INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

DETECTION OF SEPARABLE PREDICATES ON SERIES-PARALLEL SYSTEMS

Guy Dumais

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

September 1998

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Guy Dumais**

Entitled: **Detection of separable predicates on series-parallel systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 19 ____ _____

Dr. Nabil Esmail, Dean

Faculty of Engineering and Computer Science

Abstract

Detection of separable predicates on series-parallel systems

Guy Dumais

In this thesis, we address part of the predicate detection problem on distributed computations. We introduce two new classes of predicates, the monotonic predicates and the separable predicates. These classes generalize several classes of well-known predicates as the conjunctive predicates. We show that these classes are detected efficiently on distributed computations having the series-parallel property. This deviates from the approach used in the past where the detection problem was addressed for small classes of predicates to be detected on general distributed computations.

The detection algorithm is based on a decomposition of the state lattice into simple subsets called concurrent intervals. This decomposition has its own interest since it helps understanding the relation between the complexity of the state lattice and the complexity of the event structure. At the heart of this relation is a new structure: the communication graph. We show that the communication graph gives a suitable level of abstraction to deal with the predicate detection problem.

Résumé

Détection des prédicats séparables sur les systèmes séries-parallèles.

Guy Dumais

Dans cette thèse, nous nous intéressons au problème de détection de prédicats sur les calculs répartis. Deux nouvelles classes de prédicats y sont introduites, les prédicats séparables et les prédicats monotones. Ces classes généralisent plusieurs prédicats bien connus, entre autres, les prédicats formés d'une conjonction de prédicats locaux. Nous démontrons que ces classes peuvent être détectées efficacement sur les calculs répartis ayant la propriété série-parallèle. Ceci diffère de l'approche habituelle qui considère la détection de classes restreintes de prédicats sur des calculs répartis généraux.

La présentation de nos algorithmes nécessite l'introduction d'une décomposition du treillis des états en composantes simples appelées intervales simultanées. Cette décomposition possède un intérêt propre puisqu'elle aide à la compréhension de la relation entre la complexité du treillis des états et celle de l'ordre partiel des évènements. Une nouvelle structure, le graphe des communications, est au coeur de cette relation. Nous démontrons que le graphe des communications possède le niveau d'abstraction désiré pour aider au problème de détection.

Acknowledgments

I would like to express my deep gratitude to my supervisor Dr. H.F. Li. His suggestions and guidance were invariably my best sources of inspiration. I want to thank him also for his financial support and the diligent revisions of the first versions of this thesis.

I wish also to thank some of my friends in the department, Ganesh, Jaya, Louis, Manolo, Yun and Yves, who made my life at Concordia a pleasant experience.

Contents

List of Figures

List of Notations

Introduction

The omnipresence of distributed applications as in Internet applications, multilevel security and distributed databases render their study and understanding essential. As a contribution to the study of distributed applications, we give tools to analyze the behavior of the distributed computations at run-time. In particular, we address part of the predicate detection problem and give an original solution to it. Predicate detection appears in many contexts such as debugging, testing and monitoring of distributed applications.

To introduce the problem of predicate detection consider the example of a bank with several distributed branches. A client of the bank goes to her favorite branch to deposit, withdraw or transfer money. At each branch a clerk manages the accounts of his clients. The clerk is allowed to perform all the transactions on the accounts of his clients but, when a client wants to transfer money from her account to an account in another branch, the clerk must call the other branch clerk and let him know about the operation. Suppose that someday the bank president decides to buy another bank. Before doing so, she may want to know if the bank has enough money to seal the deal, say more than $1,000. Therefore, she calls each branch and asks the clerk for the total of their client savings. After calling the clerks and summing up the amounts, the president questions if this sum gives a realistic evaluation of the bank assets. To answer her concern, she looks at the following scenario:

> There are 3 branches A, B and C to the bank. When calling A, the clerk gives a total of $500. Then calling B, the clerk gives a total of $200. Meanwhile, the clerk of branch A receives a client asking for a transfer of $200 to an account to branch C. The clerk performs the transaction promptly and calls the clerk of branch C to update the other account, which is what he does immediately. Later, the clerk at branch C receives the call from the president and he returns

an amount of \$400. Therefore, the bank would have \$500 + \$200 + \$400 = \$1,100.

But, as the president realizes, if the transfer is the only transaction performed during the calling process, there is no point in time between the first call to branch A and the last call to branch C where the bank actually owns \$1,100. As shown in the space-time diagram of Figure 1, at anytime between the first and the last call, the total amount of money owned by the bank is \$900. Therefore, the evaluation of the bank assets is not consistent with the

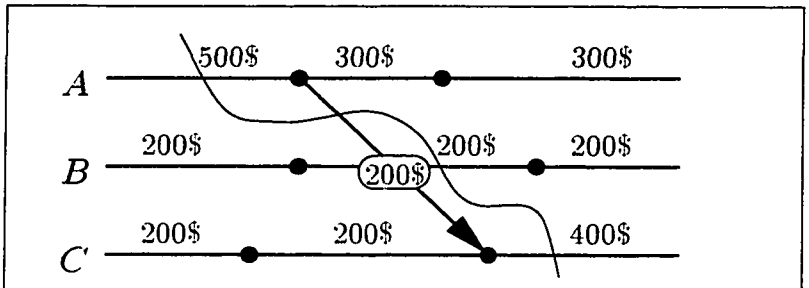

Figure 1: An inconsistent "picture" of a distributed bank

reality.

The problem of getting a "consistent picture" of the distributed computation at some instant must be addressed whenever one wants to do debugging, testing or monitoring of distributed applications. In particular, the programmer must be able to evaluate if the computation satisfies a certain property at some instant. To do so, the property must be evaluated on a "consistent picture" and not on a state that the computation will never be able to achieve. It is worth looking at two natural attempts to solve the problem in order to get a feeling of its complexity.

In our bank example, part of the problem is due to the fact that the clerks are called at different instants by the president. Suppose that each clerk has a watch which is perfectly synchronized with those of the others. To get an accurate evaluation of the money owned by the bank at some instant, the clerks may record the amount of money in their client accounts at some given time T. Because of the presence of a global time scale, the recording at time T is done simultaneously by all clerks. Therefore, the president has only to collect the records later on, and sum up their values to get a clear picture of the assets of the

bank at time T. While this solution offers an effective way to answer the problem, it is impractical. It is hard to imagine that sites that are several thousands of kilometers apart could have perfectly synchronized clocks. For that reason, an asynchronous model of distributed systems is more realistic.

Another source of the problem is due to the fact that clerks continue to perform transactions during the calling process. To fix that, the president may ask that each clerk suspends his work after the call until the calling process is completed. Under that assumption, the amount of money computed by the president gives a consistent picture of the bank assets. But again, this solution is unacceptable. For the president, the suspension of the branch activities would mean loss of money. More generally, the construction of the consistent picture of the distributed computation should not be intrusive, that is, it should not affect the underlying computation. An intrusive debugger for distributed computations has a really bad effect. It may hide some of the bugs by preventing one or more of the execution behavior to happen. These bugs may later appear when the distributed program is run without the debugger.

As mentioned above, the ability to evaluate if a computation satisfies a certain property at some point in time is important for debugging, testing and monitoring distributed computations. Most of the time, the property is phrased as a predicate on the consistent states of the computation and the property evaluation is replaced by the detection of the predicate. An example of predicate is given by the president's question:

Does the bank own more than \$1,000 at some instant?

Predicates and their associated properties have been divided in several classes. Among the most common are the stable predicates, the conjunctive predicates and a few subclasses of relational predicates. For all of these predicates, we have satisfactory detection algorithms.

In the present work, we try to enlarge the set of predicates that can be efficiently detected. In particular, we introduce two new classes of predicates, the monotonic predicates and the separable predicates. These classes are new subclasses of the relational predicates and generalized some of the common classes. Along the way, we describe a decomposition of the state lattice, a structure that contains all the consistent pictures of a distributed computation. The elements of the decomposition are simple structures called concurrent intervals. Again, the concurrent intervals are interesting in their own, and formalize similar

notions that have appeared in the past literature. The state lattice decomposition into concurrent intervals will probably prove to be a tool useful in other context.

Unfortunately, the restriction to monotonic or separable predicates does not allow us to derive a detection algorithm for all distributed computations. Algorithms are given for the detection of monotonic predicates on series systems and the detection of separable predicates on series-parallel systems. However, we believe that the series-parallel systems are general enough to be of interest in many applications.

The first chapter introduces the main definitions to be used in this thesis after a brief survey of the predicate detection algorithms. The second section will be important for the reader since it contains not only the standard definitions, but also a new structure: the communication graph. The second chapter is more abstract and presents the results regarding the decomposition of the state lattice. The properties of the decomposition are also stated and proved. The second section of the same chapter introduces the series-parallel systems to the readers. We give examples of the systems and their interplay with the decomposition. The third chapter contains all the results concerning the detection of monotonic and separable predicates. After a definition of these classes of predicates, we give the algorithm to detect them on series systems and series-parallel systems respectively. The last chapter includes further results of smaller importance. We give there an algorithm to perform the strong detection of monotonic predicates on concurrent intervals.

The presentation of our subject forces the use of several discrete structures such as related sets, partial orders and lattices. For an introduction to those topics, the reader is referred to Appendix A, where most of the needed results are summarized. The two first chapters of Davey and Priestley [DP90] contain also a good introduction to the subject. The discrete structures used in this thesis are assumed to be finite.

Chapter 1

An overview of distributed computations and predicate detection

This chapter serves as an introduction to our work. It does not contain new results but the presentation and the formal definitions of Section 1.2 place the problem of predicate detection in its proper context. In the first section, we review the previous work on predicate detection and the current state of research in that area. It includes a brief description of the detection algorithms for general predicates, stable predicates, conjunctive predicates and sequences of local predicates. References to the relevant work are also given. The second section introduces our model of distributed computations. Along with the standard definitions, we formalize a new structure: the communication graph. Even though most of the entities are defined in mathematical terms, we give their natural interpretation and examples wherever possible.

1.1 Distributed systems and predicate detection

A **distributed system** consists of several computer systems or processors, called **sites**, each running a distinct process. The sites are connected together by a communication network and the only means to communicate between two processes is by exchanging messages. The message delivery takes an arbitrary, but finite, time delay, and a series of messages sent

through a channel are received in the FIFO order. We do not assume that the sites share a common clock.

Debugging, testing, and monitoring of **distributed programs** (programs that run on a distributed system) rely firmly on the ability to evaluate whether the states of the processes satisfy some given properties. Typically a property of the process states is expressed by a **predicate**. The predicate is true if and only if the process states satisfy the property. If the predicate involves only the states of one process, it is called a **local predicate**. On the other hand, a predicate expressing the properties of the states of many processes is called a **global predicate**. In the following, we are mainly interested in global predicates. Therefore, we usually drop the adjectives and simply assume that all the predicates are global.

In this first section, we give a general view of the predicate detection problem on a **distributed computation** (an execution of a distributed program). The presentation is based on intuition more than formalism. Let us look at some examples first.

Example 1.1 Suppose we want to know if all the processes in a distributed computation have received a special message. To do so, we introduce Boolean variables, denoted b_i for process i. The variables are set to FALSE initially and turned to TRUE when the process that owns it receives the message. Consider the predicate $\mathfrak{P}_1$ = "$b_1 \wedge b_2 \wedge \cdots \wedge b_n$". This predicate is true if and only if all the processes have received the message. Detecting if $\mathfrak{P}_1$ is true is equivalent to the detection of the desired property. □

Example 1.2 Consider a server having at most five connections at one time. Any distributed program using that server must never make use of more than five connections. To make sure that a distributed program satisfies the above property, we can add a counter x_i to each process. The counter x_i would keep a count of the number of connections held by process i. A violation of the property would be detected by a predicate of the form $\mathfrak{P}_2$ = " $x_1 + x_2 + \cdots + x_n > 5$". □

Example 1.3 Assume that a distributed program needs at some point of its execution to hold at least 3 units of the same resource to make further progress. We want to know if the program can complete successfully. To monitor the property, we add a counter x_i to each

process as in Example 1.2. The property can then be expressed by the predicate $\mathfrak{P}_3$ =" $x_1 + x_2 + \cdots + x_n \geq 3$". Again, the property is satisfied if at some point of the execution the predicate $\mathfrak{P}_3$ is true. □

In order to evaluate a predicate, one might try to record the process states at some instant of the execution. But, as mentioned earlier, the processes do not share a common clock. It is therefore impossible for all the processes to record their states at the same time. Another approach to perform the detection is needed. In the absence of global time, it is meaningless to ask if two events occurred at the same time. The simultaneity of events relies on an arbitrary time scale which is not part of the distributed system. This reminds us of the counterpart in relativity, where different observers may disagree on the ordering of distant events. In a distributed system, the only well-founded ordering of events is the one imposed by the causality between events. Lamport, in his pioneering paper [La85], presented such an ordering: the happens-before relation. Intuitively, two events cannot occur at the same time (no matter the time scale) if one causally affects the other in the continuum of space and time. In other words, the cause must happen before the effect. The messages in a distributed system provide a good illustration of the relation. The sending of a message must happen before its receipt. It is this relation that must be taken as a basis to order the events in the system. Since the relation imposes only a partial order of the events occurring in the system, there are several total orders consistent with the partial order. Each of these total orders is called an **observation** and represents the ordering of events that could be given by an external observer.

The happens-before relation naturally leads us to the assumption that two process states not causally related are **consistent**, that is, they may coexist in some computation. More generally, a set of pairwise consistent process states form a consistent "picture" of the distributed computation since they may coexist at some point in time. These sets of consistent process states are called the **global states** (sometime called consistent global state).

Predicates have been grouped into different classes. A predicate that does not turn false after it becomes true, as $\mathfrak{P}_1$ in Example 1.1, is said to be **stable**. Clearly, not all predicates are stable and a good example of an **unstable** predicate is given by $\mathfrak{P}_2$ in Example 1.2. The value of an unstable predicate can alternate between TRUE and FALSE which makes this class of predicate harder to detect. As pointed out in the past literature [MN91], there

are several ways to interpret the predicate detection problem. At the moment, two of these interpretations are discussed and we postpone a more complete discussion to Chapter 4. We say that a predicate has been **weakly** detected if there exists at least one observation of the distributed computation such that, at some point of the observation, the set of process states satisfies the property expressed by the predicate. This has also been phrased as the detection of possibly($\mathfrak{P}$), where $\mathfrak{P}$ is the predicate. Weak detection is observer-dependent, that is, it may happen that for two different observations of an execution, one has a state that satisfies the predicate, while the other does not. The weak detection is particularly appropriate for safety properties [MN91]. Given that a predicate $\mathfrak{P}$ represents an error condition as in Example 1.2, then possibly($\mathfrak{P}$) represents a possible occurrence of the error, that is, a violation of the safety property. On the other hand, one might be interested to know if, regardless of the observation, there will always be a state such that $\mathfrak{P}$ is satisfied. This is called **strong** detection or detection of definitely($\mathfrak{P}$). Strong detection is observer-independent. Strong detection is more suited to express a good property that must happen. Given that $\mathfrak{P}$ is a desirable predicate as in Example 1.3, definitely($\mathfrak{P}$) represents a certainty that $\mathfrak{P}$ is satisfied regardless of the observer. Several algorithms have been proposed to detect stable and unstable predicates and we turn now to a short description of the most representative ones.

In the following, we use detection to mean weak detection unless it is stated otherwise.

1.1.1 Detection of stable predicates

Detection of stable predicates is simple and can be done efficiently on a general distributed computation. An algorithm is given by the Chandy-Lamport distributed snapshot [CL85]. A simplified version of the algorithm constructs a global state of the computation (snapshot) as follows. 1) One or more processes initiate the algorithm by saving their current states and then sending notifications through all their channels to the other processes. 2) A process receiving a notification for the first time, saves its current state and, before resuming its execution, sends notifications through all its channels. 3) If a notified process happens to receive additional notifications, it ignores them. 4) The algorithm ends when all the processes are notified. Assuming that the graph representing the communication network is strongly connected, the notification is guaranteed to reach all the processes. Furthermore,

if the messages sent along one channel are received in the same order, then the recorded process states form a global state of the system. Once a global state is known, one can test if it satisfies the predicate. Periodic construction of global states can give a detection technique for stable predicates.

The Chandy-Lamport algorithm and its successors have low cost in space and time and do not interfere with the underlying computation. Unfortunately, these algorithms cannot be used to detect unstable predicates. Indeed, a snapshot algorithm constructs only a subset of the global states and an unstable predicate may happen to be true only on global states not in that set.

1.1.2 Detection of unstable predicates

Global predicates include general Boolean expressions defined on a global state. Hence, the class of predicates is vast and includes all the example predicates given so far. We distinguish between two complementary approaches to solve the predicate detection problem. The first one is to consider the whole class of predicates and to give a general algorithm to detect any member of the class. This approach will invariably give solutions with exponential cost. Indeed, it can be shown that the detection of the predicate

$$x_1 + x_2 + \cdots + x_n = c$$

for integer variables x_i and a constant c is NP-complete ([Li]). The other approach is to get a more practical solution to the problem by restricting the set of predicates to be detected, such as the stable predicates, the conjunctive predicates and the relational predicates.

Marzullo and Neiger [MN91] used the general approach. Their algorithm detects a predicate by constructing all the global states. The algorithm necessitates the construction of $O(l^n)$ states and keep a record of $O(l^{n-1})$ states at each stage, where l is the number of local states in a process and n is the number of processes. This approach is clearly impractical for large n. In order to lower the cost in space and time, a synchrony assumption can be used. Each process carries a real clock which is approximately synchronized with the other clocks. In addition to this, it is assumed that there are lower and upper bounds on message transmission times. Under these assumptions, the algorithm is improved by constructing only the states that are in the prescribed bounds. The monitoring process can

then consider only a subset of the states for the detection.

Efficient algorithms are available for a restricted class of predicates. In particular, the detection of conjunctive predicates has attracted a lot of interest. Conjunctive predicates are useful to capture some important properties. They are also easily detectable. For example, mutual exclusion is expressible as a conjunctive predicate and its detection can be carried out in polynomial time. Formally, a **conjunctive predicate** is a conjunction of local predicates and has the following form

$$\mathfrak{p}_1 \wedge \mathfrak{p}_2 \wedge \cdots \wedge \mathfrak{p}_n,$$

where $\mathfrak{p}_i$ is a local predicate on the i^{th} process. Efficient algorithms have been presented by Venkatesan and Dathan [VD95], Garg and Waldecker [GW94, GW96], by Hurfin, Mizuno, Raynal and Singhal [HMRS95] and others.

There are two main approaches in the design of a distributed debugger: the off-line approach and on-line approach. Off-line algorithms are run after the termination of the computation. Generally, the processes involved in the computation record the relevant debugging information to be used by the debugger. Off-line algorithms are easier to design than their on-line counterparts, but they fail to give rapid feedback to the user. On the other hand, on-line algorithms are run concurrently with the underlying computation. As with most sequential debuggers, the on-line algorithms give rapid feedback and allow halting of the computation. The drawback of the on-line algorithms for distributed computations is that they may interfere with the underlying computation.

The algorithm to detect conjunctive predicates given by Venkatesan and Dathan is off-line, fully distributed and does not use the vector clock [Ma88, Fi91]. From the history of each process, it constructs the first global state that satisfies the conjunctive predicate (if this state exists!).

Garg and Waldecker have solved the same detection problem using an on-line and centralized algorithm based on vector clock. During the execution, a central process maintains the process states that satisfy the local predicates in a set of queues. As the computation goes, the states are dequeued if they cannot coexist with the other process states at the heads of the other queues. The predicate is detected when all the head of the queues can coexist. It is interesting to note that an algorithm similar to the Garg and Waldecker algorithm has been presented by Fromentin and Raynal in [FR94] to detect definitely($\mathfrak{P}$) rather

then possibly($\mathfrak{P}$). It uses the notion of inevitable state, a state presents in all observations.

Hurfin, Mizuno, Raynal and Singhal [HMRS95] improved the previous detection algorithm by giving an on-line distributed detection algorithm for conjunctive predicate.

Relational predicates form another class of predicates studied by researchers. Examples of relational predicates are given by $\mathfrak{P}_2$ and $\mathfrak{P}_3$ from the previous examples. The relational predicates form a wide class that includes the conjunctive predicates. The difficulty to detect these predicates forces researchers to concentrate on subclasses. These include Tomlinson and Garg [TG93, TG97] who address the detection of relational predicates of the form $x_1 + x_2 < c$. They give an on-line centralized algorithm as well as a decentralized version to perform the detection.

1.1.3 Detection of sequences of predicates

As pointed out by Babaoğlu and Raynal [BR95], predicates defined on a single state cannot capture properties involving a time ordering. To illustrate the problem, let us return to the distributed bank example from the introduction. Consider a bank with three branches A, B and C. Suppose the interest rate of a loan is partly decided on the changes in bank assets. Say the interest rate must be revised whenever the bank assets drop from \$2,000 to less than \$1,000. Such a property cannot be captured by a predicate on a single state but could be with a sequence of predicates. It involves the two predicates

$$Assets_A + Assets_B + Assets_C \geq \$2,000$$

and

$$Assets_A + Assets_B + Assets_C \leq \$1,000.$$

These two predicates must be detected in the given order which differs from the detection of predicates on a single state. As demonstrated in [BR95], sequences of predicates have more expressiveness and can prove to be really useful when detecting properties involving a time ordering. Again the difficulty of detecting a sequence of predicates forces the researchers to choose between tractability and generality.

Babaoğlu and Raynal followed the second option. In [BR95], they use an approach similar to the one used by Marzullo and Neiger [MN91] to solve the general predicate detection problem. In order to detect a sequence of predicates, the algorithm constructs

all the possible states and evaluates the predicate sequence on them. The result is a very general detection algorithm but with exponential complexity in space and time.

In earlier work, Miller and Choi [MC88] introduced linked predicates. Instead of being tested on a sequence of global states, a linked predicate uses sequences of process states. For example, a linked predicate may be useful to someone that wants to break the execution of a program whenever a given sequence of events (that may belong to several processes) occurs. Their algorithm performs the detection in polynomial time.

A generalization of linked predicates is given by the atomic sequences [HPR93]. Given an undesirable process state σ, we say that a sequence of process states is atomic if σ does nct appear in the sequence. A typical solution of an atomic sequence is given by the sequence of events satisfying the underlying linked predicate and free of occurrence of some given forbidden events. Again, the detection can be done in polynomial time.

