run, then it may be concluded that that property is wrong for the whole state space. However, a third possibility exists; that the property was neither satisfied nor violated. For

example, a program ensuring mutual exclusion for a distributed database is monitored for the satisfaction of the safety requirements governing mutual exclusion. If no process ever accesses the database, then the safety requirements are trivially satisfied; however, this is not an interesting result for the purposes of safety conformance checking. Thus when monitoring for predicates expressing safety, there are three possible results; the predicate is satisfied; the predicate is not satisfied; or the predicate is violated.

Safety conformance checking proceeds most efficiently when the predicates monitored express correct program behavior. This efficiency is because the ways for a program to be in error are many and various, but correctness is explicitly specified. In addition, as mentioned above, it is more useful to be able to conclude that a predicate has been satisfied than to conclude only that it was not violated. However, where the predicate chosen is sufficiently abstract to cover a wide set of the possible incorrect behaviors, monitoring for error can be a useful procedure. In addition, monitoring for specific types of error can help narrow the possible sources of violation once a violation has been detected.

7.2.1 Monitoring for correct behavior: Token-passing

Where the topology of the communication between processes is a ring (unidirectional or bidirectional), mutual exclusion between processes can be assured by a special message or *token* that circulates on the ring. Each process waits for its one-hand neighbor to pass it the token, uses the token, and sends it to its other-hand neighbor.

A method for detecting loss of the token **[Ray88]** uses two tokens, each of which serves to detect the loss of the other. Both tokens circulate on the ring. A token T1 arriving at process $proc_i$ can guarantee that the other token T2 has been lost if neither T1 nor $proc_i$ has encountered T2 since T1's last visit to $proc_i$. This method can be generalized to any number of tokens, and will work as long as at least one token remains circulating on the ring.

Let the two tokens be called 'ping' and 'pong', and let numbers 'nbping' and 'nbpong', associated with the two tokens respectively, and equal in absolute value but opposite in sign, record the number of times the tokens have met. Since ping

should have met pong the same number of times pong has met ping, the sum of nbping and nbpong should always be zero.

The algorithm to detect loss and to regenerate the token works as follows. Initially the two tokens are both in an arbitrarily chosen process and nbping=1 and nbpong =-1. Each time they meet, one number is incremented by one and the other decremented by one. Each process $proc_i$ has a local variable m_i, initially zero, that records the number nbping or nbpong, associated with the token that last passed through $proc_i$. When $proc_i$ receives a token, it compares its m_i to the associated number. Say $proc_i$ receives ping. If m_i = nbping, then pong was not the last token to pass through $proc_i$, nor have the tokens met in their passage around the ring, so the token must have been lost and $proc_i$ can generate it. Otherwise, m_i is assigned the value of nbping.

```
when received (ping, nbping) do
        if m_i=nbping then
            begin {pong is lost, regenerate it}
            nbping := nbping + 1;
            nbpong := -nbping;
            end;
        else m_i := nbping;
        end if;
end do;
when received(pong, nbpong) do
        {as before, interchanging ping and pong}
end do;
when meeting(ping, pong) do
        nbping := nbping + 1;
        nbpong := nbpong -1;
end do;
```

At each step of the algorithm the relation nbping+nbpong=0 is preserved. Thus the predicate

PT = (invariant nbping+nbpong=0)

is the constraint that the protocol should preserve. If this predicate is violated at any point, then the protocol has failed and debugging can commence.

7.2.2 Monitoring for safety violation

Mutual Exclusion:

A problem arising in connection with communicating processes is that of ensuring that when processes compete for a resource that cannot be shared, only one process may gain access to it at any given time. Mutual exclusion can be assured through various mechanisms. In general, a process desiring to enter its critical section must complete a permission-granting protocol; this often takes the form of a token-passing protocol. Violation of mutual exclusion may occur in particular when the permission-granting protocol fails, but more generally when two or more processes are operating in their critical section concurrently.

For processes $proc_1..proc_n$, let each process take the following form:

$proc_i$:: **repeat**
{some code}
obtain token
critical section{}
release token
until(false)

Satisfaction of mutual exclusion requirements is independent of any particular token protocol; mutual exclusion is violated if two or more processes are operating in their critical section concurrently.

A predicate specifying violation of mutual exclusion is constructed as follows:

Let P_i = $proc_i$ is in its critical section.

Then, for any two processes P_i and P_j, if P_i sim P_j then violation of mutual exclusion has occurred:

$\forall i \forall j$: $P_i sim P_j$

Since satisfaction of the above indicates that a safety requirement has been violated, breaking at the minimal state at which the violation occurred will be useful. Thus in monitoring the detection procedures will assume a desired breakpoint:

$\forall i \forall j$: $\min_{p}(P_i \text{sim} P_j)$

7.2.3 Monitoring for correct behavior: FIFO ordering

Lamport points out in [Lam80] that some temporal properties of programs cannot be expressed in ordinary temporal logic. One such property is the FIFO or first-in-first-out property which can be expressed as follows: 'if process 1 requests a service before process 2 requests the service, then process 2 is not served before process1'.

Let $proc_1$.r1 and $proc_2$.r2 be integer variables indicating the logical time at which process 1 and process 2 submit service requests.

Let $proc_1$.served[r1] and $proc_2$. served [r2]be boolean variables; if ($proc_1$.served[r1]), then $proc_1$ is waiting or is being served for request r1.

```
proc1 :: repeat
   request service
   wait
   service
until (false)
```

'as long as' expresses this property handily:

~($proc_2$.served[r2] $\models\Rightarrow$ $proc_1$.served[r1]) as long as ($proc_1$.r1 $\leq$ $proc_2$.r2)

This specification gives $proc_1$ a slight priority: if process 1 and process 2 submit requests concurrently, then process 1 is served first ($proc_1$.r1<=$proc_2$.r2). However, if more than one server is available, then process 2 may be served concurrently with process 1. If more than one service is available and '$proc_1$' represents not just one process, but a partition or set of processes, then service may cross ($proc_2$.served X $proc_1$.served). If only one server is available, then it always be the case that ($proc_1$.served $\models\Rightarrow$ $proc_2$.served). In no case will process 2 be served before process 1.

Chapter 8 : Detection

8.0 Introduction

Safety predicates are constructed using the relationships between instances of two predicates. All such relationships are based on the reachability relationship: given predicates PA and PB, reachability between A_0, A_f, B_0, and B_f must be detected. Reachability itself is defined in terms of causality relationships between events. Thus detection of safety predicates requires distinguishing instances of a predicate, and determining A_0 and A_f for an instance of a predicate. This information will be sufficient for deciding satisfaction of safety predicates, since causality and hence reachability can be determined given this information.

