STATE SPACE ESTIMATION AND DISTRIBUTED PREDICATE DETECTION

MARIA JANTO

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER IN COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

JUNE 1996

ISBN 0-612-18406-4

Abstract

State Space Estimation and Distributed Predicate Detection

Maria Janto

General predicate detection is a fundamental problem in the design, coding, testing and debugging of distributed programs.

A distributed system formed of communicating sequential processes has a global state space that grows exponentially with the degree of concurrency in the execution. This makes program analysis and debugging difficult. A distributed predicate defined on the local states of the processes is often useful in capturing the safety [correctness] requirements of a given system. The complexity of detecting distributed predicates is often linked with the size of the global state space. This thesis attempts to address the issue of size of global state space and the complexity of distributed predicate detection, and proposes some strategies to deal with the problem in practice.

Algorithms for estimating the size of the global state space of an observed execution are proposed. Special cases where the size of the global state space is guaranteed to be determinable in polynomial time are identified.

In addition, a detection method is introduced for some simple form of distributed predicates, and a useful strategy is proposed to deal with distributed predicate detection in general. Some experimental results are also presented.

THIS THESIS IS DEDICATED TO
MY HUSBAND,

ELMER CAESAR

Acknowledgments

I wish to express my sincere gratitude to my supervisor Dr. H.F. Li for his advise and guidance given me throughout the last two years. He also helped me to acquire the proper approach for problem solving. His support and patience were invaluable in the preparation of this thesis.

I would like to confirm my appreciation within my close family here, to Chad Loeven and Melodie Sullivan for their example and encouragement during my studies at Concordia.

I also like to take this opportunity to thank my friends, Judit Barki, Uyentrang Hoang Nguyen, Joanna Sienkiewicz, and Kim Duong for their generous support and encouragement.

Furthermore some other very special people in my life also supported and encouraged me during the past two years. I would like to thank my sister Ilona Fekete for her unconditional love, and John Nemeth and Giselle Baron for their sincere friendship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Context

In the development process, it is often required to know whether for a distributed computation a certain property holds or not. Properties are characterized as distributed predicates which are evaluated on global states.

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 systems, 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 important application domain for global predicates is the field of testing and debugging. Testing and debugging programs are more involved in distributed systems than in uniprocessor systems because of the presence of the communication medium and the inherent concurrency.

Before debugging a program can proceed, it is necessary to discover an error. A program is considered to be in error when some state of the execution violates the safety requirements [correctness] of the system. When a state of the computation violates safety requirements (expressed as general distributed predicates), then the program is said to be in error. However, checking whether a general predicate is true

in a particular distributed execution may invoke exponential cost.

Generally, the difficulty dealing with distributed predicate detection stems from two factors:

1. a distributed system has a global state space that grows exponentially with the degree of concurrency in the execution, and
2. the nature of the distributed predicate involved.

These factors interact to make some cases truly difficult as discussed in this thesis.

Furthermore, it is a challenge to understand the behavior of a distributed program, since there is a lack of adequate tools for the design and analysis. The reason for this is the complexity inherent to the causality structure, which leads to tool designs dominated by efficiency considerations. Although there are some useful tools to help the development process (e.g., vector clock mechanism to identify whether two actions in a distributed computation are concurrent or causally dependent), there is a need for more. For example, it would be useful in supporting the development of distributed systems, to detect if the analysis could be handled within the practical space and time constraint of the development environment. In other words, by knowing the size of the state space, one could proceed to decide if distributed predicate detection is practicable in a given situation. Therefore, in this thesis we provide a tool which could help the user to estimate the size of the state space for a distributed program.

1.2 Organization of the thesis

In Chapter 2 we give a general view of the direction and the related problems of distributed predicate detection. Also, the most common detection strategies are reviewed. In Chapter 3 we describe a model for a Distributed Memory. An algorithm is presented to compute the size of the state space for an arbitrary distributed execution. Also an upper and a lower bound estimation strategy is proposed with different accuracy-complexity level. Chapter 4 presents several cases (according to the topology of communication between processes) where the exact number of states can be

computed easily. In Chapter 5, we address the factors that complicate distributed predicate detection in general and some means to deal with them in practice. Based on these results, in Chapter 6, the design of the abstract algorithms are presented for the computation of the number of global states for a distributed execution. Some experimental results are demonstrated in Chapter 7. Conclusions and suggestions for further work are presented in Chapter 8.

Chapter 2

Background and Related work

2.1 Background on distributed predicate detection

Researchers have shown that a distributed program execution can be analyzed and they have raised questions relating to its correctness answered by using the notion of distributed predicates. They can be used to check that certain 'good events' occur and that certain 'bad events' do not take place.

In this chapter we give a general view of the direction of distributed predicate detection. The majority of the work reviewed here focuses on the related problems of detecting distributed predicates. As illustrated here, some of the work considers the detection of general types of predicates, and others focus mainly on the detection of some special types of predicates. Also, some of the more often used detection strategies are reviewed.

2.1.1 Detection of distributed predicates

A predicate is considered to be either stable or unstable depending on its persistence once satisfied. For the detection of stable predicates researchers have already developed efficient techniques. The recent focus is on the detection of unstable predicates.

Stable predicates:

A predicate is considered to be stable if it is once true in a global state then it remains true in all future global states. In other words, if a stable predicate is true in some global state of the computation, then it must be true in all global states reachable from it. The case where the predicate is stable leads to particularly simple and efficient solutions based on distributed snapshots [CL85, Mat93]. Being stable, the predicate evaluating to be true in the snapshot state is sufficient for concluding that it has been detected. The properties which could be detected safely by taking snapshots, for example, are termination and deadlock detection.

Unstable predicates:

The value of these types of predicates alternates between true and false. Detection of unstable predicates cannot be based on snapshots, even when they are applied repeatedly [Mat93]. No matter how frequently taken, a sequence of snapshots may give gaps that correspond to exactly those global states in which the (unstable) predicate holds. For example, when debugging a system one may wish to monitor the length of two queues, and notify the user if the sum of the lengths is larger than some threshold. If both queues are dynamically changing, then the predicate corresponding to the desired condition is not stable. Thus, an entirely different approach is required for the detection of such predicates.

Some of the researchers took the bigger task of detection of general predicates [CM91, Seg93, BR95]. However, each of them ended up with the state explosion problem present in a distributed execution. In other words, due to concurrency between events, there may be an exponential number of global states existing for a single execution. (The number of global states are exponential in the number of processes). If the predicate being detected requires to check satisfaction in each such global state, then in a large system, the computational complexity of the algorithm becomes impracticable. Therefore, some researchers considered a smaller task in order to reduce complexity: some of them restrict the global predicate to those that can be efficiently detected, such as conjunction and disjunction of local predicates [GW92, GW94, GC95, VD95], linked predicates [MC88], simple sequences [GW92, HW88], and atomic sequences

[MHR93]. In some of these cases the construction of the whole state space may be avoided. Others reduce the cost of state traversal through heuristics that are dependent on the property being detected [FR94].

2.1.2 Some detection Strategies

A detection algorithm can be either *on-line* or *off-line*. On-line technique is attractive, because all predicates are evaluated on the fly and re-execution or post-mortem analysis become unnecessary. However, on-line detection may require delaying the execution of processes and introduce an unbearable probe effect which distorts the system's behavior. In fact, by blocking some processes when the predicate becomes potentially true, we may affect the system state trajectory. A strategy that evaluates predicates after the execution is called off-line evaluation. By observing the process during execution, event-traces are collected and the detection algorithm is applied later. In this approach, a detection algorithm could even make use of the space-time diagram corresponding to the execution. However, off-line detection may not be suitable for some application programs (i.e., real-time distributed systems).

Detection algorithms can be *centralized* or *distributed* and similarly as above, different approaches are suited to different applications. For example, in testing and debugging it can be assumed that a facility for deterministic re-execution is available as proposed in [LMC87]. In this context, a centralized algorithm might be the simplest solution. For fault-tolerance applications however a distributed detection algorithm is preferable to ensure that the detection itself is fault tolerant.

The lack of global clock 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 the predicate-detection strategies.

2.2 Related works

2.2.1 Chandy-Lamport and Mattern: Distributed Snapshots

The algorithm presented in [CL85] constructs consistent global states for a distributed execution. It is often called a 'snapshot' protocol, since a snapshot initiator process 'takes pictures' of the individual process sates. The algorithm assumes FIFO message delivery. It works 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, then it is defined as red. A process initiating a snapshot turns red, saves its current local state, and sends warning messages 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 warning message on every outgoing channel. When all nodes in the system are red, the algorithm terminates. 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 dec d 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, then a distributed snapshot is not a good means for detecting. If a process initiates a snapshot each time its local element holds 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.

Mattern [Mat93] extended this method for repeated snapshots where the channels do not deliver messages in FIFO order. While this approach can detect stable properties effectively, it does not work for an unstable predicate which may turn true only between two snapshots and not at the time when the snapshot is taken. No matter how frequently taken, a sequence of snapshots may have gaps that correspond to exactly those global states in which the unstable predicate holds.

2.2.2 Cooper-Marzullo: General distributed predicate detection

They considered the set of all global states which could be generated by a distributed computation and the set of all possible sequential executions that can be associated with this computation. These executions - usually called observations - structure the set of all global states as a lattice. A path in the lattice is a sequence of global states of increasing level, where the level between any two successive elements differs by one. By construction, each such path corresponds to a possible sequence of ordered observations. Thus the lattice of global states effectively represents the set of all possible observations of the computation.
They defined a general predicate (Φ) over a set of observations as follows:

- **Possibly(Φ)**: There exists a consistent observation O of the computation such that Φ holds in a global state of O.
- **Definitely(Φ)**: For every consistent observations O of the computation, there exists a global state of O in which Φ holds.

In other words, if (Φ) represents an error condition, then Possibly(Φ) represents a possible occurrence of an error. As for Definitely(Φ), (Φ)represents something good that we desire to be true irrespective of the actual progress of the execution.
Note that the above definition of Possibly(Φ) and Definitely(Φ) have been called weak and strong formulas, respectively by Garg and Waldecker in[GW92, GW94].

In the detection algorithms for Possibly(Φ) or Definitely(Φ) they built the lattice of global states and then traversed it to look for one global state (in case of Possibly(Φ)) or one global state per observation (in case of Definitely(Φ)) satisfying the general distributed predicate. They used an on-line construction of the lattice. However, as in the worst case, the size of the lattice can be exponential in the number of processes, so this approach can become infeasible in practice.

2.2.3 Vanketasan and Dathan

Vanketasan and Dathan [VD95] considered testing a given execution for the presence of errors without explicitly considering each possible global states in the lattice. They considered to detect a class of distributed predicates, such as conjunctive form of predicate. In addition to Possibly(Φ) and Definitely(Φ), they detect First(Φ) and Last(Φ) too, where (Φ) is in the form $\Phi = LC_1 \wedge \ldots \wedge LC_i \wedge \ldots \wedge LC_n$. Similarly as in [Seg93], they made use of the fact that a predicate value may hold over a subset of global state space, and there are two global states which are unique in this set: where (Φ) becomes true in the first time, and where (Φ) becomes true in the last time.

Vanketasan and Dathan presented two distributed algorithms. One is for the detection of Possibly(Φ), and the other for the detection of Definitely(Φ). Interestingly, their algorithm for Possibly(Φ) is able to detect First(Φ) too. They proposed that the detection of Last(Φ) can make use of the algorithm for Possibly(Φ) with some modifications. In the algorithms, they used the advantage of the conjunctive form of predicates: that each process i can perform an independent evaluation of the subexpression LC_i at every event in P_i without communicating to any other process. This gives a set of spectrum on each P_i where LC_i is true. Then they analyzed distributively the spectrum at all processes to check the distributed predicate. Their algorithms are fully distributed, so all processes execute the same algorithm. However, message delivery is assumed to be FIFO in order to make sure that the deduced global states are consistent.

They have also give a description of an algorithm for the detection of Always(Φ). They defined Always(Φ) to be true, if Φ holds in every consistent cut. The algorithm is as simple as follows: each process i evaluates LC_i. If after some event in process i LC_i is evaluated to false, then Always(Φ) is false; otherwise it is true. In this case Φ may be an invariant.

2.2.4 Garg-Waldecker, and Garg-Chase

To avoid the problem of combinatorial explosion of the states, Garg and Waldecker [GW92, GW94] focused on detection of predicates belonging to a class that they believed captures a large subset of predicates interesting to the programmer. They considered Possibly(Φ) and Definitely(Φ) (they called them weak conjunctive predicates and strong conjunctive predicates respectively) where (Φ) is restricted to conjunctive predicates with each conjunct involving variables of no more than one process. Their detection algorithms for Possibly(Φ) and Definitely(Φ) were centralized and used vector clocks. Each participating process sends the vector clocks of its events during which (Φ) is locally satisfied to a checker process. The checker process receives the clock values of the events and finds a consistent cut where (Φ) holds. They perform an on-line evaluation of the predicates.

While Garg and Waldecker detected conjunctive predicates based on a centralized checker process, in [GC95] a token based algorithm presented, which distributes the computation and space requirements of the detection procedure. The main idea of their algorithm is as follows: The token carries in it a candidate global cut in which the desired global predicate could be true. A global cut consists of states from each of the processes such that local predicates are true in those states. If such a global state is also consistent, then the predicate is true. This check of consistency is performed by the process holding the token. If the process finds that the cut is consistent, then the global predicate is detected; otherwise the token is sent to the process which violates the consistency condition on the global cut. This process identifies a new candidate cut by replacing the state from this process with a local successor. The process will then check for consistency conditions again. To uniquely identify a state, a vector clock mechanism is used. Their algorithm works in an on-line fashion.
They also presented another distributed algorithm which permits greater concurrency in the algorithm for detection of a conjunctive distributed predicate, where multiple tokens are used. This algorithm does not use vector clocks, instead, it uses direct dependencies for identifying a desired state. Since the distribution of work is more equitable than in the centralized algorithm, their algorithm is somewhat more efficient than the previous algorithms presented in [GW92, GW94].

2.2.5 Raynal: Inevitable global states

In this work the cost of lattice traversal is avoided through heuristics that are dependent on the property being detected. They detected the satisfaction of a predicate by checking it in the so called inevitable global states. They called a global state inevitable if it is shared by all the observations of the computation.
They defined the computation at some abstraction level: they called it 'predicate' or 'user's level'. At this abstraction level only relevant variables can be observed. Using this abstraction they defined weak or strong precedence relations on local states as the basis for identifying the necessary and sufficient conditions for a global state being inevitable.

In this context, a distributed computation satisfies the property Φ if and only if there exists an inevitable global state satisfying Φ. They provided an algorithm for identifying these global states by using vector clock values of local states by adopting one of the algorithms defined by Garg and Waldecker.

It should be noted that their algorithm does not detect Definitely(Φ), since with incomplete explorations of the lattice structure (i.e., checking the satisfaction of the predicate only in a state which is shared by all process), detection of Definitely(Φ) would take only probabilistic interpretations.

Although the applications where their work could be useful is limited, it is useful in checkpointing: since inevitable global states have necessarily been passed through by the actual computation.

2.2.6 Haban-Weigel and Miller-Choi

While predicates, (stable or not) over a single global state are able to capture many interesting system properties they inherently lack notions of logical time or relative order. In order to characterize dynamic properties and behavioral patterns of distributed computations, predicate specifications must be extended to include a temporal component. In other words, specifying and verifying dynamic properties require

reasoning about sequences, rather than single instances of global predicates. Such proposal is due to Miller and Choi [MC88] and Haban and Weigel [HW88].

Miller and Choi [MC88] defined Simple and Disjunctive Predicates and Linked Simple Predicates. Algorithms based on the distributed snapshot algorithm are provided for halting in a state at which a predicate holds. Conjunctive predicates are defined, but no algorithms for their detection are provided. The significance of their work is that they have introduced an algorithm to detect "linked predicates' in which the event ordering can be specified. Their algorithm can not detect concurrent events.

Based on the work in [MC88], in Haban and Weigel's work [HW88] the goal was to define distributed predicates, detect and halt. They defined the same types of predicates that were considered by Miller and Choi, but somewhat in more detail. For detection, the predicate is broken down over a binary tree, where the leaves of the tree are Simple Predicates and the root is the value of the whole predicate expression. Detection algorithms are provided as part of a complete debugger architecture. For non-locally decidable predicates such as conjunctive predicates, a halting algorithm is provided, but the authors recognized that yielding a state in which the predicate holds, can not be guaranteed. To compensate for this, a local trace facility is provided that records the local states in which Simple Predicates are held.

The work provided in [MHR93] is an extension of Miller-Choi's and Haban-Weigel's research work. The authors generalized the predicate to a little bit more general form: atomic sequence of predicates. Here some events can be forbidden between each pair of consecutive relevant events. They described global properties by causal composition of local predicates augmented with atomicity constraints. These constraints specify forbidden properties, whose occurrences invalidate causal sequences. They propose a distributed algorithm to detect their occurrences. Their detection algorithms are superimposed on the distributed computation and use a simple piggybacking technique to ensure consistency of the detection.

2.2.7 H. Segel: Monitoring Distributed Systems

In [Seg93] a predicate logic that permits the specification of relationships between distributed predicates is proposed. In other words, a model is presented that permits direct specification of fundamental safety operators; (an operator is considered fundamental if some safety property cannot be expressed without it). The significance of their work is the idea of direct specification of safety properties which permits monitoring of programs for satisfaction or violation of safety requirements, facilitating the detection of error in distributed program testing. The proposed logic also permits the specification of safety primitives such as P unless Q in UNITY [CM88] using distributed predicates.

They were treating unstable predicates as non-atomic events: the predicate may hold over a number of states, where the minimal and the maximal prefix where the predicate is satisfied is uniquely identified. These states are specifiable as part of a breakpoint definition in the logic presented.

Two detection algorithms were also presented in this work. Detecting satisfaction of a predicate ensures that the property that the predicate expresses has been satisfied (similar to as Definitely(Φ)). This approach may require detecting the predicate in all possible global states in an execution. The other algorithm is about the detection of violation (Possibly(Φ)). They provide an efficient algorithm for answering the question: Is there one state in which the predicate is satisfied? The detection proceeds by just looking for the first occurrence of a predicate (they call it minimal prefix) without considering all (later) possible global states. Thus the complexity of the detection is a linear function of the number of instances of local predicates.

2.2.8 Babaouglu-Raynal: Specification and Verification of Dynamic Properties

Babaouglu and Raynal [BR95] have explored Cooper and Marzullo's work [CM91] to which their approach could be extended to capture larger classes of properties, such as dynamic aspects of distributed computation.

They introduced two global predicate classes called simple sequences and interval constrained sequences for specifying desirable states in some causality-preserving order along with intervening undesirable states as was proposed in [MC88, HW88, MHR93]. They considered sequences of global predicates by composing instances of the class of Simple Predicates, where the term Simple Predicate corresponds to a boolean expression defined over a single global state. They called this extension as Simple Sequences. In other words a sequence of global predicates is satisfied if each of the component predicates hold in distinct global states of the observation and each such component must be observed in an order consistent with the syntactic position of its respective component predicate. They also define another predicate class, the so called interval-constrained sequences for detecting the notion of atomicity. For instance, in terms of observations, atomicity of distributed actions can be expressed as transforming one global state to another with no intermediate global states that are observable. In other words, the state in which the atomic action is applied and the resulting state must be adjacent to each other.

Their approach to dynamic property detection consists of two concurrent activities: that of constructing the lattice of consistent global states by monitoring the computation and that of evaluating the formulas by traversing this lattice. Therefore, the overhead is about the same as for the algorithms used for the lattice construction in [CM91]. The verification takes place at a monitor internal to the system and concurrently with the actual computation. In other words, the detection proceeds in an 'on-line' fashion.

2.3 The contribution of our work

The goal of this thesis is motivated by the difficulties in distributed program analysis and debugging.

In this thesis, we attempt to address the size of the global state space and the complexity of distributed predicate detection in general. We observe that the complexity

may be attributed to not only to the size of the global state space, but to the nature of the distributed predicate also. For instance, even if the size of the state space is relatively small, there could be some types of predicates (e.g. the number of terms in the given predicate is exponential in terms of the number of processes) where the detection procedure would be computationally expensive.

An algorithm for estimating the size of the global state space of an observed execution is proposed. Computing the size of the state space could be considered as a tool in supporting the development of a distributed program. In other words, it could be useful in deciding if the analysis could be handled within the practical space and time constraints of the development environment.

Although some distributed predicate detection may involve exponential cost, some practical systems may not exhibit such an intolerable complexity. This could be due to the synchronization design of the distributed system where processes tend to communicate locally in the form of clusters or processes are tightly synchronized. In the former case, the size of the state space is likely to be easily determinable. In the latter case, the size of the state space is likely to be smaller. We identify some special cases where the size of the global state space is guaranteed to be determinable in polynomial time. This knowledge may be useful in the design and debugging of a distributed system.

In addition, a detection method is introduced for some simple form of distributed predicates, and a useful strategy is proposed to deal with distributed predicate detection in general.

2.4 Conclusion

Analyzing a distributed program and checking it against behavioral properties are difficult topics. Most properties useful to the computer scientists interested in distributed program analysis refer to global states of distributed computations. Usually, behavioral properties are specified as distributed predicates. Detecting the existence

of global states of a distributed system which satisfy a distributed predicate has been studied in the literature of the last decade.

Research efforts have produced efficient results for evaluating stable properties. While detecting unstable properties is notably more difficult than the case of stable ones, since their occurrences are transient, many interesting results have been obtained recently by researchers. The detection of some special type of predicates (conjunctive, disjunctive, linked predicates, simple sequences, and atomic sequences) can be done with reasonable cost. However, the detection of general types of distributed predicates still remains unsolvable in polynomial time and space.

Chapter 3

Global State Space Estimation.

3.1 Introduction

Distributed predicates can be used to express the safety properties of a distributed computation. A predicate is satisfied if there is one or more global states where the predicate evaluates to true during execution.

Predicate values are meaningful only when they are evaluated on consistent global states. Often, the global states are identified by considering the causality relationships between the objects (events, local states) of the system. A fundamental problem in distributed computing is to ensure that a global state constructed in this manner is meaningful.

Searching for satisfaction of a predicate is sometimes expensive when it involves checking at each global state reachable in an execution. Due to concurrency between events, there may be an exponential number of global states existing for a single execution (i.e., the number of global states are exponential in the number of processes). Thus, it would be useful to know if the detection could be handled within the practical space and time constraint of the system. In other words, the computation of the size of the state space could help the user to decide if distributed predicate detection is practicable in a given situation or not.

In this chapter, we begin with the description of a model for a Distributed Memory System. Also, a Consistency Diagram is described to represent such systems. In section 2, a labeling method is presented where the size of the state space is computed for an arbitrary Consistency Diagram. In the following sections two estimation strategies are proposed: one which estimates an upper bound of the number of global states; followed by another estimation technique, which computes a lower bound on the global state size.

3.2 Model of a Distributed Memory System

A distributed system consists of a set of communicating processes each of which owns its local variables. The local state of a process is given by the values of its local variables. The local variables of all processes together form a distributed memory and their values form the global state of the distributed memory system. Interprocess communication is the medium through which processes synchronize in their change of local states. A space-time view of the distributed memory system is shown in Figure 1. An event in a process depicts one of three possibilities:

1. an internal change of state to s_{ij} [the j^{th} state of process i],
2. a sending of a message, or
3. a receiving of a message.

In this model, the state of a channel is not part of the distributed memory state. However, it is possible that the local state of a process also accounts for the sending or receiving of messages. In such a case, the state of a channel is deducible from the global state formed from the local states of the processes. For example, if the local state of process 1 is after it has sent two messages to process 2 while that of process 2 is after it has received only one message from process 1, then there will be a message in transit from process 1 to process 2.