1.1.4 Our work: Detection of monotonic predicates and separable predicates

Restrictions are applied to the class of predicates to be detected in order to ensure a reasonable complexity of the detection problem. We group these restrictions in three classes: synchrony restriction, predicate restriction and communication restriction. The state explosion is mainly due to the asynchrony among the processes. This asynchrony is needed since there is no shared clock among the processes. When such a clock exists, only one observation results. In real applications, implementing an ideal common clock is not feasible. A possible goal is to synchronize the local clocks of the processes so that their skews are within some fixed bound. The bound can then be used to lower the complexity as the process states separated too far apart in their respective local times cannot be consistent. An example of this is the work of Marzullo and Neiger as described above.

The predicate restriction is the most common one. We have already seen examples of this in the context of stable predicates and conjunctive predicates. Stability is certainly a property that simplifies the detection. Since a stable predicate always stays true after turning true, there is no need to check the predicate on all the states. It is enough to perform the detection on a few states but periodically. It is less obvious why conjunctive predicates are easy to detect. One of our explainations is that a process state takes only

two values, either FALSE or TRUE.

Instead of restricting the class of predicates to be detected, one can decide to detect a large class of predicates on specialized distributed computations. Indeed, another source of complexity is given by the geometry of the distributed computation, that is, the number of processes, the number of messages, and their causality relationship. In the extreme case, the detection problem on a distributed computation involving only one process or having no messages is a much simpler problem for several classes of predicates. The goal is to find a class of distributed computations large enough to be of practical interest, but simple enough so that the detection problem becomes tractable. This is our approach to detection and we now turn to a short description of our work.

The work most closely related to ours is the one by Tomlinson and Garg [TG97]. We recall that they provided an algorithm to perform the detection of relational predicates of the form $x_1 + x_2 > c$, where x_i are integer valued variables. Our goal is to extend the detection to two larger classes of predicates. The first class, the monotonic predicates, contains predicates of the form

$$f(x_1, x_2, \ldots, x_n) < c,$$

where f is a monotonic function of the integer valued variables x_i (note that the direction of the inequality has little effect on the result). The second class, the separable predicates, contains predicates of the form

$$h_1(x_1) \oplus h_2(x_2) \oplus \cdots \oplus h_n(x_n) < c,$$

where h_i are single valued functions and max is distributive on the binary operator $\oplus$. It is not hard to see that when h_i is the identity and $\oplus$ is replaced by $+$, we get the predicate

$$x_1 + x_2 + \cdots + x_n < c.$$

This shows that the detection of separable predicates includes, as a special case, the problem addressed by Tomlinson and Garg. It is not known if the detection of separable predicates can be done efficiently on a general distributed computation, but using the series-parallel property, we show that there exist efficient algorithms on restricted distributed computation. An additional difficulty comes from the fact that the series-parallel property, applied directly

to the partial order of events, is too restrictive. To bypass the difficulty, we introduce a description of the partial order at a higher level of abstraction. We define the communication graph, a structure based on the messages in the distributed computation. When applied to the communication graph, the series-parallel property allows for a large class of distributed computations.

1.2 Distributed computation model

We recall that our distributed system model is composed of n independent sites communicating through message-passing. We further assume that there is no clock for synchronizing the processes, that the messages are delivered in FIFO (first-in first-out) order and that the message transmission delay is arbitrary but finite.

An occurrence of interest in a process is called an **event**. We distinguish among three kinds of events: an **internal** event, the **receipt** of a message and the **sending** of a message. An internal event usually corresponds to a change of the value of some process variable. All events are assumed to be instantaneous. The execution at one site defines a process which is represented as a chain of events. The j^{th} event in process i is denoted by ϵ_i^j .

The values of the process' variables at some point in time form a **local state**. Note that the process' variables can include variables used in the computation, as well as control variables like the program counter, the number of connections to a server and so on. Moreover, a local state may contain information about the messages sent and received. The occurrence of an event results in a new local state.

The receipt and sending of messages are the events of importance in the following. They are necessary to establish the causality relation among the events. To a message m in a distributed computation correspond two events, the sending of m, $send(m)$, and the receipt of m, $rec(m)$. The set of events is given a partial order structure using the happens-before relation ($\rightarrow$). More precisely, let D be the set of events (internal, sending, receipt) of the execution of the distributed program. We recursively define the relation by

Definition 1.1 Let ϵ and ϵ' be events in D. We say that ϵ **happens-before** ϵ' ($\epsilon \rightarrow \epsilon'$) if

1. ϵ and ϵ' belong to a same process and ϵ occurs before ϵ' or;

2. e is the sending of a message and e' is the receipt of the same message or;

3. there exists an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

Intuitively, events e and e' are related under the happens-before relation if e can causally affect e'. This causal relation can be broken in a sequence of events

$$e = \epsilon_1, e_2, \ldots, e_{k-1}, e_k = e'$$

such that, for all i, e_i and e_{i+1} are in a same process and e_i occurs before e_{i+1} or e_i is the sending of a message and ϵ_{i+1} is the corresponding receipt. In that case where $e \rightarrow e'$, e is a **predecessor** of e', while e' is a **successor** of e. The happens-before relation is an irreflexive partial order on D (a relation which is transitive and antisymmetric).

The happens-before relation induces a similar relation on local states. Let σ_i^j be the local state of process i after the j^{th} event is executed. We define the **consistency relation** on the set of local states by:

$$\sigma_i^j \rightarrow \sigma_k^l \text{ if and only if } \epsilon_i^{j+1} \rightarrow \epsilon_k^l.$$

A caution is needed here. While, we use the same symbol ($\rightarrow$) for both, the consistency relation and the happens-before relation, these two relations are clearly different. The context will tell the user which one is involved. Two local states, σ_i^j and σ_k^l, are said to be **consistent** under this relation if they are from two different processes and there is no event e such that $\epsilon_i^{j+1} \rightarrow e \rightarrow \epsilon_k^l$ or $\epsilon_k^{l+1} \rightarrow e \rightarrow e_i^j$. The local states σ_1^2 and σ_2^1 in Figure 2 as well as σ_1^0 and σ_2^1 are consistent. Consistent local states can coexist or more precisely, the local states of two processes at some point in time are consistent. Note that, before the execution starts, the local variables of a process are initialized to some values forming an **initial local state**. Let σ_i^0 be the initial local state of process i. Contrarily to the other local states, σ_i^0 is not preceded by any event. So, to extend the above partial order to the initial local states, we must assume that they are initiated by some fictional initial events. It follows that the initial local states of two processes are always consistent. This makes sense since all the initial local states must coexist at the beginning of the execution.

A **distributed computation** is a partial order, $(D, \rightarrow)$, where D is the set of events of the processes and $\rightarrow$ is the happens-before relation. We often denote a distributed computation by its set of events D only. A distributed computation is diagrammatically

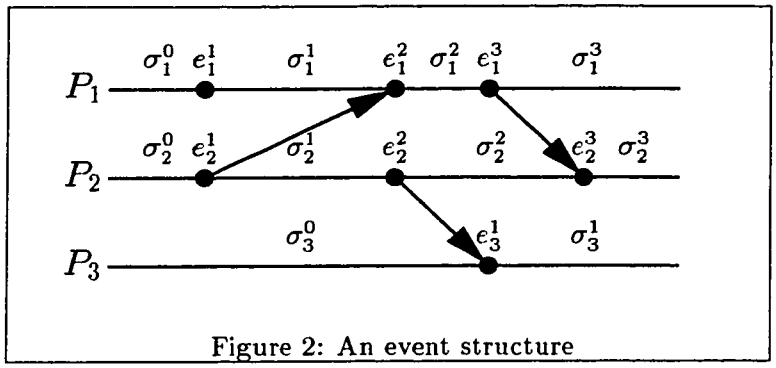

Figure 2: An event structure

represented by an **event structure** which is a graph where the nodes are events, the processes are horizontal lines and the arrows are messages (the event structure is equivalent to the transitively reduced graph of the partial order $(D, -)$). Figure 2 gives an example of an event structure.

In the previous section, we informally presented the notion of a global state. We turn now to a more precise definition. Given an irreflexive partial order $(P, \leq)$, a subset A of P is called a **down-set** if, for any $x \in A$, $y \in P$, $y \leq x$ implies that $y \in A$. We define the down-set $\downarrow(S)$ generated by a subset S of P as the smallest down-set containing S. This set must exist since P itself is a down-set containing S. The down-set $\downarrow(S)$ contains the elements of S and all their predecessors.

Definition 1.2 A (global) **physical state** is a down-set of a distributed computation.

Among the physical states of a distributed computation D two special cases deserve to be mentioned. The **initial state** σ_0 of D which is the empty down-set and the **final state** σ_f which is the down-set containing all of D. An example of a physical state is given by the set of gray dots in Figure 3. A remark about the terminology is needed here. In the literature, physical states have been called consistent global states, minimal prefixes and consistent cuts. A **cut** refers to the segment drawn in the event structure when representing a physical state. The cut associated to a state σ is denoted by $[\sigma]$. It is an n-vector where the i^{th} entry is the local state of process i after the execution of all the events in the physical state. The curved line in Figure 4 represents the consistent cut $(\sigma_1^3, \sigma_2^2, \sigma_3^0)$ which is also written as (3 2 0). A **prefix** refers to the fact that a physical state contains all the events preceding a cut. We add the adjective "physical" to distinguish physical state from logical state defined

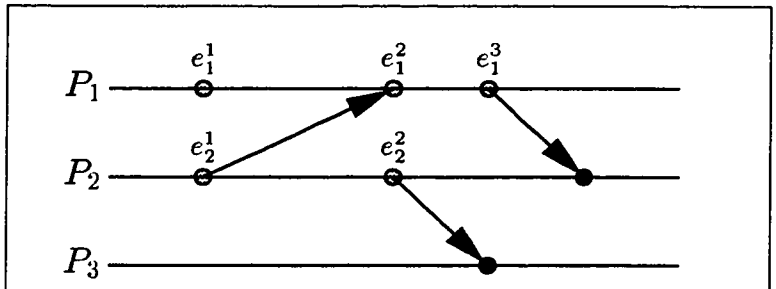

Figure 3: A physical state

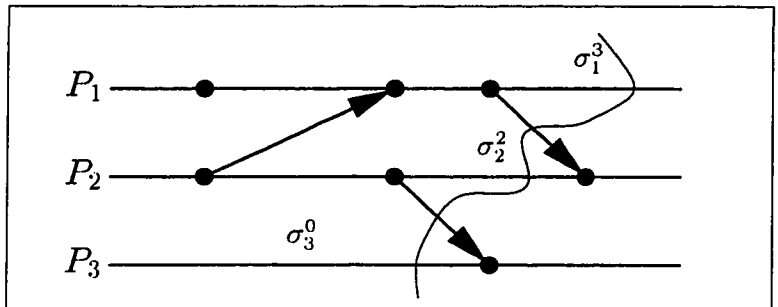

Figure 4: A physical state seen as a cut

below. It also emphasizes the fact that a physical state depends only on the event structure and not on the predicate to be detected. A set definition for the physical state is preferred to the vector definition (as in the definition of a cut) because it simplifies the notations and the proofs by naturally allowing the use of operators as $\cup$ and $\cap$.

We denote the physical state $\downarrow(e)$ associated with an event e by σ_e. By definition, it is the smallest down-set containing the event e. σ_e is called the **minimal prefix** of e and it contains e and all the events that precede it. Intuitively, the minimal prefix is the set of all events that must be executed before e can be executed. We can define in a similar way the **maximal prefix** of an event e as the maximal down-set not containing e. In particular, the set ϕ_e can not contain any successor of e. The maximal prefix can be interpreted has the set of all events that can be executed without e. In Figures 3 and 4, the state shown is the maximal prefix of the event e_2^2.

The set of physical states of a distributed computation D is denoted by $\Gamma(D)$. $\Gamma(D)$ endowed with set union and intersection operations has a natural distributive lattice structure

(see [DP90] for a detailed proof).

Definition 1.3 The **state lattice** is the set $\Gamma(D)$ of all physical states of $(D, \rightarrow)$ endowed with the natural distributive lattice structure and the attached partial order $\subseteq$ is called the **reachability** relation.

The state lattice of the event structure in Figure 2 is given in Figure 5. Each node is a triple representing a unique physical state. More precisely, the node (i j k) corresponds to the cut $(\sigma_1^i, \sigma_2^j, \sigma_3^k)$ which is the state where i events have been executed from process 1, j events from process 2 and k events from process 3. A state is reachable from another state if there is a directed path in the state lattice going from the latter to the former. As an example, the state (1 2 0) in Figure 5 is reachable from states (1 0 0) and (0 1 0). A directed path in the

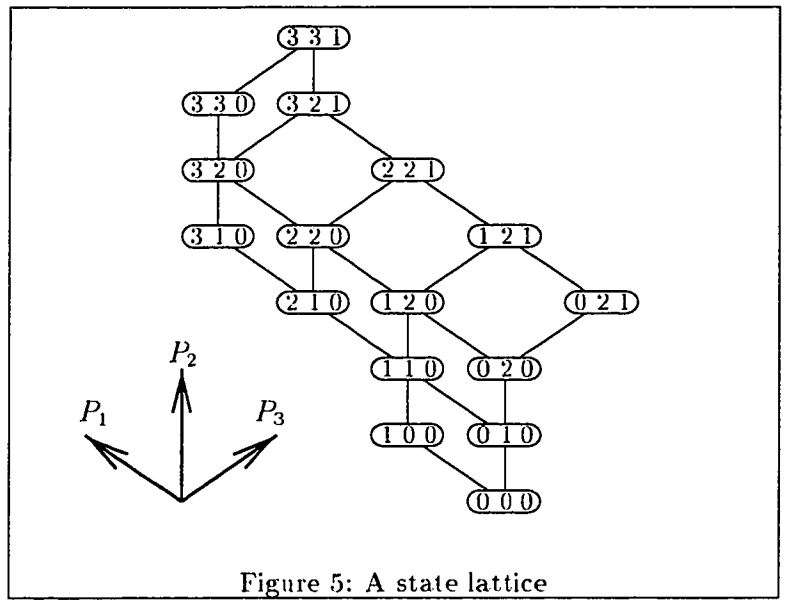

Figure 5: A state lattice

state lattice has a special meaning. It gives a possible ordering of events observed by some observer. Formally, an **observation** $\mathcal{O}$ is a total order of the events in D consistent with the happens-before relation. The sequence of events $e_1^1, e_2^1, e_2^2, e_3^1, e_1^2, e_1^3, e_2^3$ is an example of an observation of the distributed computation D in Figure 2. This total order is consistent with the partial ordering of D since, for any pair of events (e, e') such that $e \rightarrow e'$ in D, the event e appears before e' in the sequence. The above observation is given by the directed path (0 0 0), (1 0 0), (1 1 0), (1 2 0), (1 2 1), (2 2 1), (3 2 1), (3 3 1) in Figure 5.

According to Fromentin and Raynal [FR94], there exist two levels of description of a distributed computation. The first level is the one discussed above where all the changes in the local state, all the sendings and all the receipts of messages are events. This has been called the Lamport level description, but we prefer the term **physical level** since the events in that description correspond to physical occurrences like the sending of a message, the change of some value in memory and so on. As they pointed out, some of the events in the physical description might not be of interest in detecting a given predicate. Therefore, they introduce a second level of description suitable for predicate detection. We call it the **logical level** description and turn now to its presentation.

A local state is made of the value of several variables. In most cases, the system must be tested on a property that involves only a subset of those local variables. Only these **relevant variables** should appear in the predicate expressing the property. To fix the idea, consider a property expressed by a predicate $\mathfrak{P}$ that depends only on one variable per process. It is enough to include in the local states only one value. The detection of the predicate $\mathfrak{P}$ is done by monitoring the change of the variables. Figure 6 illustrates the situation. In process 1, the value of the relevant variable changes successively from 1 to 3 and then back to 1. It should be noted that the values in local states σ_1^2 and σ_1^3 are

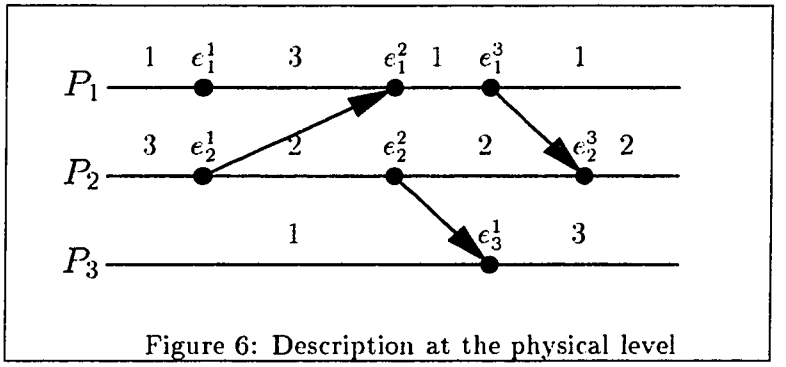

Figure 6: Description at the physical level

identical. Therefore, there is no need to distinguish among those two states. This suggests that the definition of local state in the logical description should be bound to the specific predicate and its relevant variables. A local state written in term of its relevant variable values is called a **logical local state** to emphasize the fact that it depends on the predicate to be detected. It is now clear that not all the events described above (internal, sending, receipt) causes a change of logical local states. Event e_1^3 in Figure 6 is a good example.

To emphasize this fact, we introduce a new keyword, the **relevant events**, which are the events that cause a change of logical local states. Figures 7 shows the relevant events of the previous example. The description of a distributed computation at the logical level is given by the set of relevant events endowed with the happens-before relation induces on this subset. The use of an event structure as in Figure 7 to describe a distributed computation

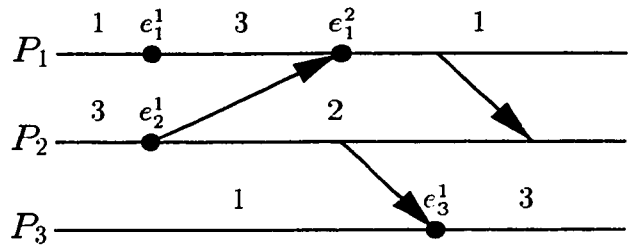

Figure 7: Description with the relevant events

at the logical level is not convenient. The problem is that the messages are no longer from event to event but may have one or both ends hanging freely. For example, the right-most message in the event structure of Figure 7 has no end "attach" to a process by an event. In that particular example, the message can be simply removed since it does not contribute to the partial ordering of the relevant events. A better representation of this event structure is given in Figure 8. There an arrow between two processes does not necessarily represent an actual message, it is simply the representation of the abstract ordering between relevant events. We call these interprocess relations **logical messages.**

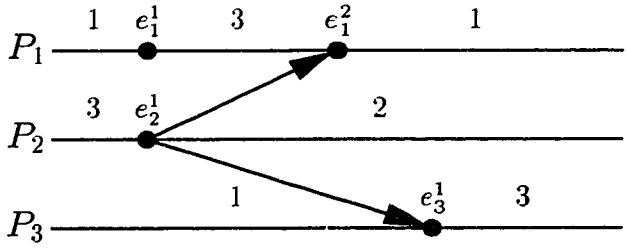

Figure 8: Description at the logical level

We could obviously push the logical level description further to introduce the notions

of logical state, logical state lattice and so on, but we would not gain much. Moreover, our work is independent of the description used. For us, an event structure is simply an abstract graph and the fact that the messages could be physical messages or logical ones does not affect our results. In the remaining, the adjectives "physical" and "logical" are omitted. So a message can be either interpreted as a logical message or physical message. However, there is a difference between the two descriptions that worth mentioning. The logical description tends to have more complex event structures than the physical one. For example, a node can have several messages going out or several messages coming in or both, which is not possible in the physical case. Note that these cases may still find natural correspondents in the form of the broadcast of messages, the gathering of message and the forwarding of messages.

As it will become clear in the following chapters, the messages of a distributed computation play an important role in the predicate detection problem. Without any message, a distributed computation is simply a set of independent processes where any pair of events from different processes are independent. The presence of messages prevents some events to occur at the same time by synchronizing the processes. We propose to introduce a new structure to capture the distributed computation at the message level. To do so, we must decide on an ordering of the messages. If we look at an event structure like the one in Figure 9, we might be tempted to order the message m_1 before the message m_3. But how to order the messages m_1 and m_2 or m_3 and m_4? An ordering defined arbitrarily might be artificial or immaterial to the processes. A better alternative is to consider the ordering as seen by a process. What would be the natural ordering of messages seen by a process? The obvious answer is to base the ordering on the natural ordering of events in a process. A message whose sending precedes that of a message m in the process would naturally be ordered before m. More generally, if a communication event (sending or receipt) of m happens before a communication event of m' in a process, then the process orders the message m before the message m'. By doing so, the process 1 in Figure 9 might have ordered the messages as m_1, m_2, m_4 and m_3(or m_2, m_1, m_4 and m_3), while the process 3 orders the messages as m_1, m_3 and m_4 since it does not know of the existence of m_2. One can notice right away that the ordering of the two processes is not consistent. Indeed, process 1 places m_4 before m_3 in the sequence, while process 3 places m_3 before m_4. This inconsistency

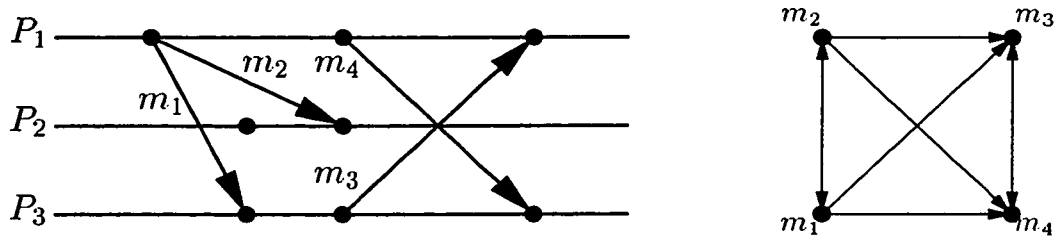

Figure 9: A distributed computation and its communication graph

cannot be avoided unless the processes exchange some additional information.

The process ordering of messages can be described more abstractly. To do so, we first define an abstract relation on the set of messages of a distributed computation.