The complexity of the detection problem depends on the various approaches to detection that are possible. As outlined in the previous chapter, possible approaches in testing or fault-tolerance are to determine if an execution run satisfied some safety predicate or satisfied some predicates specifying violation of safety. Criteria for choosing an approach will depend on the strength of result and the complexity of detection that can be tolerated by the application.

Determining membership of events in a set of concurrent events A requires determining the relationship between events. This requires some clocking mechanism isomorphic to Lamport's 'happens-before'. Various *virtual time* methods are available. Lamport's *logical clock* **[Lam78]** will be shown to be insufficient to the requirements here, as information is lost. The *vector clock* model yields sufficient information. In **[Spez89, Wal91]**, implicit vector-clock mechanisms are used; the functionality of vector-clock is duplicated, in that the information yielded is the same.

Detection algorithms can be centralized or distributed, and different approaches are suitable to different applications. For example, in testing and debugging it can be assumed that a facility for deterministic re-execution is available. In this context, a centralized algorithm might be the simplest solution. For fault-tolerance applications in a real-time environment, a distributed algorithm is preferable to ensure that the detection itself is fault-tolerant.

In Section 2, approaches to detection are discussed in the context of their costs and benefits. In Section 3, algorithms are given for the two general approaches to detection. Implementation considerations are covered in the remainder of this chapter. In Section 4, virtual time is introduced. In Section 1 the concept of logical clock is described and show to be insufficient. In Section 2, vector clock is described and shown to be sufficient for the requirements of detection. In Section 5, the definitions of causality and reachability are expressed using vector-clock. Consistent global state, minimal and maximal prefixs are also expressed using vector-clock.An optimization of search techniques is given in Section 5.1. In Section 6 approaches to control-flow in detection are considered, and conclusions are given in Section 7.

8.1 Approaches to Detection

A predicate can be satisfied, not violated, or violated. A predicate is satisfied if one or more instances of the predicate are detected during execution, and no instances of violation are detected. If no instances of the predicate are detected, then the predicate has not been violated and the safety property expressed by the predicate has been trivially satisfied. It is important to make the distinction between satisfaction and non-occurrence for the purposes of testing. An implementation may trivially satisfy a safety requirement while not actually accomplishing the intent - for example, an implementation of a distributed database that delays all writes until all reads are complete trivially satisfies consistency requirements. An implementation that does nothing is safe because it can never do anything wrong. When testing, it is therefore important to distinguish between satisfaction and lack of occurrence of a predicate. An alternate approach is to monitor for the negation of a safety predicate. Rather than detecting and checking every instance of a predicate, this approach searches for only the first state in which the predicate is violated. This approach is less complex, but the result is not as strong as no distinction can be made between lack of violation and satisfaction.

Searching for satisfaction of a predicate is usually the most expensive approach in terms of the complexity of the problem, because it involves detecting and verifying all instances of a predicate. Negation is usually much easier to detect, with the exception of comparative predicates such as (p1.x>p2.y>p3.z). The

negation of a safety predicate is less expensive to detect because all possible combinations of local instances do not have to be compared. For example, finding the first state satisfying (p1.x AND p2.y) requires finding the first states in which x and y hold, and then the first global states in which both hold. Finding a comparative predicate, by contrast, requires comparing possibly all values of x and y.

The complexity of detecting satisfaction of a predicate is based on two factors: the number of variables in the predicate, and the number of instances of local predicates holding. Let the number of variables be k, and let each variable be represented by a process line in the space-time diagram. (Thus there will be at most k process-lines.) Let the maximum number of instances of a local predicate on a line k be m, the number of instances on line m be k_m. Then the task of detection is to find all the possible combinations of each of the k local intervals such that each of the intervals is simultaneous with the other. (See Figure 27). The number of possible instances of P is thus the product of the number of instances at each line (k_m) for each line k, $(k_m)^k$.

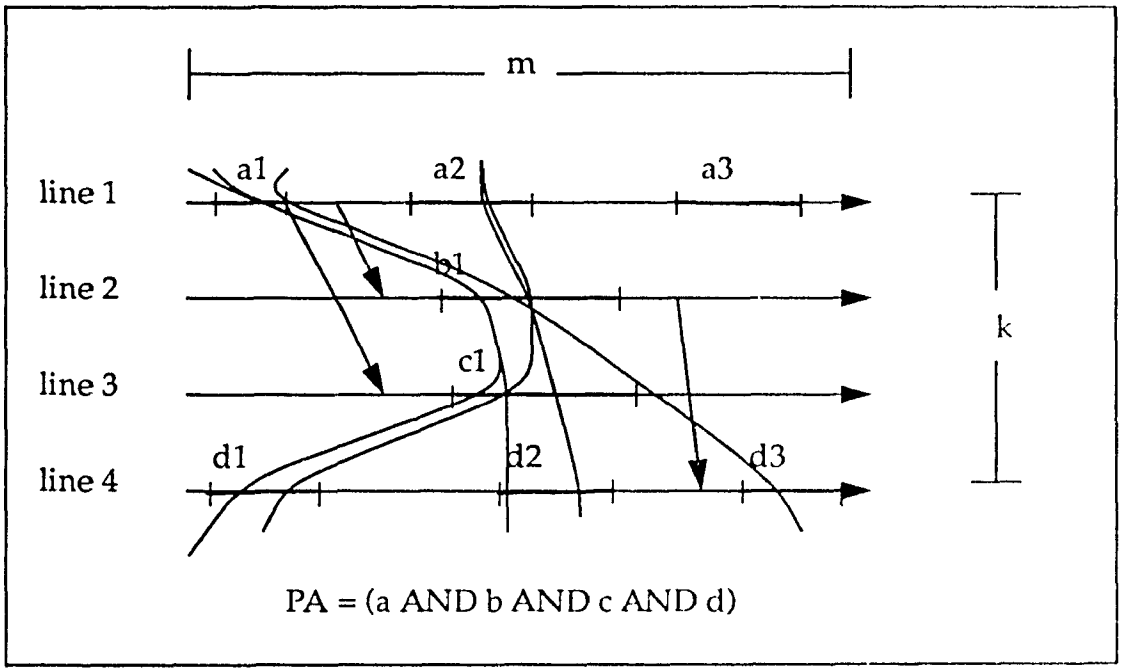

FIGURE 27: Complexity of detecting satisfaction

Some of the combinations of local instances that must be considered above are marked. For the portion of the computation shown, all instances to be considered

are [a1,b1,c1,d1], [a1,b1,c1,d2], [a1,b1,c1,d3], [a2,b1,c1,d1], [a2,b1,c1,d2], [a1,b1,c1,d3], [a3,b1,c1,d1], [a1,b1,c1,d2], [a1,b1,c1,d3]. This is $(k_m)^k$ or 3x1x1x3 instances to be considered.

An algorithm is given in the next section for this type of detection.

