Combating State Explosion in the Detection of Dynamic Properties of Distributed Computations

Richard Achmatowicz

A Thesis presented for the degree of
Doctor of Philosophy

School of Computing Science,
University of Newcastle upon Tyne,
England

July 2005

Abstract

In the context of asynchronous distributed systems, many important applications depend on the ability to check that all observations of the execution of a distributed program, or *distributed computation*, satisfy a desired (or undesired) temporal evolution of states, or *dynamic property*. Examples include the implementation of distributed algorithms, automated testing via oracles, debugging, and building fault-tolerant applications through exception detection and handling.

When a distributed program exhibits a high degree of concurrency, the number of possible observations of an execution can grow exponentially, quickly leading to an explosion in the amount of space and time required to check a dynamic property. In the worst case, detection of such properties may be defeated. This is the run-time counterpart of the well-known *state explosion problem* studied in model checking.

In this thesis, we study the problem of state explosion as it arises in the detection of dynamic properties. In particular, we consider the potential of applying well-known techniques for dealing with state explosion from model checking to the case of dynamic property detection. Significant semantic similarities between the two problems means that there is great potential for deriving techniques for dealing with state explosion in dynamic property detection based on existing model checking techniques. However, differences between the contexts in which model checking and dynamic property detection take place mean that not all approaches to dealing with state explosion in model checking may carry over to the run-time case.

We investigate these similarities and differences and provide the development and analysis of two approaches for combating state explosion in dynamic property detection based on model checking methods: on-the-fly automata theoretic model checking, and partial order reduction.

Acknowledgments

I would like to thank my thesis supervisor, Professor S.K. Shrivastava, for his help and encouragement during the preparation of this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Research Background

A *distributed computing system* is a computing system made up of several processes which may execute concurrently and which communicate with each other via a communication network, where communication is based on message passing.

Many computing systems today are distributed in nature. Distribution arises quite naturally in the need to control, via software, disparate elements of a system which are physically separated and require independent local control. In addition to the need to support the inherent distributed nature of such systems, distributed systems have many advantages over centralized (parallel) systems, in terms of scalability, reliability, low-cost, and modularity, which further account for their widespread use. Examples of distributed systems include computer networks, telecommunication systems, automated manufacturing systems, as well as ATM and point-of-sale systems, electronic auctions, other e-commerce applications.

Two important classes of distributed systems are the *synchronous* and *asynchronous* distributed systems [82]. Informally, a *synchronous* distributed system is one in which there exist bounds on the relative speeds of processors, message delays, and the time it takes a process to execute one step. Synchronous distributed systems represent an important model due to the fact that, in a synchronous system, it is possible to synchronize local clocks and use global time for global reasoning within the system. An *asynchronous* distributed system, on the other hand, is a distributed system in which there are no bounds on the relative speeds of processes and no bounds on message delay. In such systems, synchronization of local clocks is not possible, and message passing is the only means of synchronizing behaviour between processes. However, asynchronous distributed systems represent a realistic model for many distributed systems found in practice.

The development of distributed systems can present serious challenges for the software developer, for several reasons:

- Due to their distributed nature, distributed systems often depend upon specialized distributed algorithms for performing key functional tasks, examples of which include:

- the ability to structure complex algorithms into a sequence of phases, where the end of phase is characterized by some termination condition or condition of stability;
- providing mutual exclusion to shared resources in a distributed environment;
- providing concurrent access to distributed databases via distributed transactions, which may require detecting deadlock;
- providing load-balancing reconfigurations of the system, when load on one or more servers exceeds a prescribed threshold

- Due to the added complexity of distributed systems resulting from the possible interactions between concurrent components, the development of distributed systems depends heavily upon verification and validation activities, such as testing and debugging, to ensure that the distributed systems designed and implemented are indeed correct.
- Due to their ubiquitous nature, distributed systems may often represent systems with stringent reliability requirements, as for example found in mission-critical applications, and so require the provision of features such as fault-tolerance (via exception detection and handling) and security (via intrusion detection).

Each of the important problems cited above in the development of distributed systems includes a step involving the execution of some notification or reaction when the system satisfies some desired (or undesired) temporal evolution of states. For example, a temporal evolution of states can be used to represent correctness conditions for a distributed system under test, global breakpoints identifying an erroneous state in a distributed system, or the description of an exceptional execution in a fault-tolerant system. We call such a desired (or undesired) temporal evolution of the system state a *dynamic property*. We refer to the problem of detecting when a distributed computation (i.e. an execution trace of a distributed system) satisfies a specified dynamic property variously as the *dynamic property detection problem* or, sometimes, as the *trace checking problem*[1]. Given a solution to the dynamic property detection problem, solutions to the important activities defined above may be obtained by specifying an appropriate dynamic property, together with the appropriate notification or reaction. In this way, the dynamic property detection problem represents the basis of solutions to many important problems in distributed computing.

In the context of an asynchronous distributed system, solving the dynamic property detection problem faces two key challenges [83]. Firstly, due to the fact that the state of a distributed system is made up of the local states of the individual processes, and no single process has instantaneous access to these local states, any process in a distributed system wishing to perform such a detection must first construct the global state (or states) of the distributed computation on which to base the detection. Secondly, due to the relative speeds of processes and delays in communication possible in an asynchronous distributed system, different processes involved in the construction of global states for the same distributed computation may arrive at different global states being constructed, leading to differing actions being taken. This is the so-called *relativistic effect* of

[1]These two terms, *dynamic property detection* [5] and *trace checking* [60], have both been used in the literature to describe the problem stated here. Rather than choose one over the other, in the sequel, we shall use these two terms interchangeably.

asynchronous distributed systems in which observations made of a distributed computation are relative to the observer [24].

There has been a considerable amount of research invested into solving the dynamic property detection problem, particularly in the case where properties are specified by simple predicates defined on global state [83].

The detection of *stable* properties was first considered by Chandy and Lamport in [16]. A stable property is one which, once it becomes true in an execution, remains true in that execution. Examples of stable properties include termination and deadlock. Stable properties can be detected using a simple approach: periodically constructing a global state of the distributed system execution and testing the truth of the predicate on that global state. In [16], Chandy and Lamport presented a distributed snapshot algorithm which can be used to construct global states at run-time for the purpose of checking stable properties.

Unlike stable properties, an unstable property is one whose truth value may alternate between true and false throughout an execution. An example of unstable property is "the value of variable x is 1". An approach to detection of unstable properties cannot be based on the simple approach of periodically constructing global states via snapshots, as even repeated application of a snapshot algorithm to construct global states can miss possible global states through which the system passed and in which the predicate evaluates to true. Detection of unstable properties needs to be based upon the construction of observations [24]. An *observation* of a distributed computation is a sequence of global states of the distributed system in which the effect of all events occurring in the distributed computation are represented. Observations may be constructed in the following way: each process participating in the distributed computation sends notification messages to a monitor process, where notification messages describe state changes occurring at each process. On the basis of notification messages received, the monitor then constructs a sequence of global states reflecting all changes taking place at each process in the system. In this way, observations represent all events of the distributed computation and their corresponding state changes.

Unfortunately, the detection of unstable predicates cannot be based on the construction of a *single* observation. In an asynchronous distributed system, due to the relative speeds of processors and the delays in communication, it is not possible to determine the total order in which events on different processes have occurred. The best information which can be obtained in an asynchronous distributed system concerning the relative ordering of events is the ordering resulting from causal dependence between events. The *happened-before* relation [65] is a partial ordering of the events in a distributed computation which reflects causal dependence between events. Each total ordering of the events in the distributed computation which is consistent with the happened before relation represents a *possible* observation of the distributed computation. Each observation determines a corresponding sequence of global states. This state of affairs leads to the situation wherein an unstable property may be true in one observation, and false in another observation of the same distributed computation. In such a way, two monitors checking the same property over the same computation, on the basis of a single observation, could come to differing conclusions about its truth value.

In order to avoid such inconsistencies, detection based on observations needs to be *observation-independent*. In [24], Cooper and Marzullo proposed a formulation of the dynamic property

detection problem in which detection is observation-independent. Observation-independence is achieved by introducing modal operators *Pos* and *Def* to define detection over the *set* of possible observations of the distributed computation. Given a predicate Φ defined on the global state of the system, a distributed computation satisfies $Def\ \Phi$ if and only if *all* possible observations of the distributed computation pass through a state satisfying Φ. Similarly, a distributed computation satisfies $Pos\ \Phi$ if and only if *some* possible observation of the distributed computation passes through a state satisfying Φ. Detection algorithms for detecting $Pos\ \Phi$ and $Def\ \Phi$ are based on recording the happened-before relation on events during system execution and using this relation to construct and explore a structure which contains all possible observations of the distributed computation.

Simple global predicates are able to describe many interesting properties of distributed systems, but they lack any notion of time or relative ordering of events, which is important in the specification of reactive distributed systems. Unlike transformational computing systems, which are designed to transform information in a functional manner and are specified in terms of pre- and post-conditions on initial and final states, respectively, *reactive systems* [70] are designed to maintain a specified relationship with their environment over time. Examples include operating systems and network communication systems. Reactive systems are often designed not to terminate, and are specified by their behaviour over time. Consequently, the specification and detection of *sequence-based* or *temporal* dynamic properties figure highly in the development of reactive distributed systems. Several solutions to the property detection problem for sequence-based or temporal properties have appeared in the literature. In [6], Babaoglu and Raynal presented a language for specifying sequence-based predicates in terms of simple predicates, through simple composition (simple sequences), and composition with interval restrictions (interval-constrained sequences). In [5], Babaoglu, Fromentin and Raynal presented an alternative formulation of temporal properties where properties are specified as formal languages over an appropriate alphabet. Sequence-based properties, like unstable properties, also require detection based on observations and, as such, require detection which is observation-independent. As in the case of unstable predicates, the detection algorithms for such temporal dynamic properties are based on construction and exploration of a structure which contains all possible observations of the distributed computation.

The consequence of the need to make dynamic property detection observation-independent is that all observations of a distributed computation must implicitly be considered in the detection of properties defined by unstable simple predicates or the more general sequence-based predicates. Although the detection algorithms cited previously for the detection of unstable predicates and temporal predicates are linear in the size of the computation state space, the size of the computation state space grows exponentially: if $|S|$ represents the greatest number of events on any single process, and N the number of processes, then the size of the computation state space is $O(|S|^N)$. This exponential growth in the size of the computation state space and its impact on detection of dynamic properties is referred to as the *state explosion problem.* When programs exhibit a high degree of concurrency, this leads to an explosion in the number of possible states and observations which must be considered in order to check the satisfaction of a dynamic property by a distributed computation. In the worst case, the checking of such properties may be defeated.

1.2 Research Motivation

There have been various approaches to dealing with the state explosion problem in the detection of unstable properties. One of the most successful of these are *property-structural* methods, which are based on using the known structure of the property to avoid having to explore all possible observations of the distributed computation. Efficient algorithms have been developed for a number of classes of properties based on simple global predicates, classes such as disjunctive , conjunctive, relational, and others (a survey of such approaches appears in [37]). This approach has been very successful in combating state explosion for these considered classes. Despite the success in providing efficient detection algorithms for these considered classes, this approach to dealing with state explosion suffers from several disadvantages:

1. the approach does not apply to general properties, but is limited in application to properties belonging to the considered classes
2. each property class results in a different algorithm, resulting in a multiplicity of algorithms
3. the results developed using this approach are, in the main, limited to properties described by simple global predicates

We would like an approach to combating state explosion which is applicable to more general predicates, in particular, with more relevance to predicates specified by sequence-based properties.