The set of global states of a distribution computation represented by a space-time diagram is given by the set of consistent cuts [CL85, Mat89], in the diagram. A consistent cut partitions the space-time diagram into two parts such that no reverse

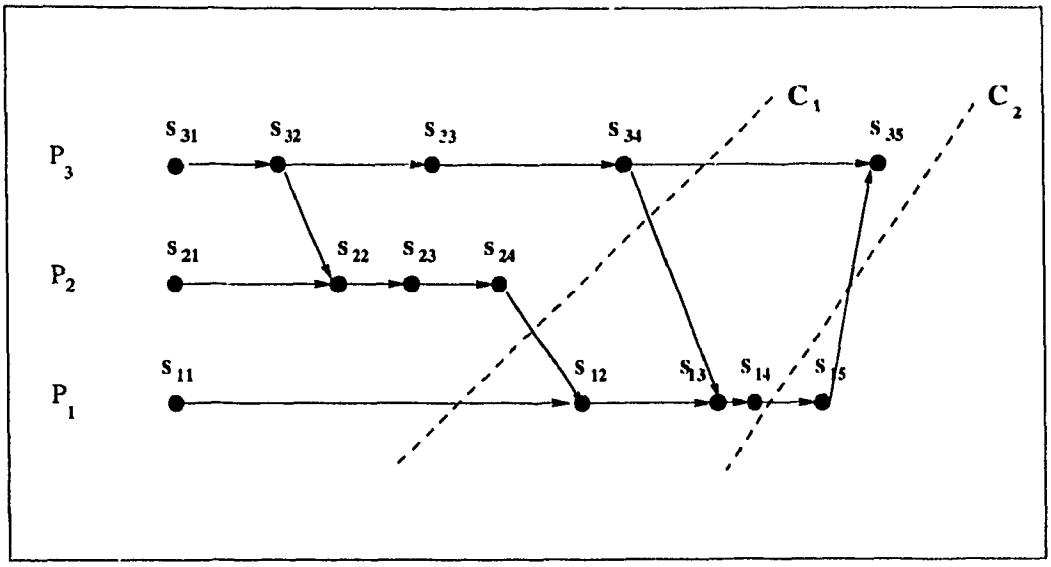


Figure 1: Space - Time Diagram.

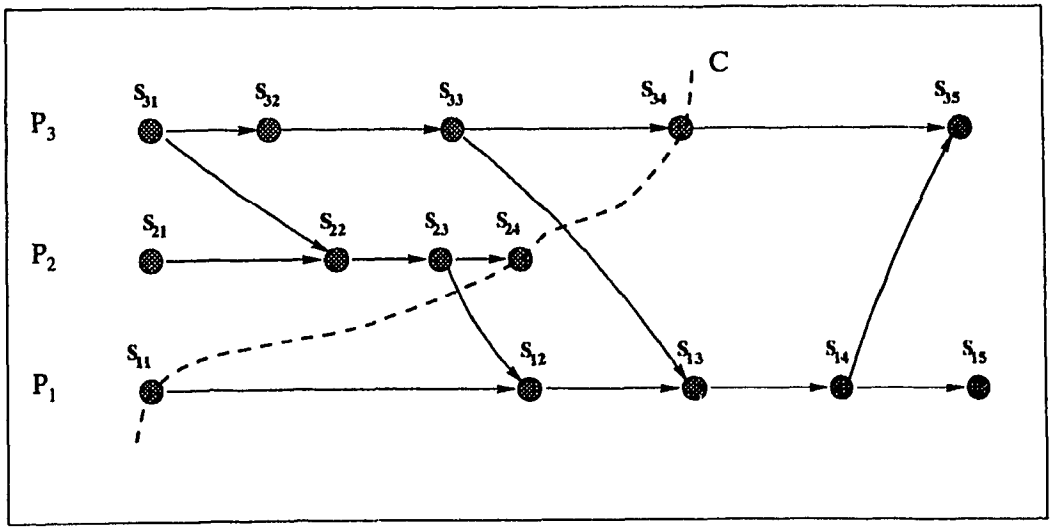


Figure 2: Corresponding Consistency Diagram.

[causal] edge is allowed to point from the second [future] partition to the first [past] partition [Mat89], for example, C_1 in Figure 1. An inconsistent cut is one where a reverse edge exists, depicting the causal dependence of some past event on some future event yet to happen, as exemplified by C_2 in the Figure. The number of consistent cuts in Figure 1 is equal to 10.

In this thesis, a slightly refined model is proposed to represent the distributed memory system. The space-time view representing explicit causal dependence between events is replaced by a Consistency Diagram. Figure 2 represents the Consistency Diagram corresponding to the space-time diagram in Figure 1. In a consistency diagram, nodes represent local states and directed edges represent inconsistency between the connected local states. Thus, the existence of a path from s_{iu} to s_{jv} means that s_{iu} and s_{jv} are inconsistent local states: they cannot coexist in any global state of the given system. In the corresponding space-time diagram, this would mean that a later event than s_{iu} causally happens before s_{jv}; process i must have changed its local state beyond s_{iu} when process j reaches s_{jv}. In the rest of this thesis, event and local state are synonymous terms.

Formally, a Consistency Diagram is a directed acyclic graph G = (N,A) where N = $\{s_{ij} \mid s_{ij}$ is the j^{th} local state of process i$\}$, and A = $\{(s_{iu}, s_{jv}) \mid s_{iu}$ has elapsed in process i before s_{jv} is reached in process j$\}$. Usually, A is represented in the transitively reduced form, i.e., a path from s_{iu} to s_{jv} in G [written as $s_{iu} \rightarrow^* s_{jv}$] also depicts the same relation as a directed edge from s_{iu} to s_{jv} [$s_{iu} \rightarrow s_{jv}$]. A *consistent cut* C in G is a subset of nodes in N which includes exactly one local state of each process such that s_{iu}, s_{jv} in C implies that neither $s_{iu} \rightarrow^* s_{jv}$ nor $s_{jv} \rightarrow^* s_{iu}$ [written as $s_{iu} \parallel s_{jv}$]. In other words, any two local states contained in C must be consistent. As an example, the cut C in Figure 2 is consistent. The following property is immediate from the definition:

Property of a Consistent Cut:
A consistent cut forms a consistent global state of the distributed memory system.

As an illustration of the application of consistency diagram to distributed predicate detection, consider the simple example shown in Figure 3 where each process state is

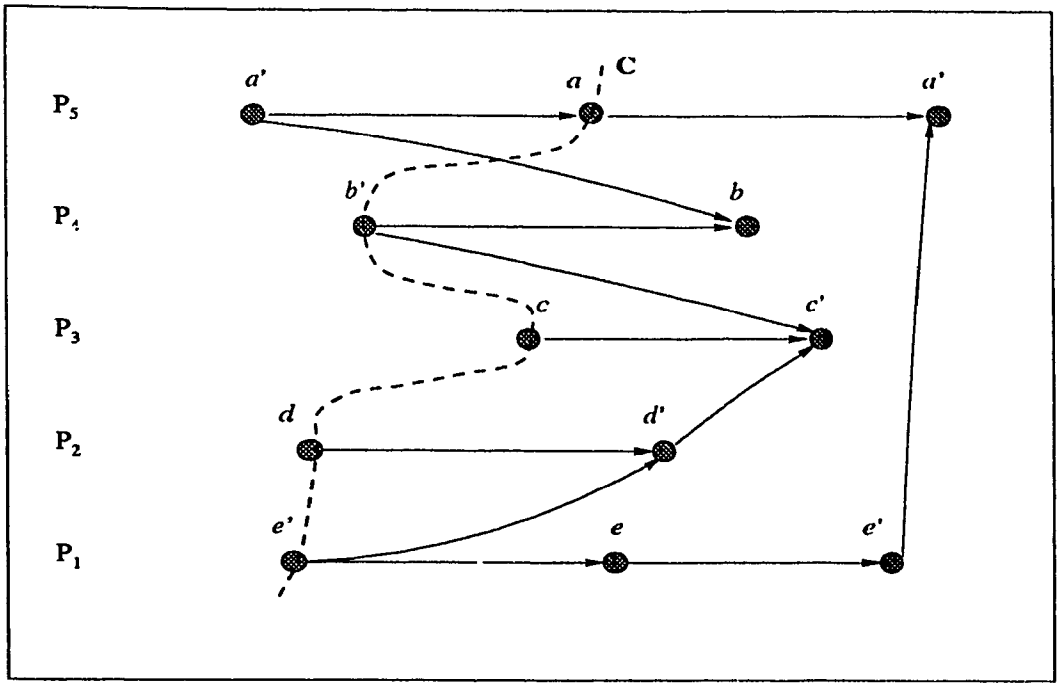


Figure 3: Cut satisfying ab'cde'.

represented by a single boolean variable. Suppose one wishes to detect the existence of a global state in which the predicate ab'cde' is true. This corresponds to finding a consistent cut in the consistency diagram where the relevant processes are at local states a, b', c, d, and e' respectively. C in Figure 3 is one such cut.

3.3 State Estimation

In this section, we explore an algorithm for computing the number of cuts in a given consistency diagram, followed by an efficient algorithm for estimating an upper and lower bound to the same number. By computing these numbers, a user may be guided to choose among various alternatives in his development process.

3.3.1 Computing the Exact Number of States

The computation procedure presented here involves successive iterations of a sequence of subgraphs of the given consistency diagram G = (N,A). In iteration i, the subgraph $G_i = (N, A_i)$ is used. In general, G_i is the subgraph of G obtained by including the

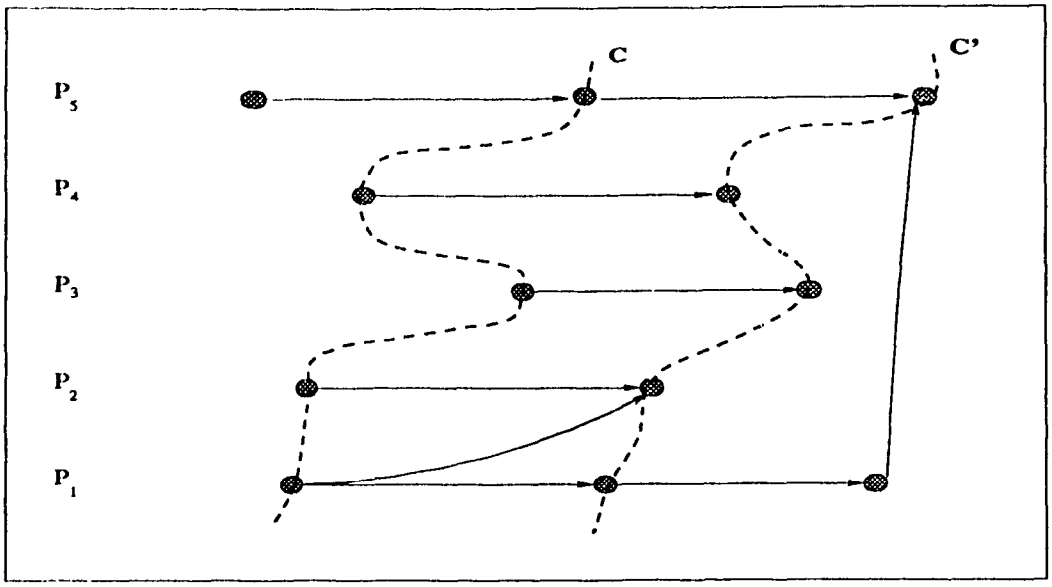


Figure 4: G_1 of G in Figure 3.

following subset of edges of G:

- edges linking the local states of each process;
- edges linking the local states of two processes provided one of them must be in process 1 through i.

For example, G_1 and G_2 of Figure 3 are shown in Figure 4 and Figure 5 respectively. Observe that G_2 differs from G_1 in only a single additional edge emanating from process 2. In general, G_i differs from G_{i-1} in exactly the same way: additional edges introduced in G_i are those that

- start from or end at process i, AND
- end at or start from process j > i.

Suppose $C = (s_1, \ldots, s_i, \ldots, s_n)$ is a cut where the local state of process j ($j \leq n$) is given by s_j. The following lemma is provable.

Lemma 1 *$C = (s_1, \ldots, s_i, \ldots, s_n)$ is a cut in both G_{i-1} and G_i iff $s_i \parallel s_j$ for $i < j < n+1$ in G_i.*

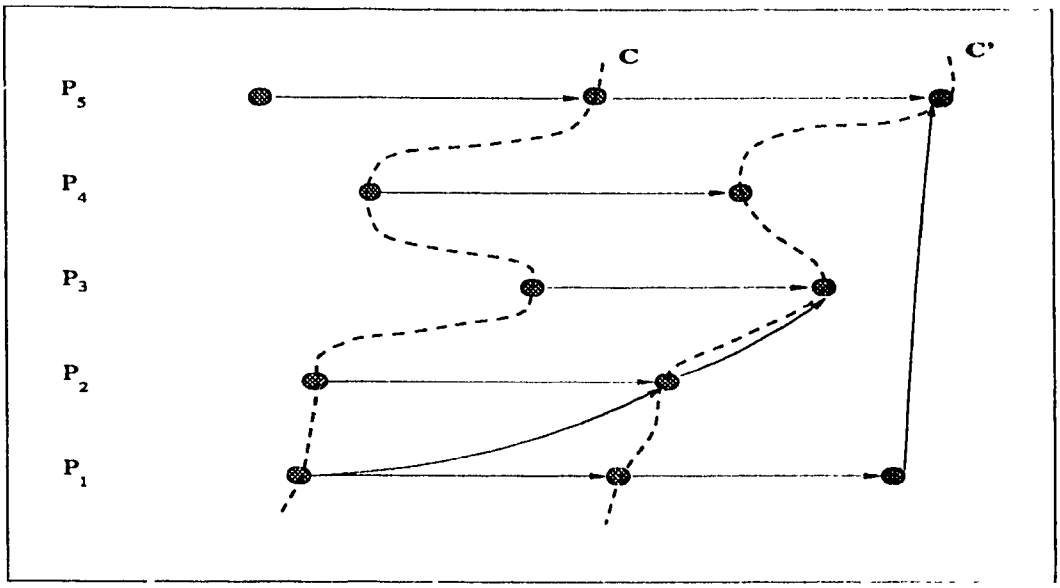


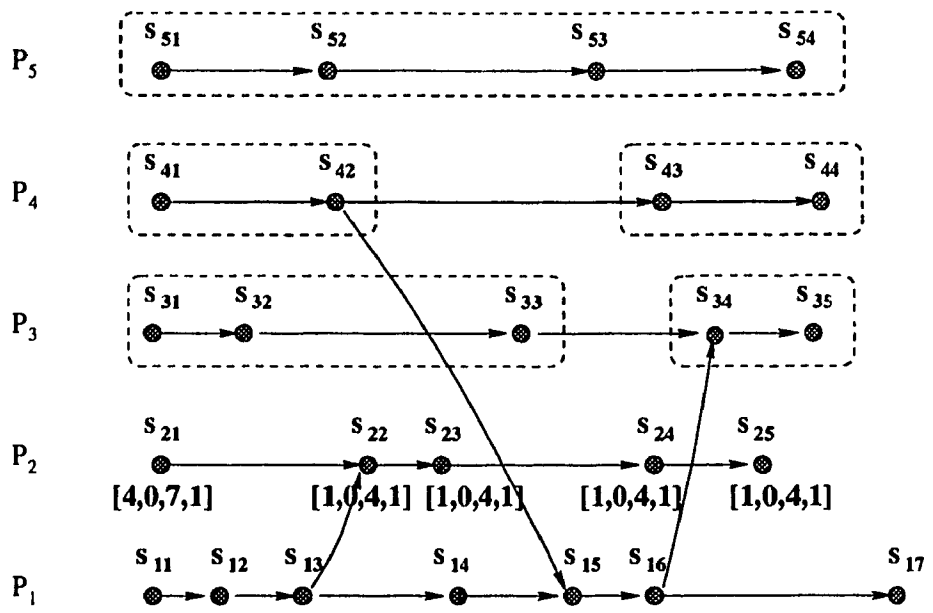
Figure 5: G_2 of G.

Proof: From definition, G_i differs from G_{i-1} in only some additional edges that start from or end at process i, and end at or start from process j > i. So $\forall j$, where $i < j < n+1 : s_i \parallel s_j$ if and only if $\forall k$, where $0 < k < i+1 : s_k \parallel s_j$. In other words, there is no edge introduced in G_i such that $s_k \rightarrow^* s_j$ or $s_j \rightarrow^* s_k$ holds. Thus the claim.

QED

In Figure 4 and Figure 5, C is a cut in both G_1 and G_2, but not C' because of the additional edge that leads from process 1 to process 3.

G_i and Lemma 1 are crucial in the development of the results contained in this paper. Consider G_1 in Figure 6. There are two edges that connect process 1 to processes 3, 4, and 5. These two edges partition the events in processes 3, 4, and 5 into 4 regions such that events from the same process in each region have the same consistency relationship with respect to the rest of the world. In other words, each region contains one partition from process 3, process 4 and process 5. For example, one of the regions, say region r_1, is $(\{s_{31}, s_{32}, s_{33}\}, \{s_{41}, s_{42}\}, \{s_{51}, s_{52}, s_{53}, s_{54}\})$. In this region, s_{41} and s_{42} have the same consistency relationship with respect to all other events, and the same holds true for s_{31} through s_{33}, and s_{51} through s_{54}. On the other hand, s_{31}, s_{41},



Regions in G1:

r1 = ({s51, s52, s53, s54}, {s41,s42}, {s31, s32, s33})

r2 = ({s51, s52, s53, s54}, {s41, s42}, {s34, s35})

r3 = ({s51, s52, s53, s54}, {s43}, {s31, s32, s33})

r4 = ({s51, s52, s53, s54}, {s43}, {s34, s35})

(label1(s21), label2(s21), label3(s21), label4(s21)) = (4, 0, 7, 1)

Figure 6: Regions in G_1 and the label for s_{21}.

and s_{52} belongs to region r_1, among others. We call (s_{31}, s_{41}, s_{52}) a *partial cut* through region r_1 in G_1. The number of cuts containing this partial cut and some event, say s_{21} in process 2, could be easily deduced and used as a label of s_{21}. Let $label_1(s_{21})$ be the number of cuts that contain s_{21} and a particular partial cut of region r_1 in G_1. For example, this is equivalent to asking the question: how many events in process 1 are consistent with event s_{21} and events in r_1. We proceed to answer this question by analyzing the edges in G_1. Edge (s_{42}, s_{15}) eliminates $\{s_{15}, s_{16}, s_{17}\}$ leaving four events in process 1 to be consistent with r_1 and s_{21}. Thus $label_1(s_{21}) = 4$. Similarly, due to edge (s_{13}, s_{22}), $label_1(s_{22}) = label_1(s_{23}) = label_1(s_{24}) = label_1(s_{25}) = 1$. A similar calculation produces the labels of other events in process 2, as indicated also in the Figure 6.

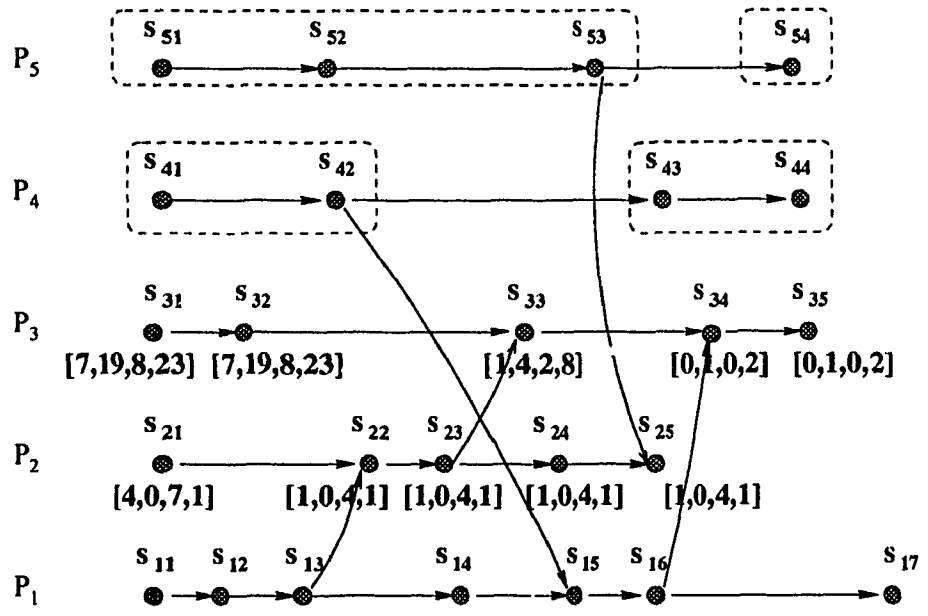
As one migrates to the next iteration involving G_2 shown in Figure 7, two additional edges are introduced: (s_{23}, s_{33}) and (s_{53}, s_{24}). Because of (s_{53}, s_{24}), events in process 5 are partitioned into two subsets: $\{s_{51}, s_{52}, s_{53}\}$, and $\{s_{54}\}$ respectively. This leads to four regions in processes 4 and 5, as shown in Figure 7. The labels of events in process 3 in G_2 can be computed using the labels of the events in process 2 computed in the earlier iteration on G_1. For example, $label_2(s_{32}) = label_3(s_{21}) + label_3(s_{22}) + label_3(s_{23}) + label_3(s_{24}) = 19$. This is because events in region 2 and s_{32} in G_2 belong to region 3 in G_1, and the two additional edges introduced in G_2 eliminate s_{25} as a possible consistent event with respect to region 2 in G_2. In other words, region 2 and s_{32} together select the sub-label of events in process 2 to be used while the edges introduced in G_2 select the events in process 2 to be used in the computation. The generalization of this idea is presented as a theorem. For convenience, we write $s_{ij} \parallel r_k$ to represent the consistency of s_{ij} with respect to events contained in r_k.

Theorem 1

$$label_k(s_{(i+1)j}) = \sum_{s_{ij'} \parallel r_k in G_i} label_{k'}(s_{ij'}) \qquad (1)$$

where $r_{k'}$ in G_{i-1} include events in r_k in G_i and $s_{(i+1)j}$.

Proof: Consider a particular partial cut through region k and event $s_{(i+1)j}$ in G_i. The number of cuts in G_i containing this partial cut is given by the extension of the



Regions in G2:

r1 = ({s51, s52, s53}, {s41,s42})

r2 = ({s51, s52, s53}, {s43})

r3 = ({s54}, {s41, s42})

r4 = ({s54}, {s43})

label2(s32) = label3(s21) + label3(s22) + label3(s23) + label3(s24)

= 7 + 4 + 4 + 4 = 19

Figure 7: Regions in G_2.