The above figure assumes a boolean predicate, so local instances exist where the local predicate holds. For comparative predicates, every change of value of the local predicate must be considered an instance, thus increasing the number of instances to be considered.

However, it should be noted that while detection is, in this case, a relatively complex task, it is less complex than searching the whole event space; the complexity is a function of the number of instances of a local predicate, rather than of the number of events at a local process. Thus the larger the difference between $(k_m)^k$ and the size of E, the greater the reduction in complexity.

Detection of violation for boolean predicates is a relatively simple problem. Detection of satisfaction of a predicate requires finding all possible combinations of local intervals; this results in complexity on the order of the number of local instances, where the number of variables is fixed. But detection of violation only asks if there exists one state in which the predicate is satisfied: detection can proceed by just looking for the first occurrence of a predicate without considering all (later) possible combinations of local instances. (See Figure 28.) Thus the

complexity of the detection is a linear function of the number of instances of local predicates.

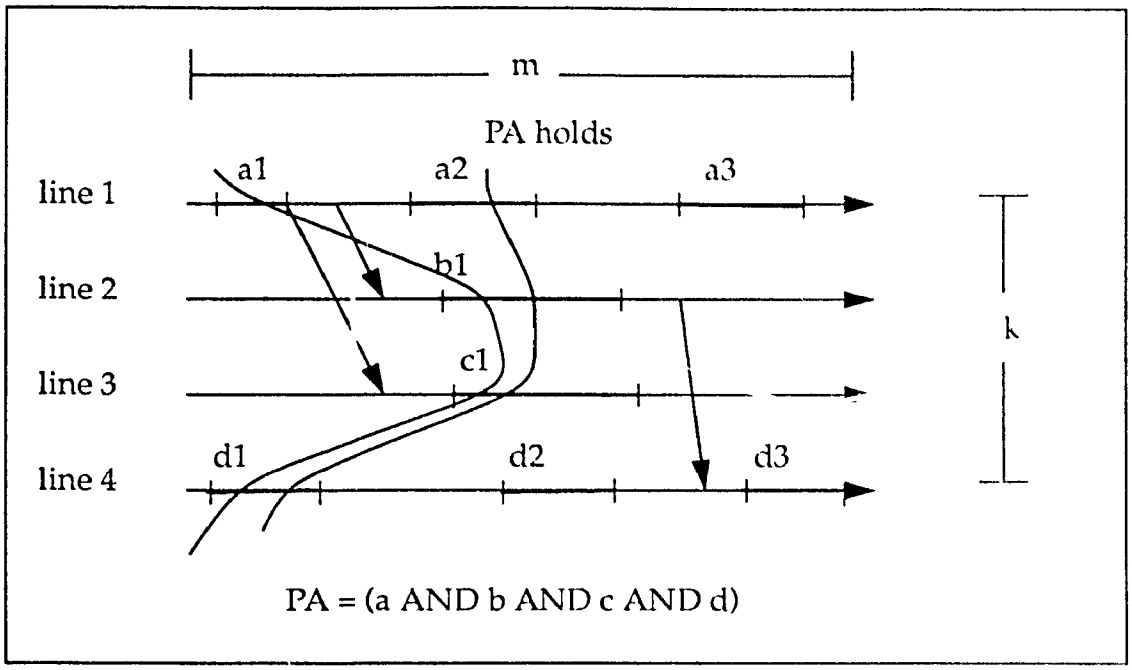

FIGURE 28: Detecting violation, or first instance only

Detecting the first instance only is relatively straightforward. Only two instances need be considered for the above figure: [a1,b1,c1,d1] and [a2,b1,c1,d1]. In other words, detection halts as soon as the first instance is found. An algorithm is given in the next section for this type of detection.

Measures can be developed at various levels of detail to gain savings. Where the predicate in question is boolean, the violation, rather than satisfaction, can be detected to take advantage of the simplicity of the detection problem. The number of comparisons required to perform the detection can be reduced using the information carried by vector clock. Designing search algorithms to avoid redundant comparisons can yield some savings. However, for off-line testing and debugging, the degree of complexity of the detection problem is not such a deterrent as it may be for fault-tolerant applications, where a change of approach to detecting violation may be more appropriate to the application and warranted by the savings in detection costs.

What are the relative strengths of conclusions that can be drawn from the different approaches? If a predicate occurs and is satisfied in a computation, then it can be concluded that the property expressed by the predicate is satisfied for that portion of the state space of the program traversed by that particular execution. Thus a result from satisfaction of a predicate in a particular run is partial, and does not imply that the predicate will hold in all execution runs of the program. If a predicate is neither satisfied nor violated during an execution run, then no further conclusion can be drawn about the state space of the program. If the predicate is violated, then a strong conclusion can be drawn: it can be concluded that the property expressed by the predicate does not hold throughout the state space of the program.

8.2 Algorithms

8.2.1 Detecting the first instance of a predicate

Let the ordered instances of a local predicate holding on a line k be referred to as $k_1, k_{1+1}, .., k_m$. Let C be an array of size k.

Initially, set C[k] := k_1.

repeat

> **if** C[x] sim C[y] **for** all x,y in k, **then** SUCCESS
> **else, for** all C[x] : C[x]<C[y], C[x]:= k_{1+1}
> if i+1>m or C[x]>C[y] **then** FAILURE

until SUCCESS or FAILURE

The algorithm works as follows: All elements of C are initially set to the first local instance for every line k. For example, in Figure 28, C is initially set to [a1, b1, c1, d1]. Each instance k_1 in C that precedes some C[x] is replaced with the next instance on line k; in the example, the next step would be C= [a2, b1, c1, d2]. If this results in a set of simultaneous local instances, then the distributed predicate can be tested for these values of the local predicates. If there are no more instances on line k (i.e. if i+1>m) or if for some line k there is no instance C[x] simultaneous with C[y] then the local instances are not anywhere simultaneous and the distributed predicate cannot hold.

The complexity of this search is linear in m and k.