The state explosion problem has been studied extensively in the context of model checking[19]. In model checking, the problem is to check that all executions of a finite state concurrent program satisfy a specification of desired behaviour, often in the form of a linear temporal logic formula [70]. The basis of one well-known solution is to exhaustively explore all possible executions of the system, based on a depth-first search of the graph representing the program state space, and check that each execution explored satisfies the linear temporal logic formula. This exhaustive analysis approach, combined with the representation of concurrency through interleaving, leads to an explosion in the number of states which has to be considered. Intensive research over the past twenty years has been conducted into developing methods for dealing with the state explosion problem in the context of model checking, and a wide range of techniques have been developed.

From the algorithmic point of view, model checking and dynamic property detection exhibit strong similarities, not only in the semantic representations used to model program behaviours and computations, but also in the required exploration of those representations during analysis. This immediately suggests that any approach to dealing with state explosion in the model checking context might be applicable to dealing with state explosion in the dynamic property detection context.

Although the two problems have several key similarities, there may well also be certain challenges in applying model checking methods to the dynamic property detection context. These challenges arise partly due to the dynamic nature of the run-time environment in which property detection is carried out, and partly due to the limitations of what we can observe at run-time in an asynchronous distributed system. For example, when observing a distributed computation, we

maintain only a prefix of the computation at any given point in time; this may rule out certain approaches to exploring the possible states and observations of the execution, which might otherwise be possible in model checking.

The overall aim of the present research is to determine if techniques for combating state explosion in model checking can be successfully applied to the problem of state explosion in dynamic property detection. In particular, we aim to answer the following questions:

- what are the broad classes of approaches used to address state explosion in model checking and dynamic property detection?
- how do the contexts in which these two problems are carried out affect the feasibility of an approach to combating state explosion?
- is it possible to adapt techniques for combating state explosion used with success in model checking to combat state explosion in dynamic property detection?

1.3 Research Tasks

Based on the above research questions, the following research tasks will be explored in this work:

1. *survey existing approaches to combating state explosion in both model checking and trace checking*
   Surveying state explosion techniques from model checking is required in order to fully understand the key approaches to combating state explosion which are available for potential application. Further, a survey of the approaches already in use in trace checking is required in order to determine if any of these approaches, or similar approaches, are being used already. This task will effectively answer the first of our three research questions.

2. *compare and contrast the contexts and identify promising candidates*
   As mentioned in the previous section, there are significant differences in the context in which model checking algorithms are based upon and trace checking algorithms need to cater to. Such differences may rule out certain algorithmic approaches, so we next need to look in detail at the differences in context. These differences in context can then be used to identify which state explosion techniques from model checking are the most promising candidates for use in a trace checking context. This task effectively answers the second research question.

3. *explore the application of one or more candidate techniques*
   Despite having identified an approach to combating state explosion from model checking as a promising candidate for use in a trace checking context, answering the question of whether model checking techniques can be used successfully can only really be borne out by carrying out the details of the development. This task can be the most informative task of all, in that the development of any complex algorithm can lead to unexpected outcomes. This task will address the third research question.

1.4 Organization

The organization of the thesis is as follows. Chapter 2 aims to further motivate the importance of the dynamic property detection problem and its role as a canonical problem in distributed computing. We first examine the relationship between model checking and dynamic property detection, from the point of view of its application in the software engineering life cycle. In particular, we differentiate between the different times at which property detection may be required. This leads to consideration of several application contexts in which dynamic property detection arises as a canonical problem: the design and implementation of distributed algorithms, validation testing and debugging of distributed programs, and implementing fault-tolerance for distributed programs, based on exception detection and handling. Here, we aim to identify application-specific requirements on solutions to the dynamic property detection problem, which will be important when considering the feasibility of model checking approaches.

Chapter 3 presents a survey of techniques for combating state explosion in model checking. The chapter begins with a supporting discussion of the various fundamental concepts which are required in order to understand the model checking problem, as well as an analysis of how the state explosion problem arises. This is followed by a discussion of the various approaches appearing in the literature for combating state explosion. In presenting and discussing each approach, we shall attempt to focus on the basic idea of the approach, key examples from the literature, and its algorithmic complexity, a measure of the degree to which state explosion is mitigated.

Chapter 4 similarly surveys existing techniques for combating state explosion in the detection of dynamic properties of asynchronous distributed computations. The chapter likewise begins with a supporting discussion of the fundamental concepts of dynamic property detection which are required in order to understand the dynamic property detection problem, as well as an analysis of how the state explosion problem arises. As in the previous chapter, when discussing state explosion techniques, we shall attempt to focus on the key idea underlying each approach, key examples from the literature, and its achieved algorithmic complexity.

Chapter 5 provides a comparative analysis of the techniques for combating state explosion in model checking and trace checking reviewed in the previous two chapters. The chapter begins by considering the differences in context between model checking and trace checking, in order to identify potential areas where model checking algorithms may require adjustment. This is a required ingredient in making a determination as to whether a model checking algorithm may be a successful candidate. Additionally, we compare and contrast the techniques for state space reduction, identifying existing and potential synergies between the two areas which could potentially be exploited. This comparative analysis leads to the identification of two techniques for combating state explosion in model checking which appear especially promising: the on-the-fly automata-theoretic approach, and the partial order reduction approach.

Chapter 6 considers the problem of developing an algorithm for combating state explosion in trace checking based on the on-the-fly automata-theoretic method of model checking. The chapter begins by presenting the necessary background on the theory of on-the-fly automata-theoretic model checking which will be necessary for the development of the algorithm. Design and related issues are then discussed, followed by presentation of the algorithm. In this chapter, based on

an existing algorithm for dynamic property detection for the case of general temporal properties, we present the successful development of an algorithm incorporating state space reduction based on the on-the-fly approach. This chapter illustrates that certain state explosion techniques from model checking can work very well with existing dynamic property detection approaches, with little adjustment required and well suited to the new context.

Chapter 7 considers the problem of developing of an algorithm for combating state explosion in trace checking based on the partial order reduction approach of model checking. Unlike the on-the-fly automata-theoretic approach, whose application to the new context is relatively straightforward, the partial order reduction approach immediately presents several alternatives for application, which are discussed initially. The chapter follows by presenting the necessary background on the theory of partial order reduction which will be necessary for the development of the algorithm. This background is followed by a discussion of the issues which affect the design of an algorithm to perform partial order reduction in a trace checking context, including a discussion of why this approach will not work for certain formulations of the dynamic property detection problem. We follow this discussion with the development of an algorithm for combating state explosion in dynamic property detection based on partial order reduction. This chapter illustrates how certain approaches to combating state explosion in model checking can prove unwieldy when transported to a new context.

In Chapter 8, we summarize the results of the research tasks and present our conclusions on the proposed research question: can model checking techniques be applied to the trace checking context, and if so, which are the promising areas of investigation. We review the results of our applications of model checking techniques, and present our view on avenues for further investigation.

Chapter 2

Applications of Dynamic Property Detection

In this chapter, we aim to explore in more detail the application contexts in which dynamic property detection is required.

In particular, we aim to examine in greater detail how dynamic property detection arises as a key component of a solution to many important problems encountered in distributed systems. To this end, we examine several key problems and, in particular, the dynamic properties required, as well as the notifications or reactions required, in applying dynamic property detection to form solutions to these problems. These considerations will in turn highlight application-specific requirements on the solution to the dynamic property detection problem. Such requirements will have a bearing on the suitability of state explosion approaches which will be considered in the survey and in later chapters.

One important aspect of the application of dynamic property detection concerns the times at which dynamic property detection may be employed: during execution of a program, or after execution of the program has completed. In order to illustrate these differences, we begin by examining in more detail the relationship between model checking and dynamic property detection.

2.1 The Relationship between Model Checking and Trace Checking

Model checking and dynamic property detection both concern determining when a distributed program satisfies a desired (or undesired) temporal evolution of states. We can compare these two activities through the lens of *verification*. There are three times at which we need to verify when a distributed program satisfies a desired (or undesired) property [5]: before execution, during execution, and after execution.

Model checking is a verification technique which is used to check whether all possible executions of a finite state concurrent program satisfy a desired property. This verification generally occurs

before the concurrent program is ever executed. In model checking, both the program and its environment are modeled, and the possible executions of the program, together with its environment, are explored. During exploration, each possible execution of the concurrent program is examined to see if it satisfies the property in question. Thus, model checking is concerned with verifying that *all possible executions of the program* (together with its environment) satisfy the property.

We may also require to verify a property *during* program execution, such as when performing validation testing of a distributed program. *Run-time property detection* involves checking a property concurrent with program execution. Unlike model checking, run-time property detection only makes claims about a single distributed computation of the concurrent program. In this case, run-time property detection is concerned with verifying that *all observations of the distributed computation* satisfy the property.

Finally, property verification may be required *after* a distributed computation has terminated. *Post-mortem analysis* refers to the process of verifying whether a distributed computation satisfies a property, after the program has terminated. Such forms of after-the-fact validation can be required in the analysis of production systems which have exhibited failures and for which the cause of the failure needs to be determined. Post-mortem property detection is similar to run-time property detection in that it checks whether all sequential observations of the distributed computation satisfy the property. It differs from run-time property detection in that this check is performed after execution has terminated. In the sequel, we consider run-time property detection and post-mortem analysis simply as two forms of the dynamic property detection problem.

Babaoglu et al. [5] also note that model checking and dynamic property detection differ in important semantic terms. For example, model checking can answer questions such as "will all executions of this program terminate?", or "will any execution of this program encounter a deadlock?". Dynamic property detection instead answers the questions "has this program execution terminated?" or "has this program execution entered a deadlock state?" (in the case of the run-time variant) or "did this program execution terminate?" or "did this program execution enter a deadlock state?" (in the case of post-mortem analysis). In this sense, dynamic property detection can be viewed as detecting certain *conditions* arising in an execution. This permits using run-time property detection as a basis for applications other than verification and validation.

Indeed, dynamic property detection, in its run-time variant, forms an important part of certain applications based on a *reactive architecture* [49, 118], in which a control component passively monitors a system and takes action when the state of the system is known to satisfy a certain condition. Examples of applications based on a reactive architecture include the implementation of certain distributed algorithms, debugging of distributed programs, and providing fault-tolerance of distributed programs via exception detection and handling, among others. In this sense, dynamic property detection is a more general problem than model checking, and its solution can be subject to a more varied set of application-specific constraints.

In the next section, we explore in more detail the varied applications of dynamic property detection and the application-specific requirements on solutions to the problem which arise from them.

2.2 Applications of Dynamic Property Detection

In this section, we consider several problems in distributed systems for which a core part of any solution involves executing some notification or reaction when the state of the system satisfies some desired (or undesired) temporal evolution of states (i.e. problems for which dynamic property detection represents an important part of a solution).

We consider several contexts: the design and implementation of asynchronous distributed algorithms; the validation of asynchronous distributed programs through testing and debugging; and the provision of fault-tolerance through exception detection and handling for asynchronous distributed programs.

For each of these application contexts, we aim to focus on the following parameters of the dynamic property detection problem:

- the dynamic properties which arise in these contexts, the aspects of the problem they represent, and how they are specified
- the corresponding notifications or reactions which are required
- the modalities of detection which arise in these application contexts and what application-specific requirements they represent
- any other application-specific requirements which influence property detection, including the suitability of run-time or post-mortem detection

These application-specific features will play an important role when we come to consider the application of specific techniques for addressing the state explosion problem in dynamic property detection.

2.2.1 Distributed Algorithm Design and Implementation

In the engineering of complex distributed systems, the implementation of key tasks in the system functionality is often carried out by one or more specially designed algorithms. Examples include structuring algorithms into phases, provision of mutual exclusion to shared resources, detection of deadlock in client-server systems, and load balancing reconfigurations of the system. Such algorithms are employed at various times during the execution of the distributed system, and represent sub-tasks of the overall system functionality.