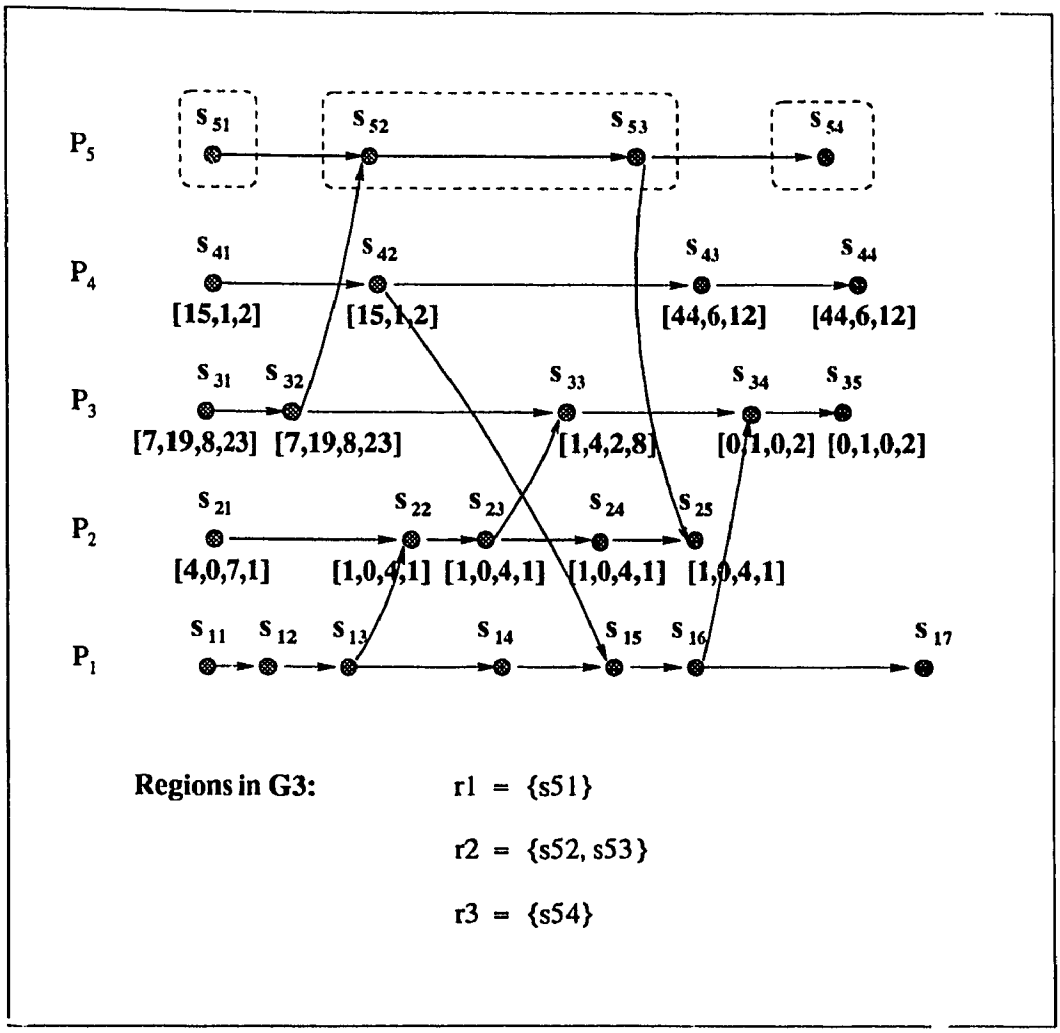


Figure 8: Regions in G_3.

partial cut to processes 1 through i. From Lemma 1, all we need to consider are those extensions that go through each $s_{ij'}$ which is consistent with events contained in r_k.

QED

Figure 8 and Figure 9 shows the application of the above labeling function in G_3 and G_4.

Corollary 1 *$label_1(s_{nj})$ gives the number of cuts passing through s_{nj} in G, where n is the number of processes in G.*

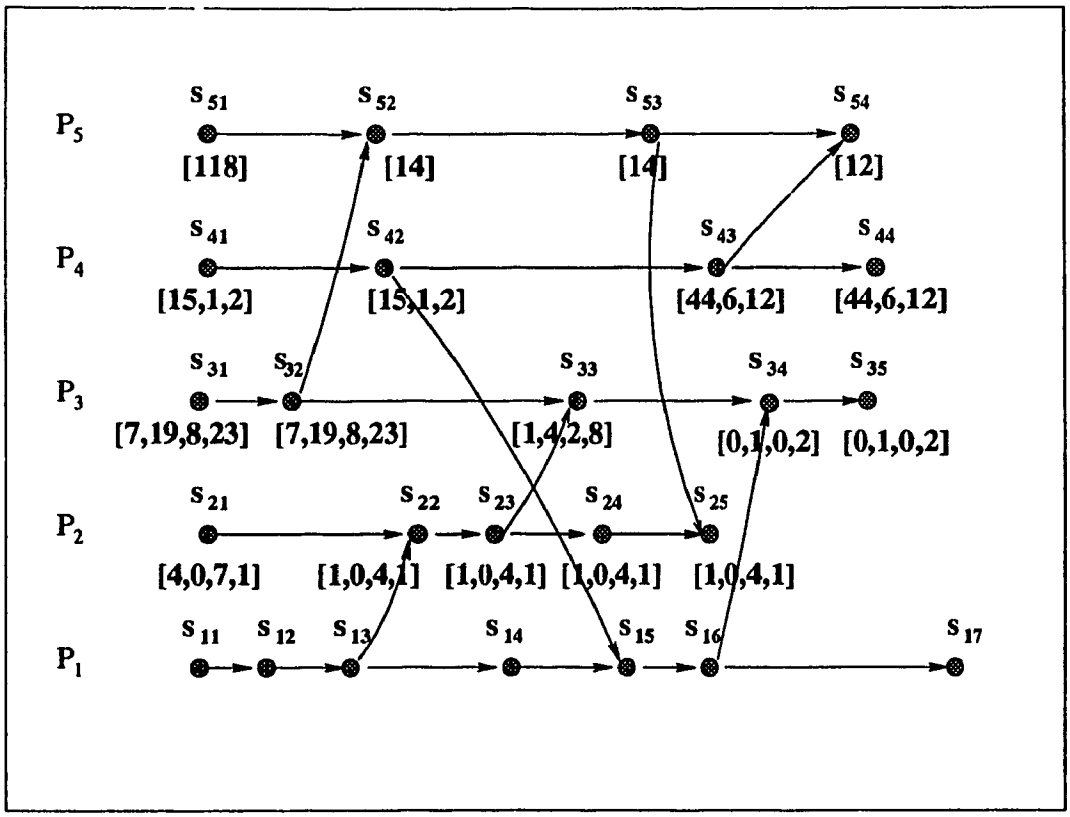


Figure 9: Regions in G_4.

Proof: $G_{n-1} = G$ [by construction]. Thus the claim.

QED

The exact number of global states of a given distributed execution can therefore be computed using the labeling function defined. For the example illustrated in Figure 6 through Figure 9, the total number of global states [consistent cuts] is given by 118 + 14 + 14 + 12 = 158.

The complexity of the labeling procedure grows with the sizes of the labels associated with the events, and thus the number of regions in G_i. The following lemma gives an upper bound on the size of the label of event s_{ij}. Let x_j be the number of edges in A_i that either starts from or ends at process j.

Lemma 2 *The size of the label associated with event $s_{(i+1)j}$ in G_i is bounded by*

$$\prod_{j>i+1} [(x_j) + 1] \tag{2}$$

Proof: From construction, each edge from or to process j must come from or end at some process $k < i + 1$. These edges partition process j into $(x_j) + 1$ subsets, leading to the claim. QED

From Lemma 2, we observe that unfortunately computing the exact number of cuts of a given execution may still invoke exponential cost. To make the task practicable, an alternate approximation technique is therefore desirable.

3.3.2 Upper Bound Estimation

Since the complexity of state counting is dependent on x_j in G_i, we could proceed with an approximation strategy that aims at minimizing the number of edges above $i + 1$ in G_i which would link processes beyond i+1 to those below i+1. We further observe that for any two given consistency graphs, G=(N,A) and G'=(N,A'), if A' is a subset of A, then a cut in G is also a cut in G'. This is stated as a lemma.

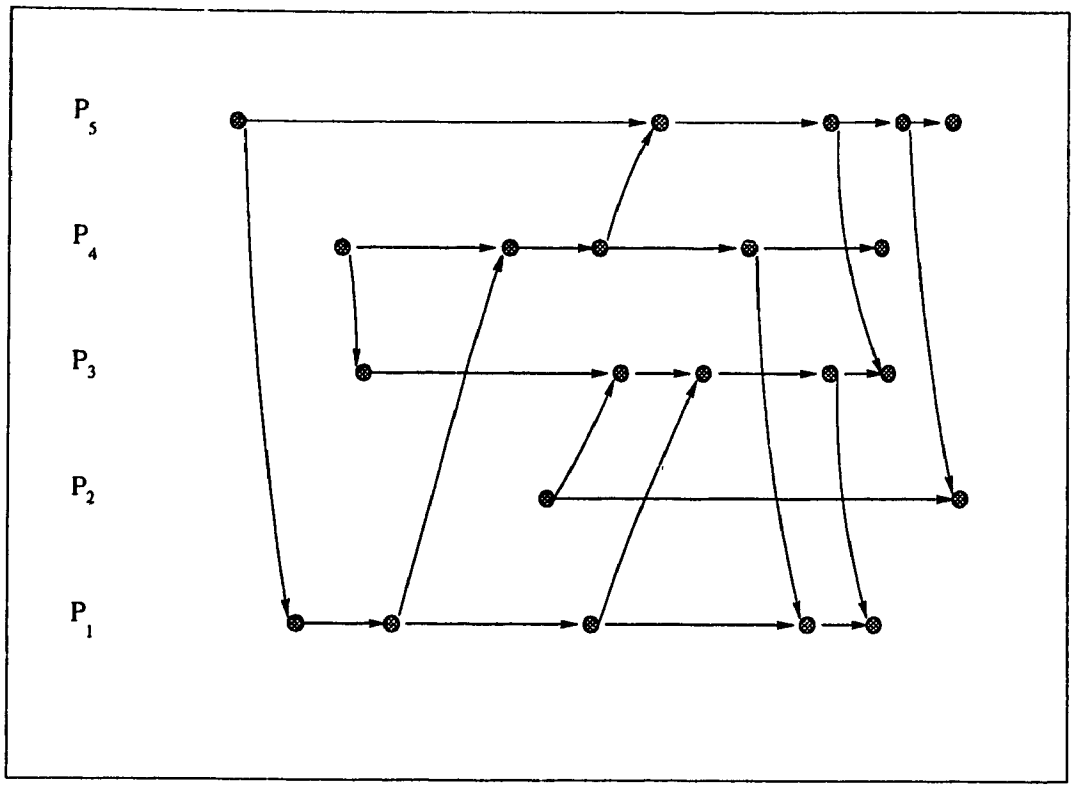


Figure 10: Original G.

Lemma 3 *Given G=(N,A), G'=(N,A') and A' a subset of A, then a cut of G is also a cut in G'.*

Proof: This is immediate from definition; $S_i \parallel S_j$ in G => $S_i \parallel S_j$ in G'.

QED

From Lemma 2 and Lemma 3, effective complexity reduction can be obtained by eliminating edges that span multiple process levels beyond process (i+1) in G_i. As an example, consider Figure 11. G_i is obtained from G_{i-1} by adding only those edges that connect process i to either process i+1 or process i+2. Suppose $y_{(i+2)}$ is the total number of edges that connect process i and process i+2. Then the size of the label associated with $s_{(i+1)j}$ in G_i is equal to $y_{(i+2)}+1$, since the events in process i+2 have been partitioned into $y_{(i+2)}+1$ subsets and those in process $j > i+2$ form a single partition. The complexity of estimation is then bounded also by a simple polynomial function of the number of events. In Figure 11, an upper bound is computed as shown.

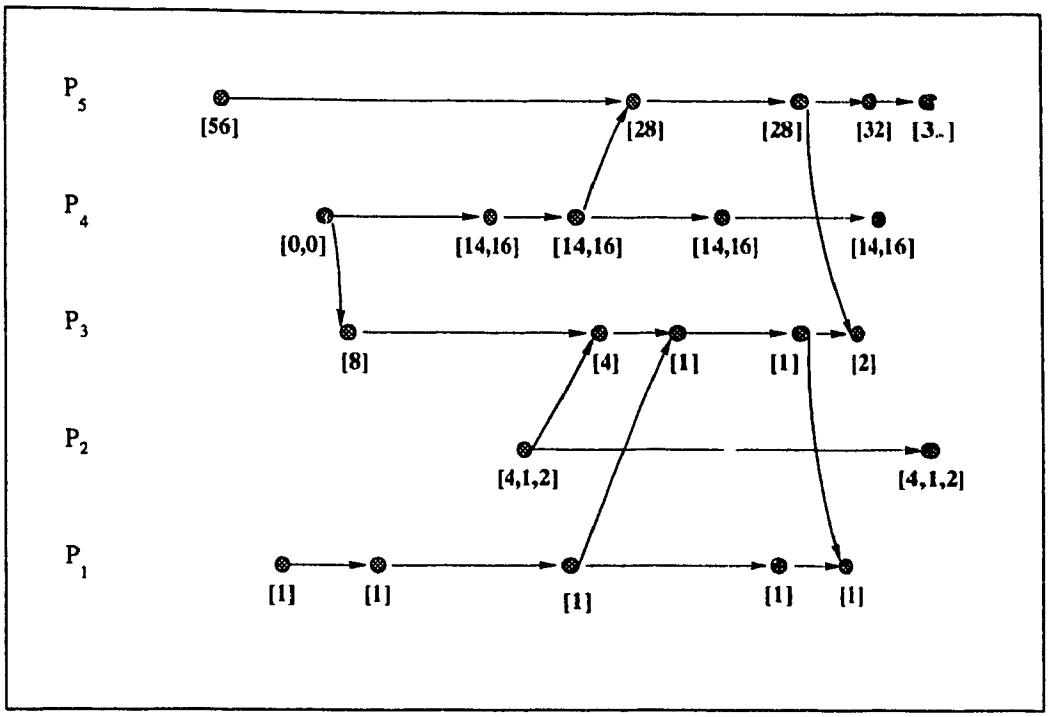


Figure 11: G' with removal of edges crossing more than one level in G.

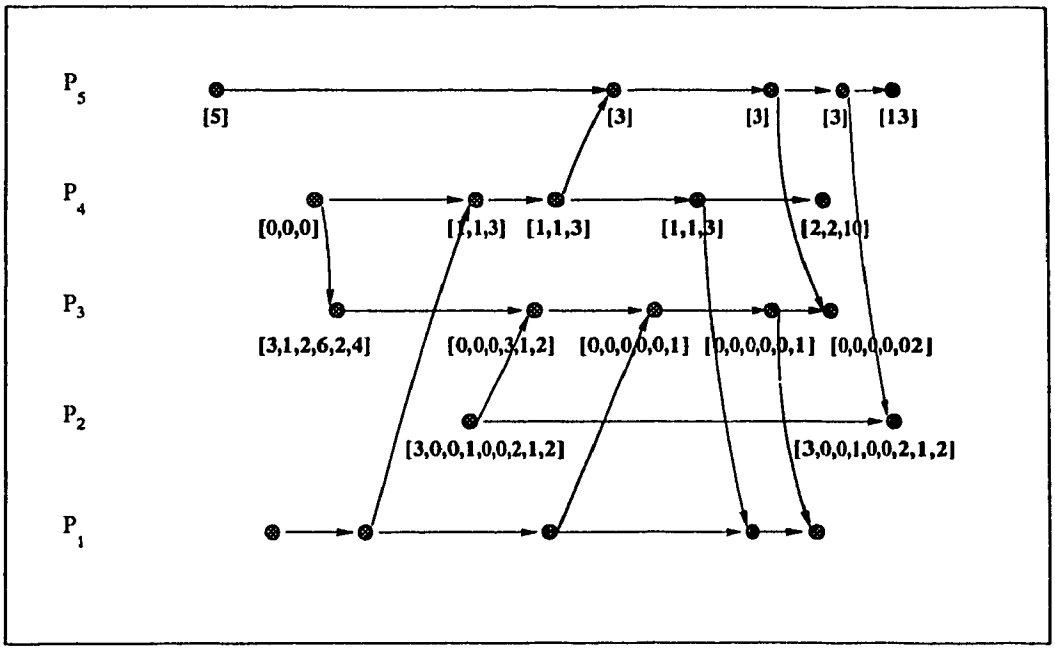


Figure 12: G" with removal of edges crossing more than two levels in G.

We could increase the accuracy of the estimation by allowing edges from more than one process to cross process i+1 in G_i. For the same example G in Figure 10, if we allow edges from two processes above process i+1 to cross the latter to process i, then the estimation is shown in Figure 12 and is a more accurate bound than the earlier one, obtained at a slightly higher cost reflected by the size of the labels shown.

3.3.3 Lower Bound Estimation

In the upper bound estimation strategy we reduce complexity by eliminating edges that span multiple process levels beyond process i+1 in G_i. In this section we present another approximation strategy where we replace each such edge with a chain of two or more edges.

Let $G = (N, A)$ and $G' = (N', A')$ be two Consistency Diagrams such that N is a subset of N' and A^* (i.e. the transitive closure[1] of A) is also a subset of A'^*. In other words, G' is obtainable from G by an augmentation procedure that introduces additional nodes and edges (which are transitively reduced afterwards to produce A').

Lemma 4 *Every consistent cut $C = (s_1, \ldots, s_i, \ldots, s_n)$ in G' containing only nodes which are also found in G is a consistent cut in G as well.*

Proof: Suppose otherwise. Then, there is at least a pair of nodes in C say, s_i and s_k, such that $s_i \rightarrow^* s_k$ in G but not in G'. This immediately contradicts the assumption that A^* is a subset of A'^*. Thus the claim.

QED

Based on Lemma 4, we aim to transform a given G into a new G' whose state size can be easily computed. The latter is then a lower bound of the state size of G. The transformation strategy is presented below.

[1]The transitive closure of any A denoted by A^* contains the pair (s_i, s_j) if and only if there is a path in A such that $s_i \rightarrow^* s_j$.

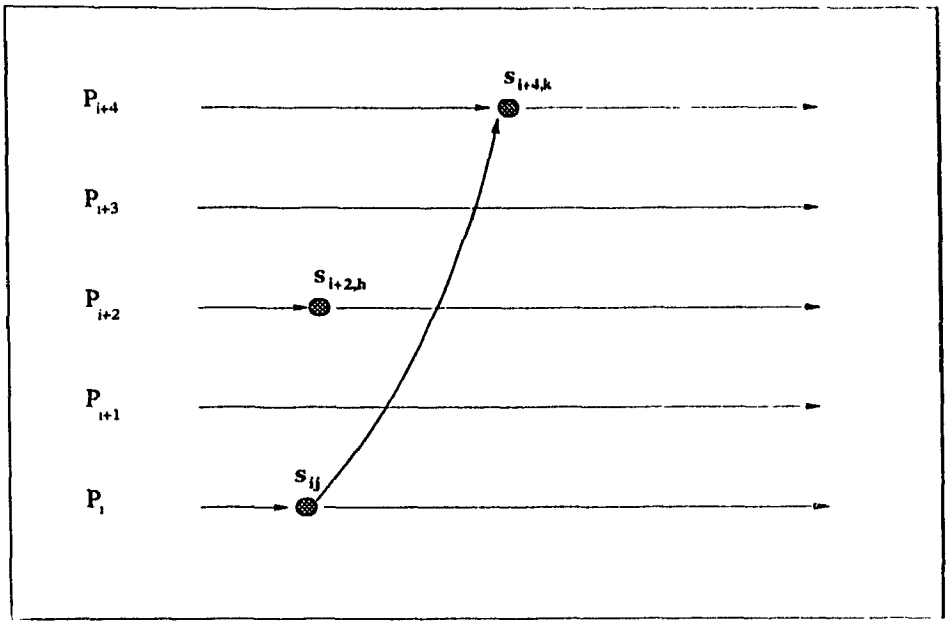


Figure 13: Original G.

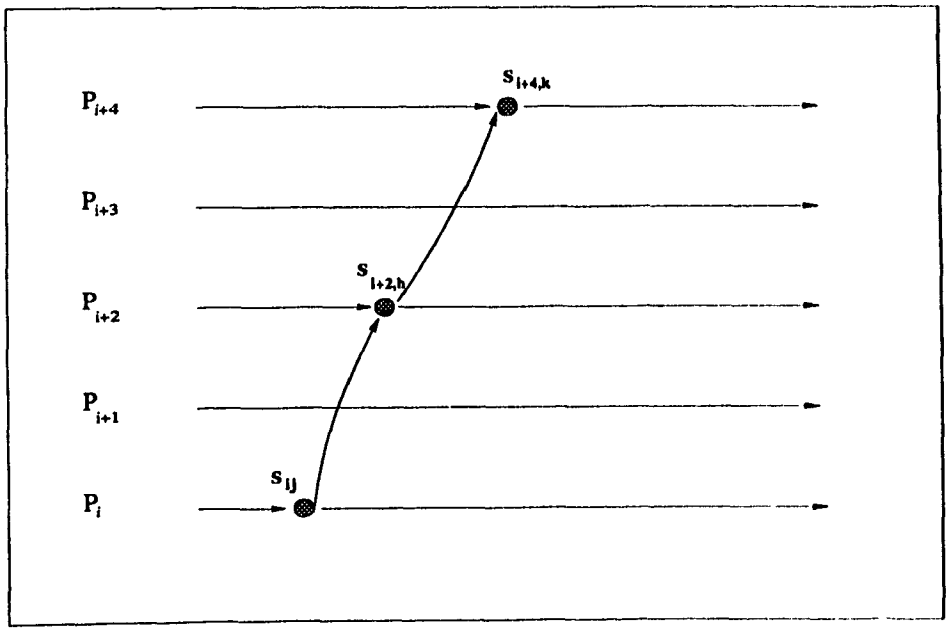


Figure 14: G is transformed into G'.

From Lemma 2, stated in Section 3.3.1, we know that if the process-distance (abbreviated by p-distance) between two nodes connected by an edge is at most two (i.e., edges do not cross more than one process), then the complexity for the computation of the state size is bounded by a simple polynomial function of the number of events. So, in transforming G to G', we try to preserve this property, i.e., every pair of nodes in G' connected by an edge has a p-distance of at most 2.

A simple strategy to proceed in the augmentation process is to replace every edge connecting two nodes whose p-distance is more than 2 by a sequence of edges such that the nodes involved have p-distances of at most 2. This is possible, for example, in Figure 13. The edge which crosses more than one process is replaced by some edges which have p-distances of at most 2, as shown in Figure 14. The transformation leads to a G' that satisfies Lemma 4 and its state size serves as a lower bound to the original G. The replacement procedure proceeds as follows:

Consider edge $s_{ij} \rightarrow s_{i+4,k}$ in G in Figure 13 again. We look for an earliest event node in process i+2, say $s_{i+2,h}$, such that $\neg(s_{i+2,h} \rightarrow^* s_{ij}) \wedge \neg(s_{i+4,k} \rightarrow^* s_{i+2,h})$ is true. We now replace $s_{ij} \rightarrow s_{i+4,k}$ by $s_{ij} \rightarrow s_{i+2,h}$ and $s_{i+2,h} \rightarrow s_{i+4,k}$. This replacement leads to a graph G' whose relationship with G satisfies Lemma 4. This is stated as another Lemma.

Lemma 5 *Let* $G = (N, A)$ *and* $G' = (N, A')$ *such that*

$$A' = A - \{(s_{ij}, s_{i+4,k})\} \cup \{(s_{ij}, s_{i+2,h}), (s_{i+2,h}, s_{i+4,k})\}.$$

If $\neg(s_{i+2,h} \rightarrow^* s_{ij}) \wedge \neg(s_{i+4,k} \rightarrow^* s_{i+2,h})$ *is true in* G, *then* G' *is also a consistent (i.e., acyclic) graph.*

Proof: The difference between A and A' is the additional edges from $s_{i+2,h}$ to $s_{i+4,k}$ and from s_{ij} to $s_{i+2,h}$. Since the reverse does not exist in G in both cases, the resultant graph G' is still acyclic and it is a consistent graph.