8.2.2 Detecting all instances of a predicate

Let the ordered instances of a local predicate holding on a line k be referred to as $k_1, k_{i+1}, .., k_m$. Let C be an array of size k.

```
for i = 1 to k
        for j = 1 to m
        C[i]:= i_j */ fix the instance k_1 /*
        for all x<>i, set C[x]:= i_1 */ first instance of each j on line i*/
            repeat
                if C[x] sim C[y] for all x, y, in i then SUCCESS
                */ predicate can be tested on the local instances in C*/
                else if C[x]<C[y] then C[x]:= x_{j+1}
                if j+1>m (for any C[x]) or C[x]>C[y] then FAILURE
                */ C[i] is not a part of any instance of the distributed predicate */
            until SUCCESS or FAILURE

        if SUCCESS then */ first instance including C[i] has been found|
            for a=1 to k, a<>i
                for b=j to m
                    C[a] := a_{b+1}
                    repeat
                        if C[a] sim C[b] for all a,b, in i then SUCCESS
                        */another instance including C[i] has been found */
                        else if C[a]<C[b] then C[a]:= x_{b+1}
                        if b+1>m (for any C[a]) or C[a]>C[b] then FAILURE
                        C[a] := x_j */reset to local instance of first instance found*/
                        */ C[i] is not a part of any instance of the dist. predicate */
                    until SUCCESS or FAILURE
```

The algorithm works as follows: The search is fixed with respect to some instance of a local predicate on line k. This is referred to as k_1 and is assigned to C[i] in the algorithm. All other elements of C are initially set to the first local instance for every line k. For example, in Figure 29, C is initially set to [a1, b2, c1, d1] when C[i] is fixed to b1. Each instance k_1 in C that precedes C[i] is replaced with the next instance on line k. If this results in a set of simultaneous local instances, then the

distributed predicate can be tested on these values of the variables. If there are no more instances on line k (i.e. if j+1>m) or if C[i] preceded some element of C, then the local instance C[i] that was fixed does not form a part of any instance of the distributed predicate. Thus the first loop finds the first instance of the distributed predicate found with respect to C[i]. The second loop just finds all other instances with respect to C[i] by examining the instances that lie between the minimal prefix of C[i] and the maximal prefix of C[i].

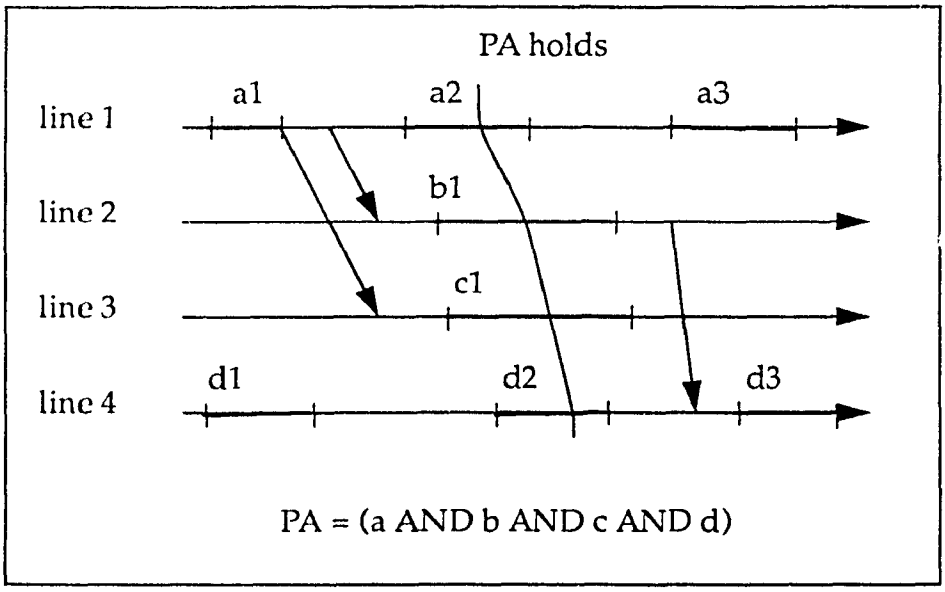

FIGURE 29: Finding all instances of PA

The algorithm may detect an instance of a distributed predicate more than once; this can be optimized at some level of implementation. For example, in Figure 29 the instance of PA labeled above is found when fixing C[i] to b1 and c1. The looping is necessary because each local instance may form part of more than one instance of a distributed predicate; for example, local instance d2 in Figure 29 forms part of two possible instances of the distributed predicate. The first is found by fixing C[i] to d2; the second by fixing C[i] to a3.

For each local instance k_i, determining if k_i forms part of an instance of a distributed predicate is linear in m(k-1). There are m instances in that line k, and k lines, making the complexity quadratic in $m^2k(k-1)$ to find the first instance of the distributed predicate with respect to k_i. To ensure that all instances are found requires the second loop of the above algorithm. Worst-case considerations for this loop bring complexity to $(k_m)^k$.

8.3 Virtual time and logical clock

Determining whether local intervals are simultaneous requires some mechanism for determining the causal relationships between the initial events and final events of intervals. Specifically, given an interval k_x and another k_y determine the causal relationship between the final event of k_x and the initial event of k_y and between the final event of k_y and the initial event of k_x must be determined. Some clocking or timestamping mechanism is required.

Various *virtual time* methods are available. Lamport's *logical clock* [Lam78] will be shown to be insufficient to the requirements here, as information is lost. The *vector clock* model yields sufficient information. In [Spez89, Wal91], implicit vector-clock mechanisms are used; the functionality of vector-clock is duplicated, in that the information yielded is the same.

8.3.1 Virtual Time

Events at a particular process are totally ordered by their local sequence of occurrence, and each receive event is ordered with respect to its corresponding send event. Events are related and the causality relation, expressing that the future cannot influence the past, is at the heart of any notion of virtual time.

In asynchronous distributed systems a common time base does not exist, but it is sufficient to create some approximation that has the desired characteristics of real time. The common concept of time is that of a set of instants with a temporal precedence ordering satisfying certain conditions:

1. transitivity
2. irreflexivity (together with transitivity this implies asymmetry)
3. linearity
4. eternity $\forall x \exists y: (y < x) \wedge \forall y \exists x: (x < y)$
5. density $\forall x, y: (x < y) \Rightarrow \exists z: (x < z < y)$, or discreteness

Density is not required for all applications - for example, digital clocks, and can be replaced by discreteness. Any implementation, in hardware or software, that satisfies these characteristics can be considered correct. Here a system of logical clocks that fulfills conditions 1 to 5 will be considered. This system of logical clocks will be used to timestamp events so that the causality relation is preserved

8.3.2 Logical clock

Recall that a computation is represented by an event structure E where E consists of a partially-ordered set of events e. A *logical clock* C is a mechanism which assigns to any event e in E a *timestamp* C(e) of some time domain T. A logical clock is a function that maps an event onto the time domain T: C:E->T. T is a partially ordered set such that:

$$(e < e') \Rightarrow C(e) < C(e').$$

In other words, if e can causally effect e', then C(e) is earlier than (has a smaller timestamp than) e'.

The following properties then hold **[Mor85]**:

1. If an event e occurs before event e' at some single process, then event e is assigned a logical time earlier than the logical time assigned to event e'
2. The logical time of a send event is always earlier than the logical time of the corresponding receive event.