Definition 1.4 Given a distributed computation with associated set of messages M, the **send-receive relation** ($\rightsquigarrow$) is defined by

$$m \rightsquigarrow m' \text{ if } send(m) \dashrightarrow rec(m'),$$

where $\dashrightarrow$ is the happens-before relation defined on events.

The send-receive relation is reflexive but not antisymmetric, nor transitive. But more important, it is consistent with the message ordering as seen by the processes. If a process orders message m before message m', then the messages are ordered, under the send-receive relation, as $m \rightsquigarrow m'$ (which does not mean that $m' \rightsquigarrow m$ is not possible also). Moreover, the send-receive relation is the weakest relation one may naturally define on the set of messages. To make this precise, consider defining on the set of messages the receive-receive relation by

$$m \overset{rr}{\rightsquigarrow} m' \text{ if } rec(m) \dashrightarrow rec(m').$$

Two messages related by the receive-receive relation must be related under the send-receive relation as well since

$$send(m) \dashrightarrow rec(m) \dashrightarrow rec(m').$$

Another interesting property of the send-receive relation is that two messages which are unrelated under that relation must belong to disjoint sets of processes.

The graph representing $(M, \rightsquigarrow)$ is called the **communication graph** . Figure 9 shows a distributed computation with its corresponding communication graph. The communication graph can be seen as a description of a distributed computation at an higher level of abstraction. We say that the communication graph is transitive if the send-receive relation is. The next chapter will show how all of this relates to the detection problem.

Chapter 2

The state lattice decomposition and series-parallel systems

It is well-known that the complexity of the state lattice grows with the increasing of the number of processes and the number of messages. The main goal of this chapter is to highlight the natural structure of the state lattice and to use this knowledge to lower the complexity of the detection problem. We propose a way to decompose the state lattice into smaller and simpler components called concurrent intervals. The first section of this chapter presents the general decomposition theorem of the state lattice. We show that the concurrent intervals have properties that may facilitate the detection of some predicates. The relation between the communication graph and the decomposition into concurrent intervals is also established. The first section ends with some interesting properties of the decomposition.

In the second section, we introduce a special class of distributed computations, the series-parallel systems. A series-parallel system is one that can be constructed using series and parallel compositions of concurrent intervals. This class of distributed computations contains two subclasses, the series systems and the parallel systems. Decomposition theorems are also proved for each of these systems. The content of this chapter forms the basis of the algorithms for detecting monotonic and separable predicates in Chapter 3.

2.1 State lattice decomposition of distributed computations

In this section, we present the theoretical background needed for dealing with the complexity of the state lattice. The main goal is to derive a decomposition of the lattice into simple subsets of states, namely, the concurrent intervals. We first return to our distributed bank example to motivate the content of the section.

Suppose that the president of a distributed bank with three branches A, B and C, wants to know the maximal bank asset on a given day. A procedure to compute that amount would have to last the whole period since the clients are constantly withdrawing, depositing and transferring money from their accounts during the day. The first idea that comes to the president's mind is to ask to each branch clerk to keep a record of the maximal amount of money kept in the client accounts from the beginning of the day. Hopefully, the maximal bank asset would be given by the sum of the branch maxima collected at the end of the day. The president suspects that the procedure does not always work correctly. To convince herself, she looks at the example depicted in Figure 10. In this figure, the event structure has three processes, each representing the fluctuations of the client account balances at one of the branches, and two messages representing the transfer of money from one branch to another. As a result of the procedure above, the clerk at branch A returns

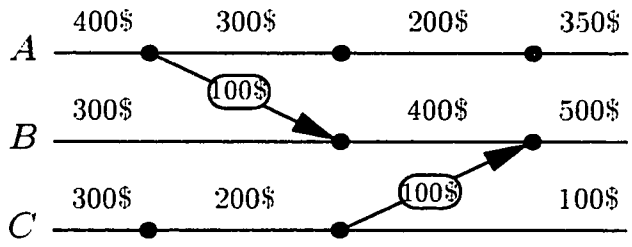

Figure 10: The amount of money in the accounts of the distributed bank

a maximum of 400$, the clerk at branch B returns a maximum of 500$ and the clerk at branch C returns a maximum of 300$. So, the maximal branch asset would be 1200$. After a more careful inspection of the event structure, the president realizes that the state $(A = 400\$, B = 500\$, C = 300\$)$ is inconsistent, that is, at no point in time did the branches own these respective amounts simultaneously. Indeed, the maximum at branch A

is reached before the transfer of 100\$, while the maximum at B is reached after the transfer. She concludes that the procedure would work only if no transfer from a branch to another is done during the day.

The above example is a good illustration of the interplay between the geometrical aspects of the event structure and the ease in detecting certain predicates. Indeed, a procedure similar to the one described above would offer an efficient way to test if the sum $x_1 + x_2 + \cdots + x_n$, is greater than some constant if x_i is local to process i and the distributed computation do not contain any message. Certainly such a distributed computation in which the processes do not communicate may be uninteresting, but a weaker constraint on the distributed communication geometry may be practicable without sacrificing easy detectability. Our approach is to divide a given distributed computation into a set of simpler distributed computations where the processes do not communicate. To formalize this, consider the next definitions.

Let $(P, \leq)$ be a partial order. We say that a set $\mathcal{S}$ of subsets of P **decomposes** P if, for all $x \in P$, there is an element of $\mathcal{S}$ that contains x. $\mathcal{S}$ is then called a **decomposition** of P. Since the state lattice of a distributed computation is a partial order under the reachability relation, it makes sense to talk about its decomposition. We are interested in decomposition into special sets. Let x and y be two elements in P such that $x \leq y$. We define an **interval** $[x, y]$ of P as the subset

$$[x, y] = \{z \in P : x \leq z \text{ and } z \leq y\}.$$

An interval is simply composed of all the elements of P between x and y. On the state lattice, the interval $[\sigma, \tau]$ contains all the states reachable from σ and from which τ is reachable. Even though the intervals are subsets of the state lattice, it is sometimes easier to represent them on the event structure. An interval forms a "window" on the event structure as in Figure 11. The window starts at $[\sigma]$, the opening of the interval represented by the left-most cut and ends at $[\tau]$, the closing of the interval represented by the right-most cut. The states (here we regard a state as a cut rather than a set) entirely included in the window are members of the interval.

Definition 2.1 An interval $I = [\sigma, \tau]$ on a state lattice is said to be **concurrent** if for any message m in M, $\sigma_{rec(m)} \subseteq \tau$ implies that $\sigma_{send(m)} \subseteq \sigma$.

We recall that the notation σ_e denotes the minimal prefix of an event ϵ. There are two geometrical interpretations of concurrent intervals that come naturally. As for intervals, concurrent intervals can be seen as a window of events in the event structure with the additional constraint that the window does not contain both the sending and the receipt of any message. Since the window cannot contain a complete message, there is no dependency between two events of different processes. As a consequence, two local states of two different processes in a concurrent interval must be concurrent. Concurrent intervals somehow generalize the notion of distributed computations with no messages. Figure 11 gives an example of a concurrent interval. The concurrency of the local states in a concurrent interval leads us to the second interpretation. In the state lattice, such a set of local concurrent states form a box or more technically a Boolean algebra (see Appendix A). Figure 14 shows some of the boxes of the given state lattice.

Local intervals, that is, consecutive events in a process, have been introduced in the work of several authors. In Hurfin, Mizuno, Raynal and Shinghal [HMRS95], the idea is pushed further by defining a set of concurrent local intervals. Our definition of (global) concurrent intervals is equivalent. A complete set of concurrent local intervals always forms a concurrent interval in our sense.

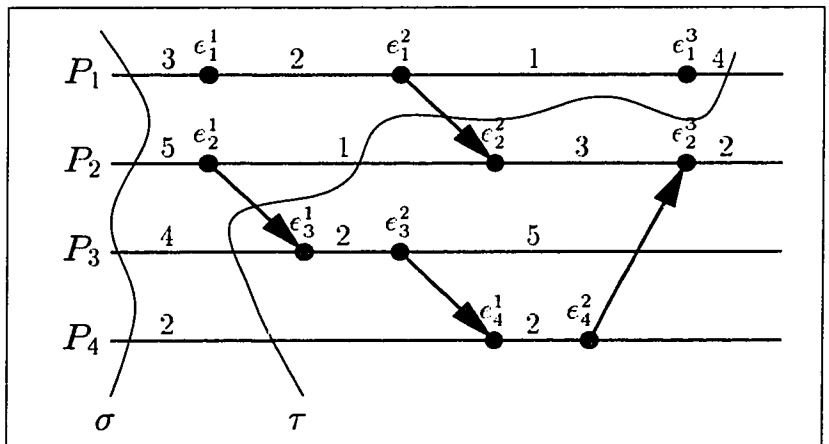

Figure 11: A concurrent interval

Concurrent intervals are an important abstraction since predicates such as

$$\mathfrak{P} = \text{“}x_1 + x_2 + \cdots + x_n > c\text{”}$$

can be easily detected on them, as illustrated in the following three steps:

1. Find the maximal value of x_i in the local interval of P_i;
2. Sum these maxima of the variables;
3. Compare the sum against the constant c.

The predicate is detected if the sum is greater than c.

Example 2.1 Consider the concurrent interval $[\sigma, \tau]$ in Figure 11. We want to know if the predicate

$$\mathfrak{P} = \text{“}x_1 + x_2 + x_3 + x_4 > 13\text{”}$$

evaluates to true on the interval. To do so, we apply the above procedure. We have $max(x_1) = 4$, $max(x_2) = 5$, $max(x_3) = 4$ and $max(x_4) = 2$. Summing the values, we get that

$$max(x_1 + x_2 + x_3 + x_4) = 15.$$

The predicate is therefore detected since $max(x_1 + x_2 + x_3 + x_4) > 13$. In particular, the predicate is true at the state (3 0 0 0). □

A state of a given distributed computation D must be included in at least one concurrent interval on D. In other words, the set of all concurrent intervals forms a decomposition of the state lattice of D. But such an unrestricted decomposition in not interesting; after all a state forms a concurrent interval on its own. A better approach is to consider only a subset of the concurrent intervals to form the decomposition. Such a subset should preferably be as small as possible. Furthermore, the elements of the decomposition should be easily computable. One subset having these properties is the set of predecessor intervals that we now define.

We discussed in the previous chapter the notion of a down-set of a partial order. There is a similar concept in the case of a related set (a set with a relation, see Appendix A). Let R be a relation on a set P. Given $x, y \in P$, we say that x is a **predecessor** of y (or x precedes y) if xRy is in the relation. Consider a subset S of P. The **predecessor set** of S is the set of all predecessors of the elements in S. Formally

$$pred(S) = \{x \in P : xRy \text{ for some } y \in S\}.$$

We also use the notations $pred(x)$ if S contains only the element x and $pred$ when the generator set S exists but is unspecified. It is not hard to see that if the relation is transitive, then a predecessor set is a down-set. Note that $pred(\emptyset) = \emptyset$ and $pred(P) = P$ are always predecessor sets.

Consider a concurrent interval I on a distributed system D. Because of the concurrency property, for any message m in D, either $send(m)$ is excluded from the "window" defined by I or $rec(m)$ is excluded. This suggests the following definition. Let $(M, \rightsquigarrow)$ be the communication graph of a distributed computation $(D, \rightarrow)$. We define

Definition 2.2 The **basic interval** of D associated with the message $m \in M$ is the subset of $\Gamma(D)$ defined by

$$I_m = \Big[\bigcup_{m'' \in pred(m)} \sigma_{send(m'')}, \bigcap_{m'' \in \overline{pred(m)}} \phi_{rec(m'')} \Big],$$

where σ_e and ϕ_e are respectively the minimal prefix and the maximal prefix of the event e as defined in Chapter 1.

Figure 12 gives an example of a basic interval associated with the message m_3. Note that I_{m_3} satisfies the two previously stated conditions. Indeed, the state

$$\Sigma_{pred(m_3)} \equiv \bigcup_{m'' \in pred(m_3)} \sigma_{send(m'')}$$

ensures that any state in I_m is reachable from all the minimal prefixes $\sigma_{send(m'')}$ for some m'' in $pred(m_3)$. On the other hand, the state

$$\Phi_{pred(m_3)} \equiv \bigcap_{m'' \in \overline{pred(m_3)}} \phi_{rec(m'')}$$

ensures that all the maximal prefixes $\phi_{rec(m'')}$ for m'' in $\overline{pred(m_3)}$ are reachable from any state in I_{m_3}.

Example 2.2 The basic interval of Figure 12 can be constructed as follows. First, the predecessor set of m_3 must be computed. We have

$$pred(m_3) = \{m_1, m_2, m_3\}.$$

Therefore, $\overline{pred(m_3)} = \{m_4\}$. Next, we evaluate the state $\Sigma_{pred(m_3)}$ which is simply the union

$$\Sigma_{pred(m_3)} = \sigma_{send(m_1)} \cup \sigma_{send(m_2)} \cup \sigma_{send(m_3)}.$$

We recall that the states have been defined as a set of events, so the union operator is naturally defined. Similarly, the computation of $\Phi_{pred(m_3)}$ gives

$$\Phi_{pred(m_3)} = \phi_{rec(m_4)}.$$

□

The set of basic intervals $\{I_m : m \in M\}$, as their name suggest, serves as a basis to decompose the state lattice. It is interesting to note the symmetry of the definition. The operators $\cup$, *pred* and *send* on the left-hand side are replaced, on the right-hand side, by their duals $\cap$, $\overline{pred}$ and *rec*.

The intervals have also a maximal property that is discussed later. The basic intervals

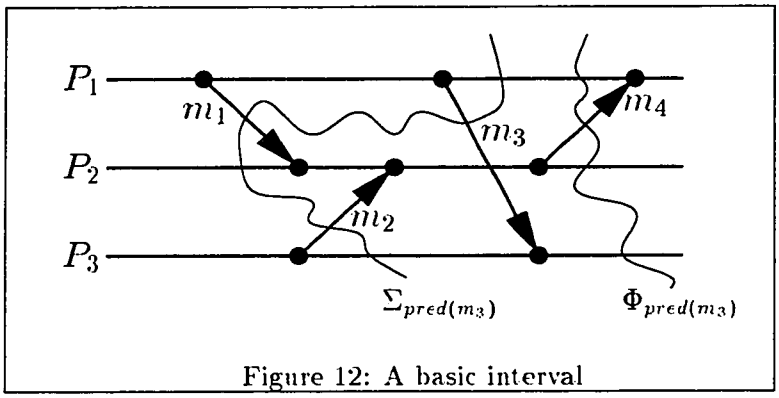

Figure 12: A basic interval

are defined in term of a single message but we can associate an interval to any predecessor set *pred* of $(M, \rightsquigarrow)$ in a similar way.

Definition 2.3 The **predecessor interval** of D associated to a predecessor set *pred* of M is the subset of $\Gamma(D)$ defined by

$$I_{pred} = \Big[\bigcup_{m'' \in pred} \sigma_{send(m'')}, \bigcap_{m'' \in \overline{pred}} \phi_{rec(m'')} \Big].$$

The interval I_{pred} is never empty. Note that the basic interval I_m is equivalent to the predecessor interval $I_{pred(m)}$. We also distinguish one special interval, the **initial interval** I_0, which is defined by

$$I_{pred(\emptyset)} = I_0 = \Big[\sigma_0, \bigcap_{m'' \in M} \phi_{rec(m'')}\Big],$$

where σ_0 is the initial state. Even though an interval on the state lattice can be as complex as the state lattice itself (after all, the whole state lattice is the interval $[\sigma_0, \sigma_f]$), the predecessor intervals have a simple structure. We have

Proposition 2.1 (concurrency) *The predecessor intervals are concurrent.*

Proof. Consider a distributed computation with a set of messages M. Let m be a message in M such that

$$\sigma_{rec(m)} \subseteq \bigcap_{m'' \in \overline{pred}} \phi_{rec(m'')},$$

where *pred* is a predecessor set of M. Then m cannot be in $\overline{pred}$ since otherwise the event $rec(m)$ would not be included in the right-hand side. Therefore, m is in *pred* and

$$\sigma_{send(m)} \subseteq \bigcup_{m'' \in pred} \sigma_{send(m'')}.$$

This proves the result. ■

From the definition of a concurrent interval, it follows that any interval contained in a concurrent interval is also concurrent. On the other hand, it is generally not true that an interval containing a concurrent interval is, itself, concurrent. Indeed, as the next proposition shows, any interval properly containing an interval I_{pred}, for some predecessor set *pred*, is not concurrent. In that sense, the interval I_{pred} is a maximal concurrent interval.

Proposition 2.2 (maximal property) *The predecessor intervals are maximal concurrent intervals.*

Proof. Consider an interval $I = [\sigma, \tau]$ on a distributed computation with a set of messages M. Assume that I contains a predecessor interval I_{pred} for some predecessor set of M. We show that I is not concurrent. We must have

$$\sigma \subseteq \bigcup_{m'' \in pred} \sigma_{send(m'')} \text{ and } \bigcap_{m'' \in \overline{pred}} \phi_{rec(m'')} \subseteq \tau,$$

where one of the inclusions is strict. Suppose that the first inclusion is strict. Then, for some message m in *pred*, there must exist an event e in $\sigma_{send(m)}$ such that e is not in σ. This implies that

$$\sigma_{send(m)} \not\subseteq \sigma$$

but

$$\sigma_{rec(m)} \subseteq \bigcap_{m'' \in \overline{pred}} \phi_{rec(m'')} \subseteq \tau.$$

That is, I is not concurrent. Similarly, we show that if the second inclusion is strict, then I can not be concurrent. ■

The maximal property derives directly from the definition of I_{pred} and the definition of concurrency. To say that, c consider the basic interval I_m in Figure 12. In each process, the opening $\Sigma_{pred(m_3)}$ of the interval is placed just after a send event corresponding to a message in $pred(m_3)$ or at the initial state if no such event exists. Any attempts to extend the window to the left forces the inclusion of a send event whose corresponding receipt is already in the window. Similarly, the closing $\Phi_{pred(m_3)}$ of the interval is placed just before a receipt event corresponding to a message in $\overline{pred(m_3)}$ or at the final state if no other receipt event exists. Again, any attempt to extend the window to the right forces the inclusion of a receipt whose corresponding send event already exists in the window.

The next theorem demonstrates that the set of predecessor intervals decomposes the state lattice. but first, let us prove a lemma. Let $(D, —)$ be a distributed computation with a set of messages M. We have

Lemma 2.3 *Given two predecessor sets pred and pred' of M. If σ is a state of I_{pred} and σ' is a state of $I_{pred'}$, then $\sigma \cup \sigma'$ is in $I_{pred \cup pred'}$.*

Proof. Let S and T be two sets of messages. We first note that the union of the predecessor sets $pred(S)$ and $pred(T)$ is still a predecessor set. Indeed, we have that

$$pred(S) \cup pred(T) \subseteq pred(S \cup T),$$

since each component is included in the right-hand side. Moreover, any m in $pred(S \cup T)$ must be before at least one message of S or T. This shows that

$$pred(S \cup T) \subseteq pred(S) \cup pred(T).$$

We have that

$$\bigcup_{m \in pred} \sigma_{send(m)} \subseteq \sigma \text{ and } \bigcup_{m \in pred'} \sigma_{send(m)} \subseteq \sigma',$$

which gives that

$$\bigcup_{m \in pred \cup pred'} \sigma_{send(m)} \subseteq \sigma \cup \sigma'.$$

We also have that

$$\sigma \subseteq \bigcap_{m \in \overline{pred}} \phi_{rec(m)} \text{ and } \sigma' \subseteq \bigcap_{m \in \overline{pred'}} \phi_{rec(m)}$$

which gives that

$$\sigma \subseteq \bigcap_{m \in \overline{pred \cup pred'}} \phi_{rec(m)}$$

and

$$\sigma' \subseteq \bigcap_{m \in \overline{pred \cup pred'}} \phi_{rec(m)}.$$

Putting everything together we conclude that

$$\bigcup_{m \in pred \cup pred'} \sigma_{send(m)} \subseteq \sigma \cup \sigma' \subseteq \bigcap_{m \in \overline{pred \cup pred'}} \phi_{rec(m)}.$$

■

If we apply Lemma 2.3 to the basic intervals, we get as a corollary that any state σ in $I_{pred(S)}$ can be decomposed into

$$\sigma = \sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_l,$$

where σ_i is in I_{m_i} for some m_i in S. Therefore, the states in the basic intervals generate the states in any interval I_{pred}. In other words, the basic intervals form a basis for the set of predecessor intervals.