QED

Unfortunately, this direct replacement is not always feasible. As it turns out, sometimes, no such event in process i+2 may be identifiable. Figure 15 and Figure 16

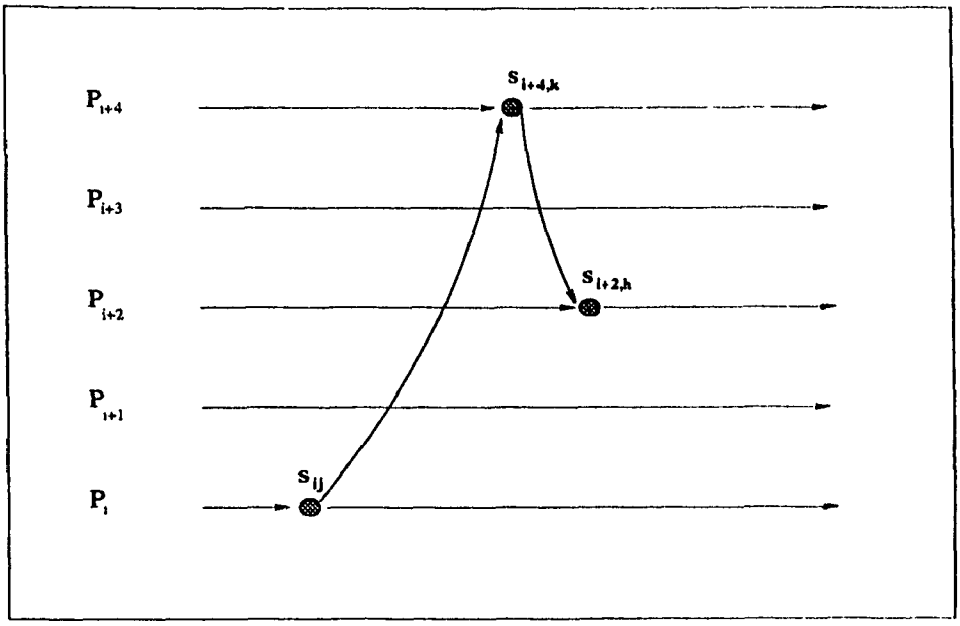


Figure 15: Irreplaceable $s_{ij} \rightarrow s_{i+4,k}$ in G.

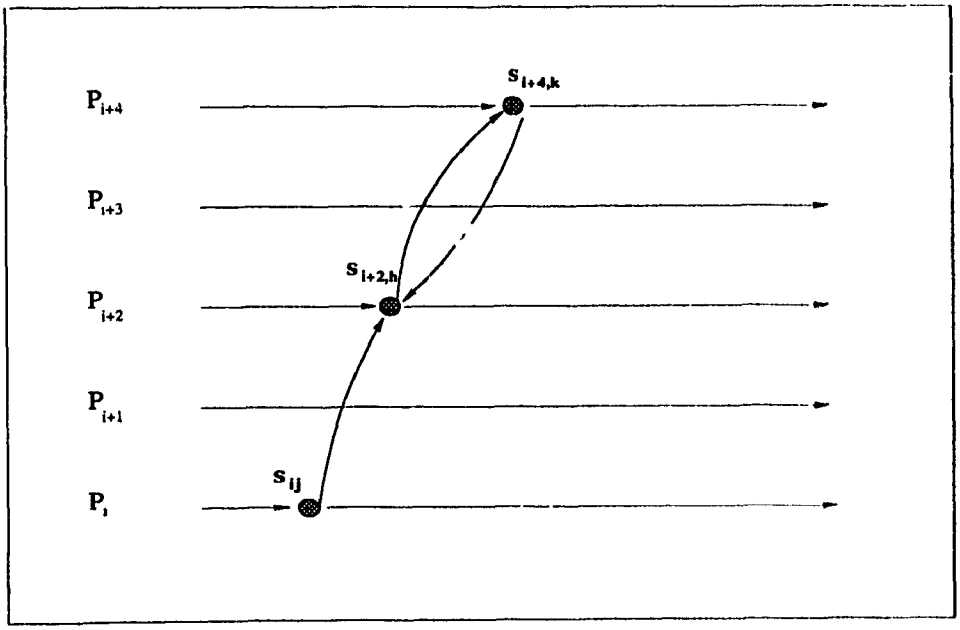


Figure 16: G' is not a consistent graph.

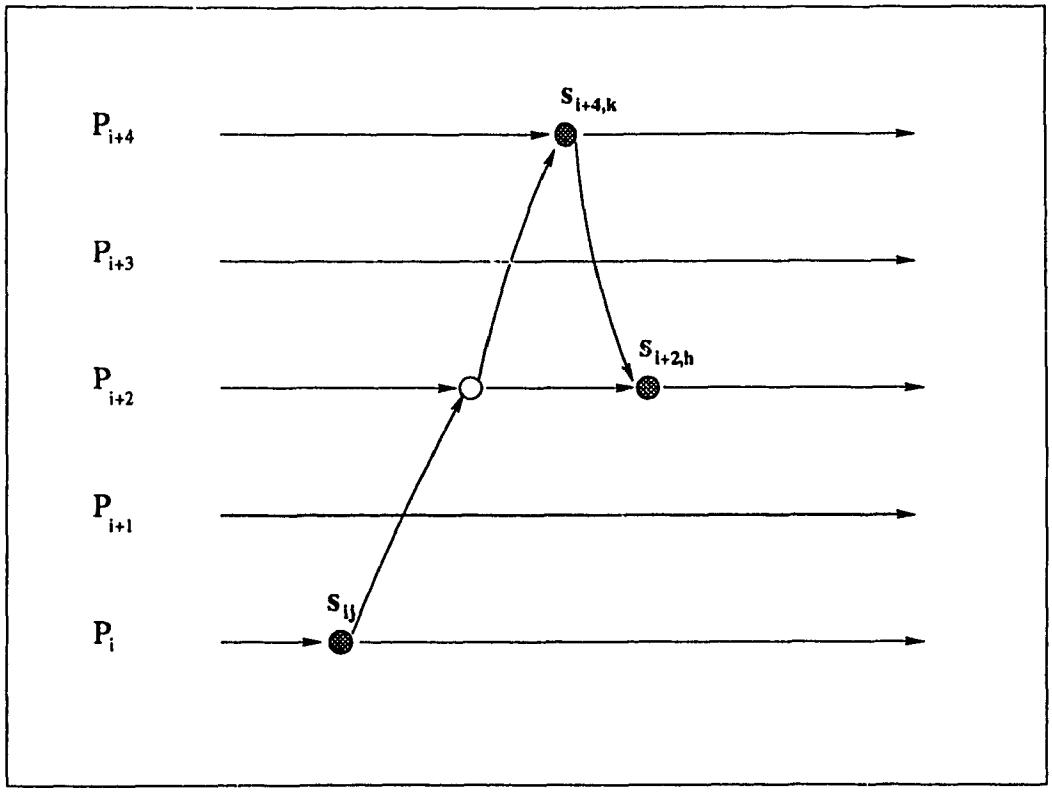


Figure 17: G' augmented with an additional node.

illustrates an example of this. In G' a cycle is present since $s_{i+2,h} \rightarrow s_{i+4,k}$ and $s_{i+4,k} \rightarrow s_{i+2,h}$. So G' is not a consistent graph, which is clearly unacceptable. To produce the desired new graph G' under such a case, we introduce an additional node in process i+2 whose relationships with respect to s_{ij} and $s_{i+4,k}$ satisfy the condition in Lemma 5. An acceptable transformation of G is shown in Figure 17. An additional node is denoted by an empty circle in the figure. It is always possible to perform such an insertion uniquely at process i+2, as will be proved next.

Lemma 6 *If no event in process i+2 satisfies the condition in Lemma 5, then the earliest event in process i+2 reachable from $s_{i+4,k}$ must come immediately after the latest event in process i+2 that could reach s_{ij}. Between these two events, a new event could be inserted to satisfy the condition in Lemma 5.*

Proof: Suppose $s_{ij} \rightarrow s_{i+4,k}$ while no event in process i+2 satisfies the condition stated in Lemma 5. Let $s_{i+2,j'}$ be the latest event reachable to s_{ij} and $s_{i+2,j''}$ be the earliest event reachable from $s_{i+4,k}$. Since $s_{ij} \rightarrow s_{i+4,k}$, this implies that $j' < j''$, otherwise, a cycle exists running through s_{ij}. Moreover, $j'' = j' + 1$, i.e., no event exists between $s_{i+2,j'}$ and $s_{i+2,j''}$; otherwise, we would have found an event in process i+2 satisfying the condition in Lemma 5. Thus we could insert a new event between $s_{i+2,j'}$ and $s_{i+2,j''}$ and in the augmented graph, this event would satisfy the condition in Lemma 5.

QED

When there is no more edge in G to be replaced, then the number of cuts in G' is computed by the labeling procedure described in section 3.3.1 with a slight modification. According to Lemma 1 we derive all consistent cuts in G' which includes nodes in G.

Similar to the upper bound estimation, we could increase accuracy of the lower bound estimation by allowing edges to cross more than one process. It is straightforward, therefore it is not elaborated further here.

Obviously, the cost of computing the lower bound comes from two factors: the cost

of the edge replacement followed by the cost of the labeling procedure. The complexity of the edge replacement procedure grows with the number of edges connecting different processes in the partial order which is bounded by k^2, where k denotes the number of events in the given Consistency Diagram. From Lemma 2 and G' the cost of the labeling procedure remains polynomial

3.4 Summary

In this chapter an algorithm is presented to compute the size of the state space when an arbitrary Consistency Diagram is given. A labeling function is defined for the computation of the exact number of global states by the successive iterations of a sequence of G_i. The labeling procedure grows with the size of the labels associated with the events. The size of the label is the product of the number of edges located at different processes, considering only those edges which cross process i+1 in G_i. As it is implied, the labeling procedure may invoke exponential cost. Therefore, upper and lower bound estimations are proposed. In the upper bound estimation strategy we reduce complexity by eliminating edges that span multiple process levels beyond process i+1 in G_i. In the lower bound estimation we transform G into G', where no edge crosses multiple process levels in G'. Then the computation of the number of cuts in G' – without considering additional events – is a lower bound of the state size of G.

As it is stated, in both strategies, the level of accuracy of the estimation procedure is reflected by the size of the labels. It gives a certain flexibility to the user to select the appropriate accuracy-complexity level for the given distributed execution.

Chapter 4

Easy cases

As examined in the previous chapter, in some cases the state space calculation requires exponential cost. However, in practical systems, it may not exhibit such an intolerable complexity. This could be due to the synchronization design of the distributed system where processes tend to communicate locally in the form of clusters. In other words, the topology of communication among processes could play an important role in terms of complexity.

Static Network topology is the most suitable for multiple computing nodes of a distributed system. Static networks are formed of point-to-point direct connections which will not change during program execution. In general, a network is represented by the graph of a finite number of nodes (processes) linked by undirected edges. The number of edges (links or channels) incident on a node is called the node degree. It is a worthwhile task to investigate, if there are systems among these, where the identification of the state space can **always** be done in polynomial time.

According to the topology of communication between processes this section presents several cases, where the exact number of states can be computed easily (within polynomial bounded space and time). From the computation, then one could proceed to decide whether distributed predicate detection is practicable in a given situation.

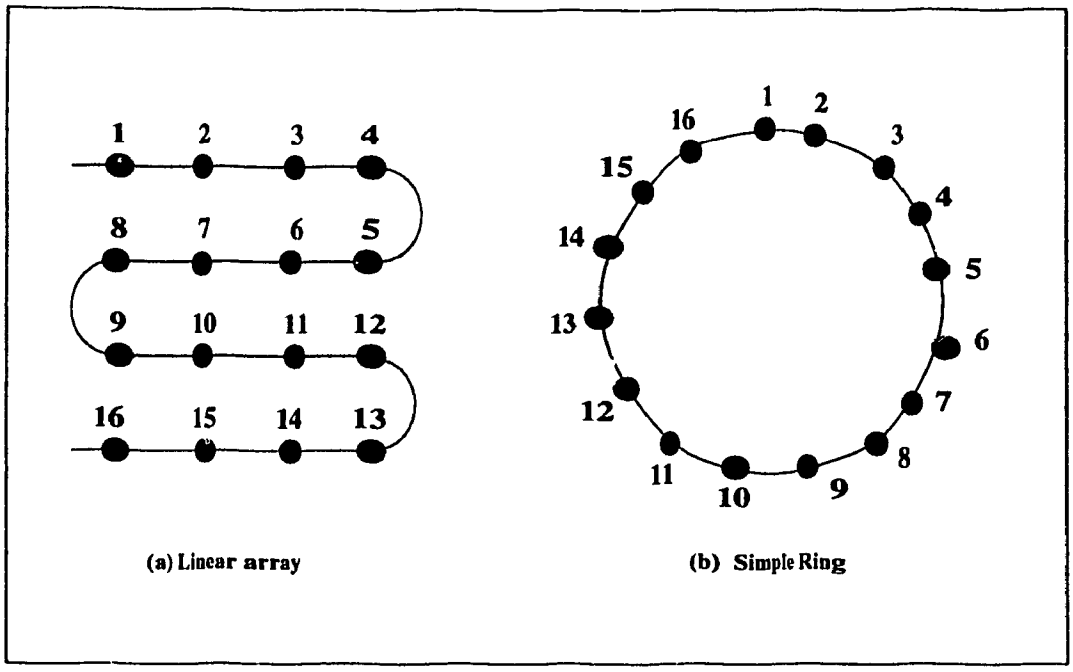


Figure 18: Linear Array and Simple Ring.

4.1 Linear Array

Linear arrays are the simplest connection topology. Each internal node has a degree of 2, and a terminal node has degree of 1 as shown in Figure 18a. If the application involves processes which communicate in the form of a linear array, then from Lemma 2, the size of a label in process i+1 in G_i is exactly 1 (no edge crossing process i+1 in G_i). Thus the cost of computing the exact number of states is at most quadratic to the number of events in G.

4.2 Ring

Among the various possibilities, in this section we analyze the most well known types of Ring topologies, in terms of complexity for the state space identification. Notice that these types of rings differ only in their node degree.

4.2.1 Simple Ring

A ring is obtained by connecting the two terminal nodes of a linear array with one extra link, as shown in Figure 18b. It is symmetric with a constant node degree of 2. If the processes operate in the form of a ring with m being the number of edges connecting process 1 and process n, then the cost of computing the exact number of state is bounded by m times that of the linear array, since the size of a label is at most m.

4.2.2 Chordal Ring of degree 3

By increasing the node degree from 2 to 3, we obtain a chordal ring as shown in Figure 19 /a. The chordal ring of this type is having links as $(i, i+1)$ for $i = 1, 2, \ldots, n-1$, and $(n, 1)$. Moreover, there are links $(i, i+3)$ where $i < n-1$, and link $(n-1, 2)$. The size of the label is m^3. Consider the possible inconsistencies for the chordal ring shown in Figure 19a and G_i. The three processes from which edges could cross process i+1 is n, $n-1$, and either $i+2$ or $i+3$. Then, according to Lemma 2, $m \times m \times m = m^3$. Then the complexity of the labeling procedure is bounded by m^3 times that of the simple ring.

4.2.3 Chordal Ring of degree 4

A chordal ring with node degree 4 is shown in Figure 19b. It has links as $(i, i+1)$ for $i = 1, 2, \ldots, n-1$, and $(n, 1)$. Moreover, there are links $(i, i+4)$ where $i < n-3$, and links $(n-3, 1)$, $(n-2, 2)$, $(n-1, 3)$, and $(n, 4)$. The size of the label is m^7, as illustrated in the Figure.

It is evident that as we increase the node degree, we obtain a higher complexity in the labeling procedure. As an example, if we increase the node degree from 4 to 5, which is the well known Barrel shifter network topology, the size of the label is $= m^{10}$, which is not shown.

4.3 Binary Tree

Suppose the processes in a binary tree are identified according to pre-order traversal of the tree, where the node degree is 3. Then in G_i, at most two processes beyond

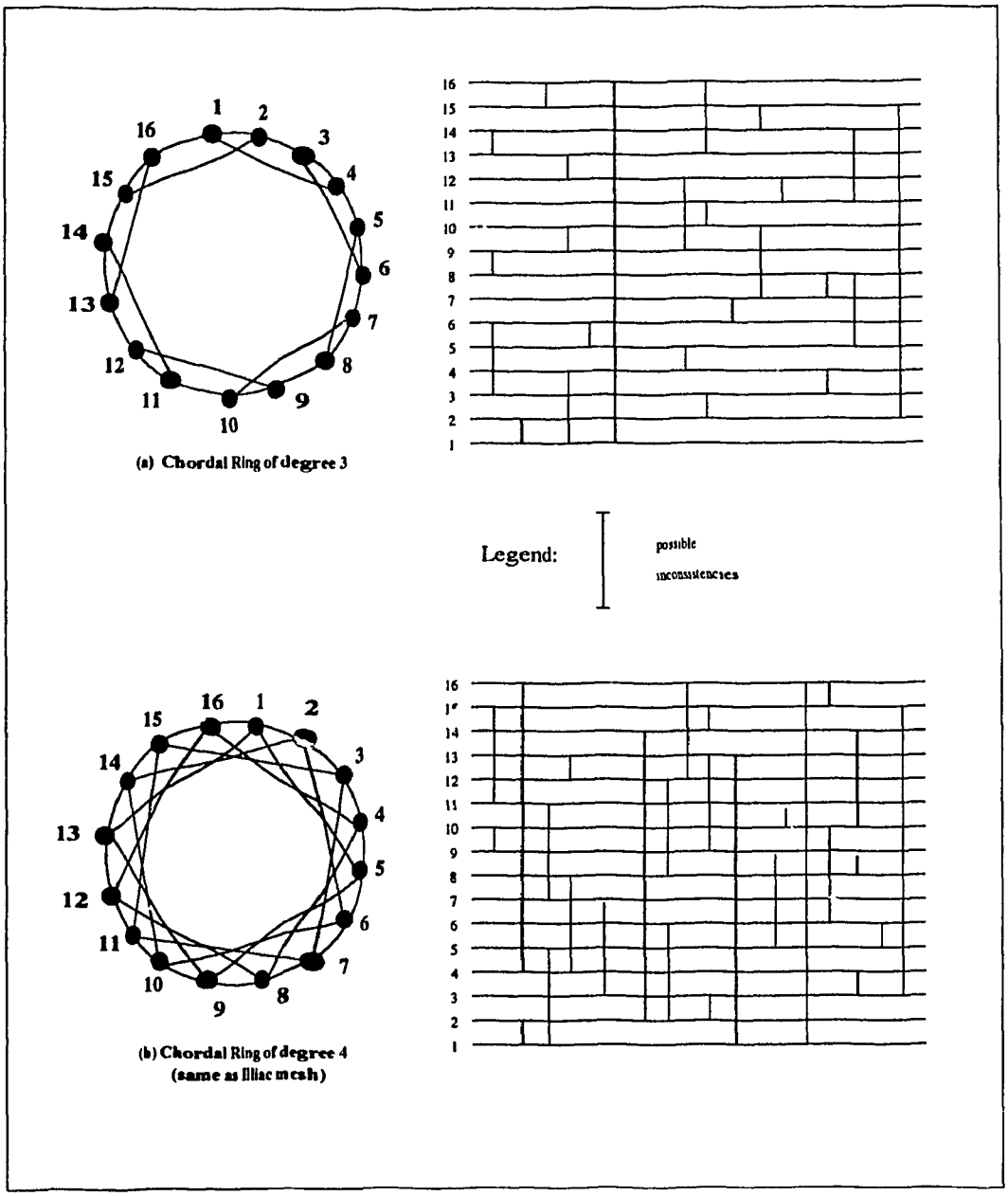


Figure 19: Chordal Rings.

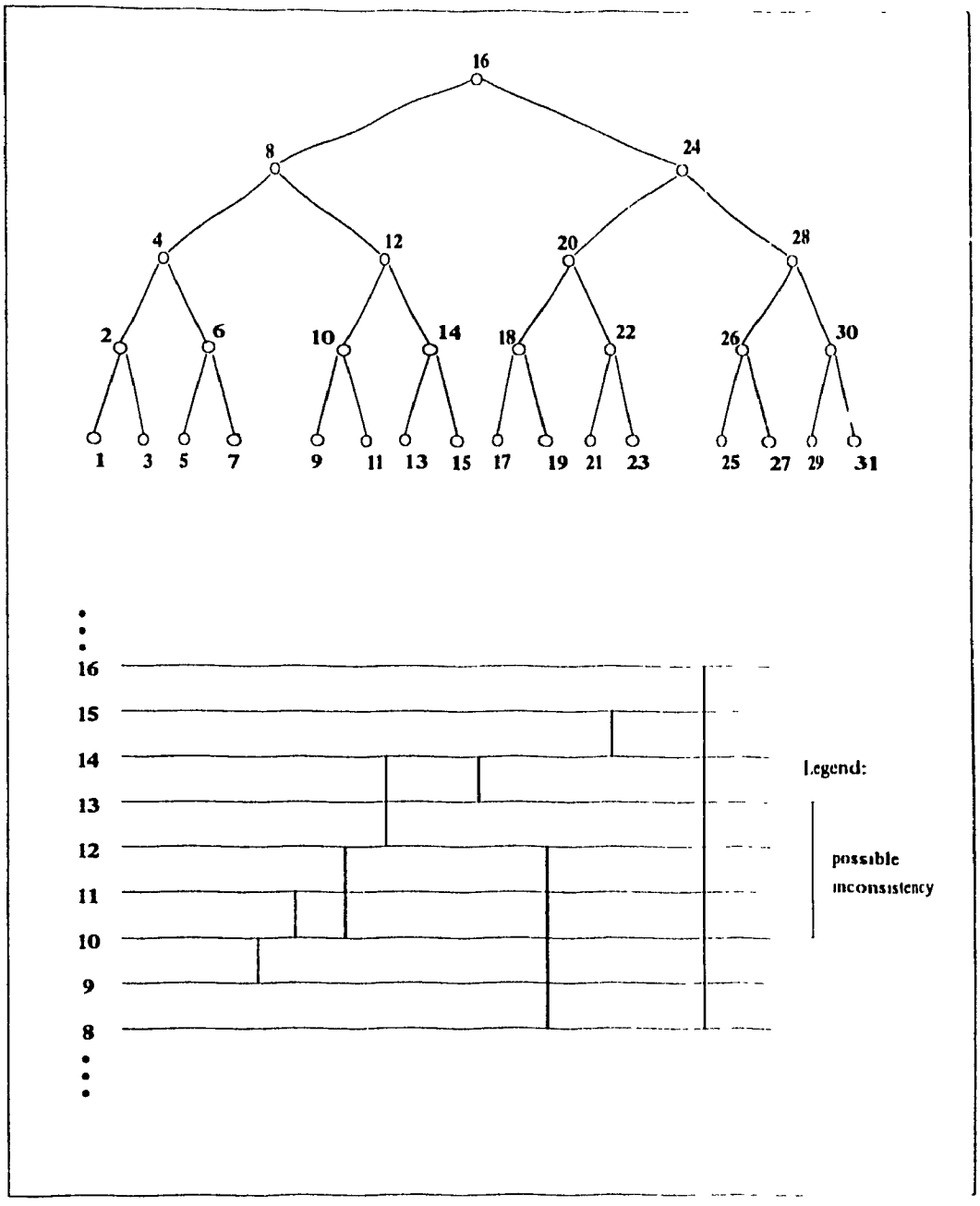


Figure 20: Binary Tree with 31 nodes. (Possible edges are exemplified above.)

process i+1 would have edges connecting to process below i+1. Thus the complexity of the computation is bounded by m^2. Figure 20 illustrates this claim.

Similarly as to the ring, we could increase the node degree in the tree, however the complexity of the labeling function increases as well.

4.4 Conclusion

In this chapter we investigated the types of network topologies, where the processes tend to communicate locally in the form of clusters. In all of the presented cases, the computation procedure for identifying the states of the system is always done in polynomial time and space. The significance of the above are twofold:

1. In systems where processes are arranged in one of the topologies discussed above, at the detection stage, we only have to be concerned with the complexity which comes from the nature of the distributed predicate.