Using a clock C_i for each process P_i and the set of integers N for the time domain T, the local clocks obey the following procedure:

1. When P_i executes a send event, a receive event, or any internal event, the clock C_i ticks: $C_i : C_i+d$ $(d>0)$
2. P_i includes a timestamp of the send event with each message sent.
3. When P_i receives a message from P_j with timestamp t, C_i is advanced:
   $C_i := \max(C_i,t) + d$ $(d>0)$

Usually the value of d is one. Updating of the clock occurs just before the local event, so that the timestamp of the event is already the new value of the local clock

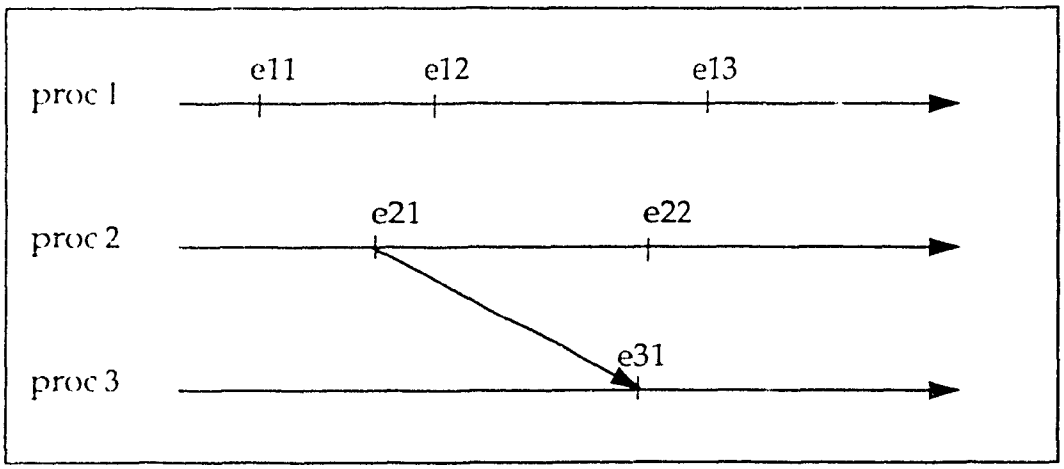

FIGURE 30: logical clock

Two events are independent (concurrent) if they have the same timestamps. For example, in Figure 30 e11, e21 and e31 are all stamped '1' and are concurrent. If C(e)<C(e') then it can be concluded that ~(e'<e), that is, the future does not influence the past. However, it cannot be concluded whether e<e' or e | | e'. This is a serious problem: by looking at the timestamps it is impossible to assert that some event could not influence some other event. This type of logical clock is thus not usable for the application of reachability, because the precise causal relationships between any two events cannot be determined.

This method is therefore not appropriate to the purposes outlined here because essential information about the relationship between events is lost; it is impossible to distinguish between causal ordering and concurrency.

As Mattern argues in **[Mat89]**, a linearly ordered structure of time is not adequate for distributed systems. A partially ordered system of vectors forming a lattice structure is a natural representation of time in distributed systems. Causality is represented in this way without loss of information.

8.3.3 Vector clock

An idealized external observer having instantaneous access to all local clocks could achieve a consistent view of the system. This information can be stored in a vector, with one element for each process's clock value. While instantaneous access to all local states is not possible, each process can build its own consistent view of the system state. Each process P_i has a clock C_i which consists of a vector of length n, where n is the number of processes in the system. At each local event, clock C_i 'ticks' by incrementing its own component in the vector, $C_i[i]$:

$C_i[i] := C_i[i] + 1$

The timestamp of any event is now a vector, C_i.

Each time P_i sends a message, it appends the timestamp of the send event. Each time P_i receives a message with timestamp C_j, P_i updates its view of the system by using the sending process's perception of system state, as follows:

$C_i[k] := \max [C_i[k], C_j[k]$ for each k from 1 to n.

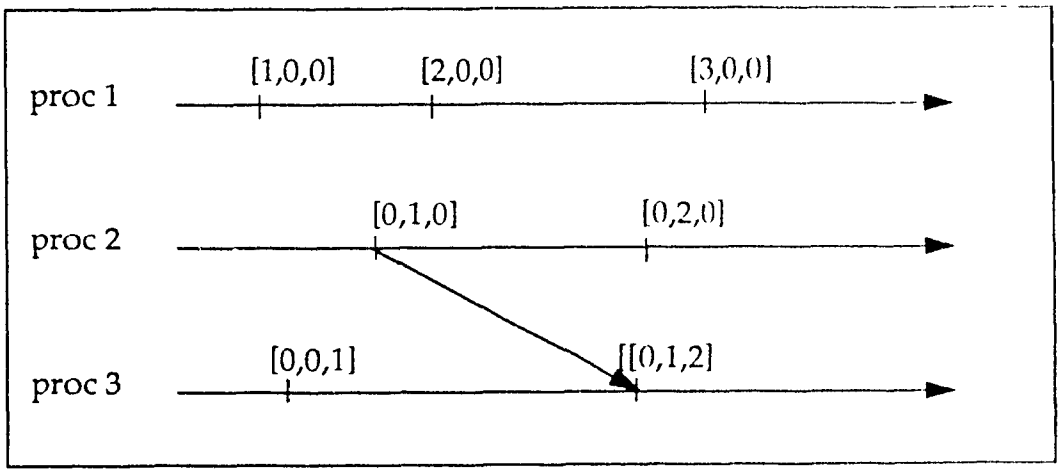

FIGURE 31: Vector Clock

In general, e<e' if $C_i(e)$ and $C_j(e')$ are related as follows:

$\forall k, C_i[k] \leq C_j[k], (i \neq j)$.

In this case, we can say that C(e)<C(e').

More specifically, e<e' if

$C_i[i] \leq C_j[i] \wedge C_i[j] < C_j[j]$.

This second definition clearly leads to a more efficient implementation.

For example, in Figure 31, the send event at process 2 is stamped [0,1,0]; the corresponding receive is stamped [0,1,2]. The send happens-before the receive by the definitions given above.

Two events e and e' are concurrent if $C_i(e)$ and $C_j(e')$ are related as follows:

$\neg(C_i < C_j) \wedge \neg(C_j < C_i)$,

and we say that **C(e)||C(e')**.

Again a more specific description is possible: e||e' if

$C_i[j] < C_j[j] \wedge C_j[i] < C_i[i]$,

which also leads to an efficient implementation.

For example, in Figure 31, the event after the send event, timestamped [0,2,0] is concurrent with the receive event at process 3, timestamped [0,1,2] by both definitions.

Thus vector clock fulfills the requirement that causal ordering be distinguishable from concurrency; using timestamps as the clock vectors gives enough information to decide the exact relationship of any two events.

8.4 Using Vector Clock

A global state of the system is given by a matrix of length *n* of clock-vectors C_i. This can be understood as an *n*X*n* matrix where a row *k* represents process k's perception of the state of the system (the local states of processes 1 through *n*), and a column *k* represents the state of process κ as perceived by processes 1 through *n*.