Theorem 2.4 (state lattice decomposition) *The set of all predecessor intervals of a distributed computation D decompose the state lattice $\Gamma(D)$.*

Proof. We must show that for any state σ in $\Gamma(D)$ there is a predecessor set $pred$ in M such that σ is in I_{pred}. We first show that for any event e, σ_e is in some interval. Consider the subset $Rec(e)$ of messages in M satisfying $rec(m) \rightarrow e$. Note that $Rec(e)$ is not necessarily a predecessor set of M. The case when $Rec(e) = \emptyset$ is simple. From the definition of the initial interval, we must have that σ_e is in I_0. If $Rec(e)$ is not empty, it must contain a maximal element, that is, a message m_{max} such that for all messages m in $Rec(e)$, we have $rec(m) \rightarrow rec(m_{max})$. The case where e is the receipt of some message is direct, simply take m_{max} to be the message whose receipt is at e. To demonstrate the existence of m_{max}, it is enough to show that for two messages m and m' in $Rec(e)$, there is always a message m'' such that $rec(m) \rightarrow rec(m'')$ and $rec(m') \rightarrow rec(m'')$. Again, we distinguish

among two cases. If $rec(m) \rightarrow rec(m')$ or $rec(m') \rightarrow rec(m)$, then m' and m respectively satisfies the property. Now, suppose that $rec(m)$ and $rec(m')$ are not related under the happens-before relation. Then the receipts of m and m' cannot be in a same process. As a matter of fact, they cannot be both in the process hosting the event e. Suppose that the receipt of m is not in that process. Since $rec(m) \rightarrow e$, there exists a message $\tilde{m}$ such that $rec(m) \rightarrow rec(\tilde{m}) \rightarrow e$ and $rec(\tilde{m})$ is in the same process as e. We similarly demonstrate that there is a message $\hat{m}$ whose receipt is in the same process a e and which satisfies $rec(m') \rightarrow \hat{m} \rightarrow e$. The receipts of $\hat{m}$ and $\tilde{m}$ being in the same process, we must have that either $rec(\hat{m}) \rightarrow rec(\tilde{m})$ or $rec(\tilde{m}) \rightarrow rec(\hat{m})$. In the first case, $\tilde{m}$ is the desired message while $\hat{m}$ is that message in the second case. We conclude that $Rec(e)$ contains a maximal element m_{max} and that $rec(m_{max})$ and e are in the same process as in Figure 13. We claim

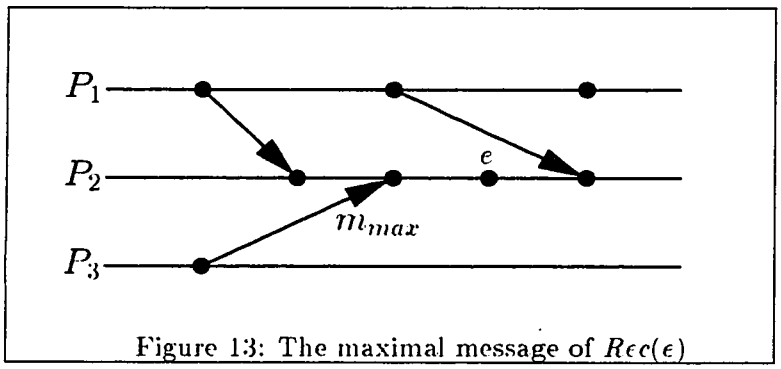

Figure 13: The maximal message of $Rec(e)$

that $I_{m_{max}}$ contains the state σ_e. From the definition of m_{max}, we have that

$$\bigcup_{m \in pred(m_{max})} \sigma_{send(m)} \subseteq \sigma_{rec(m_{max})} \subseteq \sigma_e.$$

It remains to show that σ_e is included in $\phi_{rec(m)}$ for all m in $\overline{pred(m_{max})}$. Suppose that there is an event e' in σ_e such that $rec(m) \rightarrow e'$ for some message m. Then m is in $Rec(e)$ since $rec(m) \rightarrow e' \rightarrow e$. But m_{max} is the maximal element of $Rec(e)$, we must have that m is in $pred(m_{max})$ since $send(m) \rightarrow rec(m) \rightarrow rec(m_{max})$. This shows that if m is in $\overline{pred(m_{max})}$ then $rec(m) \not\rightarrow e'$ for all e' in σ_e. We conclude that σ_e is included in $\phi_{rec(m)}$ for all m in $\overline{pred(m_{max})}$.

Any state can be written as a union of σ_e. Indeed, given a state σ in $\Gamma(D)$, we can

decompose σ as

$$\sigma = \bigcup_{e \in \sigma} \sigma_e.$$

From the first part of the proof, we know that each of the component σ_e belongs to one basic interval and from Lemma 2.3, we know that the finite union also belongs to an interval I_{pred} for some predecessor set *pred*. ■

It is not hard to see that a predecessor set on a transitively related set can always be generated by an anti-chain. We have this useful corollary.

Corollary 2.5 (anti-chain decomposition) *Consider a distributed computation D with a transitive communication graph, then, for any state σ, there exists an anti-chain of messages A such that*

$$\sigma \in I_{pred(A)}.$$

Proof. This result follows immediately from the previous theorem and the remark above. ■

Corollary 2.5 simply says that, whenever the send-receive relation on M is transitive, the set of intervals $\{I_{pred(A)} : A \text{ an anti-chain of } M\}$ decomposes the state lattice. The next example computes the state lattice decomposition for the event structure in Figure 14.

Example 2.3 The simplicity of the event structure allows us to construct directly the predecessor sets of the communication graph. We have

$$\begin{aligned} pred(m_1) &= \{m_1\}; \\ pred(m_2) &= \{m_1, m_2\}; \\ pred(m_3) &= \{m_1, m_2, m_3\}. \end{aligned}$$

The readers can convince themselves that these are the only non-empty predecessor sets of the communication graph. From Theorem 2.4, we know that I_0, $I_{pred(m_1)}$, $I_{pred(m_2)}$ and $I_{pred(m_3)}$ form a decomposition of the state lattice corresponding to the event structure. Figure 14 shows the state lattice and its decomposition. To determine the states include in the predecessor intervals, we simply apply the definition. As an example, consider $I_{pred(m_1)}$. The interval opens at $\Sigma_{pred(m_1)} = \sigma_{send(m_1)}$ which corresponds to the state (0 1 0) in the lattice and closes at $\Phi_{pred(m_1)} = \phi_{rec(m_2)} \cap \phi_{rec(m_3)}$ which corresponds to the state (3 2 0). All the states included between (0 1 0) and (3 2 0) are part of the interval. □

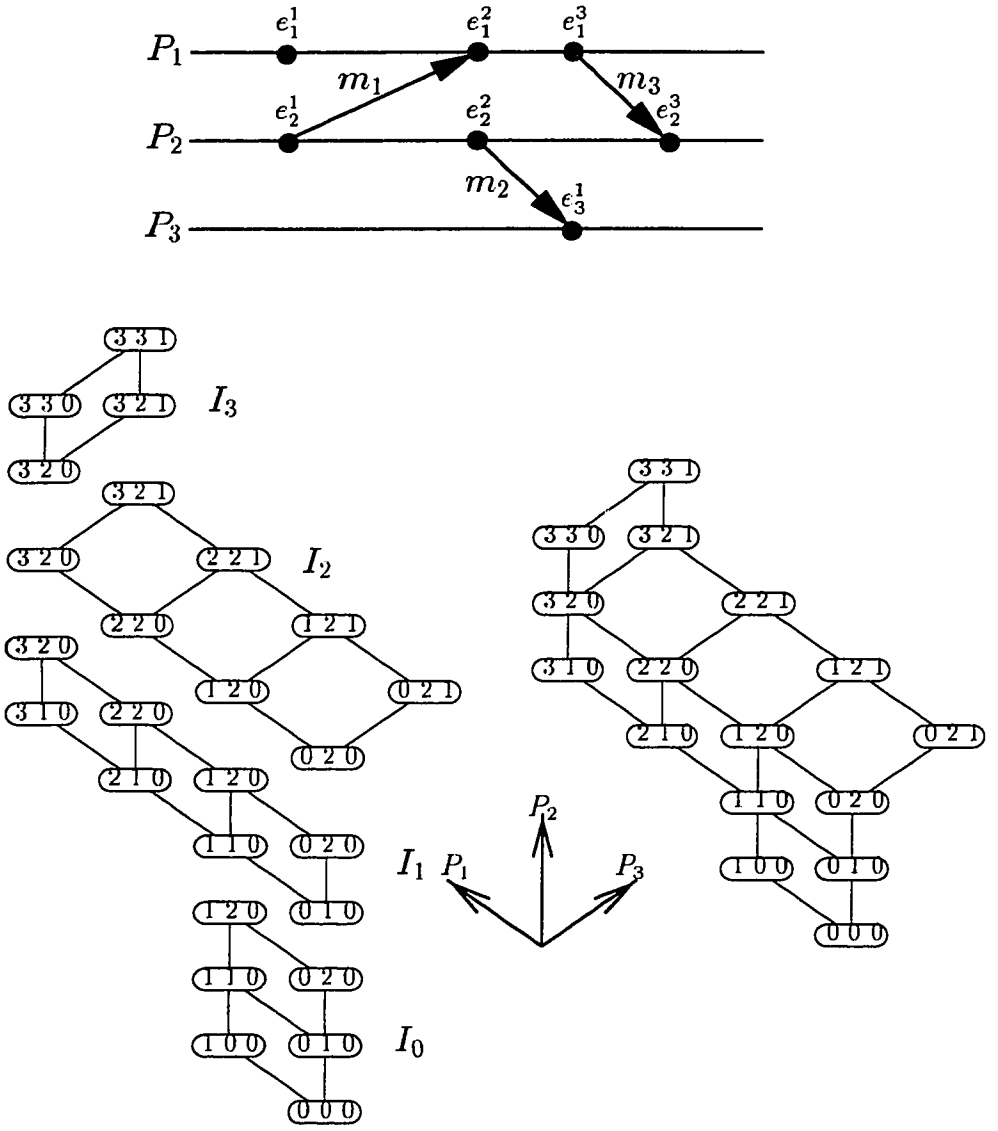

Figure 14: The decomposition of the state lattice into predecessor intervals

Theorem 2.4 and Corollary 2.5 suggest also that the complexity of the decomposition and, in particular, the number of intervals it contains, are determined by the number of predecessor sets in the communication graph.

Theorem 2.6 (minimal decomposition) *On a distributed computation with a transitive communication graph, the decomposition of the state lattice given by the predecessor intervals is minimal, that is, any other decomposition of the state lattice into concurrent intervals contains at least the same number of intervals.*

Proof. Consider the state

$$R_{pred} = \bigcup_{m'' \in pred} \sigma_{rec(m'')}.$$

There is one such state for each predecessor set. We show that if a concurrent interval contains the states R_{pred} and $R_{pred'}$ for two predecessor sets $pred$ and $pred'$, then $pred = pred'$. Let $[\sigma, \tau]$ be a concurrent interval that contains R_{pred} and $R_{pred'}$. For all m in $pred$ we have,

$$\sigma_{rec(m)} \subseteq \tau.$$

Then, from the concurrency property, we must have

$$\sigma_{send(m)} \subseteq \sigma \subseteq R_{pred'}.$$

This shows that there must be one message m' in $pred'$ such that $send(m)$ happens-before $rec(m')$. We conclude from the transitivity of the communication graph that m is in $pred'$ and that $pred$ is a subset of $pred'$. Similarly, we show that $pred'$ is a subset of $pred$. This confirms the results. ■

It comes as a surprise how much information the communication graph gives on the complexity of the state lattice decomposition. A consequence of Theorem 2.6 is that the minimal number of concurrent interval needed to cover the state lattice of a distributed computation with a transitive communication graph is equal to the number of predecessor sets in the communication graph. It is important to note here that there exist distributed computations for which the minimal number of concurrent intervals in the decomposition is exponential in the number of messages. Such a distributed computation is shown in the next example.

Example 2.4 By Theorem 2.6, it is enough to find a distributed computation for which the number of predecessor sets of the communication graph grows exponentially with the number of messages. Consider the event structure containing k messages organized as in Figure 15. None of the messages in the communication graph are related under the send-receive relation. Therefore, any subset of messages forms a different predecessor set. A decomposition into concurrent intervals of the corresponding state lattice has at least 2^k intervals. □

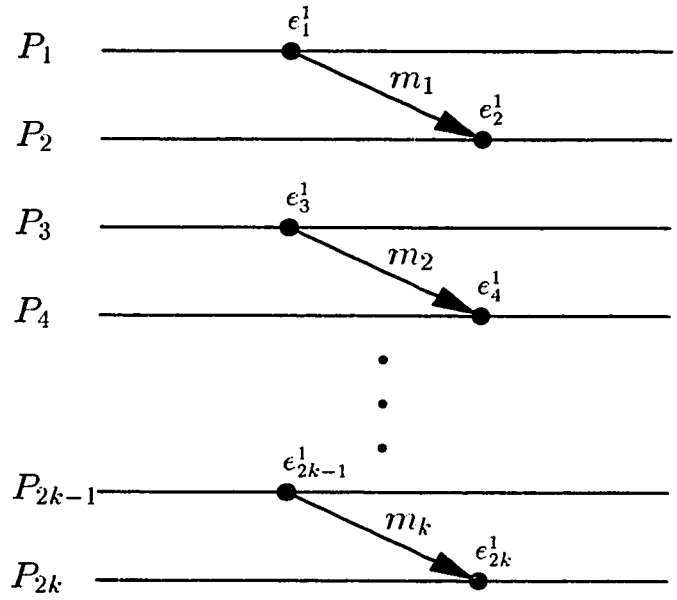

Figure 15: An event structure with an exponential number of intervals

The relation between the communication graph and the number of concurrent intervals in the state lattice decomposition is captured by the following rule of thumb.

Rule of Thumb: the more connected is the communication graph the smaller is the number of concurrent intervals needed to decompose the state lattice.

One may wonder why the decomposition of the state lattice can facilitate predicate detection. The answer is two-folds. First, there are predicates that can be easily detected on

concurrent intervals. The predicate and the procedure discussed in Section 2.1 has demonstrated this. In that case, the predicate needs to be evaluated only on a single state of the interval. Needless to say that this represents great savings in time. Second, the decomposition gives valuable information about the complexity of the state lattice. Two state lattices of the same size may exhibit properties that are quite different once they are decomposed into concurrent intervals. The next section contains more on the subject.

2.2 Series-parallel systems

The series-parallel property in graphs is studied extensively in electrical engineering. Since its appearance in computer sciences, it has found applications in the scheduling problem ([Du65]) and many graph related problems ([TNS82]). Many NP-complete problems become solvable in polynomial time if they are restricted to series-parallel graphs. Because of their importance, the recognition of series-parallel graphs has become a classical problem for the design of algorithms ([VTL82]).

In this section, we introduce a special class of distributed computations, the series-parallel systems. A series-parallel system is one that can be constructed using series and parallel compositions of concurrent intervals. This class of distributed computations contains two subclasses, the series systems and the parallel systems that we present as well. Decomposition theorems are also proved for each of these classes.

2.2.1 Series systems

We say that a communication graph is **serialized** if its messages can be ordered in such a way that, if m appears before m' in the sequence, then $m \rightsquigarrow m'$. A **series system** is a distributed computation with a serialized set of messages. The event structure in Figure 14 is an example of a series system. We use the notation

$$m_1; m_2; \cdots; m_k$$

to express a serialized set of k messages. Series systems are special cases of the series-parallel systems defined later, but they have their own interest as shown by the next example.

Example 2.5 In the client-server paradigm, there is a set of processes called the clients, conversing with one central process, the server. Any communication between two clients

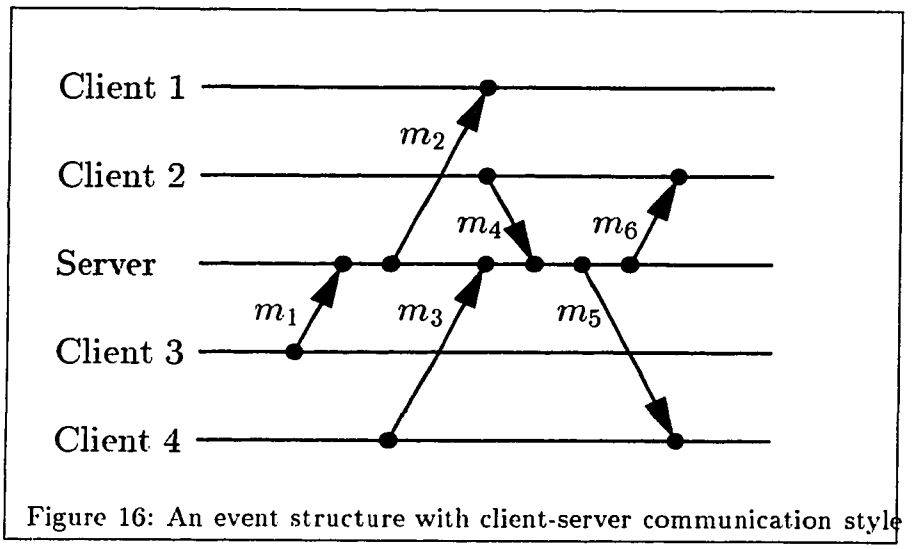

Figure 16: An event structure with client-server communication style

must be made through the server. Therefore, there is no message going from one client directly to another. The messages must go through the server first. Under these assumptions, all the messages must have an event (sending or receipt) on the server process. The natural ordering of these events ensures that the messages are serialized. Indeed, the corresponding sequence of messages satisfies the series property. Figure 16 gives an example with four clients. The messages are ordered as $m_1; m_2; m_3; m_4; m_5; m_6$ which correspond to the ordering of their respective events in the server process. □

Another interesting property of series systems is that their state lattices are easily decomposed into a set of concurrent intervals. Consider a special case where the series system has a transitive communication graph. An immediate consequence of Corollary 2.5 is that the basic intervals of a distributed computation with these properties form a decomposition of the state lattice. Indeed, a transitive and serialized set of messages is a total order (that may also have symmetric relations). That implies that an anti-chain on that set contains at most one element. This gives us an easy way to decompose the state lattice into concurrent intervals. This decomposition can be generalized to any series system as shown in the next proposition.

Proposition 2.7 *On a series system, the basic intervals decompose the state lattice.*

Proof. We show that given a state σ there is a basic interval containing it. Let $Rec(\sigma)$ be the subset of M containing the messages m if and only if $\sigma_{rec(m)} \subseteq \sigma$. Since the messages are serialized, there must be an element m_{max} of $Rec(\sigma)$ such that $m \rightsquigarrow m_{max}$ for all m in $Rec(\sigma)$. In order to show that σ is in $I_{m_{max}}$, we first remark that

$$\Sigma_{pred(m_{max})} \subseteq \sigma_{rec(m_{max})} \subseteq \sigma.$$

On the other hand, we also have that

$$\sigma \subseteq \phi_{rec(m'')}$$

for all m'' not in $pred(m_{max})$ and thus

$$\sigma \subseteq \bigcap_{m'' \in \overline{pred(m_{max})}} \phi_{rec(m'')} = \Phi_{pred(m_{max})}.$$

■

For an example of the state lattice decomposition of a series system, we refer the reader to Figure 14. It should be easy to verify that the messages are serialized as $m_1; m_2; m_3$ and that the four intervals are the basic intervals of the systems.

The next proposition gives a characterization of series systems with three processes. The result states that these series systems correspond to event structures that do not contain a 3-crown. A 3-crown is a graph with six vertices as given by the events $\{\epsilon_i^j : i, j = 1, 2, 3\}$ in Figure 17. So, the set of vertices contains three sources and three sinks. The sources precede exactly two sinks. The crown-free property appears often in graph and partial order theory. As shown in [AJF96], it seems to be particularly important to achieve consistent event ordering in distributed environments.

We have this useful characterization of series systems with three (3) processes.

Proposition 2.8 *A distributed computation with three processes is a series system if and only if the corresponding event structure does not contain a subgraph isomorphic to a 3-crown.*

Proof. Let $(D, —)$ admit a 3-crown subgraph. The presence of a 3-crown implies the existence of three messages organized as in Figure 17. It is clear that none of the messages in the computation is preceded by the other messages under the send-receive relation. This implies that the messages are not serialized.

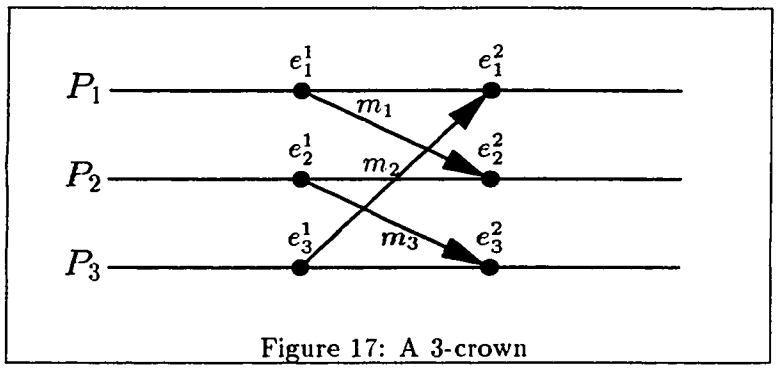

Figure 17: A 3-crown

Assume that M is not serialized. We claim that $(D, \rightarrow)$ must contain a 3-crown. Let m_1, m_2 and m_3 be a set of messages which are not serialized. We show that the set of events $\{send(m_1), send(m_2), send(m_3), rec(m_1), rec(m_2), rec(m_3)\}$ forms a 3-crown. We first mention that two messages on a distributed computation with three processes are always related under the send-receive relation. Indeed, from the four communication events of these messages, two must be on the same process. These two events ensure that the messages are related. Since, the messages are not serialized none of the messages is preceded by the two other message. We conclude that the messages must be ordered in a cycle of the form $m_1 \rightsquigarrow m_2 \rightsquigarrow m_3 \rightsquigarrow m_1$, which represents a 3-crown. ■

This characterization theorem can be used by an algorithm to decide if a distributed computation with three processes is a series system. We do not present this algorithm, but we give in Chapter 3 a general procedure to recognize the series-parallel property and therefore the series property. We turn now to the presentation of parallel systems.

2.2.2 Parallel systems

A communication graph is **parallel** if it contains no pair of messages related under the send-receive relation. A **parallel system** is a distributed computation where the set of messages is parallel. Figure 15 gives an example of a parallel system. Parallel systems do not resemble series systems. The latter have strongly connected communication graph while the formers have fully disconnected communication graph. Parallel systems have little interest on their own since the communication is limited to one message per pair of processes.

Parallel systems are a good example of disconnected distributed computation. We have

Definition 2.4 A distributed computation is **connected** if its processes $\{P_1, P_2, \ldots, P_n\}$ cannot be divided into two subsets D_1 and D_2 such that

1. $D_1 \cup D_2 = \{P_1, P_2, \ldots, P_n\}$ and,
2. the processes in D_1 do not communicate with the processes in D_2.

Any distributed computation can be written as a union of connected distributed computations. These are called the **connected components** of the distributed computation. Figure 18 shows a decomposition of a distributed computation into its connected components.

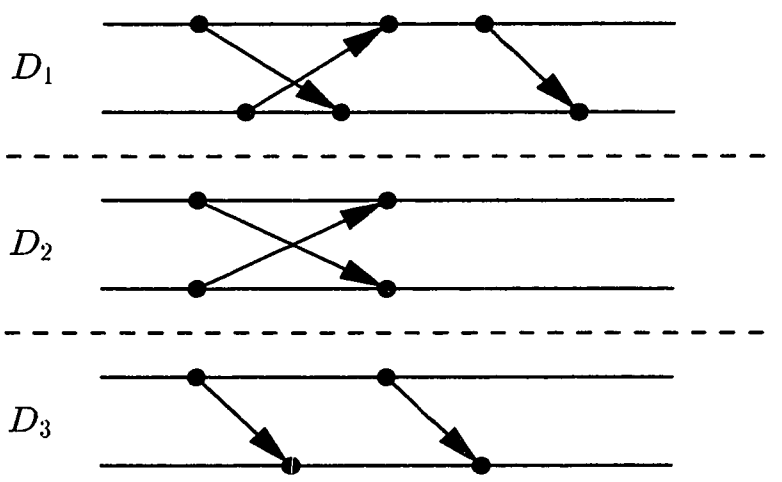

Figure 18: The connected components of a distributed computation

A connected component is itself a distributed computation. Therefore, it makes sense to talk of its state lattice. The states of a distributed computation are related to the states of its connected components as shown by the next proposition.