In the design of such distributed algorithms, we often need to reason about the combined behaviour of the system, as opposed to the behaviour of any one of its component processes. During implementation, such requirements often translate into the need to evaluate a Boolean-valued predicate Φ defined on the global system state during the execution of the system, and then execute some notification or reaction in response to the detection. By the global system state of a distributed system, we refer to the values of all local variables in all processes participating in the computation, including program counters, and the states of all channels used for communication between processes. The detection of dynamic properties specified by such simple predicates defined

on global state represents an important example of the dynamic property detection problem, so important that it has been studied in its own right, and is referred to in the literature as the *global predicate evaluation* (GPE) problem. A detailed development of the concepts and mechanisms underlying global predicate evaluation is presented in [83].

In the following, we illustrate some important examples of the global predicate evaluation problem.

2.2.1.1 End of Phase Detection

When distributed algorithms are structured into phases, there is a requirement to identify the end of one algorithm phase, so that the next algorithm phase may be initiated [16]. Unlike termination, the end of a phase is often not characterized by a lack of activity, but rather a period of stability in the computation, with respect to some defined, phase-specific stability predicate defined on global state. Indeed, during the period of stability at the end of a phase, processes may still change state and exchange messages. However, such conditions are stable, in that once they become true in a system execution, they remain true. In [16], the condition describing the completion of a phase is described by a Boolean-valued predicate on the global state of the system. The detection is required to be made concurrent with algorithm execution, as detecting end of phase forms part of the functionality of the algorithm itself. Thus, run-time property detection, as to the post-mortem variant, is required. The purpose of the notification or reaction is to initiate the start of the next phase and can be achieved, for example, by sending an "end of phase" message to each participating process.

2.2.1.2 Client-Server Distributed Systems and Deadlock

In [83], the authors present an example of how dynamic property detection arises as a requirement for the detection of deadlock in client-server-based distributed systems. A client-server based distributed system is one comprised of *servers*, which provide remote services and *clients* which require those services in order to perform their intended function. Access to services in a distributed system is often provided via remote procedure calls, wherein a client issues a request for a service, naming the desired remote procedure and procedure parameters in the request, and remains blocked until a corresponding reply, containing the procedure result, is returned. In such a system, servers may themselves depend upon the services provided by other servers, and so act as may act as clients. Executions of such systems have the potential to enter a deadlock state: a global system state in which all processes are blocked, each waiting for a reply from another blocked process. In such systems, it is important to be able to detect when the system has entered a deadlocked state, and to take action to resolve the deadlock.

The detection of deadlock can be formulated as a case of global predicate evaluation, by defining a suitable Boolean-valued predicate on global state which characterizes the condition under which deadlock occurs. This can be achieved through the use of a *waits-for-graph*, in which graph nodes model processes and edges model blocking of processes. In this approach to deadlock detection, each server process p_i keeps track of remote procedure calls which have been received from processes p_j, $i \neq j$, but not responded to. Using suitable data structures, such local information can be

used to maintain a global waits-for-graph. The condition characterizing deadlock then becomes to detect the presence of a cycle in the global waits-for-graph, which could be expressed by the predicate Φ = "a cycle exists in the wait-for-graph". Note that this property is also a stable property. The notification or response required in this case is to resolve the deadlock, in some way. This can be achieved, for example, by terminating one or more of the processes involved in the deadlock. As in the case of detecting the end of phase, the detection and resolution of deadlocks is required to be carried out concurrent with program execution.

2.2.1.3 Mutual Exclusion and Token Loss Detection

In distributed systems in which concurrent accesses to shared data are permitted, a *distributed mutual exclusion algorithm* may be employed to guarantee exclusive access to shared resources. When mutual exclusion algorithms are token-based (i.e. based on passing a token around the system, and only the process which holds the token may access the resource), tokens can get lost, and may require being regenerated. This presents a requirement for detecting token loss. The detection is required to be performed concurrently with the execution of the mutual exclusion algorithm. Token loss is a stable property: once the token is lost, it does not reappear until it is regenerated. In formulating this problem as an instance of global predicate evaluation, the global predicate characterizes the condition under which a token is not present in the system: if each process P_i holds a local variable $hasToken_i$, then the required predicate is $hasToken_1 \vee \ldots \vee hasToken_k$, if there are k processes. The notification or response will be to invoke an algorithm for regenerating the token, which may involve simply sending a token to the process which holds the token initially.

2.2.1.4 Load Balancing

As a final example, we now consider a problem in which the property to be represented is not stable.

Consider the problem of detecting the processing load on a collection of k servers in a distributed system. Assume that the local state of each server process S_i records the current processing load on that server in the variable $load_i$. The overall load on the system is then given by the global state predicate $load_1 + \ldots + load_k$. In this context, a simple load balancing strategy could be to detect when the system load exceeds a particular threshold, and then take corrective action by either adding or removing servers. The overall load of each server process and the system as a whole is dynamically changing - the load may temporarily exceed the threshold (when one particular server is receiving a lot of requests), but later fall within the constraints defined by the threshold. The condition of the overall load of the system exceeding, for example, a threshold t is an *unstable* property. The global predicate $\Phi = load_1 + \ldots + load_k > t$ identifies the condition under which the load exceeds an upper threshold (in this case, too few servers). A similar arrangement could be used to define a lower threshold. The notification or reaction in this case is to allocate (or deallocate) one or more servers to the pool of servers handling the processing load, which may depend on the difference between the total load and the threshold.

Summary

The examples above show how many diverse problems in the development of complex distributed systems depend upon the ability to detect when the state of the system satisfies a condition and then execute a notification or reaction when that property is detected.

In the examples above, the required conditions were able to be expressed in terms of a single predicate Φ defined on the global state of the system. It was also seen that some conditions (end of phase detection, deadlock detection, and token loss) are stable, in the sense that once they become true, they remain true. Other properties (such as unbalanced load) do not represent stable properties. Further, the associated notification or reaction was seen to be application-specific.

Given a solution to the dynamic property detection problem, such a solution may form part of the implementation of such complex distributed systems in which these particular problems need to be solved.

Finally, note that special purpose distributed algorithms exist for performing some of these tasks, such as stability detection [53] and deadlock detection [15, 31]. The difference between a general solution as proposed here and the special purpose distributed algorithm will be some measure of efficiency (the special purpose algorithms will be more efficient).

2.2.2 Testing and Debugging Asynchronous Distributed Applications

Testing and debugging are two related verification and validation activities, which together represent an important approach to uncovering and eliminating software design faults[1] in distributed program implementations.

Testing is the process of identifying the presence of software design faults in a software implementation. Debugging is the complimentary process of locating the source of those design faults in the software implementation in order that they may be corrected.

In this section, we aim to examine how dynamic property detection represents a core subproblem of these two important verification and validation activities.

2.2.2.1 Testing Asynchronous Distributed Computations

The purpose of testing is to make a judgment about the quality or acceptability of software with regard to its intended purpose, as defined by a set of requirements, by executing the software under controlled conditions in order to identify the presence of software design faults. Testing can be applied at various stages of software development. It can be applied to individual units or modules of the implementation (*unit testing*), assembled subsystems of the implementation (*integration testing*), or to the entire system *(validation testing)*. Here, we focus on validation testing of distributed programs, where the aim is ultimately to determine if the system, as a whole, satisfies requirements.

[1] According to standard theory on fault-tolerance, a design *fault* is the logical cause of an *error*, which represents an undesired attribute of system state, and is manifested to the user through a *failure*, which is a recognizable deviation of specified behaviour.

Further, we base the discussion around the validation testing of *reactive* systems [70]. Unlike transformational systems, which transform information in a functional manner from one state to another and are specified in terms of pre- and post-conditions on initial and final states respectively, reactive systems are specified by describing their behaviour over time. Indeed, reactive systems are very often designed for the express purpose of maintaining a specified relationship with their environment over time and, as such, are often designed not to terminate.

Reactive systems are often specified using temporal logics. Temporal logics are formal specification logics which were expressly designed for the specification of reactive systems [89]. Temporal logics allow describing individual states and sequences of states which the reactive system must pass through. Temporal sequencing of states is described through the use of modal operators, such as X ("next"), G ("always"), F ("eventually") and $\mathcal{U}$ ("until"), which can be combined in formulae to describe complex conditions occurring in distributed computations. Temporal logics permit the specification of temporal behaviour of reactive systems through the specification of properties: each desired behavioural property is described formally as a formula in the temporal logic. The overall temporal requirements of the system are then characterized by the set of associated formulae. In certain cases, the specification of temporal behaviours may not require the use of the X ("next") operator: such specifications are referred to as being *next-free*[2].

Validation testing is carried out in a two-step approach. In the first step, a *testing methodology* is used to identify a finite set of test cases for the program, where each test case specifies inputs and expected outputs for an execution of the software. In the case of reactive systems, test cases are composed of sequences of inputs, and have an associated sequence of outputs and state transitions. In a second step, a *testing environment* is set up to execute the test cases in a controlled manner. This involves executing each test case in turn, observing the execution behaviour as it progresses, and deciding whether or not the execution satisfies its temporal properties.

Testing complex distributed systems is labour-intensive and error-prone. As a result, tools have been developed [95, 29] to automate the various testing tasks and improve the integrity of the testing exercise itself.

One area where automation is required is in the observation and validation of test executions. Executions need to be observed, and their *actual* temporal behaviour compared against *intended* or specified temporal behaviour. This problem is particularly acute in the case of testing reactive distributed systems. There, testing is complicated by the fact that (i) such systems exhibit many possible behaviours and so in order to obtain adequate "coverage" of the set of possible behaviours, many test cases must be selected and (ii) each test execution requires considering potentially long executions, as reactive systems are not designed to terminate, and relative ordering must be checked throughout such long executions.

Automation of test execution validation can be achieved through the combined use of a test execution monitor and a test oracle [95]. A *test monitor* is employed to collect information about the actual observed execution generated by a test case and store it in a test execution profile. A *test oracle* is a mechanism for determining the behavioural correctness of a test execution. An

[2] An important class of temporal logic formulae, the *next-free* temporal logic formulae, are those which do not involve the "next" operator, X. Informally, next-free formulae do not impose restrictions on the specific successor states which a reactive system may enter from a given current state.

oracle consists of two parts: *oracle information*, which is an encoding of correct behaviour, and an *oracle procedure*, which describes how the behaviour encoded in the oracle information is related to the actual execution. The test oracle is then used to evaluate the data contained in the test execution profile for correctness.

In order to automate test execution validation, we require a means to compare the observed behaviour of the distributed computation recorded in a test execution profile, with the specification, which we assume is initially formulated in a temporal logic. One way to achieve this is to convert the temporal logic specification into a form which is executable. The theory of formal automata [57] provide an executable representation for properties of reactive systems specified in temporal logic. Finite state automata are state machines which are executable and can be used to represent all and only the set of sequences which are described by a formula in a number of temporal logics, including linear temporal logic and graphical interval logic. In [29], Dillon and Yu describe the process of translating reactive system specifications presented in a graphical interval logic into finite state automata.

Based on the above discussion, testing can be seen as an application which contains an important subproblem involving executing some notification or reaction when the state of the system satisfies a desired temporal evolution of states. The subproblem is the problem of test execution validation: the desired temporal evolution of states of the system is represented by the test oracle (information), and the notification or reaction required upon detection is a simple notification either that the test succeeded (the property was satisfied by the observed execution) or the test failed (the property was not satisfied by the observed execution). Note that in testing, we want to determine whether *all* possible observations of the distributed system execution satisfy the temporal property, assuming that the temporal property represents *desired* system behaviour.