2. In the design of a distributed system one could consider the above knowledge to ease development cost, if the nature of the application allows us to do it.

Chapter 5

Distributed Predicate Detection.

5.1 Introduction.

Predicates are constructed so as to encode system properties of interest in terms of state variables. When these variables are distributed among the processes, they become distributed predicates. A predicate is satisfied if there are one or more global states where the predicate evaluated to true during execution.

Distributed predicate detection is useful in many applications. In distributed debugging, it is useful for the purpose of locating errors which reveal themselves in the form of erroneous states (states that satisfy some distributed predicate). A distributed program is considered to be in error when the execution has entered a state which violates the safety requirements of the program. In general, safety violations could arise as a consequence of design or programming errors.

The complexity of detecting distributed predicates is often linked with the size of the global state space. As is well known, the size of the global state space in some cases may be exponential in terms of the number of processes in the system [CM91]. However, the complexity of detection is affected by not only the size of the global state space but also of the predicate to be detected.

This chapter attempts to address the problem of predicate detection in general. It is demonstrated how the labeling method presented in Section 3.3.1 can be modified

to solve this problem. It is illustrated here that even if the size of the global state space could be computed in polynomial time (as discussed in Chapter 4), predicate detection may be expensive for certain types of predicates. Some strategy is proposed to deal with this problem in practice.

5.1.1 Distributed predicates

A predicate is an assertion defined on the local variables of one or more processes. In the Distributed Memory model, each variable is owned by a distinct process where it resides.

Predicates can be constructed hierarchically. A local predicate (denoted by LP) is a general boolean expression defined over the local variables of a process, whereas a distributed predicate (denoted by DP) is defined on variables distributed over multiple processes.

Two classes of basic distributed predicates emerge:

1. a distributed predicate formed by the conjunction/disjunction of local predicates defined locally in the process, say LP_i in process i.
   For example, $(LP_i \ AND \ LP_j) \ OR \ LP_k$.

2. a distributed predicate involving non-boolean variables.
   For example $x_i + x_j > x_k$, where variable x_i is located in process i.

These basic distributed predicates can recursively generate general (distributed) predicates (denoted by Φ) using the following grammatical rules:

$$\Phi = DP \ \mid \ [LP_i | DP \wedge (AND|OR|NOT) \wedge LP_j | DP]$$

where

$$DP = ((LP_i \wedge (AND|OR|NOT) \wedge LP_j) \ \mid \ (x_i \wedge (< \mid > \mid =) \wedge x_j))$$

EXAMPLES:

1. Mutual exclusion problem: to check if process 1 and process 2 could be in the critical section at the 'same time': $\Phi = LP_1 \ AND \ LP_2$ where $LP_i (i = 1, 2)$ denotes local predicate which is satisfied if and only if process i is in the critical section.

2. Resource allocation: $x_1 + x_2 +, \ldots, + x_n = C$, where x_i is a integer valued variable located at process i denoting the number of units of resources held by the process, and C is the total number of units of the resources available.

3. During debugging the user may want to halt the execution at a state when $\Phi = 3 + x_1 + x_2 > 10$ becomes true.

4. Often 3-Phase commit protocol is used to commit a transaction (T) in a distributed database system.
   Then the predicate $[Committed_1(T) \vee Committable_1(T)] \wedge [Committed_2(T) \vee Committable_2(T)] \wedge [Committed_3(T) \vee Committable_3(T)]$ may be checked to be true if one expects a transaction T_i (executed on 3 sites) to commit in a particular execution.

5.1.2 Complexity

Concurrency in a system often introduces complexity in analysis; sometimes the complexity so generated can be 'tamed' by skilful design of analysis algorithms [PL90, GJ79]. But in some cases, the complexity is inherent and cannot be avoided. The main reason behind the high complexity is the exponential growth of the number of system states with concurrency. If examination of all states is inevitable, then the cost of analysis will be expensive. Consider the following example shown in Figure 21: there are n processes each with two local states and all local states not belonging to the same process are consistent. The total number of cuts [and thus global states] of the system is 2^n. Certainly with more events in each process, the size will grow even faster. Suppose there are k events at each process, then the size of the state space is k^n. Indeed, the problem of detecting if a given distributed system has a state satisfying the predicate specified in the following Lemma is NP-complete.

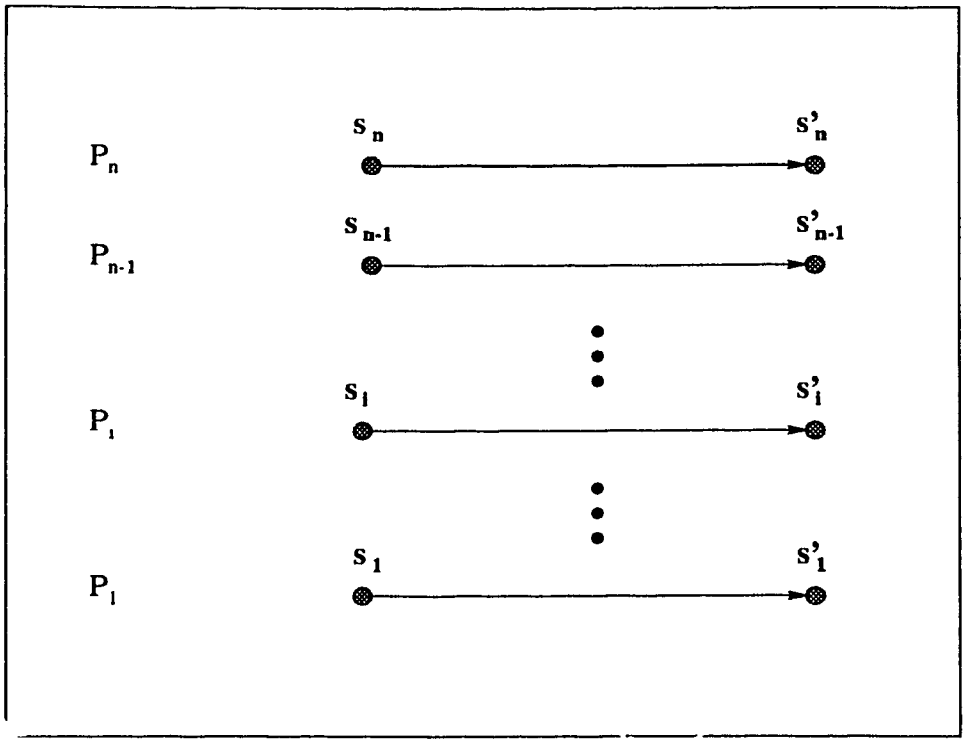


Figure 21: System with 2^n cuts.

Theorem 2 *The problem of detection if a given distributed system has a state satisfying a general predicate of the form* $S_1 + S_2 + \ldots + S_n = B$ *is NP-complete.*

For the interested reader the proof is provided in [LJ96].

From Theorem 1 and Lemma 2 it is clear that in situations as illustrated in Figure 21, the size of the state space can be computed easily, even with a large number of events. Particularly, if there are no edges between any events in the Consistency diagram, the size of the label is equal to 1. Therefore the cost of identifying the size of the state space is at most quadratic in terms of events. However, Theorem 2 implies, that even if the size of the state space can be identified in polynomial time, the detection procedure may involve intractable complexity for some predicates.

5.2 Application to Distributed Predicate Detection

The labeling method presented in Section 3.3.1 could be used directly as well as indirectly in detecting the occurrences of states that satisfy a distributed predicate.

In direct application, the labeling method is modified to evaluate the satisfaction of the given predicate iteratively from G_1 until G_{n-1}. In this approach, it is assumed that the size of the state space is identifiable with the labeling method in polynomial time. The detection of a predicate may introduce additional complexity compared to the cost of the labeling method. In some simple types of predicate detection the complexity slightly increased (because of the time it takes to evaluate the local predicates), in some other cases it may be increased substantially, but it is still remains reasonable; and it may be intractable (as in Theorem 2).

In indirect application, the labeling method is used as a means to decide if the state space is small enough to be examined somewhat exhaustively for a given predicate. In this approach, it is assumed that the size of the state space can not be identified in polynomial time that we have to deal with.

5.2.1 Direct Application

From the discussion in Section 5.1.2, it is evident that even if one could compute the exact number of states of a given execution within polynomial time, because of the nature of the given predicate, deciding whether there exists a cut satisfying the predicate could involve exhaustive computation. On the other hand, it is possible to detect some special predicates rather easily, such as a conjunctive predicate of the form: $DP = LP_1\ AND\ LP_2\ AND \ldots\ AND\ LP_n$. The decision should be made: if there exists a cut in the execution equal to $(s_1, \ldots, s_i, \ldots, s_n)$ which satisfies the given DP. Here we will demonstrate how such a decision can be made by modifying the labeling method.

The $label_k(s_{ij})$ in G_{i-1} is replaced by $sat_k(s_{ij})$ where $sat_k(s_{ij})$ denotes whether or not

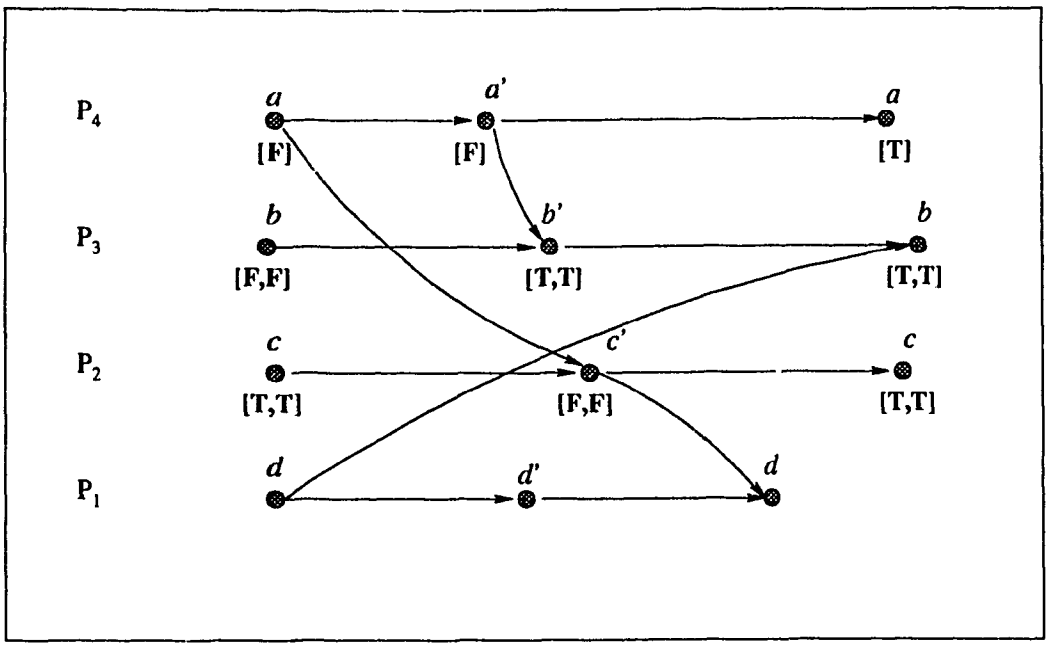


Figure 22: Detecting ab'cd

there exists a cut including events in region k and s_{ij} in G_{i-1} whose local states in processes 1 through i are given by $(s_1, \ldots, s_i)$. In other words, the partial cut satisfies the predicate partially, from S_1 up to S_i. More precisely, $sat_k(s_{ij})$ contains the result of the satisfaction of the partial predicate $DP' = LP_1 \wedge LP_2 \ AND \ldots AND \ LP_i$ on the partial cut $(s_1, s_2, \ldots, s_i)$.

As the labeling migrates to G_i, a new label in G_i stated in Theorem 1 is computed by

$$sat_k(s_{(i+1)_j}) = OR_{s_{i,j'} \| r_k in G_i}\{sat_{k'}(s_{ij'})\} \tag{3}$$

where $r_{k'}$ in G_{i-1} include events in r_k in G_i and $s_{(i+1)j}$.

The complexity increased slightly in the modified labeling procedure since the consistency comparison between each pair of events s_i and s_j is increased with the evaluation of local predicates and comparison of the value of the corresponding LP_i and LP_j. An example is illustrated in Figure 22 for detecting the conjunctive predicate $DP = ab'cd$.

A TRUE value in the label of any event s_{nj} in G_{n-1} indicates that the predicate is satisfied by the execution. This is stated as a Lemma:

Lemma 7 *Φ is satisfied, if and only if there is an event s_{nj}, whose label contains the TRUE value.*

Proof: A predicate could be evaluated either True or False. From Equation 3, as the procedure migrates from G_i to G_{i+1}, consistent cuts which yield a False value to the predicate are excluded. Therefore, at G_{n-1}, DP is evaluated on the remaining cuts at s_{nj}, placing the True value into it's label iff Φ is satisfied. Thus the claim.

QED

There are many other types of distributed predicate addition to the conjunctive form of predicate, which is possible to be detected with the modified labeling method, without any increase in the size of the label.

Consider the predicate $\Phi = x_1 = x_2 = \ldots = x_n = C$ where C refers to a constant. An example including 4 processes is illustrated in Figure 23 for the detection of the predicate $x_1 = x_2 = x_3 = x_4 = 9$. Consider s_{21} in process 2 in G_1, where $x_2 = 9$. There are two regions to be considered (see Section 3.3.1). Thus the size of the label is 2. The partial predicate to be evaluated in G_1 is $x_1 = x_2 = 9$. At event s_{21} there is one cut including events in region 1 and event s_{21}. Thus, there is one partial state including process 1 and process 2, such as (s_{11}, s_{21}). Therefore, the partial predicate is evaluated on the corresponding values of x_1 and x_2. $x_1 = x_2 = 9$, therefore the partial predicate is true. The second element of the label is conducted similarly. Note that the sign '[-]' in the label refers to the situation when there is no cut found to be considered to evaluate the predicate at that event.

Obviously, in each case, once $sat_k(s_{ij})$ is evaluated as false, then there is no need to continue keeping and using that in later iterations, leading to an actual reduction in complexity. An example of that is shown in the next section.

Predicates of the form $x_1 + x_2 + \ldots + x_n > C$ can also be easily detected using the modified labeling algorithm directly. Figure 24 illustrates this scenario. The

detection strategy is similar to as above, except the labels are slightly changed to become:

$$label_k(s_{(i+1)j}) = MAX_{s_{ij'} \| r_{k'} \cap G_i}\{label_{k'}(s_{ij'})\} \quad (4)$$

where $r_{k'}$ in G_{i-1} include events in r_k in G_i and $s_{(i+1)j}$.

Here each local state s_{ij} refers to the value of the corresponding x_i. The $label_k(s_{ij})$ contains the maximum partial sum of the predicate elements $(s_1+s_2+\ldots+s_i)$, involving cuts that include events in region k and s_{ij} in G_{i-1} whose local states in process 1 through i is $(s_1, s_2, \ldots, s_i)$. For example, in Figure 24 consider the first event at process 2. The value of the label contains 8: $MAX[(3+2),(3+0),(3+3),(3+5)] = 8$. Obviously the predicate is satisfied if there is an event found whose label contains a value that is greater than the given constant C (as in the first two events in process 4).

This technique could be extended to predicates that involve multiple terms instead of a single conjunctive term, or more complex than those which were considered here. It will be demonstrated further in the next section.

5.2.2 Direct application with reasonable complexity increase

In the preceding section we modified the labels of an event so that it contains only True/False values. But the size of each label is not increased when compared with that in computing the size of the global state space. In this section, we consider some other types of predicates, whose detection can still be done in polynomial time. However, the size of the label of an event may be increased compared with that in computing the size of the state space for a distributed execution.

Consider predicate $\Phi = DP_1\ OR\ DP_2 \vee \ldots\ OR\ DP_r$, where each DP_i is a conjunctive global predicate of the form $LP_1\ AND\ LP_2\ AND \ldots\ AND\ LP_n$. So, Φ contains r such conjunctive predicates.

Figure 25 illustrates an example of the detection of predicate $\Phi = DP_1\ OR\ DP_2$, where DP_1 and DP_2 are single conjunctive predicates shown in the figure. From the

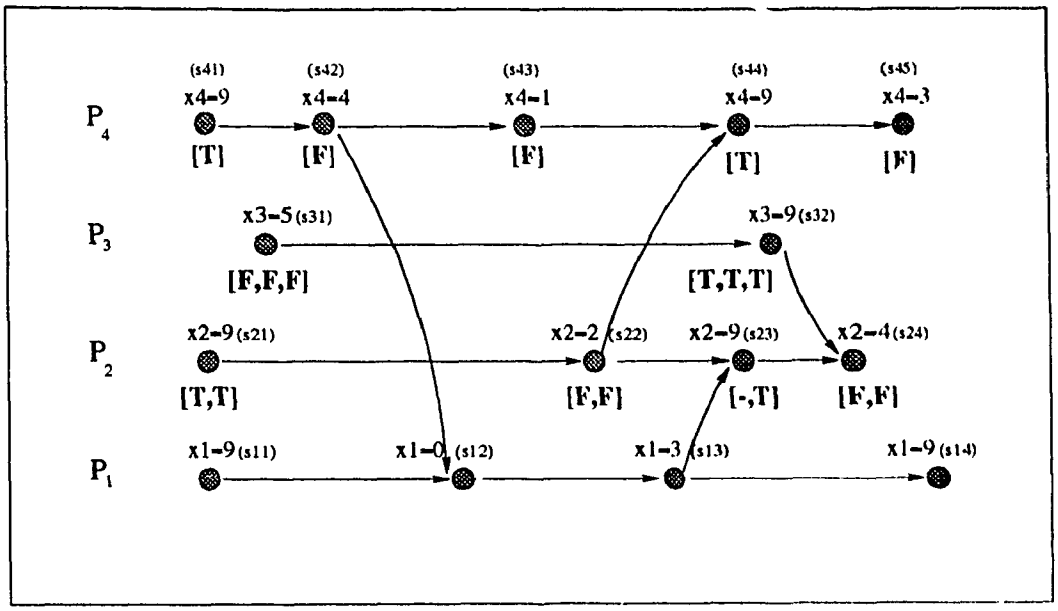


Figure 23: Detecting $x_1 = x_2 = x_3 = x_4 = 9$.

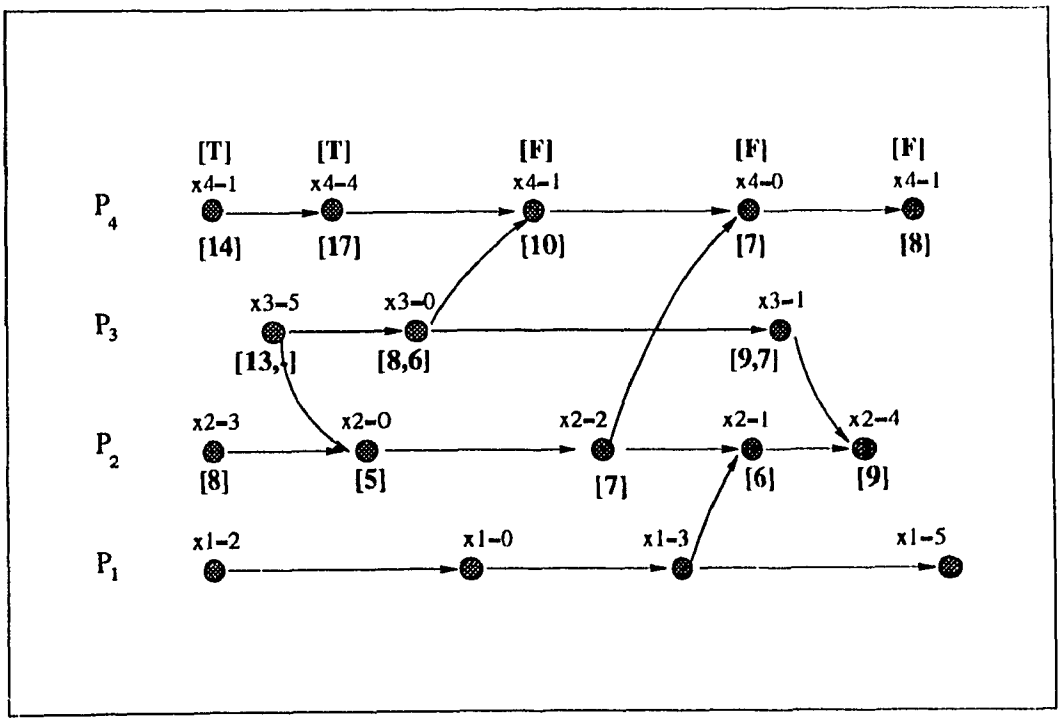


Figure 24: Detecting $x_1 + x_2 + x_3 + x_4 > 10$.

discussion in Section 5.2.1 and Lemma 2, the size of the label is equal to 2 for one of the single conjunctive term in Φ. However, we have to detect two such terms, therefore the size of the label is doubled as shown in the figure. Thus, we could state that the complexity for the detection of a disjunction of r conjunctive terms increases r times as the detection of a single conjunctive predicate discussed in Section 5.2.1.

As was mentioned in the previous section, we could further reduce complexity if we do not consider keeping the value of the label when $sat_k(s_{ij})$ is evaluated as false. As an illustration of this, in Figure 26 shown the detection for the same Φ as in Figure 25.

From the above discussion it is evident that the labeling method could be modified to handle the detection of more general predicates in some cases. But as we move into more complex types of predicates, the size of the label increases accordingly, even if the size of the state space is identifiable in polynomial time.

5.2.3 Indirect Application

The modified labeling algorithm for detecting distributed predicates works well in cases where the distributed predicates are of specific types, such as those exemplified in the preceding section. But for some other predicates, such as those used in the NP-completeness proof of Theorem 2, the theorem already implies the exponential cost of detection, i.e., the label of the events would grow exponentially independently of the consistent graph topology. However, for such predicates, we may still wish to tackle them when the state space is small. This is where the upper bound estimation algorithm could be used.