Theorem 2.9 (parallel decomposition) *Consider $D_1, D_2, \ldots, D_n$ the connected components of a distributed computation D. A state in D is uniquely decomposed as*

$$\sigma = \sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n,$$

where σ_i is a state in $\Gamma(D_i)$.

Proof. Since D_i is a subset of the event in D, we have that $\sigma = \sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n$ is a subset of D. We first show that σ is a state of D (a down-set). Given e, an event in σ, this event must belong to one connected component, say D_1. We prove that if $e' \rightarrow e$ then e' is in σ. Suppose e' does not belong to any process in the connected component D_1. Then there must exist a message between e and e'. This message links one process in D_1 with one process in the connected component containing e'. This is impossible since two different connected components are disconnected. Therefore e' must be in the connected component D_1. Since e is in σ_1 and that σ_1 is a state of D_1, σ_1 must contain the event e'. This proves the first part.

Now, we show that any state σ in $\Gamma(D)$ can be decomposed into $\sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n$ where σ_i is in the component D_i. Consider the set $\sigma_i = \sigma \cap D_i$. We show that σ_i is a state of D_i. Let e be an event of σ_i. Then, any e' in D such that $e' \rightarrow e$ is in σ. This shows that any e' in D_i such that $e' \rightarrow e$ is in σ_i. ■

From the theorem, we know how to generate a state of a distributed computation D from the states of its connected components D_i. But can we generate the concurrent intervals of D from the concurrent intervals of D_i? The answer is yes. A proof of that fact follows the proof of Theorem 2.9 with the addition that, we must also show the concurrency of the generated interval. Let $I(i)_j$ be j^{th} concurrent interval on the connected component D_i. We define an interval on D by

$$I(j_1, j_2, \ldots, j_n) = \{\sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_n : \sigma_i \in I(i)_{j_i}\}.$$

One can show that the concurrency of the intervals $I(i)_j$ implies the concurrency of the interval $I(j_1, j_2, \ldots, j_n)$. This result is important since it allows us to generate the complex intervals of a distributed computation from the simpler intervals of its connected components.

Example 2.6 To illustrate the result, we return to the event structure of Figure 15. We mentioned in Example 2.4 that the number of concurrent intervals needed to cover the state lattice was growing exponentially with respect to the number of messages. However, the distributed computation decomposes into simple connected components, each of them containing only one message. From Proposition 2.7, we know that the decomposition of the state lattice of one of the connected component contains only two concurrent intervals. A

simple computation gives a total of $2 \times k$ concurrent intervals that contain all the states in the connected components. Here is where things become interesting. The number of concurrent intervals needed to cover the state lattice of the example distributed computation is 2^k. In other words, 2^k intervals can be generated from $2k$ simple intervals. Needless to say, we gain a lot dealing with the intervals of the connected components separately. □

The use of this observation in predicate detection will be presented in Chapter 3.

2.2.3 Series-parallel systems

We are now ready to define series-parallel systems.

Definitions 2.5 A communication graph M is **series-parallel** if:

1. it contains a unique message;
2. there are two disjoint series-parallel sets of messages M_1 and M_2 such that
   (a) $M_1 \cup M_2 = M$ and for all m in M_2, $M_1 \subseteq pred(m)$ (**series composition**);
   (b) $M_1 \cup M_2 = M$ and all pairs of messages in $M_1 \times M_2$ are unrelated (**parallel composition**).

A **series-parallel system** is a distributed computation with a series-parallel set of messages. The systems presented previously arise as special cases of the series-parallel systems. If we apply only series composition, we get a series system. On the other hand, if we apply only the parallel composition, we get a parallel system.

The series-parallel systems, as defined above, form a broad class of distributed computations. The following is an example series-parallel system.

Example 2.7 An application of series-parallel systems surfaces in hierarchical communication. Consider a set of processors ordered in a tree structure. In some applications, a process must wait for the messages from the processes above it in the tree before communicating with others below it. The resulting communication graph is series-parallel. In particular, this has applications in hierarchical database and multilevel security. □

Series decomposition is written as $M = M_1 \uplus M_2 \uplus \cdots \uplus M_k$, while parallel decomposition is written as $M = M_1 \biguplus M_2 \biguplus \cdots \biguplus M_k$. The operators $\uplus$ and $\biguplus$ are called respectively **series composition** and **parallel composition.** The operator $\biguplus$ is commutative while $\uplus$ is not. Series and parallel decompositions of the communication graph generate similar decomposition of the event structure and the state lattice. Consider a distributed computation D whose communication graph M is series decomposed as

$$M = M_1 \uplus M_2 \uplus \cdots \uplus M_k.$$

The **series components** of D are intervals defined by

$$I_{M_i} = \left[\bigcap_{m'' \in M_i} \Sigma_{pred(m'')}, \bigcup_{m'' \in M_i} \Phi_{pred(m'')} \right].$$

We have

Theorem 2.10 (series decomposition) *The state lattice of a series system $(D, \rightarrow)$ with communication graph decomposition*

$$M = M_0 \uplus M_1 \uplus \cdots \uplus M_k$$

can be decomposed into

$$\Gamma(D) = I_{M_0} \cup I_{M_1} \cup \cdots \cup I_{M_k}.$$

Proof. We must show that any state in the state lattice can be found in one of the set I_{M_i}. We already know that the set of intervals I_{pred} decompose the state lattice. It is therefore enough to show that the state in any interval I_{pred} are included in at least one I_{M_i}. We first show that if $pred \cap M_i \neq \emptyset$ then $pred(M_{i-1}) \subseteq pred$. Indeed, since all the messages in M_i are preceded by the elements of M_{i-1}, we have that $pred(M_{i-1}) \subseteq pred(pred \cap M_i) \subseteq pred$. Let M_i be the set with largest i such that $pred \cap M_i \neq \emptyset$. Then $I_{pred} \subseteq I_{M_i}$. Indeed

$$\bigcap_{m'' \in M_i} \Sigma_{pred(m'')} \subseteq \bigcap_{m'' \in M_i \cap pred} \Sigma_{pred(m'')} \subseteq \Sigma_{pred}$$

Similarly,

$$\Phi_{pred} \subseteq \bigcup_{m'' \in M_i \cap pred} \Phi_{pred(m'')} \subseteq \bigcup_{m'' \in M_i} \Phi_{pred(m'')}.$$

■

Theorem 2.10 generalizes the result of Proposition 2.7. The series components of a distributed systems can be seen also as distributed computations. Indeed, the window defined by the intervals I_{M_i} are themselves distributed computations. A series decomposition of a series-parallel system may get rather complicated. To simplify the next example, we represent the concurrent intervals as subsets of the event structure rather then on the state lattice.

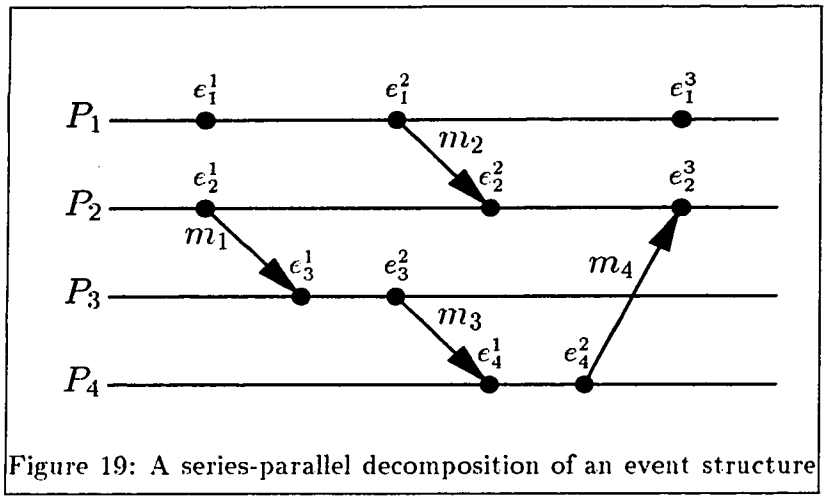

Figure 19: A series-parallel decomposition of an event structure

Example 2.8 Consider the event structure of Figure 19. The series decomposition contains four different intervals:

$$I_0 = \begin{bmatrix} 0-3 \\ 0-1 \\ 0-0 \\ 0-0 \end{bmatrix} \qquad I_{m_1} = \begin{bmatrix} 0-3 \\ 1-1 \\ 0-2 \\ 0-0 \end{bmatrix}$$

$$I_{\{m_2,m_3\}} = \begin{bmatrix} 0-3 \\ 1-2 \\ 0-2 \\ 0-2 \end{bmatrix} \qquad I_{m_4} = \begin{bmatrix} 2-3 \\ 1-3 \\ 2-2 \\ 2-2 \end{bmatrix}$$

where the first column represents the cut opening the interval, while the second column represents the cut closing the interval. The reader may notice that the interval $I_{\{m_2,m_3\}}$ completely contains I_{m_1}. This is not a coincidence as we can show that this happens any

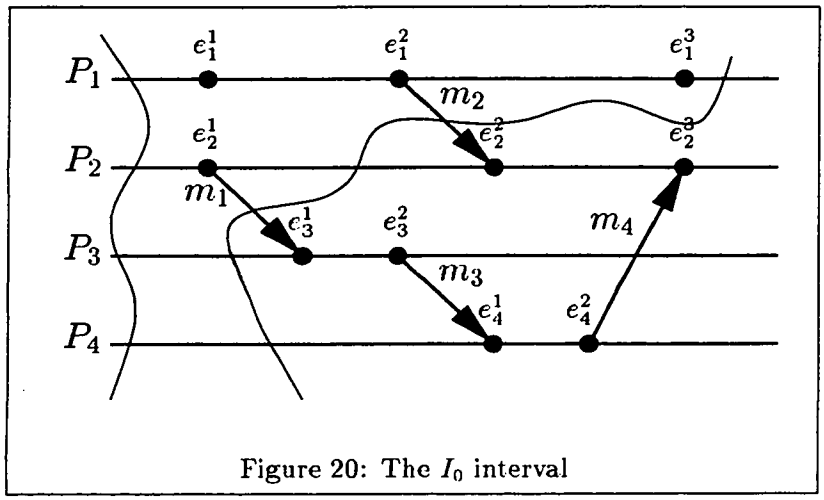

Figure 20: The I_0 interval

time the communication graph contains a parallel set of messages with more than one elements. This set of intervals covers the state lattice. It can be seen from the following figures that the intervals I_0 and I_{m_4} are concurrent while $I_{\{m_2,m_3\}}$ is obviously not. □

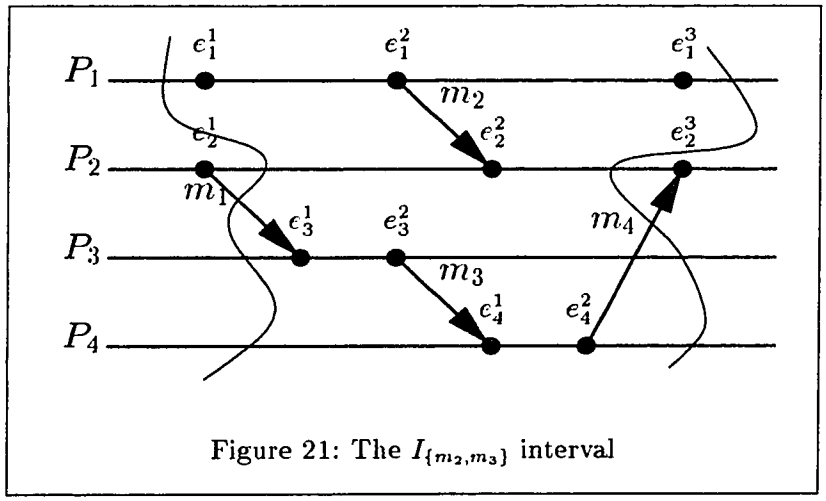

Figure 21: The $I_{\{m_2,m_3\}}$ interval

Recognition of series-parallel graphs is a classic problem and efficient sequential and parallel procedures are known. These procedures are of importance for us because our detection algorithms work only on series-parallel systems. Therefore, the detection algorithms should be associated with a recognition procedure to verify the validity of the conclusion. The presentation of any of these recognition procedures is long and we prefer referring the

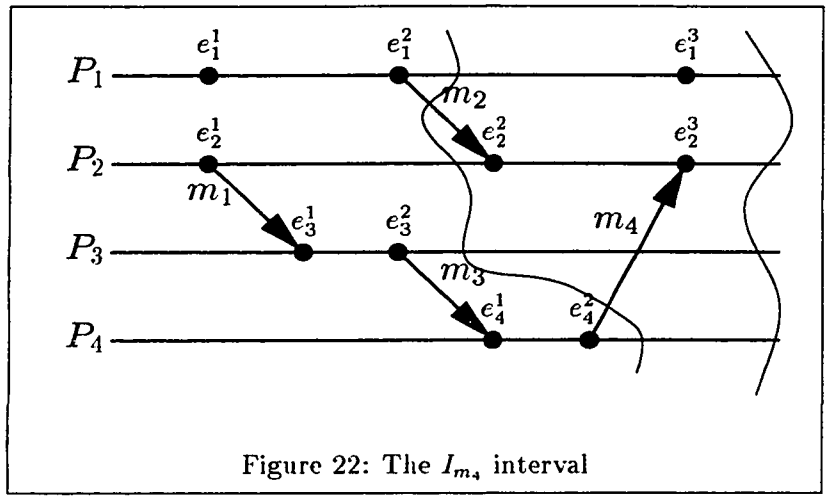

Figure 22: The I_{m_4} interval

reader to the relevant papers. For a general procedure to recognize series-parallel directed graphs, see [VTL82]. Parallel recognition algorithms are found in [Ep92] and [HY87]. A word of caution is needed here. The recognition algorithms mentioned above are generally designed for acyclic directed graphs. For us, the recognition is slightly harder because of the presence of cycles in the communication graph. For the sake of completeness, we briefly describe a way to reduce the recognition problem on a communication graph to one on acyclic directed graphs.

Given a related set S, an antisymmetric reduction of S is an antisymmetric related set A such that the symmetric closures of S and A are equal. All related sets have an antisymmetric reduction, but not all of them have a transitive antisymmetric reduction. We have

Proposition 2.11 *If a communication graph is series-parallel then it has a transitive antisymmetric reduction.*

Proof. It is enough to show that series communication graphs and parallel communication graphs have a transitive antisymmetric reduction. That can be seen easily from the definitions. ■

Let S be a related set represented by a directed graph $G(S)$. We want to know if there is a transitive antisymmetric reduction of S. This is equivalent to the problem of finding

an acyclic graph which has the same transitive closure as $G(S)$. To find a transitive antisymmetric reduction of a directed graph, we apply the reduction rules shown in Figure 23. The reduction rules are applied repetitively on the graph until none of them can be applied.

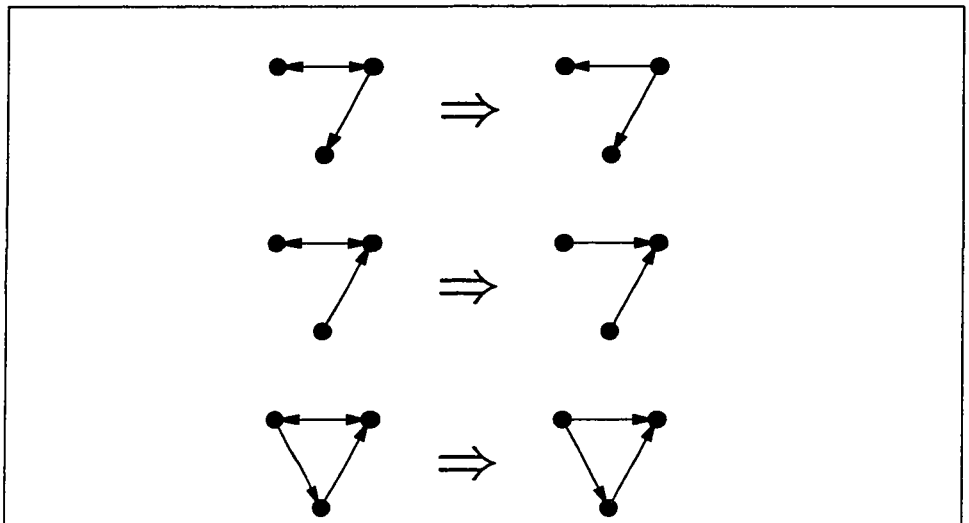

Figure 23: The reduction rules to get an acyclic communication graph

At that point, we may still be left with some cycles. It can be shown by analyzing all subgraphs with three vertices that an arbitrary choice of orientation of one edge will lead to a transitive antisymmetric reduction if it exists. Whence the choice is done, we applied again the reduction rules to the new graph. The next procedure computes a transitive antisymmetric reduction if it exists. Let G be a directed graph and let v and v' be vertices in G.

1. Using the two rules of Figure 23 remove edge of G until none of the rules can be applied;
2. If it remains two edges (v, v') and (v', v), then choose one of them arbitrarily and return to step (1);
3. If no more edge can be removed then verify if the resulting graph is transitive;
4. If the resulting graph is transitive then it is a transitive antisymmetric reduction of G, otherwise there exists no transitive antisymmetric reduction of G.

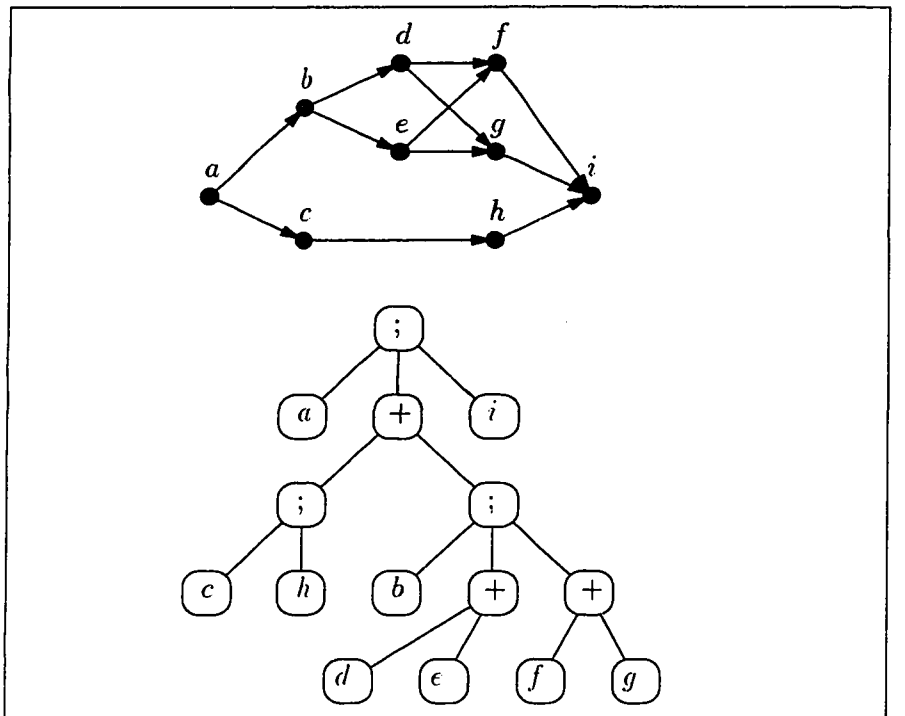

Figure 24: The decomposition tree of a series-parallel directed graph

An interesting feature of the recognition procedure given in [VTL82] is that its produces a tree representation of the directed graph. Indeed, a series-parallel directed graph can be represented in a natural way by a binary tree as shown in Figure 24. This tree is constructed by 1) associating a tree of one node with the series-parallel directed graph having one vertex and no edges, and 2) using the two rules in Figure 25 to recursively build larger trees from smaller ones. Decomposition trees will be useful later in the presentation of the detection algorithms.

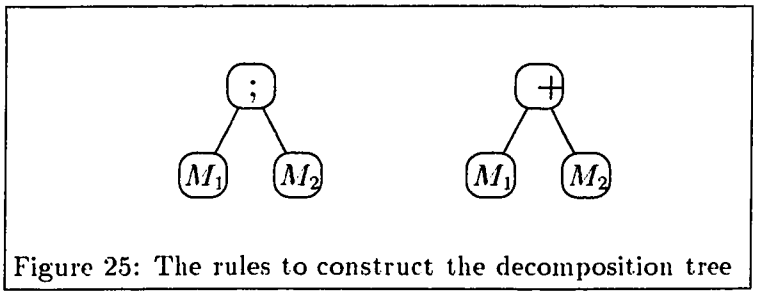

Figure 25: The rules to construct the decomposition tree

Chapter 3

Detection algorithms for monotonic predicates and separable predicates

The exponential growth of the state lattice prevents us from visiting every state when addressing the predicate detection problem. Unfortunately, this is only one of the sources of complexity. As it was shown, for some simple classes of predicates such as the stable predicates and the conjunctive predicates, the detection problem can be solved rather easily. This suggests that the predicate itself affects the complexity of the detection. It appears that some properties of the predicates make them easier to detect. Finding the necessary properties for easy detection is again far from being easy. This is why researchers have considered mainly some special properties. A contribution of this chapter is to characterize two new predicate properties, monotonicity and separability, and to give the corresponding detection algorithms. Monotonic and separable predicates, as we call them, form classes general enough to include the conjunctive predicates. Because of the generalization so introduced, the easy detection is in turn addressed only for a subclass of distributed computations.

In the first section of this chapter, we introduce the tools needed for the detection. Among them, we discuss the vector clock and the construction of concurrent intervals. The second section defines monotonic and separable predicates. For each of these classes, we present a detection algorithm on simple distributed computations and prove its correctness.

The main result of this thesis is presented in the third section. There, we describe an algorithm to detect separable predicates on series-parallel systems.

3.1 Vector clock and basic intervals construction

The ordering of events in a distributed environment is rendered more difficult by the absence of a common clock. In order to verify the consistency of a set of process states, we need a mechanism to capture the partial ordering of events. These mechanisms are called logical clocks since they implement in software the functionality of a real clock. Different logical clocks have been proposed in the literature. The most common ones are the Lamport's logical clock [La85] and the Mattern-Fidge vector clock [Ma88, Fi91]. The aim of these clocks is to provide a mean to order the events at different processes. For our purpose, a **logical clock** on a distributed computation D is a mapping

$$C : D \longrightarrow P,$$

where P is a partial order (typically $\mathbb{N}^j$ for some integer $j \geq 1$). In the following, we present the Lamport clock and the vector clock, and we show how to use the vector clock mechanism to construct the basic intervals.