Test execution validation may occur at run-time, and so concurrently with the test execution, or post-mortem, based on the information recorded in the test execution profile.

2.2.2.2 Debugging Asynchronous Distributed Computations

Once the presence of a software design fault (i.e. a "bug") has been identified through testing, (i.e. the execution of a chosen test case has produced a distributed computation of the distributed program which does not satisfy the requirements), a *distributed debugging application* or *distributed debugger* is used to locate the presence of the software design fault through controlled execution of the software. A distributed debugger is a software program which aims to provide the user with the ability to control (manage) the execution of the distributed program being debugged. Distributed debugging software is naturally structured as a *reactive architecture*, in which the distributed debugger assumes the role of the control component, and the program being debugged, the role of the environment.

Distributed debugging proceeds in ways similar to those found in traditional debuggers for centralized programs: in particular, by stepping through the execution one step at a time, or by defining *breakpoints* on the state of the distributed system and allowing the execution of the system to proceed up to the first point in the execution where the breakpoint is satisfied and then halting the program in a state in which the breakpoint holds true. Breakpoints represent significant

conditions in the execution of the distributed program being debugged. They are of interest in locating a design fault and its associated error. Many significant conditions can be characterized through the use of appropriately defined predicates on global states of the computation. Unlike centralized systems, the detection of breakpoints and the halting of the distributed computation in a globally consistent state are complicated by the fact that the system is distributed.

Debugging can take place at run-time, concurrent with program execution, or off-line, based on a recorded execution trace. One of the chief difficulties in debugging concurrent with program execution is that distributed system executions are generally not repeatable, due to non-determinism present in the program. Re-executing a distributed program with the same inputs can lead to indeterminate executions, due to the way in which non-determinism is resolved. This effectively prohibits debugging based on a cyclic debugging methodology, wherein a cycle of program execution, examination of program state and program re-execution is performed. Debugging on-line, based on deterministic replay [66] is useful when debugging programs which contain non-determinism. In the absence of a deterministic replay mechanism, in so-called *one-shot* or *trace-based debugging*, debugging must be based solely upon the information is recorded in the execution trace. In both approaches to debugging, however, specification and detection of breakpoints combined with halting are used to inspect program states of interest.

Debugging asynchronous distributed programs can be seen as yet another application which contains an important subproblem involving executing some notification or reaction when the state of the system satisfies a desired temporal evolution of states: the desired temporal evolution of states of the system is represented by the breakpoint specification, and the notification or reaction is to halt the system in a consistent state in which the breakpoint holds, if possible. In fact, much early research into dynamic property detection was motivated by this application.

The aim of specifying a breakpoint is to identify a point of interest in the program execution which may shed light on the source of an error. Breakpoints are specified as debugging progresses, and so breakpoint specifications should be easily specified and understandable. They should also be flexible enough to describe a wide range of conditions of interest in a distributed program execution. Two types of breakpoint specifications have been considered in the literature: Boolean-valued predicates on global states, and sequence-based predicates defined on sequences of global states. Predicates on global state are useful for defining conditions on global state for halting, for the same reasons as in centralized debugging. For example, we may want to halt the program when the global state reaches a point in which all communication queues are empty. The specification and detection of predicates on global state fall within the scope of the global predicate evaluation problem. On the other hand, sequence-based breakpoints can be used to describe interesting conditions involving the relative ordering of program states of interest. For example, we may wish to halt the program execution when a state satisfying the predicate $\Phi_1 \equiv (x == 1)$ is immediately followed by a state satisfying $\Phi_2 \equiv (y == 1)$. The specification and detection of sequence-based properties was considered in [6], where sequence-based properties are specified in terms of simple predicates, simple sequences, and interval-constrained sequences. The specification language results in compact, easily understandable specifications of breakpoints, such as $\Phi_1; [false]\Phi_2$ to represent the predicate mentioned above. Unlike temporal specifications encountered in testing, sequence-based breakpoint specifications are generally not next-free: debugging

may require isolating specific successors of a state which manifest an error.

Unlike testing, where we generally want to check that *all* possible observations of the distributed computation satisfy the specification, there are two possible modalities of detection useful in debugging. When a property specification represents a desired temporal evolution (such as the satisfaction of a global invariant), we want to ensure that *all* observations of the distributed computation satisfy the property. On the other hand, when the property represents an undesired temporal evolution of states (such as the violation of a mutual exclusion condition), we want to check if *some* observation of the distributed computation satisfies the undesired property. The existence of some observation satisfying an undesired property indicates that the bug may be revealed due to the relative speeds of processors.

The notification or reaction to detecting as breakpoint is to halt the distributed computation in a state in which the predicate holds. A discussion of how this is achieved is beyond the scope of this discussion. One solution to this problem in the case of breakpoints defined by predicates on *local* states was presented by Miller and Choi [80] and based on a modification of the snapshot protocol.

2.2.3 Providing Fault-tolerant Asynchronous Distributed Applications

The more we rely on distributed computing systems, the more we require that the systems be *reliable*. Reliability is "the probability of failure free operation of a computer program in a specified environment for a specified period of time" [110], where failure free operation in the context of software is interpreted as adherence to its requirements. Software does not fail due to the degradation of physical components as hardware components do, but due to the activation of software design faults introduced during software development. Fault-removal techniques, such as reviews, analyses, and in particular testing and debugging, are used to attempt to identify and remove software design faults from the distributed program. However, such fault-removal activities are not able to conclusively prove that all software design faults have been eliminated from a software system: they can demonstrate the presence of software design faults, but not prove their absence. This means that some software design faults will remain latent in the final software product, and possibly be activated in response to particular combinations of inputs. Therefore, *software fault-tolerance* is used in order to deal with software design faults which may be activated during execution.

One approach to implementing software fault-tolerance is based on using an *exception mechanism* to alter the normal sequence of control specified in a program when a software design fault is activated. Informally, an *exceptional execution* is an execution of a distributed system for which it is known that, from some point on, that execution cannot satisfy its specification if normal continuation of execution is followed. This often coincides with the system being in an inconsistent state (i.e. a state containing an error, or undesired attribute). When such a determination is made, there is no point in continuing with the normal continuation of execution. An exception mechanism allows interrupting the normal continuation of the program once an exception has been detected, in order to take some form of corrective action. Corrective action is achieved through the use of error recovery techniques, such as *forward error recovery*, and *backward error recovery*.

Forward error recovery is based on the use of redundant data and algorithms which repair the system by analyzing the detected error, and returning the system to a correct state. Backward error recovery, on the other hand, returns the system to a previous error free state without requiring any knowledge of the errors. For example, check-pointing techniques have been used for recovering consistent states.

Software fault tolerance techniques are often tightly integrated with the modular structure of the computer system, and the detection of exceptional executions and initiation of recovery procedures used to provide fault-tolerant system components often reflect this. For example, exception detection and handling have been used to provide fault-tolerance in coordinated atomic actions [92], an object-oriented atomic action paradigm for concurrent systems. The system is structured into atomic actions in which several processes may participate. Therefore, the exact form of specification (and so the means of characterizing exceptional executions) will be dependent on the modular structure of the system in question. However, for the purposes of this discussion, we do not consider the issue of modularity and note simply that temporal specifications will be required in some form.

The provision of fault-tolerance based on exception detection and handling can also be seen as an application which involves executing some notification or reaction when the state of the system satisfies a desired (or undesired) temporal evolution of states. The undesired temporal evolutions of states of the system, in this case, are the exceptional executions of system, and the desired notification or reaction required upon detection is either an initiation of recovery, through forward or backward recovery, or notification of the user. In many cases, included in the set of the exceptional executions of the system are those which violate any global temporal requirements φ, and these can often be specified as the negation of the specification, $\neg\varphi$.

Furthermore, it should be noted that this application of dynamic property detection is restricted to run-time property detection. In the case of error recovery through forward exception handling, which is based on analysis of the exceptional execution, there may be a requirement to provide information concerning the particular exceptional execution, in the form of an error trace. Furthermore, a key element of successfully tolerating design faults is the ability to detect within a reasonable time when the system is in an exceptional execution [27]. Thus, timeliness of detection is an important application-specific requirement.

2.3 Summary

This chapter has presented several important application contexts in which dynamic property detection forms an important sub-problem. We review the key observations:

- *distributed algorithm design and implementation*: many key tasks involve the detection of conditions which may be encoded as global predicates on system state, such as end of phase and token loss. Certain key tasks involve properties which are stable, such as end of phase detection and deadlock; others, such as load balancing, involve detection of properties which are unstable. Detection needs to occur concurrent with system execution.

- *testing asynchronous distributed systems*: the automation of test execution validation requires determining if a test execution satisfies its temporal specification, followed by notification of test success, or production of error trace. Dynamic properties represent test oracle information, in the form of finite state automata derived from temporal properties, often initially specified in a temporal logic. Detection generally needs to establish that all possible observations of the distributed system execution satisfy the required temporal evolution. Depending on whether test execution validation is performed concurrent with test case execution or not, property detection may be required on-line or post-mortem.

- *debugging asynchronous distributed systems*: debugging asynchronous distributed systems requires the specification and detection of breakpoints on global state, which represent significant conditions in the execution. Both simple predicates on global state as well as sequence-based predicates are useful for specifying breakpoints. In the case of identifying breakpoints, detection may require identifying whether some possible observation satisfies the required breakpoint; in the case of checking an invariant, detection may require identifying whether all possible observations satisfy the required invariant. Dynamic property detection may be required on-line or post-mortem, depending on the debugging strategy.

- *providing fault-tolerance*: provision of fault-tolerance requires the specification and detection of exceptional executions. Dynamic properties are used to describe exceptional executions, which may be specified as the negation of temporal specifications on system state. In order to facilitate forward error recovery, an error trace describing the exceptional execution may be required. Detection needs to occur concurrent with system execution, and should be timely, in order to successfully recover from failure.

Each of these important applications involves dynamic property detection as an important subproblem, and so can be defeated by the state explosion problem. This is especially true for temporal properties, such as required in testing, debugging and fault-tolerance.

Beginning with the next chapter, we begin a detailed look into the state explosion problem, how it manifests itself in both model checking and dynamic property detection, and the existing techniques for combating state explosion. Our aim will be to arrive at a position where we can compare these techniques and develop improved techniques for combating state explosion in dynamic property detection, particularly for the case of temporal properties.

Chapter 3

Model Checking and The State Explosion Problem

Specification is the process of describing a system and its desired properties. Verification is the complimentary process: that of analyzing a system to determine if it does indeed satisfy those desired properties.

Temporal logic model checking is a technique for verifying that a concurrent system satisfies its specification. Specifically, given a concurrent system P and a temporal logic formula φ representing a specification, the model checking problem is the problem of checking that all executions of P satisfy the formula φ.

In contrast with other verification techniques, such as simulation, testing and theorem proving, model checking has certain advantages. Unlike simulation and testing, which can only explore a small fraction of the executions of a concurrent system, model checking can determine whether all executions of the system satisfy the specification. And unlike theorem proving, model checking is fully automatic, and can often verify a system in minutes. Model checking also provides an error trace: if the concurrent system does not satisfy the temporal formula, an example execution which does not satisfy the formula can be produced, aiding debugging.

Model checking involves three distinct activities: specification, modeling, and verification.

Specification: Specification involves describing the desired properties of the concurrent system, in a formal language. Temporal logics, such as CTL [22, 20] and LTL [89, 70], are used to specify temporal behaviours of reactive concurrent programs.

Modeling: The modeling phase concerns the construction of a finite state model, or *validation model*, from the description of the concurrent system. The finite state model represents the concurrent system in an abstract way, and should reflect all aspects of the system which are relevant to the specification. Details of the system irrelevant to the property being verified are eliminated from the model through the process of abstraction.