The upper bound estimation of the state size can be performed for a given execution. This is guaranteed to be computable in polynomial bounded time. If the state size has an actual number which is manageable within the practical constraint of the given system, for example, there are one million states and the system is capable of traversing a state in one-tenth of a millisecond, then checking an arbitrary predicate may be performed in 100 seconds and may be deemed acceptable in a practical debugging session. In that case despite the complexity introduced by the arbitrary predicate,

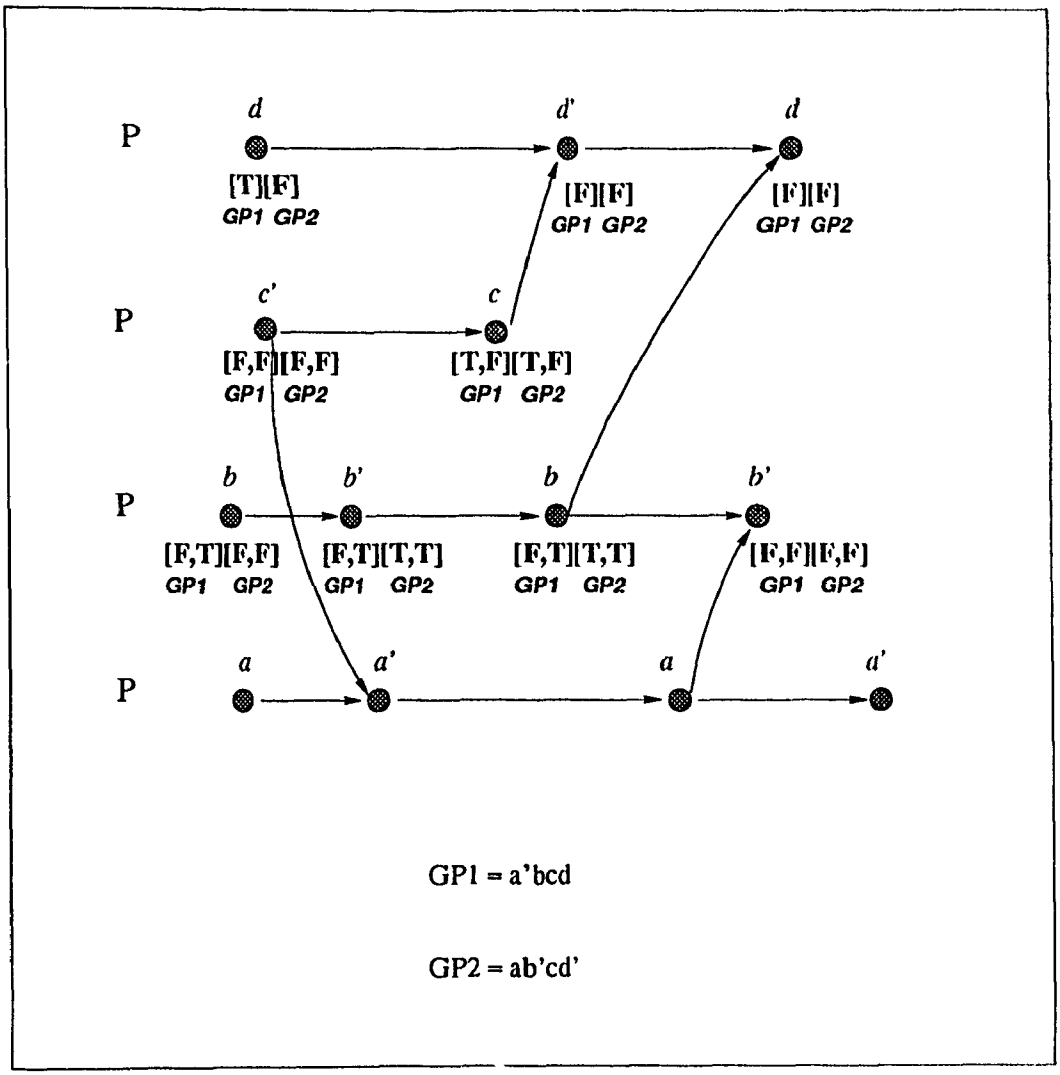


Figure 25: Detecting $\Phi = DP_1 \vee DP_2$.

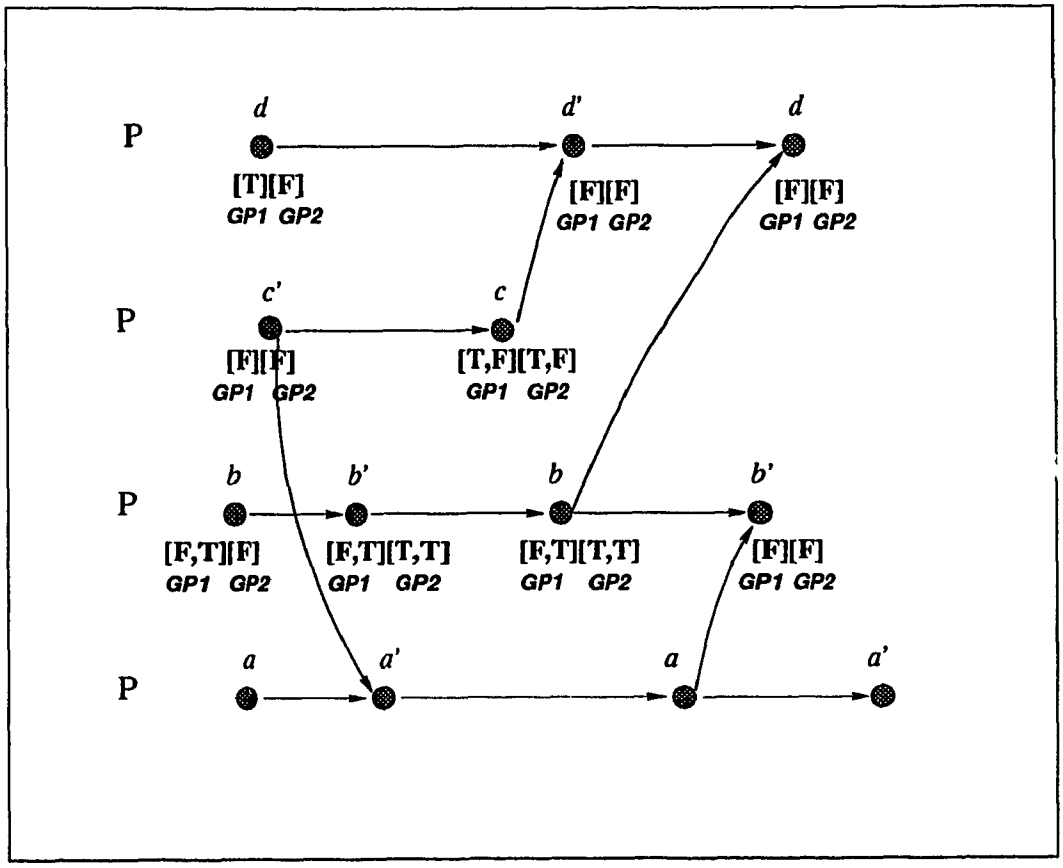


Figure 26: Detecting $\Phi = DP_1 \vee DP_2 = a'bcd \vee ab'cd'$.

the system can still provide the needed service to the user.

5.3 Summary

In this chapter we demonstrated how the labeling method presented in Section 3.3.1 can be modified to detect distributed predicates. This method could be used quite efficiently for relatively simple types of predicate detection such as simple conjunctive predicates, for predicates in the form $x_1 = x_2 = \ldots = x_n = C$, or for predicates of the form $x_1 + x_2 + \ldots + x_n > C$. As we consider more general (e.g. by adding more terms) predicates, the size of the label might grow accordingly. However, in some cases, such as for the detection of a predicate in the form of a disjunction of some conjunctive predicates, complexity still remains reasonable.

Nevertheless, there are cases where: either the identification of the state space are expensive, or the type of predicate to be detected involves very high increase in complexity; or both. In this case, using the upper bound estimation may help the user to decide if the detection is still reasonable with the state traversal technique.

Chapter 6

The design of the algorithms

In Chapter 3, theoretical results are presented for the calculation of the state space size in a Distributed Memory System. From those results, in this chapter we present descriptions of algorithms to compute of the exact number of states, its upper bound estimation as well as it lower bound estimation. The algorithms make use of vector clocks to keep track of consistency relationships among nodes of a given Consistency Diagram.

6.1 Virtual time, global clock

Given a distributed computation in the form of a space-time diagram, the concurrency among its events can be maintained by means of a couple of virtual time methods. In particular, Lamport's *logical clock* [Lam78] labels events in such a way that if an event precedes another event then the label of the former is smaller then that of the latter. The *vector clock* mechanism proposed in [Mat89] labels events more strictly and reflects the precedence relationship exactly.

In Lamport's *logical clock* method, each process p_i maintains a local variable LC that maps events to the positive natural numbers. The value of the logical clock when event e_i is executed by process p_i is denoted by $LC(e_i)$. The following update rules define how the logical clock is modified by p_i with the occurrence of each new event e_i:

$$LC(e_i) := \begin{cases} LC + 1; & \text{if } e_i \text{ is an internal or a send event of process i} \\ max\{LC; TS(m)\} + 1; & \text{if } e_i \text{ is a receive event receiving message } m. \end{cases}$$

where TS(m): Time Stamp of the message m indicates the logical clock of the corresponding send event.

In other words, when a receive event is executed, the logical clock is updated to be greater than both: the previous local value, and the timestamp of the incoming message. Otherwise, the logical clock is simply incremented. It is easy to verify that for any two events e, e', if e *causally effects*[1] e', then the logical clocks associated with them are such that LC(e) < LC(e'). However, as it is shown in [Mat89], logical clocks are not sufficient to determine if event e precedes event e' in a graph. Therefore we will use a Vector Clock mechanism in achieving this purpose.

6.1.1 Vector Clock

A simple mechanism for tracking causality of events is Vector Clock [Mat89]. Specifically, each process p_i has a clock VC_i which consists of a vector of length n, where n is the number of processes in the system. Each process p_i executes the following protocol:

1. At each local event e_i, the i^{th} component of clock $VC(e_i)[i]$ is incremented by 1.

2. When process p_i sends a message, it appends the timestamp (i.e., the current VC_i) to the end of the message.

3. If e_i is a receive event and T is the timestamp of the message, $\forall j$, $VC(e_i)[j] = max(VC(e_i)[j], T[j])$. In other words, each time process p_i receives a message, it updates its own timestamp with the maximum of the value received and the value it previously held for each of the components of VC_i.

[1] Event e may *causally effect* another event e' if and only if $e \rightarrow e'$ (i.e., there is a directed path in the Space-time diagram starting at e and ending at e').

Thus, one could decide if event e_i causally effects event e_j as follows:

$$VC(e_i)[i] \leq VC(e_j)[i]. \tag{5}$$

Also, two events e_i and e_j are said to be concurrent if the following vector clock condition holds:

$$VC(e_i)[i] > VC(e_j)[i] \wedge VC(e_j)[j] > VC(e_i)[j]. \tag{6}$$

Thus, using vector clocks provides enough information to decide if two events are causally effecting each other or they are concurrent as it is proven in [Mat89].

6.1.2 Using Vector Clock in our Model

In our algorithms, we need to decide if two local states e_i and e_j are consistent or not. In other words, we want to conclude if either of the three condition: $e_i \rightarrow^* e_j$, or $e_j \rightarrow^* e_i$, or $\neg[(e_i \rightarrow^* e_j$, or $e_j) \wedge (e_j \rightarrow^* e_i)]$ holds.

According to Mattern the following property is true [Mat89]:

Property of Vector Clock:
Given a space-time diagram, e_i happens before e_j iff $VC(e_i)[i] \leq VC(e_j)[i]$.

In terms of the corresponding Consistency Diagram, if we use s_i to denote the corresponding local state upon the occurrence of e_i and label $VC(s_i) = VC(e_i)$, then from definition s_i is consistent with s_j ($s_i \parallel s_j$) iff e_i does not happen before another event in process i which in turn happens before e_j in process j and vice versa. This leads to the following lemma:

Lemma 8 *In a Consistency Diagram, $s_i \parallel s_j$ iff*

$$VC(s_i)[i] > VC(s_j)[i] \text{ and } VC(s_j)[j] > VC(s_i)[j].$$

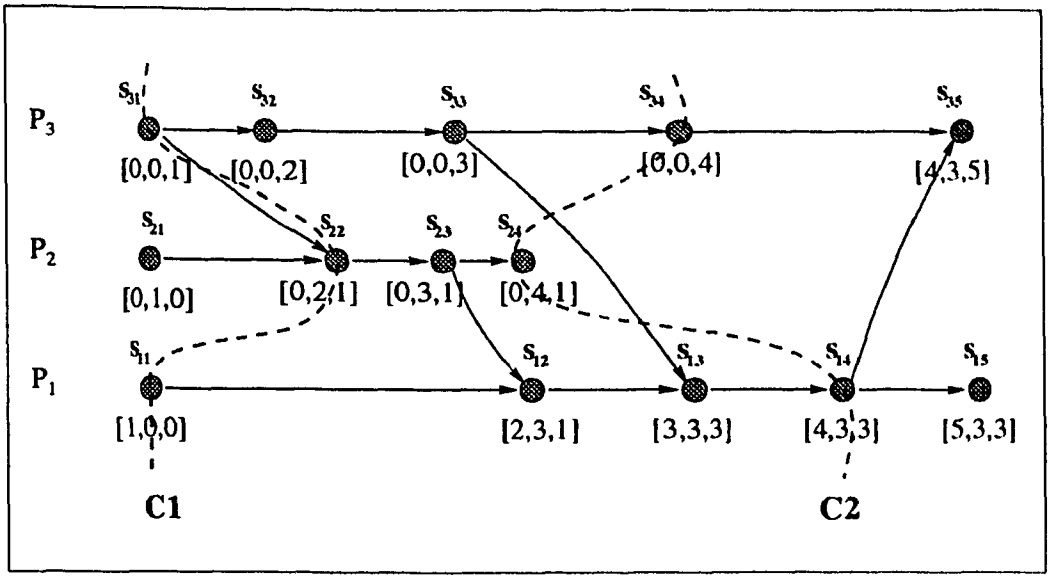


Figure 27: Vector clock values in a Consistency Diagram.

Proof: This follows immediately from the vector clock property and the definition of $s_i \parallel s_j$.

QED

In Figure 27 the vector clock values are illustrated in a Consistency Diagram. While cut C_2 is a consistent cut according to Lemma 8, cut C_1 is not. The clock value at state s_{31} = [0,0,1], the clock value at state s_{22} = [0,2,1], which implies that $VC(s_{31}[3] \leq VC(s_{22}[3]$. Therefore, s_{31} and s_{22} can not be involved in the same concurrent cut.

6.2 The algorithms

In all the algorithms presented in the following sections consistency relationship between events is identified by using the vector clock mechanism explained in the previous section.

6.2.1 The algorithm for global state counting

Let the set $R^i = \{r_1^i, r_2^i, \ldots r_m^i\}$ denote the regions in G_i, and there is an array size 1...m associated with each event $e_{i+1,j}$ which holds the value of the label. Then the algorithm for computing the number of global states is constructed as shown in Table 1 and Table 2.

The explanation of the terms used in the algorithms are as follows:

typeA(edge) = edges linking local states of each process only;

typeB(edge) = edge that either starts from or ends at process i, and starts from or ends at process j > i.

$\mathbf{P^j}$ is a set of partitions in process j. Initially, $\forall j(3 \leq j \leq n)$: there is only one partition which contains all events at process j.

$\mathbf{R^i}$ is a set of regions, where each region r_k^i contains a nonempty *partition* from each process i+2 through n, and $\forall k, h$: if $r_k^i \in R^i$ and $r_h^i \in R^i$ then $r_k^i \neq r_h^i$.

FindIndex(k, s_{i+1}) is a function which returns the integer q for which the following condition hold: $r_k^i \subseteq r_q^{i-1}$ AND $s_{i+1} \in r_q^{i-1}$.

The algorithm works as follows: In the first part (i.e., steps (1) through (5)), of ExactStates, G_{i-1} is augmented with edges – those which start at or end at process i, and start at or end at process j>i; – to form G_i. Every time G_{i-1} is augmented with a particular edge, the necessary changes are made to the partitions at process j. These changes occur according to the algorithm Partition shown in Table 2. When G_{i-1} is augmented with a forward edge for instance, the partition which contains the event to where the edge points to is divided into two. An example is shown in Figure 29, where G_{i-1} (in Figure 28) is augmented with the edge (s_{ik}, s_{jh}). Also, $partition_x^j = \{s_{j,h-1}, s_{jh}\}$ in G_{i-1}. Thus, following steps (5) through (8) in algorithm Partition, $partitionB = \{s_{jh}\}$ according to step (5), and $partitionA = \{s_{j,h-1}\}$ according to step (6). Then $partition_x^j$ is 'replaced' in $\mathbf{P^j}$ by $partitionA$ and $partitionB$. When G_{i-1} is augmented with typeB edges, then each region in $\mathbf{R_i}$ is identified according to its definition. In the second part of the algorithm ExactStates (i.e., steps

```
NumOfStates := 0;
G = {N, A^0}; where A^0 includes typeA(edge) only;
label(s_1)[1] := 1;

begin
   for i=1 to n-1
     begin
(1)        for each typeB(edge)
             begin
(2)            A' := A^{i-1} ∪ {edge};
(3)            if p-distance of the edge > 1 then
(4)                 Partition(edge); /* described in Table 2 */
             end;

(5)        Identify R^i;

(6)        for each s_{i+1}
(7)          for each r_k^i
(8)            for each s_i
(9)              if (s_i||s_{i+1}) AND (s_i||r_k^i) then
                   begin
(10)                   q := FindIndex(k, s_{i+1});
(11)                   label(s_{i+1})[k] := label(s_{i+1})[k] + label(s_i)[q];
                   end;
     end;

(12)  for each s_n
(13)    NumOfStates := NumOfStates + label(s_n)[1];

   Return NumOfStates;
end;
```

Table 1: Algorithm ExactStates.

```
partitionA = partitionB = {∅};
∃partition_x^j ∈ P^j : s_j ∈ partition_x^j;

begin

repeat
(1)     if typeB(edge) is (s_j, s_i) /* it is a reverse edge */
            for each s_j' ∈ partition_x^j
(2)           if s_j' →* s_j OR s_j' = s_j then
                  s_j' ∈ partitionA;
(3)           else
                  s_j' ∈ partitionB;

(4)     if typeB(edge) is (s_i, s_j) /* it is a forward edge */
            for each s_j' ∈ partition_x^j
(5)           if s_i →* s_j' then
                  s_j' ∈ partitionB;
(6)           else
                  s_j' ∈ partitionA;

(7)     if partitionA ≠ {∅} AND partitionB ≠ {∅}
(8)       P^j = P^j − {partition_x^j} ∪ {partitionA, partitionB};
until there is no more typeB edge
end;
```

Table 2: Algorithm Partition.

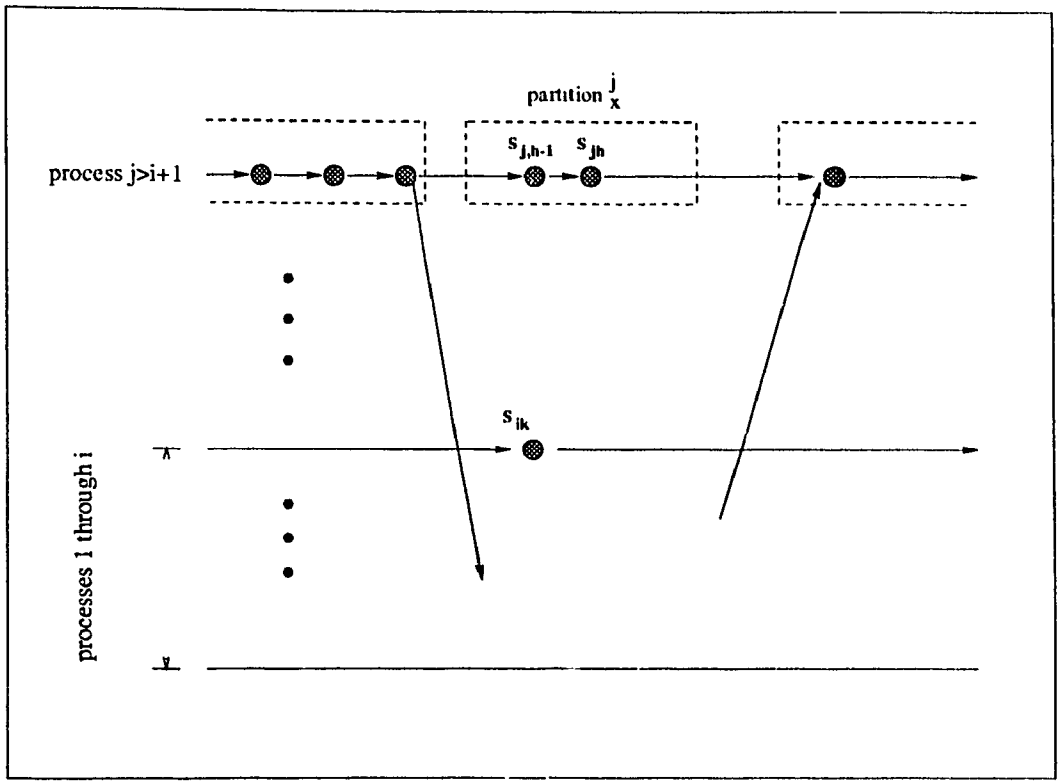


Figure 28: G_{i-1}.

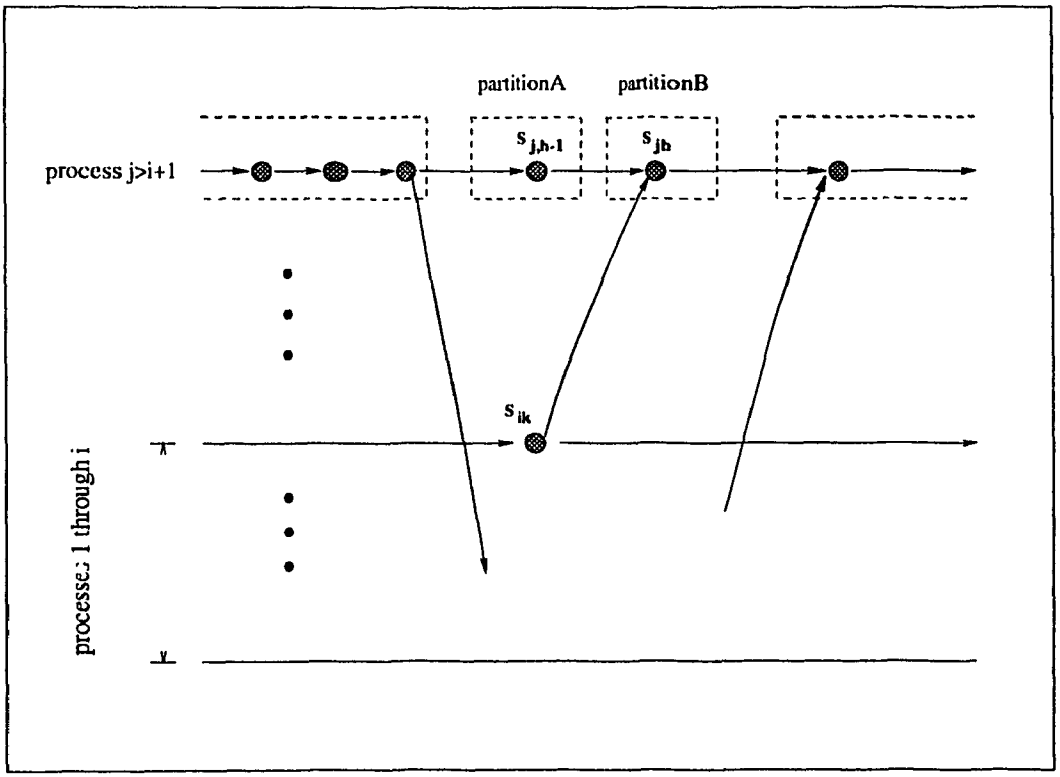


Figure 29: Augmentation of G_{i-1} with edge (s_{ik}, s_{jh}).

(6) through (11)), each item in the array referring to the label at each event s_{i+1} is computed by summing the corresponding items at each event in process s_i according to Theorem 1 in Section 3.3.1. Finally, — at step (12) — the total number of states are computed by summing the terms in the labels at each event in process n.

As an example, in Figure 30 the results are shown (after iteration 1) for a given Consistency Diagram. There is one partition at process 4 and 5, and there are two at process 3 in G_1. Suppose there are two edges (s_{23}, s_{52}) and (s_{42}, s_{25}) to be included at the second iteration to form G_2. According to algorithm Partition, the partition $\{s_{51}, s_{52}, s_{53}\}$ at process 5 is replaced by two partitions: $partition_1^5 = \{s_{51}\}$ and $partition_2^5 = \{s_{52}, s_{53}\}$ [since edge (s_{23}, s_{52})]. Similarly, the partition $\{s_{41}, s_{42}, s_{43}\}$ at process 4 is replaced by $partition_1^4 = \{s_{41}, s_{42}\}$ and $partition_2^4 = \{s_{43}\}$ [since edge (s_{42}, s_{25})]. Now the 4 possible regions formed in G_2 are : $r_1 = \{s_{51}\}, \{s_{41}, s_{42}\}$, $r_2 = \{s_{51}\}, \{s_{43}\}, r_3 = \{s_{52}, s_{53}\}, \{s_{41}, s_{42}\}$, and $r_4 = \{s_{52}, s_{53}\}, \{s_{43}\}$.
At the second part of algorithm ExactStates (i.e., step 6 through step 11) the label is computed for each event in process 3. Consider the first element in the label at event s_{31}. Since s_{31} belongs to r_1 in G_1, function $FindIndex(1, s_{31})$ returns 1. It means, that the algorithm sums the first elements in the label at each event s_2 which is concurrent with event s_{31} and region 1 in G_2. Since there are 4 such events [namely $s_{21}, s_{22}, s_{23}, s_{24}$], the first element in the label at event s_{31} is $2 + 1 + 1 + 1 = 5$. This process is repeated until all the elements in the label are computed at all events s_3. Then the algorithm ExactStates continues with the next iteration until the labels are computed (and summed at step 13) at each event in process n. Finally, algorithm ExactStates returns the total number of concurrent cuts for the given Consistency Diagram. Next, we proceed with the proof of correctness for the algorithm.