The Lamport logical clock is maintained distributively. A process implements a **Lamport clock** as an integer-valued counter which is incremented with the following rules:

1. the counter is incremented by one each time the process executes an internal event;
2. before sending a message, a process increases by one the current value of the counter and sends the new value along with the message as a time stamp;
3. upon reception of a message, the process sets the counter to the maximum of its current value and the time stamp attached to the message, plus one.

The clock value associated to an event in process P_i is simply the counter value of P_i right after it has executed that event. Suppose that the counters are initialized to zero before the computation starts. Then, the clock value at an event ϵ is the size of the longest chain of predecessors of ϵ. The advantage of Lamport clock is the low cost in space. Only one integer variable is needed per process and the size of the information appended to the messages is

fixed regardless of the number of processes involved in the computation. Unfortunately, the clock is not accurate enough to characterize the happens-before relation. Indeed, it may happen that, for an external observer, an event with a larger clock value occurs before an event with a smaller clock value. As an example, consider the event structure of Figure 26 where each node is labeled with its Lamport clock value. Let $C(\epsilon)$ denotes the Lamport clock value at event e. On one hand, it can be shown that for any successor event e' of e, the value of the Lamport clock at e' is larger than its value at e. More formally, we have that

$$e \rightarrow e' \Rightarrow C(\epsilon) < C(\epsilon'),$$

where $\rightarrow$ is the happens-before relation and $<$ is the total order on integers. On the other hand, if it is given that $C(\epsilon') > C(\epsilon)$, we cannot conclude that ϵ' is a successor of ϵ. Indeed, consider $e' = \epsilon_3^3$ and $\epsilon = e_2^1$ in the same figure. We have that $C(\epsilon') = 4 > 1 = C(\epsilon)$ but ϵ_3^3 is not a successor of e_2^1. This shows that the Lamport clock does not completely characterize

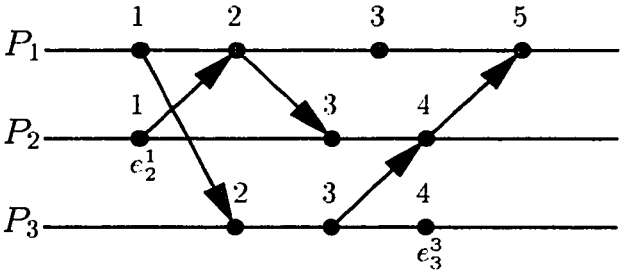

Figure 26: Lamport clock in action

the happens-before relation.

The vector clock solves this problem. To each event in a distributed computation with n processes, we associate a vector of n integers. Let V_i be the i^{th} coordinate of a vector V. In order to make the set of n-vectors a partial order, we say that a vector V is smaller than a vector V' if $V_i \leq V_i'$ for all i, where the inequality is strict for at least one coordinate. Moreover, we can define the maximum of two n-vectors $V' \vee V$ to be the component-wise maximum of their coordinates, that is

$$(V \vee V')_i = max\{V_i, V_i'\}.$$

To implement the **vector clock** in a distributed system with n processes, each process

maintains a vector of n integer variables V. Again, the process vectors are updated distributively when a process changes its state or executes a communication event. Initially, the entries of the process vectors are set to zero. Then, the i^{th} process updates the vector with the following rules

1. V_i is incremented by one each time the process i executes an internal event;
2. before sending a message, process i increases V_i by one and sends the new vector along with the message as a time stamp;
3. upon receipt of a message, process i increases V_i by one and set the vector to the component-wise maximum of the current vector and the time stamp vector received with the message.

Denoted by $V(e)$ the vector clock value of an event e in process i. $V(e)$ is the value of the n-vector in process i just after it executes the event e. Figure 27 gives the vector clock values for the previous example. The advantage of the vector clock is its accuracy. An event

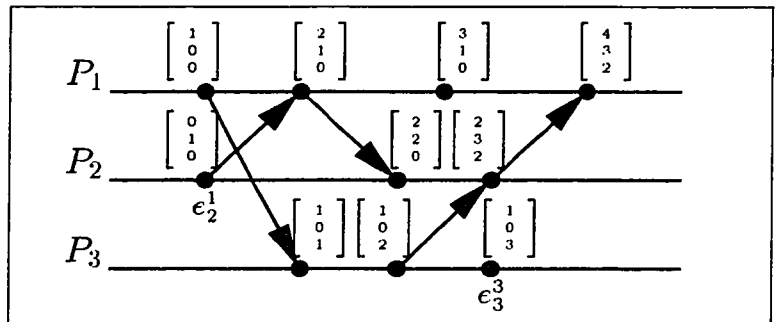

Figure 27: Vector clock in action

e happens before an event e' if and only if the vector clock value at e is smaller than the vector clock value at e'. The vector clock satisfies this strong property

$$e \rightarrow e' \iff V(e) < V(e'). \tag{3.0}$$

Consider the events e_2^1 and e_3^3 in Figure 27. Using the vector clock, we know that e_2^1 does not happen before e_3^3 since

$$V(e_2^1) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \not\leq \begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix} = V(e_3^3).$$

This accuracy is obtained at the price of extra space complexity. Indeed, a vector of n integers is appended to each message, where n is the number of processes. Since the number of processes can be large, the space cost can be excessive. It has been proved (see Charron-Bost [CB91]) that this space complexity is necessary in order to maintain accuracy. Moreover, Valot [Val93] showed that the accuracy of a logical clock is proportional to the space requirement.

Intuitively, the i^{th} coordinate of a process' vector clock represents its knowledge of process i progress. In other words, the i^{th} coordinate of $V(e)$ reflects the number of events in σ_e that occur in process i. This suggests a relation between vector clock and cut as defined in Chapter 1. We recall that a cut $(\sigma_1^{i_1}, \sigma_2^{i_2}, \ldots, \sigma_n^{i_n})$ is also represented by a n-vector $(i_1 i_2 \cdots i_n)$. For convenience, we use the notation $[\sigma]$ to denote either the cut or its vector representation. It can be shown that, for a minimal prefix σ_e, the vector representation of the cut $[\sigma_e]$ and the vector clock value at e (or more correctly its transpose since one is a column vector while the other is a row vector) are the same. Formally, for any event ϵ, we have

$$V(\epsilon) = [\sigma_\epsilon]^T,$$

where T is the vector transposition. Moreover, the union of states becomes the component-wise maximum and their intersection becomes the component-wise minimum. We denote by $\wedge$ the component-wise minimum operator on vectors and by $\vee$ the component-wise maximum operator on vectors. Hence

$$[\sigma \cup \tau] = [\sigma] \vee [\tau]$$

and

$$[\sigma \cap \tau] = [\sigma] \wedge [\tau].$$

The last properties and the vector representation of states allow us to write algorithms that use simpler data structures.

The vector clock has two important properties; it is able to capture the partial ordering of event, and the vector clock of an event is the minimal prefix at that event. The usefulness of vector clock surfaces in the design of distributed algorithms.

Using the vector clock, we give an example of an off-line algorithm to compute the basic intervals of a distributed computation.

In the first algorithm, we use the identity between the vector representation of states and the vector clock to compute the basic intervals. Rather than using the state $\sigma_{send(m)}$, we use the vector clock value $V(send(m))$ in order to compute the intervals. To imple-

```
function Compute_Concurrent_Interval(m : message) : interval;
var opening : vector;
var closing : vector;
begin
/* it is assumed that a vector clock was implemented */
/* and that the communication graph was computed */
    opening := [σ_0]; /* initial state */
    closing := [σ_f]; /* final state */
    for all messages m'' in pred(m) do
        opening := opening ∨ [σ_send(m'')];
    end for;
    for all messages m'' in ¬pred(m)
        closing := closing ∧ [φ_rec(m'')];
    end for;
    return [opening, closing];
end.
```

Algorithm 1: An algorithm to compute the concurrent interval

ment the algorithm, we assume that during the execution, the processes maintain a vector clock and record any new value for the clock. It is also necessary to record with the vector clock values the type of the corresponding event. For example, on a send event, the process stores the vector clock value, the event type "send" and the message ID. This information is mandatory to construct the communication graph. After the termination of the computation, a central process uses the recorded information to construct the communication graph and apply Algorithm 1 to construct the basic intervals. If the distributed computation contains n processes and k messages, then the complexity of the algorithm Compute_Concurrent_Interval is of order $O(n \times k)$. Since there are k basic intervals, the overall complexity to compute all the basic intervals is of order $O(n \times k^2)$.

The second algorithm constructs the connected components of a distributed computation. The algorithm receives a distributed computation and returns the resulting decomposition. It picks a process in the "remaining_processes" and finds all the processes linked to that process until a connected component is created. We use the special notations

```
function Detect_Connected_Components(D : computation) : set of computations;
var 𝔇 : set of computations;
var connected_component : computation;
var remaining_processes : computation;
begin
    𝔇 := ∅;
    remaining_processes := D;
    while remaining_processes ≠ ∅ do
        for some process P in remaining_processes do
            connected_component := {P};
        end for
        for all messages m in remaining_processes do
            if sender(m) in connected_component then
                connected_component := connected_component ∪ {receiver(m)};
            end if
            if receiver(m) in connected_component then
                connected_component := connected_component ∪ {sender(m)};
            end if
        end for
        remaining_processes := remaining_processes − connected_component;
        𝔇 := 𝔇 ⊎ connected_components;
    end while
end.
```

Algorithm 2: An algorithm to compute the connected components

$sender(m)$ to denote the process that sends the message m and $receiver(m)$ to denote the process that receive the same message. The complexity of Detect_Connected_Components is given by $O(n \times k)$ since the while loop is taken at most n times and each of the for loop is taken at most k times. In the next section, we use these algorithms as building blocks to design other detection algorithms.

3.2 Monotonic predicates and separable predicates

In the work of Tomlinson and Garg [TG93, TG97], relational predicates have been studied. This class of predicates is probably the most general that have been studied. However, the generality of the class also makes the detection problem harder. The approach adopted in [TG97] is to reduce the length of the predicate so that the detection can be done on any

distributed computation. More precisely, they considered the detection of the predicate

$$x_1 + x_2 > c,$$

where x_i is an integer valued variable and c is a constant. Such a predicate directly involves only two processes through the values of x_1 and x_2. The other processes in the distributed computation simply have to forward the dependencies (through a logical clock mechanism) to the two processes involves in the detection. Our approach is different. Rather than restricting the class of predicates so much, we restrict the class of distributed computations. More precisely, we give efficient algorithms to detect monotonic and separable predicates of any length on series-parallel systems. Eventhough the detection is performed only on series-parallel systems, this approach is more general than the one used in [TG97].

Let S be a set. A (global) **predicate** is a mapping

$$\mathfrak{P} : S \longrightarrow \{\text{FALSE}, \text{TRUE}\}.$$

We are interested in predicates of the form

$$\mathfrak{P} : \mathbb{P}_1 \times \mathbb{P}_2 \times \cdots \times \mathbb{P}_n \longrightarrow \{\text{FALSE}, \text{TRUE}\},$$

where $\mathbb{P}_i$ denotes the value space of the relevant variables in process i of some distributed computation. Each physical state σ corresponds to an element $\langle\sigma\rangle$ of $\mathbb{P}_1 \times \mathbb{P}_2 \times \cdots \times \mathbb{P}_n$, where the i^{th} component of $\langle\sigma\rangle$ is the value of the local state in process i. The vector $\langle\sigma\rangle$ is called the **state value** or simply the value of σ (this should not be confused with the cut $[\sigma]$). The predicates of interest are therefore mappings from the state values to $\{\text{FALSE}, \text{TRUE}\}$.

We mentioned in Chapter 1 a few classes of predicates that have been studied in the past. Among them, the class of conjunctive predicates is of great importance. We recall that a conjunctive predicate is one that can be written as a conjunction of local predicates. In this case, all value spaces are the same and are given by the set $\{\text{FALSE}, \text{TRUE}\}$. It is important to note that the value space can, in most cases, be seen as a total order. For example, in the case of conjunctive predicate, the set $\{\text{FALSE}, \text{TRUE}\}$ can be given the natural total order FALSE $<$ TRUE. In the following, we make this simplifying assumption:

Assumption The value space at each process is a total order.

The implicit assumption that $\mathbb{P}_i = \mathbb{P}_j$ for all i and j can be removed and is taken for simplicity only.

Consider a distributed computation where the value space at each process is a total order P and consider a mapping $f : P^n \rightarrow P$. A **relational predicate** is one of the form

$$\mathfrak{P}[f > c] : \mathbf{v} \longmapsto [f(\mathbf{v}) > c],$$

where $\mathbf{v} = (v_1, v_2, \ldots, v_n)$ is an element of P^n and $[f(\mathbf{v}) > c]$ evaluates to TRUE or FALSE. So, the predicate is true on vectors in P^n for which f is greater than c, otherwise it is false. Note that the direction of the inequality, $<$ or $>$, has no effect on the results. An algorithm to detect a predicate $\mathfrak{P}[f > c]$ can as well detect a predicate of the form $\mathfrak{P}[f < c]$ by reversing the total order relation. Before going further, let us look at an example.

Example 3.1 We show in this example that the conjunctive predicates are instances of relational predicates. Consider a value space {FALSE, TRUE} with the natural total order discussed above. The detection of the relational predicate $\mathfrak{P}[f >$ FALSE$]$ where

$$\begin{aligned} f : \{\text{FALSE}, \text{TRUE}\}^n &\longrightarrow \{\text{FALSE}, \text{TRUE}\} \\ (v_1, v_2, \ldots, v_n) &\longmapsto v_1 \wedge v_2 \wedge \cdots \wedge v_n, \end{aligned}$$

is equivalent to the detection of the corresponding conjunctive predicate. □

The detection of a relational predicate $\mathfrak{P}[f > c]$ is equivalent to the problem of detecting the state σ in the state lattice for which $f(\langle\sigma\rangle)$ is maximal. A solution to one problem gives a solution to the second. Our approach is to efficiently evaluate the maximal value of f (denoted by $max\{f\}$) on the state lattice. Let P be a total order.

Definition 3.1 A function

$$f : P^n \longrightarrow P$$

is **monotonic** if, for all i, $f(v_1, \ldots, v_i, \ldots, v_n) \leq f(v_1, \ldots, w_i, \ldots, v_n)$ whenever $v_i \leq w_i$.

A relational predicate $\mathfrak{P}[f > c]$ is a **monotonic predicate** if f is monotonic. Monotonic predicates are such that local decisions can be used to form a global decision. To make this even more precise, consider the next definition. A **box** in P^n is a subset of the form $S_1 \times S_2 \times \cdots \times S_n$, where $S_i \subseteq P$. If B is a box in P^n and f is a monotonic function, then

$$max\{f(\mathbf{v}) : \mathbf{v} \in B\} = f(max\{v_1 : \mathbf{v} \in B\}, \ldots, max\{v_n : \mathbf{v} \in B\}).$$

The evaluation of $max\{f\}$ on the box can be done by evaluating $max\{v_i\}$ on each component of the box. This property of monotonic functions ensures that the evaluation of $max\{f\}$ can be done efficiently on a concurrent interval. The next proposition establishes this result.

Proposition 3.1 *If I is a concurrent interval then $\langle I \rangle = \{\langle \sigma \rangle : \sigma \in I\}$ is a box in P^n and the equality*

$$(3.1) \qquad max\{f(\langle \sigma \rangle) : \sigma \in I\} = f(max\{\langle \sigma \rangle_1 : \sigma \in I\}, \ldots, max\{\langle \sigma \rangle_n : \sigma \in I\})$$

holds for a monotonic function f.

Proof. We must show that there are subsets $S_1, S_2, \ldots, S_n$ of P such that

$$\langle I \rangle = S_1 \times S_2 \times \cdots \times S_n.$$

Let $S_i = \{\langle \sigma \rangle_i : \sigma \in I\}$. From the definition of the sets S_i we have that

$$\langle I \rangle \subseteq S_1 \times S_2 \times \cdots \times S_n.$$

To show the other inclusion, we simply remark that a concurrent interval has no process dependencies. Therefore, any vector of n local states in the interval, one for each process, defines a cut in the interval. Equation 3.1 follows directly from the definition of monotonic functions. ■

From this proposition, we derive an algorithm to detect a monotonic predicate $\mathfrak{P}[f > c]$ on concurrent intervals. As mentioned earlier, it is enough for the algorithm to compute the maximal value of f for all states in the interval. Therefore, the algorithm takes a function f and an interval I as input, and returns an integer. It is guaranteed that, if the input f is monotonic and I is concurrent, then the value returned is the maximal value of f on I. Let n be the number of processes in the computation and let v_i be the relevant variable in process i. The third algorithm below evaluates efficiently a monotonic function on concurrent intervals. The correctness of the algorithm follows directly from the monotonicity property. Let l be the maximal number of local states in a local interval. The time complexity of the algorithm is given by $O(l \times n)$.

We turn now to the detection of monotonic predicates on series systems. The detection algorithm is applied after the termination of the computation. The general idea is to

```
function Evaluate_On_Concurrent_Interval(f : monotonic; I : interval) : integer;
    var max : array[1..n] of integers;
    var i : integer;
begin
/* it is assumed that I is concurrent */
/* and that f is monotonic */
    max := 0;
    for i = 1 to n do
        max[i] := max of v_i on I;
    end for;
    return f(max[1], max[2], ..., max[n]);
end.
```

Algorithm 3: Evaluation of monotonic functions on a concurrent interval

decompose the state lattice into concurrent intervals and to apply Algorithm 3 to evaluate the monotonic function on each of these intervals. The maximal value taken by the function is then the maximal value of these interval evaluations. A comparison of the value obtained with the constant c gives a solution to the detection problem. Algorithm 4 offers an efficient way to evaluate a monotonic function f on a series system D. The correctness of the

```
function Evaluate_Monotonic_On_Series(f : monotonic; D : computation) : integer;
var max : interger;
var I : interval;
begin
/* it is assumed that D is series */
    I := Compute_Concurrent_Interval(0); /* evaluate on the initial interval first*/
    max := Evaluate_On_Concurrent_Interval(f,I);
    for all messages m in D do
        I := Compute_Concurrent_Interval(m);
        if max < Evaluate_On_Concurrent_Interval(f,I) then
            max := Evaluate_On_Concurrent_Interval(f,I);
        end if;
    end for;
    return max;
end.
```

Algorithm 4: Evaluation of monotonic functions on series systems

algorithm is obvious at this point. It is based on the results proved in Chapter 2.

Proposition 3.2 *Monotonic predicates can be detected in polynomial time on series systems.*

Proof. Proposition 3.1 and Theorem 2.10 of Chapter 2 prove the correctness of the algorithm. Indeed, since the set of message is serialized, the set of basic intervals, together with the initial interval form a decomposition of the state lattice. In order to evaluate the time complexity of Algorithm 4, we analyze each of the operations involved. Assume that we have n processes, k messages, a maximum of l local states in each process and that the ordering mechanism allows us to do comparison of two events in $O(n)$ (this is what we would get for vector clocks). We know that the algorithms Evaluate_On_Concurrent_Interval and Compute_Concurrent_Interval have complexities of $O(n \times l)$ and $O(n \times k)$ respectively. Since the for loop is taken k times, the total complexity is $O(k \times n \times l) + O(n \times k^2)$. ■

The next example illustrates the algorithm.

Example 3.2 In this example, we find the maximal value of the monotonic function

$$f(x_1, x_2, x_3) = x_1 + 3x_2x_3$$

on the series system of Figure 28. We first compute the basic intervals. We get

$$I_0 = \begin{bmatrix} 0-1 \\ 0-3 \\ 0-0 \end{bmatrix} \qquad I_{m_1} = \begin{bmatrix} 1-1 \\ 0-3 \\ 0-3 \end{bmatrix} \qquad I_{m_2} = \begin{bmatrix} 1-2 \\ 1-3 \\ 0-3 \end{bmatrix}$$

$$I_{m_3} = \begin{bmatrix} 1-3 \\ 2-3 \\ 0-3 \end{bmatrix} \qquad I_{m_4} = \begin{bmatrix} 1-3 \\ 2-4 \\ 3-3 \end{bmatrix}$$

The series property ensures that these intervals cover the whole state lattice. Next, we evaluate $max\{f\}$ on each of them. Since f is monotonic, the maximal value of f on a concurrent interval is obtained at the maximal local variables. As an example, the state maximizing f on I_{m_2} is (1 2 3) whose local variable value in process 1 is 5 , 3 in process 2 and 4 in process 3. So, the maximal value of f on I_{m_2} is $5 + 3(3 \times 4) = 41$. Similarly, we

evaluate f on all the intervals

$$max\{f(\langle\sigma\rangle) : \sigma \in I_0\} = 32 \qquad max\{f(\langle\sigma\rangle) : \sigma \in I_{m_1}\} = 41$$
$$max\{f(\langle\sigma\rangle) : \sigma \in I_{m_2}\} = 41 \qquad max\{f(\langle\sigma\rangle) : \sigma \in I_{m_3}\} = 41$$
$$max\{f(\langle\sigma\rangle) : \sigma \in I_{m_4}\} = 53$$

From the algorithm, we know that the maximal value of f on all the states is given by the maximal value above, that is, $max\{f(\langle\sigma\rangle) : \sigma \in \Gamma(D)\} = 53$. So, the predicate $\mathfrak{P}[f > 50]$ is detected on D but not the predicate $\mathfrak{P}[f > 55]$. It is important to note that the combination of local states (1 4 1) gives a larger value to f. Indeed, the local variable values are 5 in process 1, 4 in process 2 and 5 again in process 3. But this combination does not form a consistent state. □

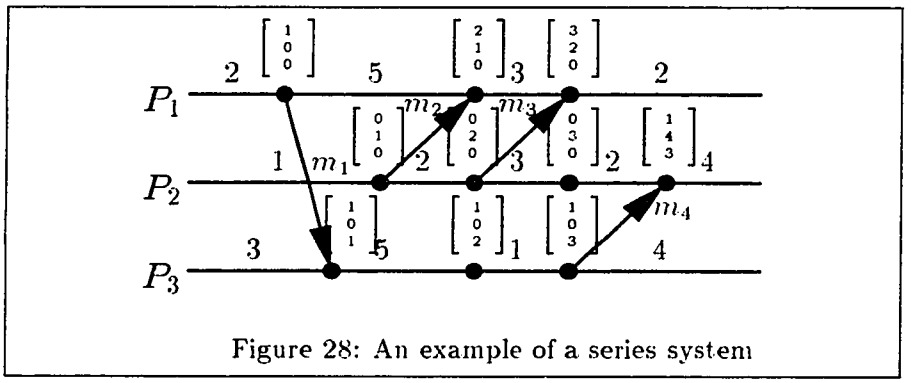

Figure 28: An example of a series system

It is not known if monotonic predicates can be detected efficiently on a parallel systems. But as we will see, the following additional property of relational predicates enables us to detect them efficiently on parallel systems. Let P be a total order.