Verification: Verification involves analyzing all possible behaviours of the validation model in order to check that each execution satisfies the specification. If not, the model checking tool provides an example of a system execution which violates the property, which aids debugging.

The main limitation of model checking as a verification technique is the *state explosion problem*, which arises when the system being verified is composed of a number of components which can operate in parallel. The state explosion problem effectively places an upper bound on the size of systems which can be verified using the model checking technique.

In the next section, we briefly present background relevant to the model checking problem, and examine how the state explosion arises in model checking. The remainder of the chapter is devoted to examining approaches to combating state explosion in model checking.

3.1 Fundamentals of Model Checking

Model checking addresses the verification of reactive, concurrent systems. A *concurrent program* is a set of processes $P_1, \ldots, P_n$ which communicate via a communication medium. There are many types of concurrent programs, which vary in their mode of execution (synchronous or asynchronous execution of program steps), mode of communication(via shared variables or message passing), and architecture (centralized or distributed). Concurrent programs can be reactive or transformational. A *reactive* program [70] is a program which maintains a relationship with its environment, by accepting inputs from the environment and reacting to those inputs by performing a local state change and producing outputs to the environment. Reactive programs are thus generally not designed to terminate. Examples include communication protocols and telephone switches. Unlike transformational programs, where behaviours are described in terms of initial and final states, reactive programs are described by their behaviour over time, by the sequence of states they pass through.

Modeling Concurrent Programs

Although there are many formal models of concurrent systems, many are based on the notion of a finite-state transition system. A *(non-deterministic) finite-state transition system* is a tuple $< V, \Sigma, \sigma, v_0 >$ where V is a finite set of states, Σ, a set of actions, $\sigma : V \times \Sigma \rightarrow 2^V$, a transition relation between states, and v_0 an initial state. Transition systems model the states of a system and the possible transitions between states. Each individual process P_i of a concurrent program can be modeled by a finite state transition system, $P_i =< V_i, \Sigma_i, \sigma_i, v_{0i} >$. The concurrent system itself $P = P_1 \| \ldots \| P_n$ is modeled as a composition of the transition systems P_i. The form of composition used depends upon the mode of communication within the concurrent system. For example, concurrent systems $P = P_1 \| \ldots \| P_n$ with synchronous communication can be modeled by the *partly synchronous composition* of transition systems, $P =< V, \Sigma, \sigma, v_0 >$, where $V = \prod_{i=0}^{n} V_i$, $\Sigma = \bigcup_{i=0}^{n} \Sigma_i$,$v_0 = (v_{01}, \ldots, v_{0N})$ and the transition relation σ is such that $(v'_1, \ldots, v'_n) \in \sigma((v_1, \ldots, v_n), a)$ if and only if (i) $v'_i \in \sigma_i(v_i, a)$ for each i such that $a \in \Sigma_i$ and (ii) $v'_i = v_i$ for each i such that $a \notin \Sigma_i$. Concurrent systems $P = P_1 \| \ldots \| P_n$ with asynchronous communication are modeled by introducing a finite-state transition system for each communication channel where transitions of channels are synchronized with their respective processes. The transition system of the system as a whole is formed by taking the partly synchronous composition of the transition systems representing the processes and the channels. Given a concurrent program $P = P_1 \| \ldots \| P_n$, the finite state transition system resulting from the composition of the state transition systems

representing the processes and channels, if any, represents the *global state transition system* of the concurrent program P. We shall refer to this global state transition system for the concurrent program P as the *program state space.*

A *Kripke structure* is a model of the global state transition system of a concurrent program in which states and transitions between states are represented, but process structure is no longer explicitly represented. Atomic propositions are used to characterize aspects of states which are of interest in relation to the specification. Let AP be a set of atomic propositions. Formally, a Kripke structure M over AP is a four-tuple $M = (S, S_0, R, L)$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation that must be total (for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$ and $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state. A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $R(s_i, s_{i+1})$ holds for all $i \geq 0$.

Temporal Logics

Temporal logics have proved successful in the specification of behaviour of reactive, concurrent systems. A propositional temporal logic (PTL) is an extension of propositional logic with temporal operators, such as X("next"), G("always"), F("eventually") and U("until"). Temporal operators permit description of the relative ordering between states in a concurrent program execution. Temporal logic formulae are interpreted over Kripke structures.

Temporal logics can be classified into *linear-time* temporal logics and *branching-time* temporal logics. These two variants differ in the way in which the temporal logic formulae are interpreted over the Kripke structure. In linear-time model, formulae are interpreted over infinite paths in the structure: each path represents an infinite sequence of values from 2^{AP}. In the branching-time model, formulae are interpreted over computation trees, a tree-like structure representing all possible sequences of elements of 2^{AP} starting from an initial state in S_0. Unlike linear time logics, branching time logics are augmented with path operators A("for all") and E("for some") which allow specifying paths within the computation tree for which the formula must hold. In the linear-time model, at each state, only a single next state is presented; in the branching-time model, all possible next states are represented. LTL is an example of a linear-time temporal logic. CTL is an example of a branching time temporal logic.

Properties of reactive concurrent systems can be classified into *safety* properties and *liveness* properties. Informally, a safety property describes execution behaviours in which "bad things don't happen". A liveness property describes execution behaviours in which "good things do happen". For example, in the case of a mutual exclusion protocol designed to ensure access to a shared resource, a "bad thing" would be entering a state of the system in which two processes had access to the resource(violation of mutual exclusion). In the same example, a "good thing" would be that every request for the resource was eventually granted(absence of starvation).

In [3], Alpern and Schneider present automata-theoretic characterizations of safety and liveness. An important result of that paper concerns the nature of the automata required in order to recognize safety and liveness properties. In the case of safety properties, the authors show that, in order to identify violations of safety properties, we need only consider finite prefixes of interleaving sequences. They also show that safety properties can be recognized by prefix closed automata on

finite words. In the case of liveness properties, they reason that the defining characteristic of liveness properties is that, for any finite prefix, there is some continuation of that prefix which will satisfy the property, and that in order to identify violations of liveness properties, we must consider infinite suffixes of interleaving sequences. Summarizing, a safety property can only be violated on finite prefixes of interleaving sequences, whereas liveness properties can only be violated on infinite suffixes of interleaving sequences.

Explicit State Model Checking

The idea behind model checking is to view the global state transition system of a finite state concurrent program as a finite propositional Kripke structure. The *model checking problem* is to determine if all executions of a concurrent program, represented abstractly by a Kripke structure $M = (S, S_0, R, L)$, satisfy the specification of the concurrent program, represented by a temporal logic formula φ.

The first model checking algorithms [91, 22] used the branching-time temporal logic CTL as a specification language. The algorithms were based on exploiting fix-point representations of CTL operators and playing a "labeling game", where the Kripke structure representing the concurrent program is labeled with the sub-formulae of the temporal formula which are true in each state, starting from the atomic propositions and working up. Later, Lichtenstein and Pnueli [68] provided a model checking algorithm for the linear-time logic LTL, based on a tableau procedure. In this case, a certain product graph was created, and the terminal strongly connected components of that product graph checked for reachable acceptance states. The complexity of the algorithm of [68] was shown to be linear in the number of states of the Kripke structure. These model checking algorithms are referred to as explicit state model checking algorithm, as they decide the model checking problem by explicitly examining the states of the global state transition system of the concurrent program.

In these model checking algorithms, the generation of the state transition graph (represented by the Kripke structure) is carried out by exploring all states reachable from the initial state. This is generally performed by a depth-first traversal of the graph representing the program state space, although other graph traversal algorithms, such as breadth-first traversal, may be used as well. Figure 3.1 shows such a depth-first graph traversal algorithm. The algorithm maintains a hash table H of visited states, and a stack *Stack* to assist in managing the recursive organization of the search. It is assumed that the set of possible reachable states is finite, so that the traversal of the state space is guaranteed to terminate. However, the state space can still be unmanageably large: the construction of the state transition graph can suffer from an explosion in the number of states and transitions which need to be considered in order to construct the graph.

State Explosion Problem

The key problem with explicit state model checking is in dealing with large state spaces. Although the complexity of the model checking problem is linear in the size of the state space of the program, represented by the associated Kripke structure, the size of the Kripke structure can be exponential in the size of the concurrent program, represented by the number of processes n, due to the state explosion problem.

```
1   Stack := empty;
2   H := empty;
3   push initial state onto Stack;

4   while (Stack is not empty)
5       pop s from Stack;
6       if (s is not in H) then
7           insert s in H;
8           T := enabled(s);
9           foreach t in T do
10              s' := the state that results from executing t in s;
11              push s' onto Stack;
12          endforeach
13  end while
```

Figure 3.1: Algorithm for depth-first search of full state space

The state explosion problem arises when processes in a concurrent program may make independent transitions in parallel. To see this, consider the following example of a concurrent program $P = P_1 || \ldots || P_n$, where execution is asynchronous and processes do not communicate with each other. The state space of the concurrent program will be $P = P_1 \times \ldots \times P_n$, having size $|P_1| \times \ldots \times |P_n|$. Viewing the size of the concurrent program as the number of processes n, the corresponding size of the state space is $O(m^n)$, where $m = max\{|P_1|, \ldots, |P_n|\}$.

Early model checking algorithms, such as those described by Clarke and Emerson [22] and Lichtenstein and Pnueli [68], made no attempt to deal with state explosion. Their algorithms assumed that the Kripke structure was constructed in a separate stage, and that this Kripke structure, along with the temporal logic formula, were inputs to the model checking algorithm.

The representation of concurrency by interleaving is a major factor in contributing to the size of the state space of a concurrent program. Indeed, the model checking problem for linear temporal logic was shown to be $PSPACE$-complete by Sistla and Clarke [101], and this is due in large measure to the need to explore the state space, which has size exponential in the size of the problem. However, it is not the only such factor. Additionally, the number of variables in each process and the size of the domains of those variables contribute to the size of the state space of a concurrent program, by increasing the size of the potential state space, $P = P_1 \times \ldots \times P_n$. This holds for concurrent programs with synchronous execution, as well as for those with asynchronous execution. As a consequence, techniques to handle large state spaces are required in the model checking of synchronous concurrent programs as well.

The problem of dealing with large state spaces in general, and the state explosion problem in particular, are the key to making the automatic verification technique of model checking applicable to realistic verification problems. In the next section, we survey the broad classes of approaches to dealing with the state explosion problem.

3.2 Techniques for Alleviating State Explosion

In this section, we survey the broad classes of approaches to dealing with the state explosion problem found in the literature. The approaches have been divided into the following categories[1]:

- automata-theoretic methods: based on a formal language approach. Both the set of executions which satisfy the formula, and the set of executions of the concurrent program are viewed as formal languages (i.e. a set of words over a specific alphabet). The model checking problem is then viewed as a decision problem: deciding language containment. This approach has the benefit of providing algorithms which are independent of the specification language and modeling language used.

- symbolic methods: these methods are based on binary decision diagrams, which permit the compact representation of Boolean functions. The methods avoid the explicit enumeration of states by representing the Kripke structure as a Boolean function, and solving the model checking problem by Boolean function manipulation.

- model extraction-based methods: abstraction is used to eliminate irrelevant states and transitions from the state space which have no bearing on the specification. This results in smaller state spaces which need to be checked. This approach is based on abstracting away irrelevant detail.

- partial-order methods: this approach to state explosion aims to use an alternative semantic model (partial order semantic models) in order to avoid having to explore all possible paths through the Kripke structure. This could be summarized as a semantic approach to state explosion.