A consistent global state of the system is given by a vector-clock matrix that conforms to the following constraints:

$\forall i, j: C_i[i] \geq C_j[i]$

By contradiction: if

$\exists i, j: C_i[i] < C_j[i]$

then process j has received a message originating in process i that process i has not yet sent, as recorded by $C_i[i]$; otherwise process j has only received messages originating in process i that process i has sent.

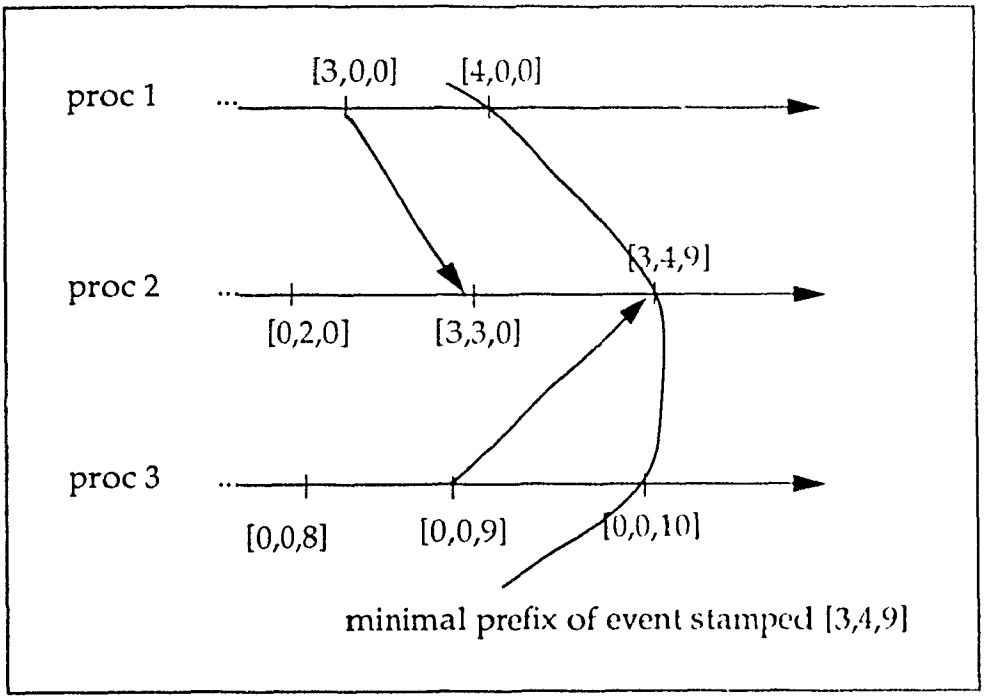

FIGURE 32: Minimal prefix using Vector Clock

Associating a set of concurrent events A with their timestamps, the relations of Chapter 3 can be directly restated in terms of vector-clock.

Let N be an array of nXn clock vectors C_i, where N conforms to the requirement given for consistent global state.

Let a timestamp C_i be associated with each element of A, and let the set of these timestamps be called A_c.

Let A_c be considered to be a kXk array, where $k \leq n$, so that every element of A is an element of N. A also conforms to the requirement for consistent global state given above.

Then a predicate PA can be said to hold at A_c just as it was said to hold at A.

The minimal prefix of the computation found with respect to the set A_c is an nXn matrix such that A_c is a kXk array in N and for all elements i,j in N

$\forall i, j: C_i[i] = C_j[i]$.

The maximal prefix of the computation found with respect to the set A_c is an nXn matrix such that A_c is a kXk array in N and for all elements i,j in N

$\forall i, j: C_i[i] \geq C_j[i]$

Definitions of reachability relations are given without proofs, as they follow directly from the definitions of $C_1(e)<C_1(e')$ and the reachability definitions given in Figure 3. Only the most specific version of each will be given for simplicity in developing detection strategies.

$C(A) \sim> C(B)$ iff $\forall a \in A, \forall b \in B: \neg(C(a) < C(b))$

$C(A) \| C(B)$ iff $\forall a \in A, \forall b \in B: (C(a) \| C(b))$

$C(A) X C(B)$ iff $\exists a, a' \in A, \exists b, b' \in B: ((C(a) < C(b)) \wedge (C(b') < C(a')))$

$C(A) \models\!\!> C(B)$ iff $(\exists a \in A, \exists b \in B: (C(a) < C(b))) \wedge$
$\forall a \in A, \forall b \in B: \neg(C(a) < C(b))$

$C(A) \angle C(B)$ iff : $C(A) \subseteq C(B)$

$C(A) \mid\!\!\rightarrow C(B)$ iff $\forall a \in A, \forall b \in B: ((C(a) < C(b)))$

Similarly, definitions of $\boldsymbol{A}$, $\boldsymbol{A_0}$, $\boldsymbol{A_1}$, and I (the set of initial states) can be found by substituting C(e)<C(e') for e<e' in all definitions.

These equations define the detection algorithm requirements at the most general level. Given the vector-clock basis, any detection algorithm must simply provide a structure for deciding if the vector-clock timestamps satisfy the desired relation

8.4.1 An optimization using vector-clock properties

In each detection algorithm is a line describing the task

determine if a set of local instances of k variables are simultaneous

(determine if C[i] sim C[j] for all i,j in k).

Specifically, this means that for an interval a marked by events a_0 and a_f, and an interval b marked by intervals b_0 and b_f, the two intervals are simultaneous if $(a_0<b_f)$ and $(b_0<a_f)$.

The number of comparisons required to complete this task is 2(k-1)! (where the array C is of size k). While this cost is not a critical factor in determining the order of complexity, reducing this cost can greatly increase the efficiency of an implementation. The cost of finding a set of k_i which are all simultaneous can be reduced to 2(k-1) by using an incremental approach and by using the properties of vector-clocks.

Recall that e<e' if $C_i(e)$ and $C_j(e')$ are related as follows:

$$\forall k, C_i[k] \leq C_j[k], (i \neq j).$$

Rather than doing k! comparisons, the minimum of each $C_j[k]$ must be found. Let the minimum of $C_j[k] := x$. This requires only k comparisons - one for each $C_i[k]$. If $C_j[k]<x$, for each k, then $C_i<C_j$.

Initially comparing one element of C to another gives the seed for the running component-wise comparison. At each step this component-wise score is updated. This means that for each C[i], instead of comparing it to all C[j], (j from 1 to k], only two comparison need be made, that between C[i] and the running component-wise comparisons, to determine if C[i] is simultaneous with C[j].

Given an array C of local instances, determine if C[i] sim C[j] for all i, j.