Definition 3.2 A function $f : P^n \to P$ is **separable** if it can be written as

$$f(\mathbf{v}) = \bigoplus_{i=1}^{n} h_i(v_i),$$

where $h_i : P \to P$ are functions of one variable and $\oplus$ is a binary operator on P such that max is distributive on $\oplus$.

A relational predicate $\mathfrak{P}[f > c]$ is a **separable predicate** if f is separable. Here are three examples of separable predicates.

Example 3.3 The three functions

$$\begin{aligned} f(\mathbf{v}) &= v_1 + v_2 + \cdots + v_n \\ g(\mathbf{v}) &= v_1 + v_2 + \cdots + v_n - v_{n+1} - v_{n+2} - \cdots - v_{2n} \\ h(\mathbf{v}) &= v_1 v_2 \cdots v_n \end{aligned}$$

are separable. The predicate $\mathfrak{P}[f > c]$ can be used to detect the maximal number of server connections that a distributed program uses. The predicate $\mathfrak{P}[g > c]$ can be used to detect if the number of server connections is balanced between two sets of processes. □

Since *max* is distributive on $\oplus$, the separable function

$$f(\mathbf{v}) = v_1 \oplus v_2 \oplus \cdots \oplus v_n$$

is monotonic. Therefore, Algorithm 4 detects efficiently separable predicates on series systems. There exists also an efficient algorithm to detect a separable predicate on parallel systems. Algorithm 5 uses the fact that the messages in a parallel system belong to different connected components. The separability property allows us to evaluate the function on each of these components individually. Let D_i be a connected component containing the processes $\{i_1, i_2, \ldots, i_l\}$ and let f be a separable function. The restriction of f to D_i,

$$f|_{\mathcal{S}(D_i)}(v_{i_1}, v_{i_2}, \ldots, v_{i_l}) = h_{i_1}(v_{i_1}) \oplus h_{i_2}(v_{i_2}) \oplus \cdots \oplus h_{i_l}(v_{i_l})$$

is again a separable function. The evaluation of $f|_{\mathcal{S}(D_i)}$ on a parallel system D is easy since each connected component contains at most one message. Algorithm 4 can be used for that purpose. The distributivity of *max* on $\oplus$ leads to

$$max\{f(\langle\sigma\rangle) : \sigma \in \Gamma(D)\} = \bigoplus_i max\{f|_{\mathcal{S}(D_i)}(\langle\sigma\rangle) : \sigma \in \Gamma(D_i)\}, \tag{3.3}$$

where i runs over all connected components of D. Equation 3.3 says that the sum of the local maxima of the restriction of f gives an evaluation of $max\{f\}$ on the whole computation. This is proven in the next proposition where we show also the correctness of Algorithm 5.

Proposition 3.3 *Separable predicates can be detected in polynomial time on parallel systems.*

```
function Evaluate_Separable_On_Parallel(f : separable; D : computation) : integer;
var max : integer;
var 𝔇 : set of computations;
begin
/* it is assumed that D is parallel */
    max := 0;
    𝔇 := Compute_Connected_Components(D);
    for all components D'' in 𝔇 do
        max := max⊕ Evaluate_Monotonic_On_Series(f|_{S(D'')},D'');
    end for;
    return max;
end.
```

Algorithm 5: Evaluation of separable functions on parallel systems

Proof. Consider a parallel distributed system with messages $M = \{m_1, m_2, \ldots, m_k\}$ and let $D_1, D_2, \ldots, D_l$ be the connected components of the distributed system. Suppose for simplicity that D_i contains the processes $2i-1$ and $2i$. By Theorem 2.9 of Chapter 2, any state of the parallel system can be uniquely written has

$$\sigma = \sigma_1 \cup \sigma_2 \cup \cdots \cup \sigma_l,$$

where $\sigma_i \in \Gamma(D_i)$. Then, given a separable predicate with $f(x_1, x_2, \ldots, x_n) = \bigoplus_{i=1}^{n} h_i(x_i)$, we must have

$$\begin{aligned} max\{f(\langle\sigma\rangle) : \sigma \in \Gamma(D)\} &= max\{\bigoplus_{i=1}^{l} f|_{S(D_i)}(\langle\sigma_i\rangle) : \sigma_i \in \Gamma(D_i)\} \\ &= \bigoplus_{i=1}^{l} max\{f|_{S(D_i)}(\langle\sigma_i\rangle) : \sigma_i \in \Gamma(D_i)\}, \end{aligned}$$

where $f_i(x_{2i-1}, x_{2i}) = h_{2i-1}(x_{2i-1}) \oplus h_{2i}(x_{2i})$. Since D_i is a series system, $max\{f_i(\langle\sigma_i\rangle : \sigma_i \in \Gamma(D_i)\}$ can be calculated as in Algorithm 4. The evaluation of the complexity goes as follows. Let n be the number of processes in the computation, k be the number of messages and l be the maximal number of local states in a process. As discussed before, the computation of the connected components is done in $O(n \times k)$. In Lemma 3.2, we evaluate the complexity of the algorithm Evaluate_Monotonic_On_Series. Since all connected components have at most one message and two processes, the given complexity reduces to $O(l)$. Finally, the algorithm loop is executed at most n times. This gives an overall complexity of $O(n \times k) + O(n \times l)$.

■

The next section presents a generalization of the previous evaluation algorithms. It shows that separable functions can be evaluated efficiently on series-parallel systems.

3.3 Detection of separable predicates on series-parallel systems

In Chapter 2, we discussed how a series-parallel system can be decomposed into series or connected components. These results, together with the detection algorithms of the previous section, suggest that it might be possible to detect efficiently separable predicates on series-parallel systems. Although this detection algorithm exists and is closely related to the previous detection algorithms, it is less straightforward. Therefore, before describing our detection algorithm, we give a similar algorithm on a simpler structure.

Consider a series-parallel partial order P where an integer weight has been assigned to each element. As in the case of general series-parallel partial orders, these weighted orders can be represented by a decomposition tree. An example of a weighted partial order and one of its decomposition trees is given in Figure 29. Let the weight of an anti-chain be the sum of the weights of its elements. We consider the following problem:

Maximal weight problem Find the anti-chain in a weighted series-parallel partial order with the maximal weight.

Note that the number of anti-chains may grow exponentially with the number of elements in the partial order. Thus, any attempt to solve the problem by finding the weight of each anti-chain is unrealistic. However, the above problem is of polynomial complexity and an algorithm to solve it is given below. It is also interesting to note that the same problem on general partial orders is NP-complete. One can show that the clique problem is a special case of the above problem.

The series-parallel property allows us to use a divide-and-conquer strategy. Each node in the decomposition tree is the root of a subtree. For each subtree, we find the anti-chain with maximal weight and we assign this weight to the root. To do so, we start at the leaf level and progressively compute the weight of each node in the tree. Obviously, a leaf node is assigned the weight of its corresponding element in the weighted partial order. The nodes of two subtrees rooted at a series node are related. Therefore, anti-chains of these subtrees

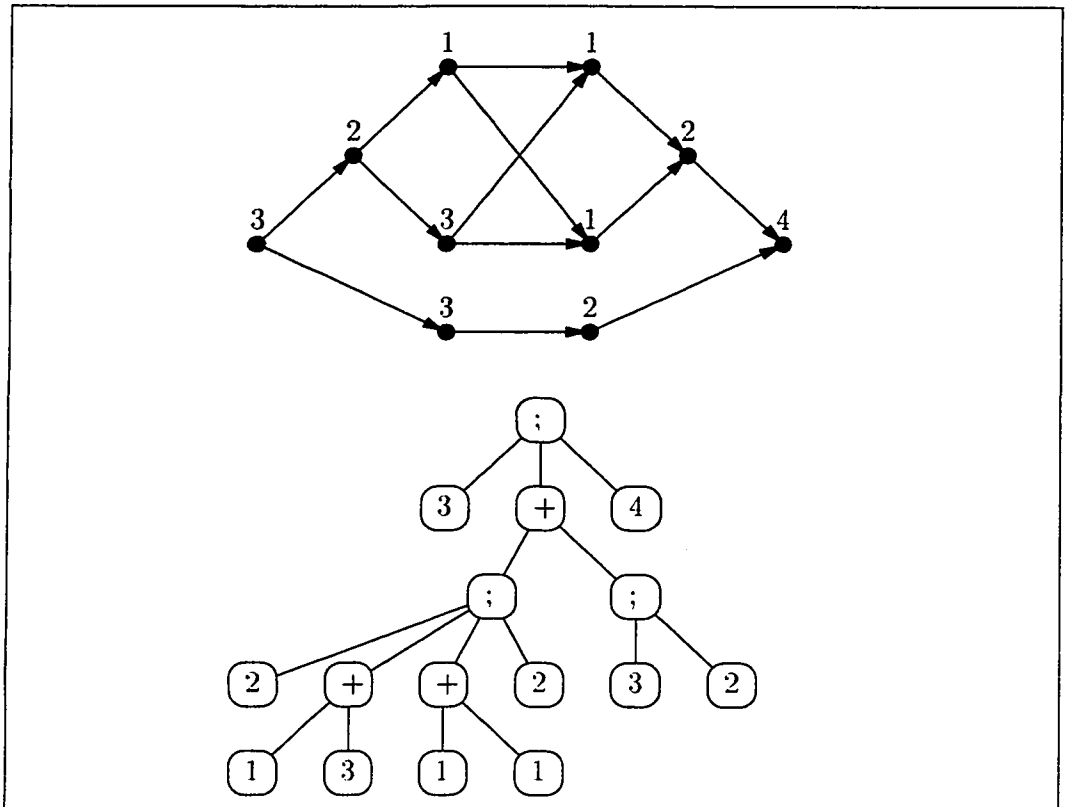

Figure 29: A weighted series-parallel partial order and its decomposition tree

cannot be combined together to form a longer anti-chain. We conclude that the maximal anti-chain of a tree with series root must belong entirely to one of its subtrees. On the other, if the root of a tree is a parallel node, the nodes in one subtree are not related to the nodes in another subtree. So, anti-chains of different subtrees can be added together to make a longer anti-chain. The maximal anti-chain of a tree rooted at a parallel node is the combination of the maximal anti-chains of all its subtrees. From these remarks, we get this mechanical procedure to compute the weight of the maximal anti-chain:

1. the weight of a leaf node is the weight assigned to the element at the leaf;
2. the weight of a series node is the **maximum** of the weight of its children;
3. the weight of a parallel node is the **sum** of the weight of its children.

This procedure gives rise to a recursive algorithm to compute the weight of the nodes. We have that the maximal weight of the anti-chains in a weighted partial order is given by the weight assign to its decomposition tree's root.

Let *root* be a tree and *root.child* be the subtree rooted at a child. Algorithm 6 recursively computes the weight of the root. The correctness of the algorithm should be obvious from

```
function weight(root : tree) : integer;
begin
    /* we assume that the input */
    /* tree is series-parallel */
    if root = leaf then
        weight(root) := root.weight
    else if root =; then
        weight(root) := max{weight(root.child): for all child of root}
    else if root = + then
        weight(root) := Σ_{child of root} weight(root.child)
    endif
end.
```

Algorithm 6: A recursive algorithm to compute the maximal weight

the remarks above and its complexity is of order $O(k)$ which corresponds to the time needed to traverse a decomposition tree with k leaves. The recursive *weight* function resembles the detection algorithm of separable predicates on series-parallel systems presented next.

Example 3.4 We apply the weight algorithm on the weighted partial order of Figure 29. The weights of the leaf nodes are already shown in the decomposition tree. The weights at the two parallel nodes on the left tree are simply given by the sum of their child's weights. This gives a weight of 4 for the left-hand side node and a weight of 2 for the right-hand side one. At the next level, there are two series nodes. The weight of a series node is given by the maximum of its child's weights. Hence, the series node at the left-hand side has a weight of 4, while the one at the right-hand side has a weight of 3. We continue the same procedure for the two next levels. We have a weight of 7 for both, the root, and the parallel node just below it. This shows that maximal weight of an anti-chain in the partial order is 7. This can also be easily verified from the partial order graph. □

Before going to the presentation of the algorithm, we recall a few results from Chapter 2.

A distributed system with a series-parallel communication graph can be either decomposed into connected components or series components. Moreover, each of these components containing more than one message can also be series or parallel decomposed. This recursive process takes at most k iterations, that is, the number of messages in the communication graph. We can therefore design an efficient algorithm to recursively compute these components and calculate their corresponding intervals. Then, an evaluation algorithm for separable functions can use this decomposition to evaluate the partial functions on each of the intervals and construct the maximal value of f from the partial evaluations as we did with the maximal anti-chain construction.

To help us find the series and parallel decomposition of a communication graph, we use the decomposition tree. The subtrees rooted at a series node correspond to the series components, while the subtrees rooted at a parallel node correspond to the connected components. Consider the decomposition tree of a series-parallel graph M. Let $M(node)$ be the set of messages rooted at $node$. Let the space $\mathcal{S}(D)$ of a distributed computation D be the set of processes D contains. Algorithm 7 efficiently evaluates a separable function f on a series-parallel system D. We recall that the initial interval is not included in the decomposition tree. Therefore, the algorithm must be completed with an evaluation of f on the initial order interval. This can be done efficiently since f is separable. When combined, the algorithm and the evaluation on the initial interval return the maximal value of f on all the states in the state lattice of D.

The first step in the first for all loop requires some clarifications. We have not presented yet a construction algorithm for series components. This algorithm is in fact almost identical to the concurrent interval construction given in Section 3.1. The computation of the series components uses the vector clock and applies directly the definition of the series components. We further discuss the complexity of this operation in the following. The correctness of the Algorithm 7 is given in the next theorem.

Theorem 3.4 *Separable predicates can be detected in polynomial time on series-parallel systems.*

Proof.

Correctness We first note that exactly one of the if segments get executed on each call of the function Evaluate_Separable_On_SP. Therefore, it is enough to show the correctness of

```
function Evaluate_Separable_On_SP(f : separable; root : state tree;
                                  D : computation) : integer;
var max : integer;
var 𝔇 : set of computations;
begin
/* it is assumed that root is a decomposition */
/* tree of the series-parallel system D */
    max := 0;
    if root = leaf then
        max := Evaluate_Monotonic_On_Interval(f,I_M(node));
    else if root = ; then
        for all children child of root do
            compute the series component I_M(child);
            if max < Evaluate_Separable_On_SP(f, child, I_M(child)) then
                max := Evaluate_Separable_On_SP(f, child, I_M(child));
            end if
        end for
    else if root = + then
        𝔇 := Compute_Connected_Components(I_M(root));
        for all connected components D'' in 𝔇 do
            max := max⊕ Evaluate_Separable_On_SP(f|_S(D'') , child, D'');
        end for
    end if
end.
```

Algorithm 7: Evaluation of separable functions on series-parallel systems

these segments individually. Their correctness is clear from Theorems 2.10, and 2.9, and from Equation 3.1.

Complexity analysis We must find the complexity of the series components construction in the first for all loop. This complexity can be at most k times the complexity of the basic interval construction since the set $M(child)$ contains at most k elements. This gives a complexity of $O(n \times k^2)$. The decomposition tree contains $O(k)$ nodes that must be visited, where k is the number of messages. There are therefore at most k recursive calls to the function Evaluate_Separable_On_SP. The complexity of the algorithm Evaluate_Monotonic_On_Interval is of order $O(n \times l)$ and the construction of the connected components is of order $O(n \times k)$. The overall complexity is $O(k \times (n \times k^2 + n \times k + n \times l)) = O(nk \times (k^2 + l))$. Note that the complexity can be reduced substantially if the series components and the connected components are computed first. With this approach, an evaluation

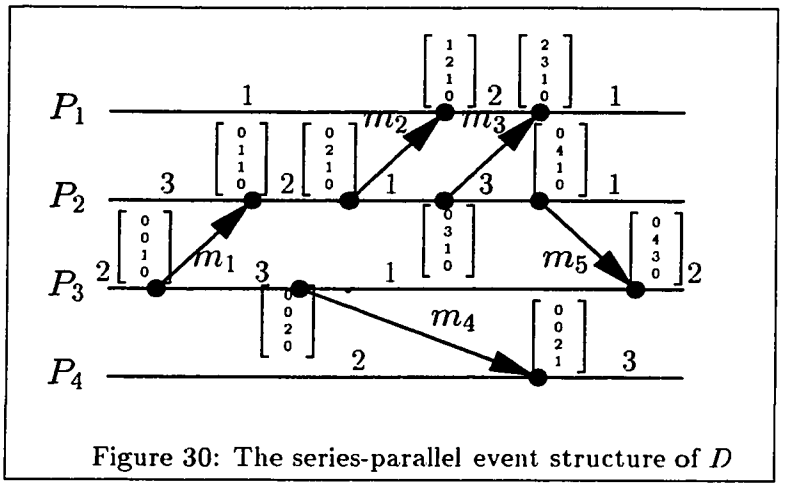

Figure 30: The series-parallel event structure of D

of f must only retrieve the information stored rather than calculating it each time the function is called. ■

To illustrate the algorithm, we use the following example.

Example 3.5 Figure 30 gives the event structure of the series-parallel system D. In order to verify that D satisfies the series-parallel property, we construct its communication graph. In this case, the graph contains only five nodes organized as in Figure 31 and its series-parallel property is obvious from the corresponding decomposition tree in Figure 32. A

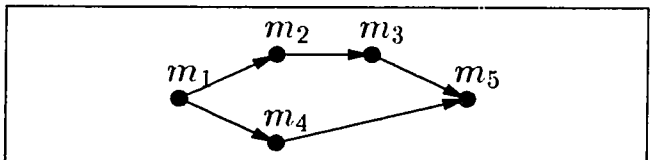

Figure 31: The communication graph of D

closer look to the decomposition tree of D reveals that the communication graph is series decomposable into

$$\{m_1\} \uplus ((\{m_2\} \uplus \{m_3\}) \uplus \{m_4\}) \uplus \{m_5\}.$$

This gives a decomposition of the state lattice into the three series components:

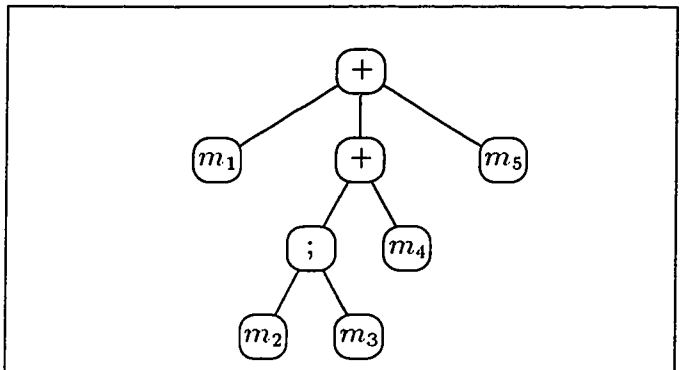

Figure 32: The decomposition tree of D

$$I_{\{m_1\}} = \begin{bmatrix} 0-0 \\ 0-4 \\ 1-2 \\ 0-0 \end{bmatrix} \quad I_{\{m_2,m_3,m_4\}} = \begin{bmatrix} 0-2 \\ 0-4 \\ 1-2 \\ 0-1 \end{bmatrix} \quad I_{\{m_5\}} = \begin{bmatrix} 0-2 \\ 4-4 \\ 2-3 \\ 0-1 \end{bmatrix}$$

to which we must add the initial interval

$$I_\emptyset = \begin{bmatrix} 0-0 \\ 0-0 \\ 0-2 \\ 0-0 \end{bmatrix}.$$

An evaluation algorithm for $f(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4$ must evaluate f on each components. Among the series components above, three of them $I_\emptyset, I_{\{m_1\}}$ and $I_{\{m_5\}}$ are concurrent, so that we can evaluate immediately f on each one. We have

$$max\{f(\langle\sigma\rangle) : \sigma \in I_\emptyset\} = 9 \qquad max\{f(\langle\sigma\rangle) : \sigma \in I_{\{m_1\}}\} = 9$$

$$max\{f(\langle\sigma\rangle) : \sigma \in I_{\{m_5\}}\} = 8.$$

The component $I_{\{m_2,m_3,m_4\}}$ can be further decomposed into its connected components. We get

$$I_{\{m_2,m_3,m_4\}}|_{\{1,2\}} = \begin{bmatrix} 0-2 \\ 0-4 \\ - \\ - \end{bmatrix} \quad I_{\{m_2,m_3,m_4\}}|_{\{3,4\}} = \begin{bmatrix} - \\ - \\ 1-2 \\ 0-1 \end{bmatrix}.$$

Again, further decomposition is possible. Each connected component can be series decomposed as

$$I_{\{m_1\}}|_{\{1,2\}} = \begin{bmatrix} 0-0 \\ 0-4 \\ - \\ - \end{bmatrix} \quad I_{\{m_2\}}|_{\{1,2\}} = \begin{bmatrix} 0-1 \\ 2-4 \\ - \\ - \end{bmatrix} \quad I_{\{m_3\}}|_{\{1,2\}} = \begin{bmatrix} 0-2 \\ 3-4 \\ - \\ - \end{bmatrix}$$

and

$$I_{\{m_1\}}|_{\{3,4\}} = \begin{bmatrix} - \\ - \\ 1-2 \\ 0-0 \end{bmatrix} \quad I_{\{m_4\}}|_{\{3,4\}} = \begin{bmatrix} - \\ - \\ 2-2 \\ 0-1 \end{bmatrix}.$$

respectively. Once the restriction of f is evaluated on these components, we get

$$max\{f(\langle\sigma\rangle)|_{\{1,2\}} : \sigma \in I_{\{m_1\}}|_{\{1,2\}}\} = 4 \qquad max\{f(\langle\sigma\rangle)|_{\{1,2\}} : \sigma \in I_{\{m_2\}}|_{\{1,2\}}\} = 5$$

$$max\{f(\langle\sigma\rangle)|_{\{1,2\}} : \sigma \in I_{\{m_3\}}|_{\{1,2\}}\} = 5$$

and

$$max\{f(\langle\sigma\rangle)|_{\{3,4\}} : \sigma \in I_{\{m_1\}}|_{\{3,4\}}\} = 5 \qquad max\{f(\langle\sigma\rangle)|_{\{3,4\}} : \sigma \in I_{\{m_4\}}|_{\{3,4\}}\} = 4.$$

Since f is separable we must have that the maximal value of f on $I_{\{m_2,m_3,m_4\}}$ is given by the sum of the maximal value of the restriction of f on the connected components. This gives that

$$max\{f(\langle\sigma\rangle) : \sigma \in I_{\{m_2,m_3,m_4\}}\} = 10.$$

We conclude that the maximal value of f is given by the maximum of the maximal value on each series components, which is 10. □

Chapter 4

Further results: strong detection of monotonic predicates

In the previous chapter, we presented algorithms to detect specialized classes of predicates. In all cases, we wanted to know if there was a state of the distributed computation for which the predicate was true. This kind of detection is called weak detection since it may happen that one observation has no state for which the predicate is true. As we mentioned briefly in Chapter 1, there are other kinds of detection. For example, the strong detection is one where any observation contains at least one state satisfying the predicate. We present in this chapter an example of a strong detection algorithm.