- distribution-based methods: in this approach, the problem of constructing and exploring the Kripke structure is distributed over a set of processes - a network of workstations connected by a communication network. This approach addresses state explosion by (i) increasing the amount of main memory available, which permits larger models to be explored, and (ii) introducing parallelism, which offers a potential speedup in the exploration of the state space.

In what follows, we consider each approach based on the following criteria: the basic approach to dealing with the state explosion problem, the details of the approach, the resulting improvement in complexity, and the variations of the general approach which appear in the literature.

3.2.1 Automata-theoretic Methods

The automata-theoretic approach to model checking was presented by Vardi and Wolper in [114].

The automata-theoretic approach to model checking is based on the notions of formal languages and finite automata which recognize them. In this approach, the Kripke structure M is viewed as

[1] The list of surveyed approaches does not include *compositional* approaches to combating state explosion. Compositional approaches make use of the fact that many concurrent systems are composed of multiple processes running in parallel, and this known process structure can be used to to address the state explosion problem. We refer the reader to [19, Chapter 12] for discussion of these approaches.

a language generator, the specification, in the form of a linear-time temporal logic formula φ, is viewed as a language acceptor, and the model checking problem is then viewed as the automata-theoretic problem of deciding *language containment*: the problem is to determine if the language generated by the Kripke structure is contained in the language accepted by the formula. The basic automata-theoretic approach in and of itself does not achieve reduction in memory used in the model checking; however, state space reduction can be achieved using the on-the-fly variant of the automata-theoretic approach. In this approach, only the portion of the state space of the program which needs to be explored to check the property is explored, by using the automaton representing the property to guide the search.

In the following sections, we review the key concepts and details of this approach.

Key concepts: The automata-theoretic approach relies on certain results from formal language theory; in particular, upon the ability to represent the set of execution sequences which satisfy the linear-time temporal logic formula φ by a finite state automaton.

Certain formal languages can be represented compactly by finite state automata [57]. For example, *regular languages* are formal languages on finite sequences which can be described by regular expressions, or equivalently, recognized by deterministic finite state automata.

Definition 3.1. A *deterministic finite automaton* (DFA) on finite words is a tuple $A = (Q, \Sigma, \delta, q_0, Q_F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \rightarrow Q$ is a deterministic transition function,
- $q_0 \in Q$ is an initial or starting state, and
- $Q_F \subseteq Q$ is a set of accepting (or final) states.

Let w be a finite word over Σ of length $|w|$. A *run* of A over $w = a_1 a_2 ... a_{|w|}$ is a sequence of states $q_0, q_1, \ldots, q_{|w|}$, where $q_0 \in Q_0$ and $q_i \in \delta(q_{i-1}, a_i)$, for all $1 \leq i \leq |w|$. A run over w is *accepting* if it ends in an accepting state (i.e. $q_{|w|} \in F$). The word w is *accepted* by the automaton A if there is an accepting run of A over w. The *language* of A, denoted by $L(A)$, consists of all the words accepted by A.

Note that the transition function of a deterministic finite automaton requires that every (state,input symbol) combination lead to a state of A. If this condition does not hold, the automaton does not represent a DFA. However, it can be viewed as a non-deterministic finite automaton (NFA), and the well-known *subset construction* used to convert it to a DFA with a *dead* (or *trap*) state: a non-accepting state that leads to itself on every possible input symbol [57].

Regular languages are suitable for specifying safety properties. Alpern and Schneider [3] have shown that safety properties can be characterized by prefix closed finite automata (finite automata in which all states but possibly the dead state are accepting).

ω-regular languages are formal languages on infinite sequences which can be specified by ω-regular expressions, or equivalently, recognized by Buchi automata.

Definition 3.2. A *Buchi automaton* is a tuple $A = (Q, \Sigma, \delta, Q_0, Q_F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $\delta : Q \times \Sigma \rightarrow 2^Q$ is a non-deterministic transition function,
- $Q_0 \subseteq Q$ is a set of starting states, and
- $Q_F \subseteq Q$ is a set of accepting states.

A state $q \in Q$ is *deterministic* if $|\delta(q,a)| \leq 1$ for all $a \in \Sigma$. If $|Q_0| = 1$ and all states are deterministic, then A is said to be deterministic. A *run* of A over an infinite word $w = a_1 a_2 \ldots$, is a sequence $q_0, q_1, \ldots$, where $q_0 \in Q_0$ and $q_i \in \delta(q_{i-1}, a_i)$, for all $i \geq 1$. A run $q_0, q_1, \ldots$ is *accepting* if there is some accepting state of the run which repeats infinitely often (i.e. for some $q \in Q_F$, there are infinitely many i's such that $q_i = q$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The set of infinite words accepted by A is denoted $L(A)$.

Buchi automata are suitable for specifying general safety and liveness properties. They have the advantage of being closed under union, intersection and complementation. However, complementation is expensive: the complementation problem for a Buchi automaton with n states has complexity $O(16^{n^2})$ [100]. The construction for representing the set of execution sequences which satisfy the linear-time temporal logic formula φ as a Buchi automaton is presented in [114] and has complexity $O(2^{3|\varphi|})$, exponential in the length of the formula.

Details of the Approach: In the automata-theoretic approach, the Kripke structure M is viewed as a language generator, generating a language $L(M)$ over the alphabet 2^{AP}. The specification, in the form of a linear-time temporal logic formula φ, is viewed as a language acceptor, accepting a language $L(\varphi)$ of sequences over the same alphabet. The model checking problem is then viewed as the problem of deciding *language containment*: the problem is to determine if $L(M) \subseteq L(\varphi)$. By elementary set theory, the language containment test $L(M) \subseteq L(\varphi)$ is equivalent to testing the relation $L(M) \cap \overline{L(\varphi)} = \emptyset$, where $\overline{L(\varphi)}$ is the complement of the language $L(\varphi)$. In order to perform this test, the connection between formal languages and finite automata is exploited. By defining Buchi automata $A(M)$ and $\overline{A(\varphi)}$ which accept the languages $L(M)$ and $\overline{L(\varphi)}$, respectively, the language containment problem is in turn reduced to checking the non-emptiness of the Buchi automaton, here denoted by $A(M) \cap \overline{A(\varphi)}$, which accepts the intersection of the two languages accepted by $A(M)$ and $\overline{A(\varphi)}$. This automaton is well-defined as Buchi automata are closed under intersection. If the automaton $A(M) \cap \overline{A(\varphi)}$ accepts a sequence, then this sequence represents both a valid program execution and an execution satisfying the negation of the specification (i.e. violating the specification).

Based on this theory, the automata-theoretic model checking procedure involves several steps: (i) computing the automaton $A(M)$ for the language $L(M)$ (ii) computing the automaton $\overline{A(\varphi)}$ for $\overline{L(\varphi)}$ (iii) computing the automaton $A(M) \cap \overline{A(\varphi)}$ representing the intersection of the two languages, and (iv) checking that the automaton $A(M) \cap \overline{A(\varphi)}$ is non-empty. It can be shown that checking the non-emptiness of the automaton $A(M) \cap \overline{A(\varphi)}$ can be reduced to determining if there is a reachable acceptance state of the intersection $A(M) \cap \overline{A(\varphi)}$ which is reachable from itself. This in turn can be achieved by computing the strongly connected components (SCCs) of

the automaton representing the intersection using Tarjan's algorithm [107], and checking if any one of these components contains an acceptance state which is also reachable from the initial state.

Vardi and Wolper cite as advantages of this approach the fact that (i) specification and program are described using the same formalism (formal languages and automata), resulting in an algorithm which is easy to understand, and (ii) the approach can be extended to various logics: all that needs to be provided is a translation from the formula in the new logic to an automaton.

On-the-fly model checking: With respect to state explosion, the basic automata-theoretic approach described above achieves no reduction in memory used, due to the sequential manner in which the steps are performed, and the fact that Tarjan's algorithm requires full exploration of the product state space in order to determine the strongly connected components of the product state space. In particular, the full state space of the program is always constructed, in the initial step. However, if we instead *combine* the exploration of the program state space with the checking of the intersection, so that the state space is constructed *on-the-fly*, substantial reductions in the number of states which are required to be explored can be achieved. In this approach, the Buchi automaton representing the negation of the formula $A(\neg\varphi)$ is computed in an initial step (this approach produces the same language as $\overline{A(\varphi)}$ but is computationally more efficient than computing the complement). Then, viewing both the program $P = P_1 \| \ldots \| P_n$ and the automaton $A(\neg\varphi)$ as finite state transition systems, the synchronous product of the two transition systems is explored. The conditions under which an execution of the synchronous product satisfies both the concurrent program and the negation of the property depend upon the property being verified; that is, the approach varies depending on whether safety properties only are verified, or general safety and liveness properties.

The first application of this approach to dealing with the state explosion problem appeared in [59]. There, the approach was applied to the case of checking safety properties only. Safety properties can be characterized in terms of finite automata on finite sequences. In such a case, we need only find an acceptance state which is reachable from the initial state. In the on-the-fly approach, it is possible to discover a violating execution path, without having had to build the full state space. Worst case complexity is realized when the intersection is indeed empty - i.e. the system is correct - and the full product state space needs to be explored.

The on-the-fly approach was extended to the case of verifying general LTL properties in [26]. General LTL properties are expressed using Buchi automata, and the acceptance conditions for such automata require that we discover reachable acceptance states contained in cycles. There, a 'double' depth first search was used to identify reachable acceptance states which are contained in cycles: a first depth first search is used to locate reachable acceptance states; the second search, initiated from each reachable acceptance state, aims to discover a cycle leading back to the acceptance state within that portion of the state space already explored. In this approach, the secondary depth first searches used to detect cycles must be performed in the order in which backtracking occurs in depth first search (i.e. postorder). This property of depth-first search is an important part of the proof of correctness of the algorithm.

One of the disadvantages of the approach in [26] is that computation of the Buchi automaton representing the property, or property automaton, is still performed in a separate preliminary step. The property automation can have exponential size, of order $2^{O(n)}$, where n is the number of sub formulas in the property formula. In [44], the authors present an algorithm in which both

the property automaton *and* the system model are constructed on-the-fly.

Examples from the literature: The basic automata-theoretic approach is presented in [114], where the discussion focuses on the theoretical foundations of using automata and formal languages to restate the model checking problem as a problem in formal languages. On-the-fly variants of the approach, which focus on achieving a reduction in the size of the state space which needs to be considered in order to check the property, appear in [59] for the case of checking safety properties only, and in [26, 44] for the case of general safety and liveness properties.

On-the-fly model checking has been used with success in the model checking tool *SPIN* [55].

Complexity: The complexity of the basic automata-theoretic approach is $O(|M| \cdot 2^{O(|\varphi|)})$, where $|M|$ is the size of the Kripke structure (in some appropriate encoding) and $|\varphi|$ is the length of the formula (the number of variables and operators). In practice, model checking algorithms based on the automata-theoretic approach face two complexity-related limits:

- the size of the automata, both for the concurrent system and the property, as execution time is proportional to the product of the number of states in the automata
- the size of that part of the product automaton which has to be kept in memory in order to check emptiness, as available memory sets a firm bound on the size of problems that can be treated

Both of these issues are addressed by the on-the-fly variant on the basic automata-theoretic approach. The degree of reduction achieved is however dependent upon both the Kripke structure representing the concurrent program and property being checked.

3.2.2 Symbolic Methods

Symbolic model checking is an example of an approach to model checking in which the states and transitions of the Kripke structure are not explicitly constructed or enumerated. In this approach, Boolean functions are used to encode both system states and transitions between states, effectively replacing the Kripke structure. The model checking algorithm is reduced to a sequence of manipulations of those Boolean functions. This approach alleviates state explosion by not explicitly enumerating the states of the Kripke structure. Symbolic model checking has proved to be very successful in the verification of synchronous and asynchronous circuits: systems with over 10^{120} explicit states have been successfully verified using symbolic model checking [14]. In the following sections, we review the key concepts and details of this approach.