Let $C[i_0]$ be the vector-clock timestamp of the minimal event in local instance C[i]; let $C[i_f]$ be the timestamp of the maximal event in local instance C[i]. Let CC be an array having the same structure as C, initially empty.

```
CC[1] := C[1]
        for i = 2 to k
        start := element-wise minimum of each CC[x0], for x=1..i-1.
        finish := element-wise maximum of each CC[xf], for x= 1..i-1.
        if ~(C[if] < start) and ~(finish < C[i0])
        then CC[i] := C[i]
        else FAILURE */exit */
```

Example:

Let C be initially [a1, b1, c1, d1]. Assume the first iteration has occurred and CC:=[a1, b1], and C[i] = c1. Then at this point start = [3223] and finish = [3333]. Using the vector-clock definition of '<' defined in the previous section, c1f<start, leading to failure: a1, b1, c1 are not all simultaneous.

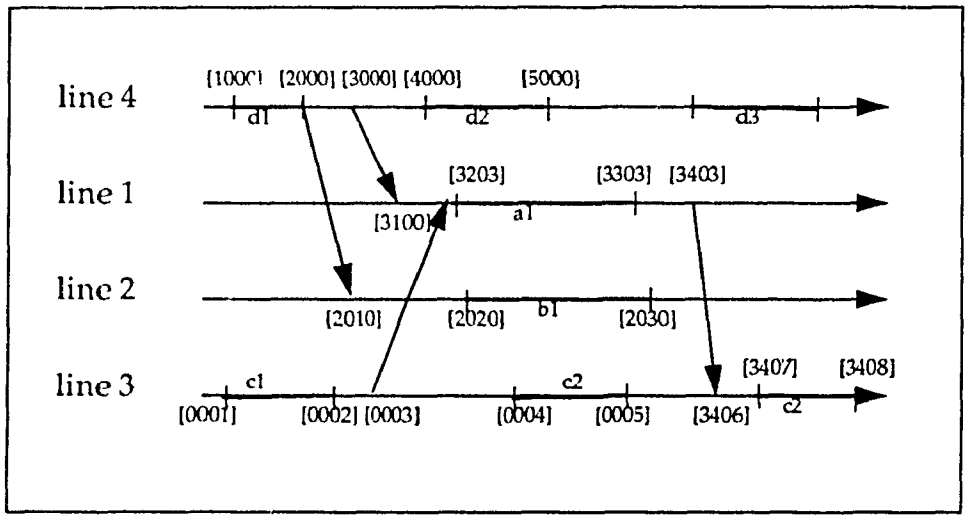

FIGURE 33: optimization using vector clocks

Now let C=[a1,b1, c2, d1]. Here after the first iteration we have the same values of start and finish as above. When C[i]=c2, start<c2f and c20<finish, so c2 is simultaneous with each other local instance in CC.

This reduces the number of comparisons necessary from 2(k-1)! to 2(k-1). For an array C, determining if an element C[i] is simultaneous with all other elements in C required 2(k-1) comparisons; to check all elements of C against each other required 2(k-1)!. Using the above method determining if C[i] is simultaneous with all other elements in CC requires just 2 comparisons. Checking for each k in the worst case thus costs 2(k-1). ((k-1) because C[i] need not be checked against itself; by definition a local instance is simultaneous with itself.)

8.5 Control

8.5.1 Centralized

A centralized approach assumes that the information collected about the local predicates instances by the process or by some local debugging monitor is forwarded to a central location for evaluation.

The detection scenario assumed is that the initial program run occurs in the context of deterministic re-execution. The information necessary for detection (the local intervals at which predicates hold) is collected during this run or during an re-execution. This information can then be evaluated off-line by a central monitoring process, and satisfaction or violation of specified predicates decided. If a breakpoint is desired, the time-stamp of the events at which the breakpoint should occur, corresponding to a minimal prefix, can be provided through the analysis done by the central monitoring process. This state can then be re-created through the deterministic re-execution facility.

8.5.2 Distributed

Programs for continuous detection of safety requirements can be composed with a distributed program to provide error detection and fault-tolerance. Where safety requirements are monitored on an on-going basis, a record is created of safe states; when a violation is detected, rollback to a recent safe state can proceed. Such a detection method should be distributed to ensure that error-detection is itself tolerant of processor failure.

1. P1 keeps track of its local predicate instances. It includes in each message sent its record of instances at which the local predicate holds.

2. Each time P1 receives a message that has an instance record appended to it, P1 checks its own records against those received to discover if there is any simultaneity between instances.
3. If an interval on P1's record is not simultaneous with any interval in the received record, and PA does not hold in the interval, then the interval is obsolete and can be discarded.
4. If an interval in P1's list is concurrent with some interval in the received record, or PA holds in the interval, then elements have tentatively been found for A_0 and A_f for that instance of PA. The timestamps of the simultaneous intervals are recorded in A_0 and A_f and are sent with the record of instances. P1 forwards this record with all messages.
5. If P1 is not a predicate process then P1 simply appends received records to outgoing messages without inspecting or modifying such records.
6. When P1 receives a list of m-1 simultaneous intervals, and has some item on its record of local instances that is also simultaneous, then an instance of PA has been detected.

Some comments:

1. Marker messages are redundant. Since causality can only be established by application messages, marker messages can only increase the 'speed' of detection; if the use of marker messages results in a measurable improvement in real-time detection, then their use is justified.
2. All occurrences of a predicate are detected, because each occurrence of a local predicate propagates among processes until it is determined that it is not concurrent with some other element of PA and PA does not hold.
3. PA may be detected at any predicate process. This highly increases the fault-tolerance of the detection strategy; if processes are still communicating then any instance of the predicate will still be detected.

8.6 Conclusion

Two approaches to detection were considered. Detecting satisfaction of a predicate ensures that the property that the predicate expresses has been satisfied non-trivially. This approach requires detecting all possible instances of the distributed predicate. The complexity of this detection technique is $(k_m)^k$. Detecting violation or the first state satisfying the predicate guarantees only that the predicate was not violated. However, the complexity of this detection technique is only linear in the number of instances of local predicate m and the number of variables, k.

To decide if a safety predicate holds it is necessary to decide if reachability between predicates holds; to decide reachability between predicates, it is necessary to decide reachability between global states; to decide reachability between global states, there must be some means of deciding if two events are concurrent or have a causal relationship. Hence the root of detection is a model that will permit the determination of the relationship between any two arbitrary events. The vector-clock mechanism provides a model of virtual time that is appropriate to distributed systems and within which the specific relationship between two events is discernible. Definitions of consistent global state and minimal and maximal prefix are also defined within the vector-clock model. Properties of vector-clock can be exploited in reducing the complexity of determining if sets of instances are all simultaneous. Once instances of a predicate are labelled with vector-clock timestamps, this information can be evaluated either on-line, using a distributed control algorithm, or off-line, using a centralized control algorithm.