Lemma 9 *The algorithm ExactStates is correct.*

Proof: The steps (6) through (11) in the algorithm ExactStates conform to Equation 1, which is proven to be correct in Section 3.3.1. Therefore, we have only to show that the construction of the regions in G_i is correct: steps (1) through (5) in algorithm ExactStates identifies partitions correctly. Particularly, we have to show that each $s_{jh} \in partition_x^j$ ($partition_x^j \in r_k^i$) has the same consistency relationship with respect to any event in G_i. We provide a proof by induction.

Base: Initially, given $G_0 = \{N, A^0\}$, there is one partition at each process. Since A^0 contains edges linking local states of each process only, it is clear that $\forall i, j$, where $i \neq j$: $s_i \parallel s_j$. Which means that all events in $partition_x^j$ are consistent with all events located in any other processes. Thus the statement is true.

Now assume that it is true for G_{i-1}. In other words, $\forall s_{kx}$, where $j \neq k$; and for each $s_{jh} \in partition_x^j$ one of the following is true:

1. if $s_{jh} \rightarrow^* s_{kz}$ then $\forall y$, where $s_{jy} \in partition_x^j \Rightarrow s_{jy} \rightarrow^* s_{kz}$.
2. if $s_{kz} \rightarrow^* s_{jh}$ then $\forall y$, where $s_{jy} \in partition_x^j \Rightarrow s_{kx} \rightarrow^* s_{jy}$.
3. if $(s_{jh} \parallel s_{kz})$ then $\forall y$, where $s_{jy} \in partition_x^j \Rightarrow s_{jy} \parallel s_{kx}$.

As a next step, we will show that these conditions are true for each partition in G_i also. Consistency between events in G_i may change by augmenting G_{i-1} with typeB edges. Suppose there is a typeB edge (s_{ig}, s_{jh}) introduced into G_{i-1}, and $s_{jh} \in partition_x^j$ in G_{i-1}. Then according to steps (5) and (6) in algorithm Partition, $partition_x^j$ is divided into two disjoint subsets: $partitionA$ and $partitionB$. Since (s_{ig}, s_{jh}) is a forward edge, it is clear that $partitionB$ always has at least one event in it: s_{jh}. Moreover, in case there are more events in $partitionB$, s_{jh} is the earliest. Now assume that there is an s_{kx} for which one of the three conditions above holds in G_{i-1}. If condition (1) or condition (2) holds, then the same condition holds for all the elements in $partitionA$ and $partitionB$ because of transitivity. Now suppose condition (3) holds in G_{i-1} when forward edge (s_{ig}, s_{jh}) is introduced in G_i, and $(s_{kz} \rightarrow^* s_{jh})$ is true in G_i. As $s_{kx} \rightarrow^* s_{jh}$, and s_{jh} is the earliest event in $partitionB$, so $\forall y$, where $s_{jy} \in partition_x^j$: $s_{kz} \rightarrow^* s_{jy}$. Hence, the statement is true for $partitionB$. If $partitionA$ is nonempty, then clearly, no inconsistency is introduced in G_i involving events in $partitionA$. So for each each s_{kx} condition (3) holds the same way as in G_{i-1}. Now, suppose $partitionA = \{\emptyset\}$. Then $partition_x^j = partitionB$, so the statement is also true. With similar reasoning, it is easy to show that the statement is true when G_{i-1} is augmented with a reverse edge. Thus the claim.

QED

The time complexity of the algorithm is determined by the number of comparisons made between local states in the second part of the algorithm (i.e., steps (6) through (11)). Assume there are q events at each process. Each event s_{i+1} is compared with

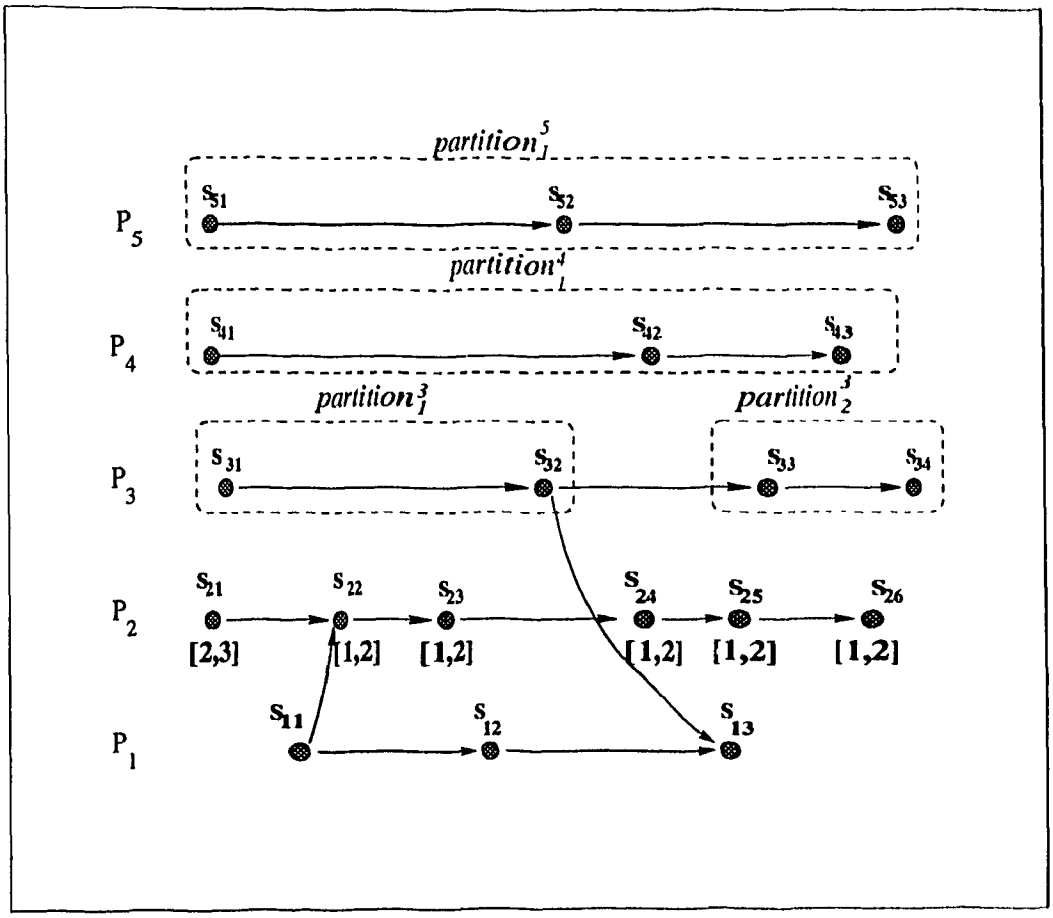


Figure 30: G_1 (after iteration 1).

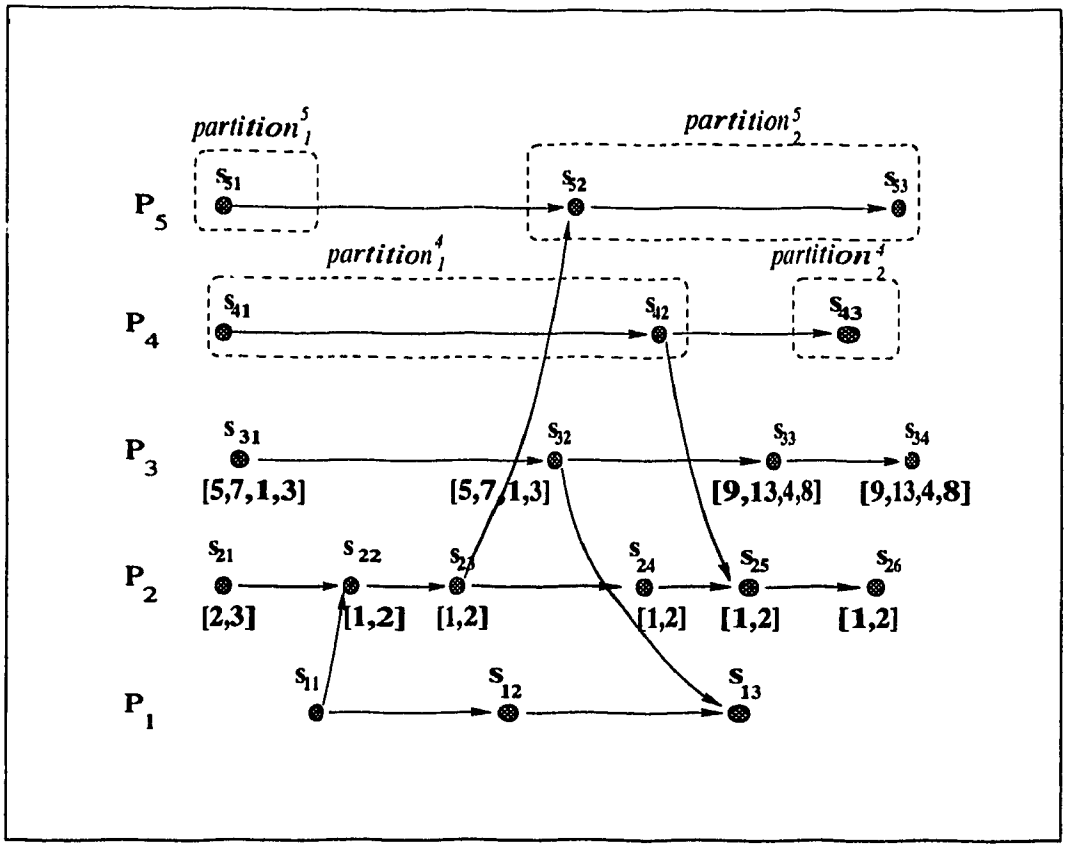


Figure 31: G_2 (after iteration 2).

each event s_i, and each s_i compared with the events in a region r_k^i. Since the events have the same consistency relationship in each partition in the region r_k^i, it is sufficient to compare s_i with only one event from each partition. In the worst case there are $n-2$ such events, therefore it gives us $(n-2)q$ comparisons for identifying one element in the label associated with event s_{i+1}. Since there are m regions, the computation of the label at each s_{i+1} takes $q^2(n-2)m$ comparison. In the worst case the outer loop brings the time complexity to $O(q^2n^2m)$.

The algorithm could be made more efficient with the following optimization strategies. As the algorithm migrates from process i-1 to process i, the label at each event below process i-1 could be discarded since there is no further use for these values later. Furthermore, events belonging to the same process with same consistency relationship could share the same label. For example, in Figure 6 in Chapter 3, events s_{22} through s_{25} may share label [1,0,4,1], which may reduce the space requirements of the algorithm greatly.

6.2.2 The algorithm for upper bound estimation

The number of global states in an *arbitrary* Consistency Diagram can be computed using algorithm ExactStates presented in the previous section. So, the same algorithm can be used in estimating the upper bound on the number of global states also. However, with some modifications made to the ExactStates algorithm there is a more efficient way to get an upper bound estimation, as we will show in this section. We will derive an upper bound estimation on a given G, where the p-distance of any edge is at most 2.

Suppose that the partitions at the same process are numbered $1,\ldots,m$, and $p\_num(s_{ij})$ denotes the partition number in which event s_{ij} belongs. Then we leave lines (1) through (4) in algorithm ExactStates as it is, except the **typeB(edge)** is redefined as follows [i.e., to make sure that there are no edges included in G_i with p-distance>2]:

typeB(edge) = edge that either starts from or ends at process i, and starts from or ends at process i+1 or process i+2.

Moreover, line (5) through (11) in algorithm ExactStates are replaced with the following.

```
begin
   for each s_{i+1}
      for each partition_k at process i+2
         for each s_i
            if (s_i||s_{i+1}) AND (s_i||partition_k) then
                  label(s_{i+1})[k] := label(s_{i+1})[k] + label(s_i)[p_num(s_{i+1})];
end;
```

In other words, we modify the algorithm ExactStates by comparing the consistency between each event s_i with each partition located at process i+2 in G_i, instead of comparing s_i with all regions in G_i. An example is shown in Figure 32. The three regions, to be compared with s_i in the modified ExactStates algorithm are listed below.

$$\mathbf{r_1} = \{s_{n1}, s_{n2}\}, \ldots, \{s_{j1}, s_{j2}\}, \ldots, \{s_{i+2,1}\}$$

$$\mathbf{r_2} = \{s_{n1}, s_{n2}\}, \ldots, \{s_{j1}, s_{j2}\}, \ldots \{s_{i+2,2}\}$$

$$\mathbf{r_3} = \{s_{n1}, s_{n2}\}, \ldots, \{s_{j1}, s_{j2}\}, \ldots, \{s_{i+2,3}\}$$

It means, s_i would be compared with at least one element in $partition_1^n$, ... , and one element in $partition_1^j$, ... , and one element in $partition_1^{i+2}$. However, observe that any event s_i is always consistent with any event s_j [j > i+2], because of the construction of G_i. So, in the modified algorithm here, we only need to compare each s_i with one element in $partition_1^{i+2}$ in order to compute the first element in the label associated with any event s_{i+2}. Based on these observations, the following lemma is

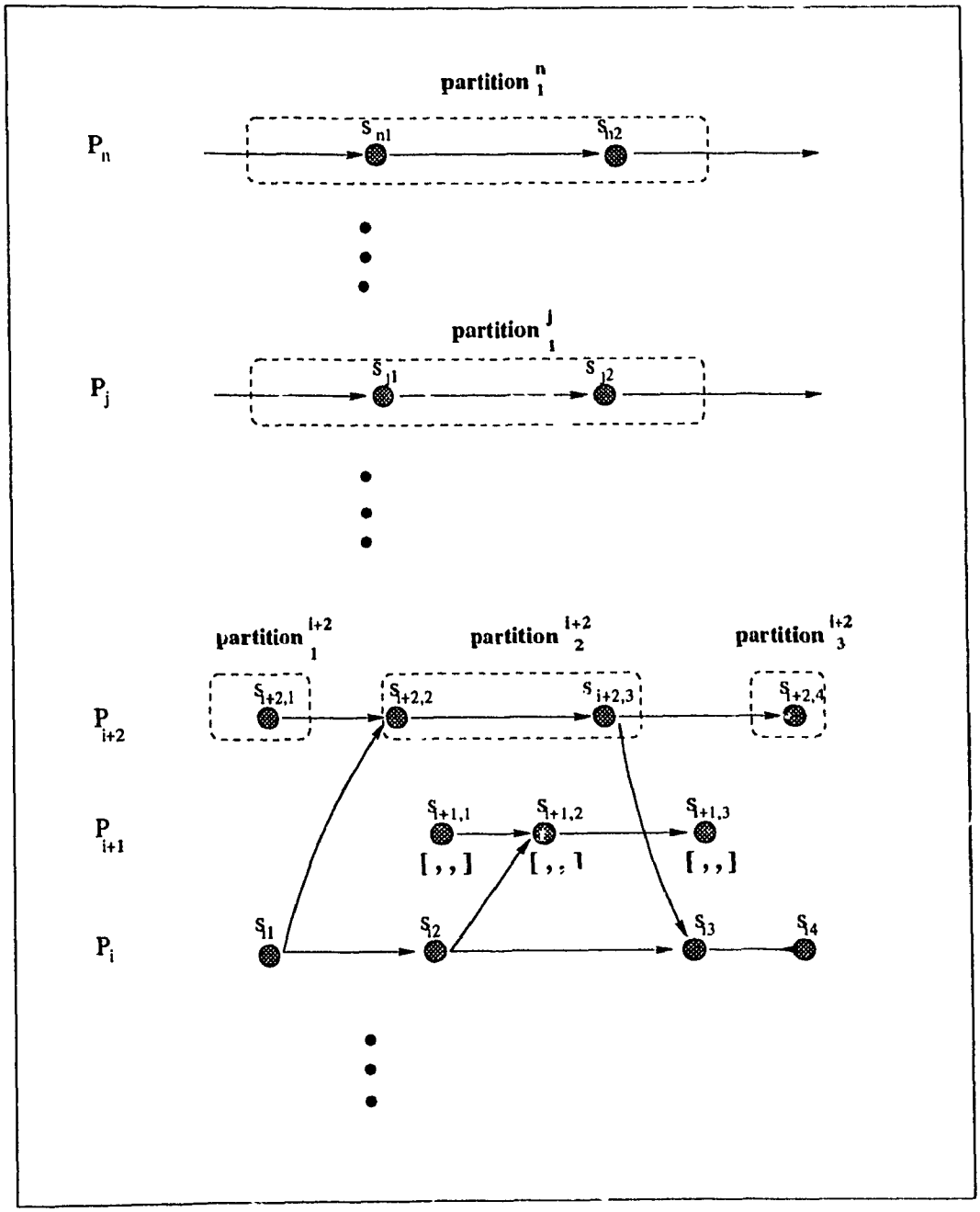


Figure 32: Partitions in G_t.

provable.

Lemma 10 *The modified algorithm is correct.*

Proof: We only have to show that partitions at process i+2 reflect the same consistency relationship for regions in G_i when each edge has p-distance of no more than 2. In other words, we have to show that if $\exists s_i$ such that $s_i \parallel partition_k^{i+2}$, then $s_i \parallel r_k^i$ in G_i where $partition_k^{i+2} \in r_k^i$. Suppose there is an r_k^i, which includes $partition_k^{i+2}$, and one partition from each process j [where j>i+2]. Also, $\exists s_i$ such that $s_i \parallel partition_k^{i+2}$ but $\neg(s_i \parallel r_k^i)$. It means that $\exists s_j \in partition_1^j$ such that either $s_i \rightarrow s_j$ or $s_j \rightarrow s_i$. However, this is impossible since events at processes i+3 through n are not 'connected' to events located at other processes. So, it is a contradiction. Thus the claim.

QED

In the modified algorithm here, we compare consistency between events and a partition located at process i+2, instead of comparing events with all the partitions in the corresponding region. Thus the time complexity of the algorithm is reduced to $O(q^2nm)$. When compared with the ExactStates algorithm, the partitions at a process here do not change as the algorithm migrates from level i to level i+1. Thus, according to Lemma 2, $m \leq q$ [i.e., since there are no edges with p-distance>2]. Hence the algorithm has a time complexity $O(q^3n)$.

6.2.3 The algorithm for lower bound estimation

Based on the discussion in Section 3.3.3, it is straightforward to design a simple algorithm to augment a given G to become G' such that every edge has a p-distance of at most 2. Table 3 demonstrates the abstract algorithm for the augmentation.

The following notations are used in the algorithm Augmentation:

P = prefix processed so far, initialized to NIL.

R(P) = set of events which are 'ready' given **P**. In particular, s in **R(P)** and $s' \rightarrow^* s$ implies that $s' \in$ **P**.

```
P := {};
s := some event in R(P);
Repeat
   if an edge emanates from s with p-distance > 2 then
     begin
       apply Lemma 5 and Lemma 6
       to replace that edge with a sequence of edges whose
       p-distances are not more than 2;
       P := P ∪{s};
     end;
   else
     begin
       if s is a typeB event then
         P := P ∪{s};
       else    /*it is a local event*/
         P := P ∪{s};
     end;
   R(P) := R(P) −{s};
   s := some event in R(P);
until R(P) is empty;
```

Table 3: Algorithm Augmentation.

typeA(s) is a local event from where an edge emanates to the next event in the same process.

typeB(s) is a event from where an edge emanates to another event located in another process with p-distance not more than 2.

The algorithm Augmentation scans through the given Consistency diagram visiting each event once. The priority between events is encoded in the `if-then-else` statements. In other words, an event from where an edge emanates with p-distance > 2 is considered first, followed by the typeB events. Finally typeA events are considered in prefix **P**. This permits more local events remaining in **R(P)** which could be chosen

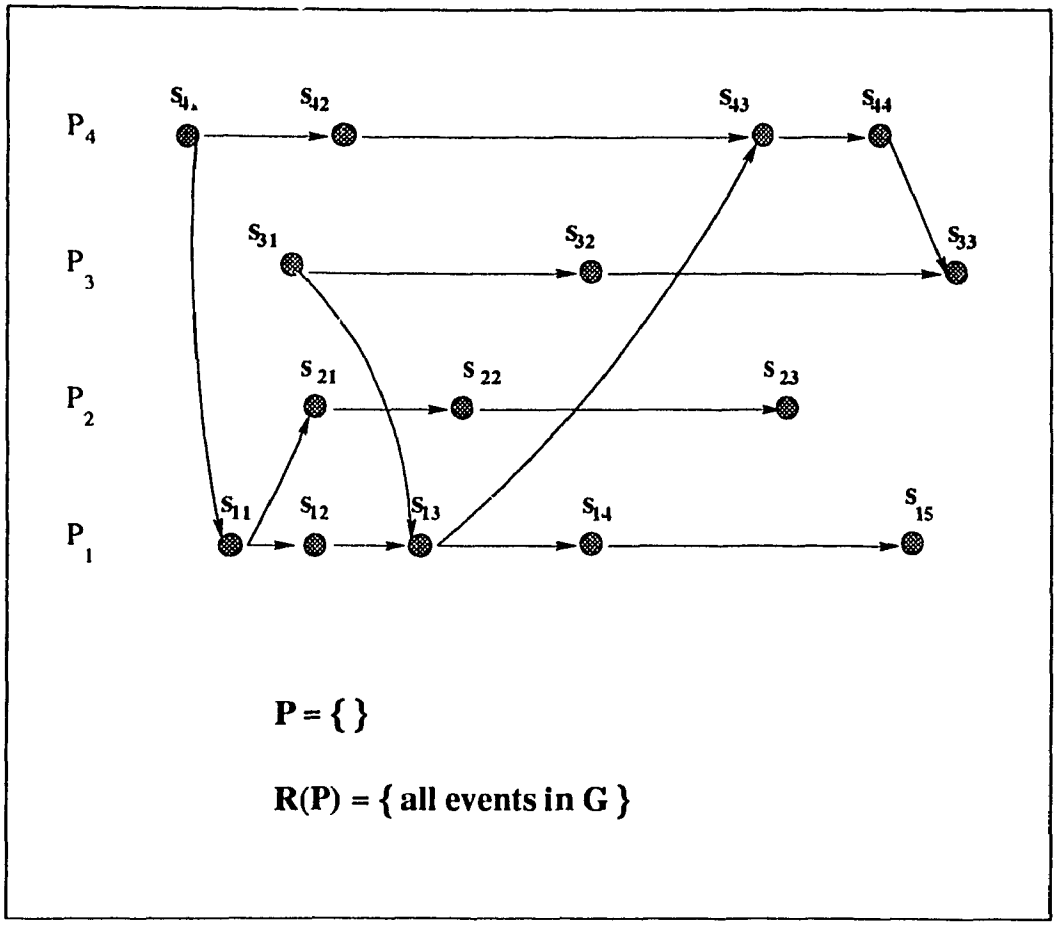

P4
P3
P2
P1
S41
S42
S43
S44
S31
S32
S33
S21
S22
S23
S11
S12
S13
S14
S15
P = { }
R(P) = { all events in G }

Figure 33: G.