Key Concepts: Let $V = (v_1, \ldots, v_n)$ be a set of Boolean variables. A Boolean function is a function from V to the Boolean values $\{0, 1\}$. Many problems in digital design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on boolean functions [13]. *Ordered binary decision diagrams*(OBDD) are canonical form representations for Boolean functions based on directed acyclic graphs. It is possible to represent Boolean functions using binary decision trees. A *binary decision tree* is a tree with two types of nodes: terminal and non-terminal. Non-terminal nodes are labeled with variable names, and terminal nodes are labeled with the values 0 and 1. A binary decision tree effectively represents the function values for all possible combinations of inputs values. Figure 3.2 shows the binary tree for the Boolean function $f(x_1, x_2) = x_1 \wedge x_2$.

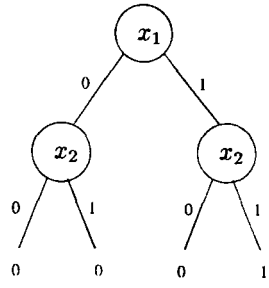


Figure 3.2: Binary decision tree for formula

Binary decision trees have essentially the same size as truth tables, and contain a lot of redundancy. *Binary decision diagrams*(BDDs) are more compact representations, based on directed acyclic graphs, and essentially obtained from binary decision trees by removing redundancy according to certain rules. These rules result in BDDs having canonical form properties and the functions that manipulate them having efficient complexity. OBDDs are a further refinement of BDDs, in that all variable labeling is based on a common order, to minimize blowup of the representation. Given a Boolean function f , the OBDD for f can be used for various purposes: to evaluate the value of function for a given set of inputs, determine satisfaction (test whether there is a set of input values which make the function true), and to determine if two Boolean functions are equivalent. OBDD function representations may be combined using the standard logical operators $(\wedge, \vee, \neg, \Rightarrow)$, and well as first order quantification $(\exists, \forall)$. OBDDs have certain important advantages when it comes to complexity. Brayton has noted that OBDDs do have complexity-related problems. For example, OBDDs need to perform operations such as satisfaction and equivalence, and that these problems are NP-complete and $coNP$-complete. He also notes that there are Boolean formulas for which the OBDD representation is exponential in the size of the formula. Further, the size of an OBDD representation is sensitive to the variable ordering, which must be the same for all functions. The advantages of OBDDs are that (i) most commonly encountered functions have a compact representation (ii) the performance of a program based on OBDDs is bounded by the size of the function graphs (iii) the representation is a canonical form, which makes it easy to determine satisfaction and equivalence. The disadvantages of OBDDs are that they are highly sensitive to variable ordering, and that there are some functions which have an exponential blowup.

Details of the Approach: The symbolic model checking approach is based on two steps: (i) representing the reactive concurrent program using first order logical representations and their OBDD representation, and (ii) reworking the model checking algorithm to process OBDDs instead of the states in the Kripke structure. Using first order logical representations, it is possible to represent the Kripke structure of a concurrent, reactive program in terms of Boolean functions. This requires representing states, sets of states, and transitions between states. States over the Boolean domain can be represented by Boolean variables. Variables over a finite domain D can be represented in terms of a Boolean domain defining a set of Boolean variables $x_1, \ldots, x_n$ and mapping Boolean values from $\{0,1\}^n$ to D. n-ary relations Q on the domain $D \times \ldots \times D$ (n times) can be represented by the characteristic function $f_Q(x_1, \ldots, x_n) = 1$ if and only if $Q(x_1, \ldots, x_n)$ holds true. Transition relations $R(s, s')$ require representing the initial state of the transition s

together with the final state s'. This is achieved by introducing a new set of variables $x'_1, \ldots, x'_n$ and describing the transitions by the conjunction of the description of the initial state and the final state. Transition relations over all states are represented by a disjunction of individual transitions. Having translated the description of the system (program text) into first order representations. OBDDs can be used to encode those relations and we thus have a representation of the concurrent program in terms of OBDDs. The OBDD representing the system as a whole is constructed by applying OBDD operators to combine the various high-level elements of the system, represented as OBDDs.

The model checking algorithm now operates on OBDDs instead of a Kripke structure. In the symbolic model checking algorithm presented in [19, Chapter 6], CTL is used for symbolic model checking as it permits the characterization of temporal logic operators in terms of fix points. These fix point operators in turn can be expressed in terms of set-based operations on OBDDs. The model checking algorithm uses a procedure called $Check(f)$ which returns the OBDD that represents exactly the states of the system that satisfy the CTL formula f . The function $Check()$ is defined inductively, on the syntax of CTL:

- if $f = a$ where a is an atomic proposition, then $Check(f)$ returns the OBDD representing the set of states satisfying a
- if $f = \neg f'$ or $f = f' \wedge f''$, then $Check(f)$ returns the OBDD resulting from applying the OBDD manipulation function $Apply(obdd_{f'}, \neg)$ and $Apply(obdd_{f'}, obdd_{f''}, \wedge)$

In the case of temporal operators, $Check()$ is defined in terms of operations which compute the least (or greatest) fix point of the corresponding least (or greatest) fix-point representation of the operator in question.

Examples from the literature: The first implementation of a symbolic model checker was the SMV system presented by McMillan [78]. Burch et al. [14] present a success story for symbolic model checking in which they successfully model check an asynchronous circuit having more than 10^{120} explicit states. Alur [4] has investigated the combination of symbolic model checking with partial order reduction.

Complexity: There are a number of points to make regarding the complexity of the symbolic approach. Symbolic methods alleviate the state explosion problem by not explicitly constructing the Kripke structure corresponding to the concurrent, reactive system in order to perform the verification, but instead represent the states and transitions between states implicitly, using Boolean functions. The Boolean functions are in turn represented by OBDDs and manipulated by OBDD operations which are relatively efficient. However, as Bryant explains, it is sometimes not possible to entirely avoid representations which are exponential in size.

Thus, symbolic model checking does not *eliminate* the problem of dealing with large state spaces, but *alleviates* it in many cases, allowing larger systems to be checked. Two cases where state explosion (now in the form of exponentially sized OBDD representations) still does arise in symbolic model checking:

- The BDD representing the transition relation can still be exponential in the size of the system, as shown by Clarke [19, Chapter 6]. This necessitated the use of special techniques, such

as partitioned transition relations and lazy parallel composition, in the case of synchronous systems to avoid exponential blowup

- Alur [4] investigated addressing this problem in the case of symbolic approaches to reachability analysis. He noted that symbolic verification of the leader election protocol runs out of memory (with $N = 5$ processes) . This motivated the development of partial order reduction techniques which could be used in combination with symbolic model checking, in order to reduce the size of memory used.

3.2.3 Model extraction-based Methods

The model extraction-based approach is based on the view that the key to applying model checking to large systems is not to build clever model *checkers*, but to build clever model ***builders***.

A key step in model checking is the modeling phase: constructing the finite state validation model from the description of the concurrent system (code or design). If it is possible to construct a smaller Kripke structure which is equivalent in some sense to the full Kripke structure, and if the formulae which are to be verified are insensitive to this equivalence, then we may use verification of the smaller model in order to reason about properties in the full model. We refer to these methods for dealing with state explosion as *model extraction* methods, as they are applied during the modeling phase, and before model checking actually takes place.

There are three key principles for reducing the size of models [25]:

- *irrelevant component elimination*: some of the program components (e.g. statements, variables, processes) may not be relevant to the property being verified. Such components may be safely eliminated from the model before verification begins. Techniques related here are slicing and cone of influence reduction.

- *data abstraction*: some variables might be recording more detail than necessary for the property being verified. By reducing the size of the variable domains, we can reduce the size of the potential state space which needs to be explored.

- *component restriction*: if irrelevant component elimination and data abstraction still do not result in a small model, we can restrict the full generality of the program and consider a restricted set of execution behaviours.

In the following sections, we review the key concepts and details of this approach.

Key concepts: *Program slicing* is a technique used to achieve irrelevant component elimination when the system is presented in terms of program text. A *program slice* P_{slice} is an executable program which is obtained from an original program P through the deletion of zero or more statements. A *slicing criterion* is a pair (n, V) where n is a program statement and V is a subset of the program's variables. A *slice* with respect to the slicing criterion satisfies the following property: whenever P halts for a given input, the slice P_{slice} also halts for that input, producing the same values for the variables in V whenever the statement n is executed. Slices were introduced in [117], where sequential, transformational programs were studied. Slices can be classified as *static* or *dynamic*. A *static* slice is one which makes no assumptions about the program's input; a *dynamic* slice is one which pertains to some specific execution, as, for example, determined by

a test case during program testing. In dynamic slicing, only those dependencies which occur in a specific execution are taken into account. Program slices are constructed by considering the *control dependencies* and *data dependencies* between statements, as well as control predicates present in the program. Control and data dependencies are defined in terms of the control flow graph (CFG) representing the program. The dependencies are recorded in a program dependency graph (PDG) and used by the slicing algorithm. The slicing algorithm aims to remove those program elements which do not affect the computation at the criterion statements. For a survey of slicing approaches, see [108].

The program slicing approach has been extended to reactive, concurrent programs written in Java in [25], with the aim of producing reduced models for model checking LTL formula. Given an LTL formula, a slicing criterion consisting of a set of program statements $\{s_1, \ldots, s_n\}$ is produced, based on considering the atomic propositions of the temporal logic formula and those statements of the program which can affect the atomic propositions. The correctness of the slicing criterion is adjusted to refer to the satisfaction of the temporal formula, and involves the definition of an equivalence between the program P and its slice P_{slice} based on bisimilarity [50].

The *cone of influence reduction* [19, Chapter 13] is another technique for achieving irrelevant component elimination, which is applicable to the verification of synchronous circuits (i.e. where the system description is presented as a synchronous transformation of state variables). Suppose we are given a circuit whose design can be modeled by a set of simultaneous equations $v_i = f_i(V)$, $1 \leq i \leq n$. Given a set of variables V, the *cone of influence* is a minimal set of variables C such that (i) $V \subseteq C$ and (ii) if for some $v_l \in C$ its f_l depends on v_j, then $v_j \in C$. A reduced system is constructed by removing all equations whose left hand side variables do not appear in the cone of influence, C. It can be shown that the reduced system is bisimilar to the original system, and that a CTL^* formula holds for the original system iff it holds in the reduced system.