Chapter 9 : Concluding Remarks

9.0 Conclusions

In debugging distributed programs a distinction is made between an observed error and the program fault, or bug, that caused the error. Testing reveals an error; debugging is the process of tracing the error through time and space to the bug that caused it.

A program is considered to be in error when some state of computation violates a safety requirement of the program. Expressing safety requirements in such a way that a computation can be monitored for safe behavior is thus a basic preliminary step in the testing-debugging cycle. Safety requirements are usually expressed as predicates; these predicates define safe states and safe state transitions of programs. When a state of the computation violates such a safety predicate, that state can be said to be in error.

The capability to express safety primitives such as *P unless Q* using distributed predicates developed in this work represents an increment in the tools and methodology available for testing and debugging distributed programs.

In the process of debugging, the user should have the facilities to define and detect predicates expressing safety, and also to define and specify situation-specific program conditions. The capability to specify the relationships between distributed predicates represents an increment in the scope and precision of conditions that can be specified and detected.

A breakpoint is *meaningful* if the prefix found includes or excludes events in such a way that the causal relationship between the events and the value of the breakpoint predicate are unambiguous. Part of the increment in precision is the capability to specify meaningful breakpoints: the minimal and maximal prefixs at which a breakpoint predicate holds.

The utility of extending a predicate logic to specify safety, to specify more complex user-defined predicates, and to specify meaningful breakpoint states is demonstrated by example.

The lack of global clock and global state in asynchronous distributed systems means that obtaining an instantaneous global state of the system is not possible. Implementing a virtual time mechanism with vector clock permits information to be gathered for the extraction of global state. It also permits a relatively efficient implementation of predicate-detection strategies. Complexity of detection was determined to be $(k_m)^k$ for a worst-case scenario using the approach of detecting all instances of a predicate. Complexity of detection in a best-case scenario is linear in the number of variables on which the predicate is defined, using the approach of searching for the first instance only of the predicate.

However, it should be noted that while detection can in the worst case be a relatively complex task, this task is less complex than searching the whole event space; its complexity is a function of the number of instances of a local predicate, rather than of the number of events at a local process. Thus the larger the difference between $(k_m)^k$ and the size of E, the greater the reduction in complexity.

9.1 Suggested further work

1. Exploration of efficient detection strategies for specific constructs. For fault-tolerant applications, detection of violations of *P unless Q* constructs at minimal cost and maximal real-time speed is a particular concern. Speed of detection is an issue that can be expanded on, over and above the work in [Li92]
2. Implementation of the model and detection strategies presented in this work in the context of deterministic re-execution (replay) for incorporation in a distributed debugging system.
3. Further examination of the implications of approaches to complexity, as suggested in Chapter 8. A study of the trade-offs between strength of results and cost of obtaining those results is indicated prefixs.
4. An exploration of the utility of the model presented here in the specification of non-atomic algorithms, i.e., algorithms that do not assume that primitives provide atomicity.

References

[Bat83] Bates, P., and Wileden, J.C., High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach, *Journal of Systems and Software*, 3(4), (1983), pp. 255-264.

[Bat88] Bates, P., Debugging Heterogenous Distributed Systems Using Event-Based Models of Behavior, *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, ACM, (1988), pp.11-12.

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

[Chan85] Chandy, M. and Lamport, L., Distributed Snapshot: Determining Global States of Distributed Systems, *ACM Transactions on Computing Systems*, 3(1), (Feb. 1985), pp.63-75.

[Coop91] Cooper, R., and Marzullo, K., Consistent Detection of Global Predicates, *Proceedings of the ACM/ONR Workshop on Distributed Debugging*, (May, 1991), pp. 163-173.

[Fow90] Fowler, J., and Zwaenepoel, W., Causal Distributed Breakpoints, *Proceedings of the 10th International IEEE Conference on Distributed Computing Systems*, (May-June 1990), p.134-141.

[Hab88] Haban,D., and Weigel, W., Global Events and Global Breakpoints in Distributed Systems, *Proceedings of the 21st Hawaii International Conference of System Sciences*, IEEE, (1988), pp.166-175.

[Hela90] Helary, J.M., Plouzeau, N, and Raynal, M., Computing Particular Snapshots in Distributed Systems, *Proceedings of the 9th International Phoenix Conference on Computers and Communications*, (March, 1990), pp. 116-123.

[Kraw92] Krawczuk, V., Distributed Debugging Based on Deterministic Re-execution Methodology and Design of a Working Prototype, Master's Thesis, Concordia University, (1992).

[Lam78] Lamport, L., Time, Clocks, and the ordering of events in Distributed Systems, *Communications of the ACM*, 21(7), (1978), pp.558-565

[Lam80] Lamport, L., 'Sometimes' is sometimes 'not never', *Proceedings of the ACM Symposium on Principles of Programming*, ACM, (1988), pp 174-184.

[Lam86a] Lamport, L., On interprocess communication Part I: Basic Formalism, *Distributed Computing*, (1986), pp.77-85.

[Lam86b] Lamport, L., On interprocess communication Part II: Algorithms, *Distributed Computing*, (1986), pp.86-101.

[LeBl87] LeBlanc, T.J., and Mellor-Crummey, J.M., Debugging Parallel Programs with Instant Replay, *IEEE Transactions on Computers*, C-36(4), (April 1987), pp.471-482.

[Li92] Li, H.F., Dash, B., Segel, H., Detection of Safety Violations in Distributed Systems, *Proceedings of GRIAO*, (Oct. 1992), pp. 2.37-2.42.

[Mat89] Mattern, F., Virtual Time and Global States of Distributed Systems, *Parallel and Distributed Algorithms*, Elsevier Science Publishers B.V. (North-Holland), (1989), pp. 215-226.

[McD88] McDowell, C.E., and Helmbold, D.P., Debugging Concurrent Programs, *ACM Computing Surveys*, 21(4), (Dec. 1988), pp.594-621.

[Mil88] Miller, B. and Choi, J., Breakpoints and halting in Distributed Systems, *Proceedings of the 8th International IEEE Conference on Distributed Computing Systems*, (June, 1988), pp. 141-150.

[Mis83] Misra, J., Chandy, K.M., Termination detection of Diffusing Computations in Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems*, 4(1), (1982), pp. 37-43.

[Mor85] Morgan, C., Global and Logical Time in Distributed Algorithms, *Information Processing Letters*, 20 (1985), pp.189-194.

[Ray88] Raynal, M., Distributed Algorithms and Protocols, John Wiley and Sons, (1988).

[Spez89] Spezialletti, M., A Generalized Approach to Monitoring Distributed Computations for Event Occurrences, University of Pittsburgh Doctoral Dissertation, (1989).

[Wal91] Waldecker, B., Detection of Unstable Predicates in Debugging Distributed Programs, University of Texas at Austin Doctoral Dissertation, (1991).