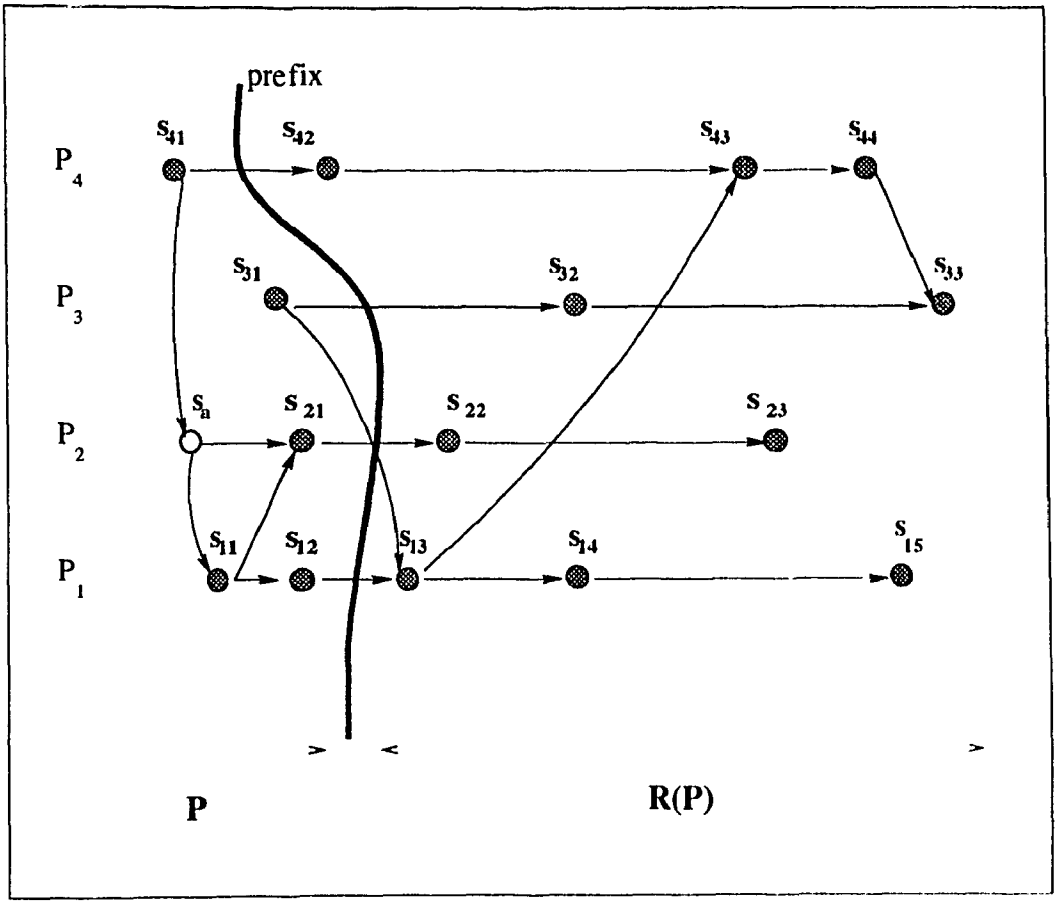


Figure 34: Advancing prefix **P** after replacing edge (s_{41}, s_{11}).

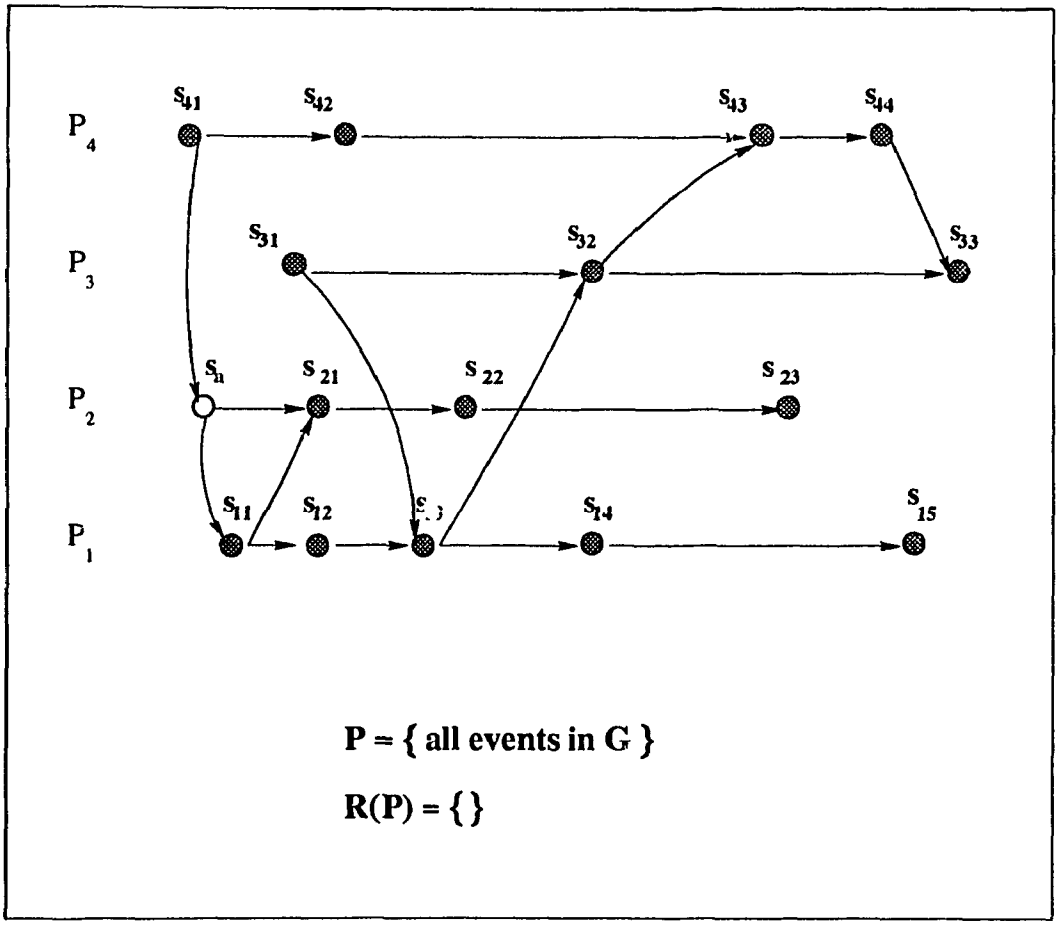

P4
P3
P2
P1
S41
S42
S43
S44
S31
S32
S33
S21
S22
S23
S11
S12
S14
S15
P = { all events in G }
R(P) = { }

Figure 35: G'.

later in the augmentation procedure, and heuristically reduces the number of additional events.

Figure 33 through Figure 35 illustrate the steps in the algorithm on a simple example. In Figure 33 the original G is shown. Since there is an edge that emanates from (s_{41}) and its p-distance is more than 2, it is replaced with a sequence of edges (s_{41}, s_a) and (s_a, s_{11}). Event s_a is an additional event, and it is denoted by an empty circle in the figure. Then the prefix **P** is advanced as shown in Figure 34. The next event from which an edge is initiated with a p-distance more than 2 is s_{13}. It is replaced with the chain of edges (s_{13}, s_{32}) and (s_{32}, s_{43}) as shown in Figure 35. Since the remaining events in **R(P)** are either type(A) or type(B), the prefix **P** is advanced until it contains all events in G.

The correctness of the algorithm follows immediately from Lemma 5 and Lemma 6. Using this augmented graph, we could proceed to obtain a lower bound on the state size of the original G involving a polynomially bounded number of computational steps.

6.2.4 The algorithm for the detection of a simple predicate

In Section 5.2.1 we discussed detection method for simple distributed predicates by modifying the labeling procedure. Here we show how the ExactStates algorithm (in Table 1 explained in Section 6.2.1 can be used with slight modification for the detection of a conjunctive distributed predicate. The only modification made to algorithm ExactStates is as follows:

1. line (11) in algorithm ExactStates is replaced with the following:

```
if the state of s_{i+1} is True AND label(s_i)[q] = True then
   label(s_{i+1})[k] := True;
else
   label(s_{i+1})[k] := False;
```

2. line (12) and (13) is replaced with:

```
if ∃s_n such that label(s_n)[1] = True then
   Return True;
else
   Return False;
```

The elements in the label in this modified algorithm contain True or False values according to the evaluation of the partial predicate. The algorithm returns True if there is at least one global state where the predicate turns true. If there is no such state, then it returns False. The complexity of the algorithm is increased slightly, as the evaluation of each local predicate is required with the comparison of these values including event s_{i+1} and s_i.

6.3 Summary

A general algorithm ExactStates is designed for the computation of the state space for a given *arbitrary* Consistency Diagram. It is quite good and can be used to derive an upper bound estimation as well. However, for the upper bound estimation we propose a more efficient way, by making some modification to algorithm ExactStates. This way, the cost of computing can be decreased n times when computed with the original ExactStates algorithm. Obviously, the complexity reduction is substantial in case of a large number of processes are included in the execution. In case of a

lower bound estimation, it is shown that based on the results in Section 3.3.3, with a simple algorithm the given G can be transfered to G', where edges with p-distance > 2 are replaced with a chain of edges. Then, the computation of the lower bound of the state space can use algorithm ExactStates or the modified algorithm described in Section 6.2.2. For the detection of a conjunctive predicate another simple alteration of ExactStates is necessary. Here, the elements in the label associated with an event reflect the resul' of the evaluated partial predicate.

In any of the algorithm discussed here, when it comes to decide if two events are consistent or not, we use the well-known Vector Clock mechanism.

Chapter 7

Experimental results

Examples are carefully chosen to demonstrate experimental results for computing the state size in different types of systems. The following applications have been chosen:

1. Calculation of the value of Π: A single co-ordinating process performs certain functions on behalf of the others - the synchronization mimics the centralized 'client/server' model;

2. Matrix factoring: Processes communicate through broadcast;

3. Hirschberg and Sinclair's elective algorithm: A token ring forms the underlying synchronization structure.

For each example the performances of the various labeling algorithms are obtained with varying system parameters. These parameters include # of events, # of edges and the size of the Consistency diagram. In these experiments the *estimation accuracy* and the *speedup of estimation* of the lower and upper bound approximation algorithms are also obtained. In each case the maximum size of the label is noted.

The 'estimation accuracy' is defined as follows:

$$estimation\ \ accuracy = MAX(\frac{C}{C^*}, \frac{C^*}{C}) \tag{7}$$

where C refers to the state size computed by the exact algorithm; and C^* refers to that by an estimation algorithm. For lower bound estimation, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the optimal solution is larger than the approximate solution. Similarly, for upper bound estimation, $0 < C \leq C^*$, and the ratio

C^*/C gives the factor by which the approximate soluti is larger than the optimal solution for the given input size. Hence, the 'approxi ... accuracy' ≥ 1 in each case. Since the optimal (i.e. exact bound) algorithm ratio = 1, the closer the 'estimation accuracy' to 1, the better the approximation.

The 'speedup of estimation' is a ratio between the execution time of the lower (or upper) bound algorithm and that of exact algorithm.

We define accuracy-speedup ratio $\wp$ as follows:

$$\wp = estimation\ accuracy \times speedup.$$

Obviously the larger $\wp$ is, the better is the estimation algorithm.

7.1 The calculation of the value of Π

A simple numerical integration formula for generating the value of Π is [Rag91] :

$$\Pi = \int_0^1 \frac{4}{1+x^2} dx \tag{8}$$

An approach to evaluate Equation 8 is to divide the area under the curve into a number of evenly-spaced strips. Each strip is approximated as a rectangle. The value of the function at the midpoint of the strip is taken as the height of the rectangle. Pi, therefore, is calculated as the sum of the strips in the area under the curve.

The calculation of the areas of the rectangles is distributed among the processes. Each node evaluates the function for an assigned set of strips and sums the values for those strips. This leaves certain tasks to the co-ordinator process: collecting and summing the partial sums calculated by each node.

Table 4, Table 5 and Table 6 shows the experimental results for the computation of the value of π for the exact algorithm, lower and upper bound estimation algorithms respectively. In Experiment 3., the estimation accuracy returns an unacceptably high value (i.e. it should be close to 1) for the lower bound estimation algorithm, even if the speedup of estimation is very good. However, the upper bound algorithm behaves

Experiment	#events	#edges	size of label	exact number of cuts
1.	42	8	27	10,406
2.	22	8	27	82
3.	88	18	6561	215,219,239
4.	47	18	6561	19,684

Table 4: Exact number of cuts for calculating Π.

Experiment	#events	#edges	size of label	lower bound	estimation accuracy	speedup of estimation
1.	42	8	5	311	49.5	4 times
2.	22	8	5	27	10	4 times
3.	88	18	12	605	631039	13,566.4 times
4.	47	18	12	72	273.4	95.2 times

Table 5: Lower bound estimation for calculating Π.

extremely well. This is especially so in Experiment 3., where $\wp \approx 165,600$ which is $\gg 1$, and it is still quite close to the value of the speedup of estimation.

7.2 Matrix factoring

One of the well known algorithms is the Gaussian elimination algorithm for factoring a square matrix into lower and upper triangular factors [Rag91].

Experiment	#events	#edges	size of label	upper bound	estimation accuracy	speedup of estimation
1.	42	8	3	15,386	1.479	8 times
2.	22	8	3	270	3.293	4 times
3.	88	18	3	504,831,859	2.346	72,354.3 times
4.	47	18	3	174,960	8.888	1,903 times

Table 6: Upper bound estimation for calculating Π.

Experiment	#events	#edges	size of label	exact number of cuts
1.	180	28	36	1,256,243
2.	472	48	150	36,970,233

Table 7: Exact number of cuts for Matrix factoring.

The given matrix is partitioned into columns which are assigned to the processes in round-robin manner. The calculation starts with a process which owns the first column. It finds the pivot element, swaps the pivot row with the first row, and divides the pivot column by the pivot. Afterwards, the process broadcasts the pivot column and the pivot row number as a message to all the other processes. No further computation is then required on the pivot column. When another process receives this message, it swaps its pivot row with its first row. Then all of the processes, including the owner of the pivot column, subtract the pivot column, appropriately scaled, from each remaining column.

In the parallel algorithm below, the number of processes is p, and the index of the processor is *me*. The value of *me* is different for each process and is in the range $0 <= me <= p$. Aside from that, each process uses the same algorithm.

```
me ← index of processor
for i = 1, ..., N-1 do
    if (i-1 mod p) = me then
        Find the index ip of the maximal element of the i^th column of A
        Swap rows of i and ip of A, so A_ii is the pivot element.
        Divide the pivot column, A_i+1,i through A_Ni, by the pivot A_ii
        Broadcast the pivot column, A_i+1,i through A_Ni, and the row number ip to swap.
    else
        Receive the pivot column and the pivot row number.
        Swap rows i and ip.
    endif
    Scale and subtract the pivot column from each remaining column.
end for
```

Table 4, Table 5 and Table 6 show the experimental results with two different Consistency Diagram as input. In this case the result of both approximation algorithms

Experiment	#events	#edges	size of label	lower bound	estimation accuracy	speedup of estimation
1.	180	28	7	942,683	2.3	2.8 times
2.	472	48	12	25,795,922	2.8	2 times

Table 8: Lower bound estimation for Matrix factoring.

Experiment	#events	#edges	size of label	upper bound	estimation accuracy	speedup of estimation
1.	180	28	4	2,190,710	1.744	5.4 times
2.	472	48	6	72,226,687	1.95	5 times

Table 9: Upper bound estimation for Matrix factoring.

are quite good, as shown.

7.3 The Hirschberg and Sinclair elective algorithm

There are algorithms where there is a particular process which performs some 'managerial' task, called co-ordinator process (as in Section 7.1). However, in case the co-ordinator process fails, another process should be selected to take over the role of the co-ordinator. Often, the remaining processes conducting a negotiation among themselves, and a process is chosen which has the highest (or lowest) identifier. The algorithms associated with such negotiations are called *election algorithms*.

In the Hirschberg and Sinclair elective algorithm [Ray88] processes are arranged around a bidirectional ring in arbitrary order. Each process in the ring is able to execute the following communication primitives :

- send the same message to both the left-hand and right-hand neighbor
- pass the message received from the right-hand neighbor to the left (or conversely) without modification
- send a response to the neighbor from which a message has just been received

Experiment	#events	#edges	size of label	exact number of cuts
1.	111	54	7	14
2.	257	123	7	13

Table 10: Exact number of cuts computed for Election.

Experiment	#events	#edges	size of label	lower bound	estimation accuracy	speedup of estimation
1.	111	54	7	12	4.5	0.7 times
2.	257	123	7	5	24.6	0.35 times

Table 11: Lower bound estimation for Election.

The idea of the algorithm is to perform a sequence of elections on increasingly large subsets of the processes until all have been included. It works as follows:
The process which initiates the procedure declares itself as a candidate and seeks to find if its own identifier is greater than that of either of its two neighgbours by sending its id to both of them. These neighbors compare their id with the received id and if their id is found to be greater than the received id then the corresponding neighbor declares itself as candidate; otherwise the initiated process remains to be the candidate. As a next step, the candidate process tests its id against a larger set of processes, in fact doubling the number of processes consulted. If a candidate is defeated at any election its task is simply to pass any message received from one hand to the other. This continues until all the processes have been consulted.

Table 10, Table 11 and Table 12 show the results for the cases where there are 111 events with 54 edges, and 257 events with 123 edges respectively. Again, the lower bound estimation do not perform very well. However, the upper bound estimation results are still reasonable to be used.

Experiment	#events	#edges	size of label	upper bound	estimation accuracy	speedup of estimation
1.	111	54	1	36	2.57	2 times
2.	257	123	1	49	3.77	1.5 times

Table 12: Upper bound estimation for Election.

7.4 Conclusion

The experimental results confirm the usefulness of the upper bound estimation as the resulting $\wp \gg 1$ in almost all cases. In fact, it performs extremely well especially in the third Experiment of the first example. In this case the exact bound algorithm is very slow (since the size of the label is very high), while the upper bound estimation algorithm performs its task quickly and with good accuracy.

Chapter 8

Concluding Remarks

8.1 Conclusion

Distributed programs are difficult to develop and analyze. This is due to their inherent characteristics such as concurrency, unavailability of global state and global time. The fact that these aspects have still not been completely mastered at the conceptual level is one of the reasons for the lack of adequate tools for the design and analysis of distributed systems.

The motivation behind this thesis is the difficulties in distributed program analysis and debugging. As one attempts to analyze program behavior through distributed predicate detection, one may end up with an exponential cost in the detection process. The main reason behind high complexity is the exponential growth of the number of system states with concurrency. If examination of all states is inevitable, then the cost of analysis may be expensive. Therefore, we attempt to address here the size of the global state space and the complexity of distributed predicate detection in general, and propose some strategy to deal with the problem in practice.

We observe that the complexity may be attributed to two factors:

1. The size of the global state space, and
2. The nature of the distributed predicate.

In practice, we may have to identify whether both unfavorable conditions exist in the application, and if they do, we may have to find a suitable last resort to deal with them. This thesis presents results leading to answers to these questions.

An algorithm for estimating the size of the global state space of an observed execution is proposed. While we have not succeeded in finding [and strongly conjecture the non-existence of] a polynomially time bounded algorithm to compute the exact number of cuts for an arbitrary consistency diagram, the upper bound estimation technique is still useful to be included in a distributed debugger to help users to move as far as possible in the debugging process. In addition, a detection method is introduced for some simple form of distributed predicates, and a useful strategy is proposed to deal with distributed predicate detection in general.

Although some distributed predicate detection may involve an exponential cost, practical systems may not exhibit such an intolerable complexity. This could be due to the synchronization design of the distributed system where processes tend to communicate locally in the form of clusters. Such special cases, where the the size of the global state space is guaranteed to be determinable in polynomial time are identified in this work.

Examples are carefully chosen to demonstrate experimental results for identifying the state space in different systems. Therefore, experiments are conducted using the following three different applications: (i) Calculation of the value of Π: a single co-ordinating process performs certain functions on behalf of the others - the synchronization mimics the centralized 'client/server' model; (ii) Matrix factoring: Processes communicate through broadcast; and (iii) Hirschberg and Sinclair's elective algorithm: A token ring forms the underlying synchronization structure. These experiments show that the upper bound estimation is quite effective.

The design of adequate tools may help the development and analysis of distributed systems, but finding correct and efficient algorithms for general distributed predicate detection still remains to be a challenge.

8.2 Suggested further work

1. Most of the theories about distributed predicate detection are based on some abstract models (where the notion of processes in the sense of linearly ordered disjoint subsets of events does not exist). It would be useful to identify a mathematical model for the representation of a distributed system which would be more suitable.

2. Implementation of the model and the algorithms for the computation of the number of global states presented in this thesis in a distributed debugging system.

3. An examination of the significance of the state space estimation strategy in different applications, other than distributed debuggers.

4. Identification of the different classes of distributed predicates and the amount of complexity increase that each would contribute to a given distributed execution.

Bibliography

[BR95] O. Babaoglu and M. Raynal. Specification and verification of dynamic properties in distributed computations. In *Journal of Parallel and Distributed Computing*, volume 28, pages 173-185, 1995.

[CL85] M. Chandy and L. Lamport. Distributed snapshot: Determining global states of distributed systems. In *ACM Transactions on Computing Systems*, volume 3, pages 63-65, February 1985.

[CM88] M. K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163-173, Santa Cruz, California, May 1991.

[FR94] E. Fromentin and M. Raynal. Inevitable global states: A concept to detect unstable properties of distributed computations in an observer independent way. In *Proc. of the 6th IEEE Symposium on Parallel and Distributed Processing*, pages 242-248, Dallas, Texas, October 1994.

[GC95] V.K. Garg and C.M. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the 15th Int. Conf. on Distributed Computing Systems*, pages 423-430, Vancuver, Canada, June 1995.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability - A guide to the theory of NP-completeness.* W.H. Freeman Publishers, 1979.

[GW92] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of the 12th Int. Conf. on Foundations of*

Software Technology and Theoretical Computer Science, pages 253-264, Springer Verlag, LNCS 652, New Delhi, India, December 1992.

[GW94] V.K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, pages 299-307, March 1994.

[HW88] D. Haban and W. Weigel. Global events and global breakpoints in distributed systems. In *Proc. of the 21st Hawaii Int. Conf. on System Sciences*, pages 166-175, Jan 1988.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in distributed systems. In *Communications of the ACM*, volume 21, pages 558-565, July 1978.

[LJ96] H.F. Li and M.G. Janto. State space estimation and distributed predicate detection. Technical report, Concordia University, Montreal, Quebec, in preparation, 1996.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant reply. In *IEEE Transactions on Computers*, pages 471-482, April 1987.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215-226, Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mat93] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*, volume 18, pages 423-433, August 1993.

[MC88] B.P. Miller and J.D. Choi. Breakpoints and halting in distributed programs. In *Proc. of the 8th IEEE Int. Conf. on Distributed Computing Systems*, pages 316-323, San Jose, California, June 1988.

[MHR93] N. Plouzeau M. Hurfin and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 32-42, San Diego, California, May 1993.

[PL90] D.K. Probst and H.F. Li. Using partial order semantics to avoid the state-explosion problem in asynchronous system. In *Workshop on Computer Aided Verification*, pages 143-155, June 1990.

[Rag91] S. Ragsdale. *Parallel Programming.* McGraw-Hill, Inc., 1991.

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

[Seg93] H. Segel. *Monitoring Distributed System.* Concordia University, Master's Thesis, 1993.

[VD95] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *IEEE Transactions on Software Engineering*, pages 163-177, February 1995.