4.1 Strong detection of monotonic predicates

Strong detection is considered to be harder than the weak one. Hence, we do not intend to give a complete solution to the strong detection problem. Our solution is restricted to the strong detection of monotonic predicates on concurrent intervals.

We recall that an observation is a directed path in the state lattice going from the initial state to the final state. Let D be a distributed computation and let $\mathcal{L}(D)$ be the set of observations on D. We have a strong detection of a relational predicate $\mathfrak{P}[f > c]$ if

$$\forall L \in \mathcal{L}(D), \exists \sigma \in L : f(\langle \sigma \rangle) > c.$$

We use the notation $\forall\mathfrak{P}[f > c]$ for strong detection of a monotonic predicate $\mathfrak{P}[f > c]$.

Given a monotonic predicate $\mathfrak{P}[f > c]$, we define the **weight** of a state σ by the evaluation $f(\langle\sigma\rangle)$. The predicate $\forall\mathfrak{P}[f > c]$ is detected on a computation if all the observations contain a state with weight greater than c. Since strong detection is often used to describe good properties, we call an observation where no state has a weight greater than c, a **faulty path**. We say that a state is **fault-reachable** from another state if there exists a (partial) faulty path from the latter to the former. Our detection algorithm construct a faulty path if it exists.

Example 4.1 Consider the event structure in Figure 33 and its corresponding state lattice. We have appended at each node of the lattice the weight of the function

$$f(x_1, x_2) = x_1 + x_2.$$

A faulty path for the predicate $\forall\mathfrak{P}[f > 4]$ is given by the thick lines. One can verify that none of the states in the path has a weight greater than 4. □

4.2 Strong detection of monotonic predicates on concurrent intervals

Before presenting the detection algorithm, we describe the function reach_minimal given in Algorithm 8. The function reach_minimal($[\sigma, \tau]$,f,c) returns σ if it has a weight greater than c, otherwise it returns a state fault-reachable from σ with minimal weight. Note that there may exist several such states. To build a faulty path, reach_minimal function increment the current state to a new state with smaller weight. To do so, the function scans the next local states in one process until it reaches a local state with weight smaller than the weight of the current local state. If the new state μ is fault-reachable from the current state, then the current state is set to μ. This procedure continues until the state τ is reached or no state with smaller weight is fault-reachable from the current one. The procedure has clearly the effect of reducing the weight of the current state. The algorithm to detect $\forall\mathfrak{P}[f > c]$ on a concurrent interval uses the function reach_minimal. Given a concurrent interval $[\sigma, \tau]$, the algorithm has these three steps:

1. Apply function reach_minimal() on $[\sigma, \tau]$ and get the state with minimal weight μ_d fault-reachable from σ;

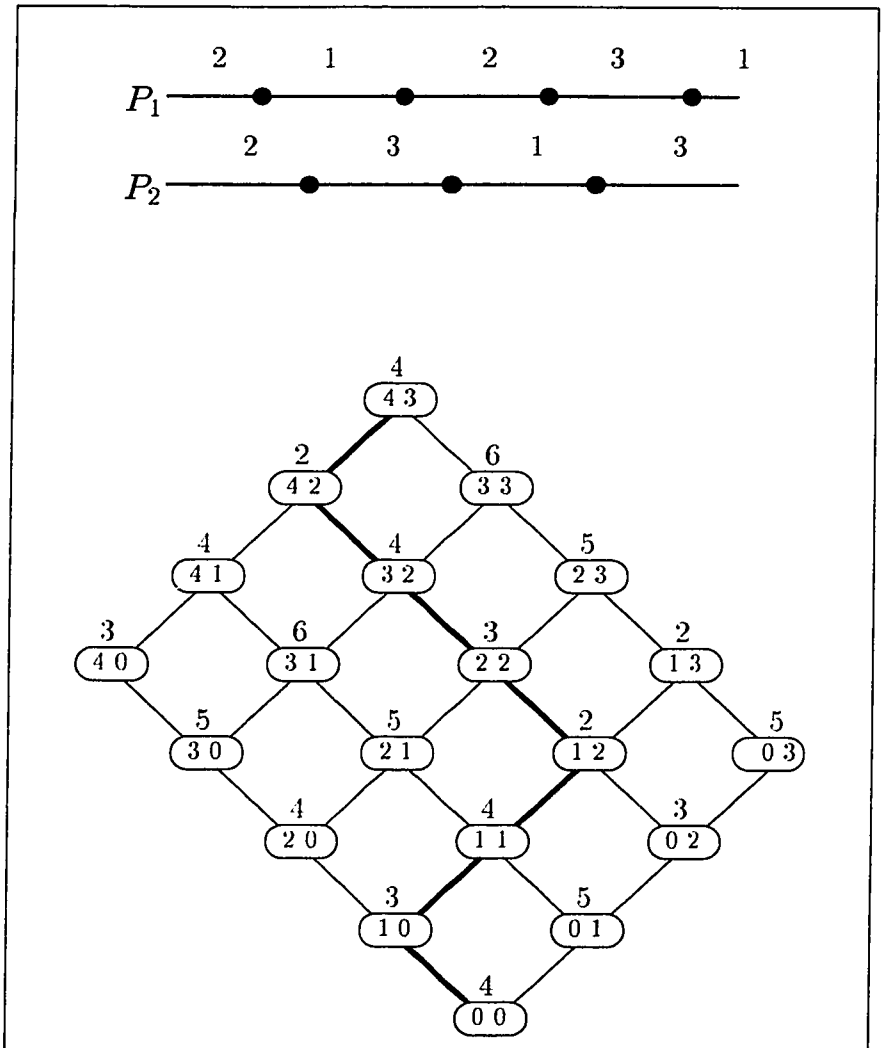

Figure 33: A faulty path for the predicate $\forall\mathfrak{P}[f > 4]$

```
function reach_minimal([σ, τ] : interval; f : monotonic; c : integer) : state
var change : boolean;
var current_state : state;
var local_state : local state;
begin
/* it is assumed that [σ, τ] is concurrent */
    change := TRUE;
    if f(⟨σ⟩) < c then
        return σ;
    else
        current_state := σ;
    end if
    while change do
        change := FALSE;
        for i = 1 to n do
            find next local state in P_i with ⟨local_state⟩ ≤ ⟨current_state⟩_i;
            if local_state fault-reachable from [current_state]_i then
                [current_state]_i := local_state;
                change := TRUE;
            end if
        end for
    end while
    return current_state;
end.
```

Algorithm 8: An algorithm to find the path to the state with minimal weight

2. Apply function reach_minimal( ) on $[\tau, \sigma]$ where the partial order has been reversed and get the state with minimal weight μ_u fault-reachable from τ;

3. If $\mu_u \subseteq \mu_d$ then $\forall\mathfrak{P}[f > c]$ is false, otherwise it is true.

The correctness of the procedure is shown by the next lemma and the following theorem.

Lemma 4.1 *Given a monotonic predicate $\mathfrak{P}[f > c]$ and a concurrent interval $I = [\sigma, \tau]$, if there exists a faulty path in I, then there is a faulty path passing through any state with minimal weight.*

Proof. Let $\sigma = \mu_1 \subseteq \mu_2 \subseteq \cdots \subseteq \mu_k = \tau$ be a faulty path in I. We show that we can modify this observation to one containing any state with minimal weight. Consider the path

$$\mu_1 \cap \mu_{min} \subseteq \cdots \subseteq \mu_k \cap \mu_{min} \subseteq \mu_{min} \subseteq \mu_1 \cup \mu_{min} \subseteq \cdots \subseteq \mu_k \cup \mu_{min},$$

where μ_{min} is a state with minimal weight. We show that each state in the above path has a weight smaller than c, that is, we show that the path is a faulty path. To see this, it is enough to realized that since μ_{min} has minimal weight each of the local states of μ_{min} has minimal value (because of the monotonicity of f). We know that the cut $[\mu \cup \mu_{min}]$ is a vector with either the local states of μ or μ_{min}. Since the local values of μ_{min} are minimal, the weight of $\mu \cup \mu_{min}$ is smaller than the weight of μ. We have

$$f(\langle \mu \cup \mu_{min} \rangle) < f(\langle \mu \rangle) < c.$$

Similarly, we show that the state $\mu \cap \mu_{min}$ has weight smaller than c. ■

Theorem 4.2 (strong detection) *Strong monotonic predicates can be detected in polynomial time on concurrent intervals.*

Proof. We first prove the correctness of the algorithm. By construction, state μ_d is fault-reachable from σ and state τ is fault-reachable from μ_u. If we have $\mu_u \subseteq \mu_d$, then, by the previous lemma, there is a faulty path from μ_u to τ passing through μ_d. Therefore, there is a faulty path from σ to τ passing through μ_d. This shows that the algorithm asserts FALSE only when there is at least one faulty path.

Consider the case where $\mu_d \subset \mu_u$. By construction of the states, there can not be a faulty path from μ_d to μ_u. We note also that the state μ_d is the smallest state in $[\sigma, \mu_d]$. In particular, each of the local state in μ_d have a weight smaller than or equal to the corresponding local state of σ. Similarly, μ_u is the smallest state in $[\mu_u, \tau]$. An argument similar to the one used in the previous lemma shows that if there is a faulty path from $[\sigma, \tau]$ then there is a faulty path from $[\mu_d, \mu_u]$. This would lead to a contradiction since there is no such path in $[\mu_d, \mu_u]$. This shows that the algorithm assert TRUE only when there is no faulty path.

The complexity analysis gives that the for loop is executed n times and that the while loop is executed $O(l \times n)$. Hence, the overall complexity is $l \times n^2$. ■

Conclusion

The ability to detect predicates is important in monitoring, debugging and testing of distributed computations. While researchers succeeded in designing algorithms to detect specialized classes of predicates, there are still much work to be done in this area. In particular, it is not yet known which properties of a predicate make it easy to detect or if the number of states in a distributed computation can be evaluated in polynomial time. The first question is especially important since it would tell which kind of predicates a programmer can use to debug a distributed computation.

In the past, the main approach to detection consisted exclusively in restricting the class of predicates to simplify the detection problem. This approach succeeded in showing that certain classes of predicates, like the conjunctive predicates, can be detected efficiently on any distributed computation. But it is also known that not all the predicates have this property. In this thesis, we use a different approach. We want to provide an efficient detection algorithm for a large class of predicates. To achieve this goal, we have to restrict the computation on which to do the detection. We show that the detection problem on computations with the series-parallel property can be addressed for larger classes of predicates. In particular, two new classes are introduced, monotonic and separable predicates, and detection algorithms are also given. Monotonic predicates and separable predicates have a better expressiveness than most of the common classes of predicates studied in the past. But our approach poses another difficulty. Since the detection is done only on specialized structures, one must have an algorithm to recognize a computation with the given property. We address briefly this question for the series-parallel property and give the relevant references.

Another important result in this thesis is the decomposition of the state lattice. We

show that the state lattice of a distributed computation can be decomposed into concurrent intervals. By doing so, we also establish a relation between the state lattice structure and the communication graph which gives an higher level of abstraction suitable from the detection of predicates. It is interesting to note that the series-parallel property is probably not the only property that ensures a simple decomposition of the state lattice. Other properties might exist for which a decomposition theorem and a detection algorithm can be proved.

In order to enhance the practical interest of the detection algorithms in this thesis, a conversion to online and/or distributed algorithms should be attempted. This would bring a better understanding of the necessity of the series-parallel property in both the decomposition and the detection problems.

Appendix A

A review of related sets, partial orders and lattices

Since our model relies intensively on discrete structures, we include this appendix. The purpose is to give a short review of the relevant definitions in this area. For a more complete introduction, it is suggested that the reader refers to Davey and Priestley's book [DP1].

A.1 Related Sets

A standard example of a related set is given by the subsets of a set under the inclusion relation. We enumerate some important definitions.

Definition A.1 A **relation** R on a set S is a subset of $S \times S$.

A set S on which a relation has been defined is called a **related set**. Let x, y be two elements of S, the notation xRy means that (x, y) is an element of R. In that case, x is a **predecessor** of y and y is a **successor** of x. We say that x and y are **related** if xRy or yRx, otherwise they are **unrelated**. The relation is

- **reflexive** if for all x in S, xRx;
- **irreflexive** if for all x in S, (x, x) is not in R;
- **symmetric** if xRy implies yRx;
- **antisymmetric** if xRy and yRx imply that $x = y$;

- **transitive** if xRy and yRz imply that xRz.

An **anti-chain** is a subset of a related set that contains no pair of related elements.

A.2 Partial orders

A partial order is a common structure in computer sciences and in many other areas.

Definition A.2 A **partial order** P on a set S is a relation on S which is irreflexive, antisymmetric and transitive.

The relation of a partial order is often denoted by $<$. Next, we give an example of a partial order that we use in this thesis.

Example A.1 Consider the set of integer vectors $\mathbf{N}^n$. Let V_i be the i^{th} coordinate of a vector V. $\mathbf{N}^n$ is a partial order under the relation $<$ defined as

$$V < V' \text{ if and only if } V_i \leq V_i' \text{ for all } i,$$

where at least one inequality is strict. One can show that the relation $<$ is irreflexive, antisymmetric and transitive. □

A subset σ of a partial order P is a **down-set** if, for any element x in σ, σ contains all the predecessors of x. A **total order** is a partial order where each pair of distinct elements are related.

Definition A.3 A **graph** is a pair $G = (V, E)$ where V is a set called the vertices and $E \subseteq V \times V$ is a set called the edges.

If the pairs in E are ordered, then the graph is **directed**, otherwise it is **undirected**. A vertex v in a directed graph is called a **source** if no edges enter v and a **sink** if no edges leave v. Graphs are represented diagrammatically by a **Hasse diagram**. In such a diagram, each vertex becomes a node and each edge (e, e') becomes an arrow from node e to node e' in the case of a directed graph or a line in the case of an undirected graph. There is a one-to-one correspondence between directed graphs and partial orders.

A **series-parallel** directed graph is defined recursively as follows:

1. a directed graph with a single vertex is series-parallel;

2. if $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are series-parallel then the directed graphs constructed by each of the following rules are also series-parallel:

   (a) parallel composition: $G = (V_1 \bigcup V_2, E_1 \bigcup E_2)$;

   (b) series composition: $G = (V_1 \bigcup V_2, E_1 \bigcup E_2 \bigcup (T_1 \times S_2))$, where T_1 is the set of sinks of G_1 and S_2 is the set of sources of G_2.

We can define series-parallel partial orders in a similar way.

A.3 Lattices

A lattice is a partial order with additional structure. An element z of a partial order P is an **upper bound** of $x, y \in P$ if $x < z$ and $y < z$. The element z is a **least upper bound** or the **join** of x and y if $z \leq t$ for all upper bounds t of x and y. The **lower bound**, the **greatest lower bound** and the **meet** are defined dually. We denote by $x \vee y$ and $x \wedge y$ the join and the meet of x and y respectively. In a partial order P, there is at most one element 1 such that $x \leq 1$ for all $x \in P$. This element, if it exists, is called the **maximal element** of P. Dually, the **minimal element** is the unique element in P, if it exists, such that $x \geq 0$ for all $x \in P$.

Definition A.4 A **lattice** is a non-empty partial order where the meet and the join exist for all pair of distinct elements.

Example A.2 The example partial order in Section A.2 is in fact a lattice. Let V and W be two vectors in $\mathbf{N}^n$. The join and the meet are given component-wise by

$$\begin{aligned} (V \vee W)_i &= max\{V_i, W_i\} \\ (V \vee W)_i &= min\{V_i, W_i\}. \end{aligned}$$

□

A lattice L is **distributive** if the meet and the join operations satisfy:

$$(x \vee y) \wedge z = (x \wedge z) \vee (y \wedge z),$$

for all $x, y, z \in L$. The subsets of a set are an example of a distributed lattice. In that case, the partial order is given by the subset relation, the meet is the set intersection and the join is the set union. It is then easy to verify that the above distributive equation is satisfied.

Consider a lattice L with a minimal element 0 and a maximal element 1. Two elements x, y in L are **complements** if

$$x \vee y = 1 \text{ and } x \wedge y = 0.$$

L is said to be **complemented** if any element x has a complement y.

A **Boolean algebra** is a complemented distributive lattice. The set formed by all the subsets of a finite set is a typical example of a Boolean algebra. The complement of a subset is then simply given by its standard set complement.

Bibliography

[AJF96] P. Ammann, S. Jajodia and P. G. Frankl, Globally Consistent Event Ordering in One-Directional Distributed Environments, IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 6, June 1996, pp. 665-670.

[BR95] Ö. Babaoğlu and M. Raynal, Specification and Verification of Dynamic Properties in Distributed Computations, J. Parallel and Distributed Computing, Vol. 28, 1995, 173-185.

[CB91] B. Charron-Bost, Concerning the size of logical clocks in distributed systems. Information Processing Letters, Vol. 39, July 1991, pp. 11-16.

[CL85] K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, ACM Transactions on Computer Systems, Vol. 3, No. 1, Feb. 1985, pp. 63-75.

[DP90] B. A. Davey and H. A. Priestley, Introduction to Lattices and Order, Cambridge University Press, Cambridge, 1990.

[Du65] R. J. Duffin, Topology of series parallel networks, J. Math. Anal. Appl. 10, pp. 303-318.

[Ep92] D. Eppstein, Parallel Recognition of Series-Parallel Graphs, Information and Computation, No. 98, 1992, 41-55.

[Fi91] C. Fidge, Logical time in distributed systems, IEEE Computers, Aug. 1991, pp. 11-16.

[FR94] E. Fromentin and M. Raynal, Inevitable Global States: a Concept to Detect Properties of Distributed Computations in an Observer Independent Way, technical report No. 2317, INRIA, IRISA, Aug. 1994, 18 pages.

[GW94] V. K. Garg and B. Waldecker, Detection of Weak Unstable Predicates in Distributed Programs, IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 3, Mar. 1994, pp. 299-307.

[GW96] V. K. Garg and B. Waldecker, Detection of Strong Unstable Predicates in Distributed Programs, IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 12, Dec. 1996, pp. 1323-1333.

[HY87] X. He and Y. Yesha, Parallel Recognition and Decomposition of Two Terminal Series Parallel Graphs, Information and Computation, Vol. 75, 1987, pp. 15-38.

[HMRS95] M. Hurfin, M. Mizuno, M. Raynal and M. Singhal, Efficient Distributed Detection of Conjunctions of Local Predicates, technical report No. 2731, INRIA, IRISA, Nov. 1995, 35 pages.

[HPR93] M. Hurfin, N. Plouzeau and M. Raynal, Detecting Atomic Sequences of Predicates in Distributed Computations, In Proc. of the 3^{rd} ACM/ONR Workshop on Parallel and Distributed Debugging, May 1993, pp. 32-42.

[La85] L. Lamport, Time, clock and ordering of events in a distributed computation, Comm. ACM, vol. 21, Jul. 1985, pp. 558-564.

[Li] H. F. Li, personal communication.

[Ma88] F. Mattern, Virtual time and global state of distributed systems, Proc. Parallel and distributed algorithms Conf., (Cosnard, Quinton, Raynal, Robert Eds), North Holland, 1988, pp. 215-226.

[MN91] K. Marzullo and G. Neiger, Detection of Global State Predicates, Lectures Notes in Computer Science, 579, Springer Verlag, 1991, pp. 254-272.

[MC88] B. P. Miller and J.-D. Choi, Breakpoints and Halting in Distributed Programs, In Proc. of the 8^{th} IEEE Int. Conf. on Distributed Computing Systems, June 1988, pp. 316-323.

[TNS82] K. Takamizawa, T. Nishizeki and N. Saito, Linear-Time Computability of Combinatorial Problems on Series Parallel Graphs, J. ACM, Vol. 29, No. 3, July 1982, pp. 623-641.

[TG93] A. I. Tomlinson and V. K. Garg, Detecting Relational Global Predicates in Distributed Systems, In Proc. of the 3^{rd} ACM/ONR Workshop on Parallel and Distributed Debugging, May 1993, pp. 21-31.

[TG97] A. I. Tomlinson, V. K. Garg, Monitoring Functions on Global States of Distributed Programs, Journal of Parallel and Distributed Computing, Vol. 41, 1997, pp. 173-189

[Val93] C. Valot, Characterizing the Accuracy of Distributed Timestamps, In Proc. of the 3^{rd} ACM/ONR Workshop on Parallel and Distributed Debugging, May 1993, pp. 43-52.

[VD95] S. Venkatesan and B. Dathan, Testing and Debugging Distributed Programs Using Global Predicates, IEEE Transactions on Software Engineering. Feb. 1995, pp. 163-177.

[VTL82] J. Valdes, R. E. Tarjan and E. L. Lawler, The recognition of series parallel digraphs, SIAM J. Comput., Vol. 11, 1982, pp. 298-313.