Data abstraction involves defining a mapping between a potentially large set of variable values and a small set of abstract values. The abstraction mapping is then extended to states of the system and the transitions between states of the system, resulting in a structure which 'simulates' the original system in some sense and can be considerably smaller. If the simulation relation can be shown to preserve the properties defined by specification, the smaller abstract version of the system may be used for analysis. The data abstraction approach can be used when the specification makes reference to the *properties* of data values, as opposed to the values themselves. For example, if the system contains a variable x over a domain D, and the specification only references the atomic propositions $x > 0$, $x == 0$, and $x < 0$, the (potentially infinite) domain D could be replaced by the abstract domain $A = \{a_{neg}, a_0, a_{pos}\}$. Clarke et al. [19, Chapter 13] describe transformations required for implementing data abstraction in terms of Kripke structures generated during model extraction. Informally, the transformation is based on the defining a mapping between actual data values D in the system and and a small set A of abstract values, using this relation to update the labels of the Kripke structure to reflect abstract versions of the propositions (e.g. $x = d$ is replaced by $\hat{x} = a$, where $\hat{x}$ denotes an abstracted version of the variable x), identifying all resulting abstract states in the resulting structure which have the same labeling, and finally ensuring that each transition between actual states in the original Kripke structure is represented with a corresponding transition between abstract states in the reduced Kripke structure. The abstract system M_{abs} obtained by their transformation is shown to simulate the original system

M: the simulation relation guarantees that every possible execution of the original system M will be a possible execution of the abstract version of the system M_{abs}. This permits a verification approach based on the simulation relation: if a property φ holds for the abstract system M_{abs}, then the property also holds for M. The converse relation does not hold: any error trace for M_{abs} need not be an error trace of M. The data abstraction approach has also been applied to the case of the verification of concurrent programs written in Java [25]. The transformation is based on first describing programs abstractly using flow chart language (FCL), and then applying parameterized transformations which encode the mapping from concrete to abstract data values. A similar notion of correctness is used, which is based on an equivalence notion. The result of the transformation is an abstract version P_{abs} of the original program, which has the property that it contains all executions of the original system. The formal proof of correctness is described in [51].
Examples from the literature: The Bandera tool [25] enables the automatic extraction of safe, compact, finite state models from program source code using the principles of irrelevant component elimination, data abstraction, and component restriction. Bandera takes as input a Java program and generates a validation model corresponding to the program in the input language of one of several existing verification tools. The approach is based upon using the property to eliminate (via program slicing) irrelevant components of the program from the validation model, as well as (via data abstraction) adjust the range of data types to smaller domains. This creates a reduced model of the program: the specification holds for the program only if the specification holds in the reduced version of the program. The authors prove that the reduced model is correct for LTL-based specifications, using a theory of program dependencies [50]. Clarke has also reported use of the the data abstraction approach for sequential circuit verification, in [23, 19] .

Complexity: We want to consider the power of model extraction techniques in reducing the size of state spaces. In the case of irrelevant component elimination via slicing, the effectiveness of slicing for reducing the size of program models varies, depending on the structure of the program: when program components are tightly coupled, or where large sections of the program are relevant to the specification, the slicing reduction is only moderate. In the case of data abstraction, the technique can not only be used to reduce the size of large, finite state models, but also, in the case of programs with infinite states, such as Java programs, it has the potential to create a finite state model from an *infinite* state space.

3.2.4 Partial Order-Based Methods

Partial order-based methods are based on the observation that the execution of a concurrent program defines a partial order: events occurring within an execution of a concurrent system may be unordered with respect to each other. This state of affairs arises due to the fact that transitions in a concurrent program may depend only on a subset of local states, and so can be independent (in a precise sense) of other transitions, and so execute concurrently.

When concurrency is represented by interleaving, as in the interleaving semantics model, concurrent system behaviours are represented by interleaving sequences, in which the set of events occurring in an execution is totally ordered. Interleaving semantics models concurrency *implicitly* by representing all possible interleavings of concurrent events. Partial order semantic models for concurrent programs represent concurrency *explicitly*, by defining only a partial ordering on

events. Two events are concurrent if they are unordered with respect to the partial order. The advantage of using a partial order semantic model lies in the fact that the Kripke structure, which contains all possible interleaving sequences of the program and whose construction is the source of the state explosion problem, need not be constructed.

Partial order methods exploit these facts in order to combat state explosion in model checking. In this section, we consider two such methods: *partial order reduction*, and *the method of unfoldings*. Both of these methods are based on the view of concurrent system executions as partial orders.

Partial order reduction is based on the observation that interleaving sequences corresponding to the same concurrent system execution contain related information, and that it may not be necessary to explore all interleaving sequences in each concurrent system execution in order to check a property. The approach is based on generating a reduced state space which is guaranteed to contain enough interleaving sequences in order to check the property. This reduced state space can often be considerably smaller than the full state space, and may be used with existing model checking algorithms to check the property in question. Although based on the ideas of a partial order semantic model (known as trace semantics), partial order reduction methods use the interleaving semantic model.

In *the method of unfoldings*, concurrent system executions are represented *explicitly* as partial orders. An *unfolding* is a (potentially) exponentially compact (as compared to interleaving semantics) representation of a concurrent system's behaviour. Unfoldings are generally infinite, and unsuitable as a basis for model checking algorithms. However, in the case of finite state concurrent programs, it is possible to construct a finite prefix of the unfolding which is guaranteed to contain all reachable states of the program. This opens up the possibility for efficient model checking algorithms which operate on this compact representation.

In this section, we review the key concepts and details of these methods.

Key concepts: *Semantic models for concurrent programs:* Semantic models for concurrent programs differ in the way they represent *non-deterministic choice* (alternative continuations) and *concurrency*. *Interleaving semantic models* model concurrency by interleaving: that is, concurrency is represented implicitly. In interleaving semantic models, conceptually, only one transition is active at any point in time and global states are therefore represented explicitly. Interleaving semantic models exist in two forms: interleaving sequences, and computation trees. Interleaving sequences model concurrent system behaviours as a total order of global states (or transition instances). In this model, possible non-deterministic choices between transitions enabled in the same global state, leading to alternative continuations of the execution, are not represented: each state reached has one unique successor state. In this model, a concurrent system execution from a given initial state is represented by a set of interleaving sequences. *Computation trees [22]* are another interleaving semantic model, in which possible non-deterministic choice of transitions is represented. Given a Kripke structure $M = (S, S_0, R, L)$, a *computation tree* with root labeled with $s_0 \in S_0$ is an infinite tree structure, whose nodes are labeled with states of M, such that (i) the root node n_0 of the tree is labeled with s_0, and (ii) $n \rightarrow n'$ in the tree if and only if $(s, s') \in R$, where s and s' in S are the labels of nodes n and n' respectively. A computation tree effectively collects together all interleaving sequences from a given initial state, and so describes the possible non-deterministic choices which may be made from a given state in the tree. In this model, all con-

Table 3.1: Classification of semantic models

	non-branching	**branching**
non-explicit concurrency	interleaving sequences	computation trees
explicit concurrency	Mazurkiewicz traces, pomsets, occurrence nets	branching processes, event structures

current system executions from a given initial state are represented by a single computation tree. Interleaving sequences could be referred to as *linear-time* interleaving semantics, and computation trees as *branching-time* interleaving semantics, given their use as interpretations for linear-time and branching-time temporal logics, respectively.

Unlike interleaving semantic models, partial order models of computation represent concurrency explicitly by defining only a partial order between events (or local states). Any two events (or local states) which are not ordered by the partial order are viewed as being concurrent. Partial order semantic models for concurrent programs have several advantages over interleaving semantic models: (i) they explicitly represent the inherent concurrency between events in a concurrent system execution, and so can be viewed as being more expressive, and (ii) they are potentially exponentially more compact than their interleaving counterparts. For these reasons, various partial order semantic models have been considered in the literature, including pomsets [90], Mazurkiewicz traces [76], occurrence nets [94], event structures [85] and branching processes [32]. Of these partial order models, some explicitly represent non-deterministic choice, and some do not. Table 3.1 illustrates the various possibilities for representing concurrency and non-determinism in a semantic model for concurrent programs.

Trace semantics: Trace semantics [76] is a partial order-based semantic model for concurrent programs. The semantic model is based on defining an equivalence relation on interleaving sequences. Traces are equivalence classes of interleaving sequences under this equivalence relation. The equivalence is based on the notion of *dependence relation* between transitions of the concurrent system, which is in turn based on the notion of independence between transitions. Let T represent the set of transitions of a concurrent program P. An anti-reflexive, symmetric relation $I \subseteq T \times T$ is an *independence relation* if for any $(t, t') \in I$ and any state s, (i) if $t \in enabled(s)$, then $t' \in enabled(t(s))$ (i.e. t and t' do not enable or disable each other at s) and (ii) if $t, t' \in enabled(s)$, then $t(t'(s)) = t'(t(s))$ (i.e. t and t' commute). The *dependence relation* D of T is the complement of the independence relation I: $D = (T \times T) \setminus I$. The dependence relation is a reflexive, symmetric relation on T, and so can be used to define an equivalence relation on the set of finite interleaving sequences of the program (where interleaving sequences are represented by finite strings in T^*): two paths $w, w' \in T^*$ are equivalent iff one can be obtained from the other by a finite number of permutations of adjacent, independent transitions. A *trace* is an equivalence class of paths, and is denoted by $[w]$, where w is any path in the trace. For any two paths in a trace, they contain the same transitions and lead to the same final state. The notion of traces has been extended to infinite paths [64]. A *run* of a concurrent program P is a trace that contains maximal interleaving sequences of P . Thus, the run is finite if and only if each one of its sequences cannot be extended by another operation. Trace semantics effectively partition the set of all maximal interleaving

sequences into runs, where each run represents a concurrent system execution: the set of events fired in that execution, and all possible equivalent paths containing those events.

Unfoldings: An *unfolding* [32] is a structure which represents, in a unique way, the behaviour of a concurrent program, modeled as a Petri Net, in terms of local states and transitions. Unfoldings represent non-deterministic choice and concurrency explicitly. We should like to convey *informally* the essential features of unfoldings and how they represent behaviour of concurrent programs, but it seems difficult without depending on at least some terminology related to Petri Nets. We therefore review some basic terminology. The definitions presented here are from Esparza [62].

A *net* (P,T,W) consists of a set of *places* P, a set of *transitions* T, and a function W where $W : (P \times T) \cup (T \times P) \rightarrow \{0,1\}$. Elements of the set $P \cup T$ are referred to as *nodes*. When $W(x,y) = 1$, this represents an *arc* from node x to node y. A net can be viewed as a directed graph, where the verticies are represented by nodes and edges are represented by arcs. A *path* through the net (viewed as a directed graph) is a non-empty sequences of nodes such that, for each node, there is an arc leading to the following node (if one exists). The *preset* of a node x, denoted ${}^\bullet x$, is the set $\{y \in P \cup T | W(y,x) = 1\}$. Similarly, the *postset* of a node x, denoted $x^\bullet$, is the set $\{y \in P \cup T | W(x,y) = 1\}$. A *marking* of a net (P,T,W) is a mapping $M : P \rightarrow N$ where N is the set of natural numbers. A tuple (P,T,W,M_0) is a *net system* if (P,T,W) is a net, and M_0 is a marking of (P,T,W). M_0 is referred to as the initial marking of (P,T,F,M_0). In what follows, we shall require the use of nets with associated labellings. A *labeled net* is a pair (N,l) where N is a net, and l is a labeling function which maps nodes of N to some set of labels. Informally, a net can be used to represent the local states and transitions of a concurrent system. Markings represent possible global states of the net. A net system represents a concurrent system which additionally has a specified initial state, represented by the initial marking.

The partial order semantics of net systems are based on *occurrence nets* and *branching processes*. Informally, occurrence nets are a restricted form of the general net defined above, used as a basis for the partial order representation of execution behaviour of a net system. Branching processes, in turn, are occurrence nets paired with a mapping which ties the occurrence net to a specific net system. Before introducing occurrence nets and branching processes formally, we introduce some useful terminology. Again, viewing a net as a directed graph, we may define a number of key relations between nodes in the net. For $x,y \in P \cup T$, x *causally precedes* y, denoted $x < y$, if there is a directed path from x to y in $(P \cup T, W)$. The reflexive, transitive closure of the relation $<$ on nodes is denoted by $\leq$. For $x_1, x_2 \in P \cup T$, x_1 and x_2 are *in conflict*, denoted $x\#y$, if there exist distinct transitions $t_1, t_2 \in T$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, and $t_1 < x_1$ and $t_2 < x_2$. For $x \in P \cup T$, x is in self-conflict if $x\#x$. For $x,y \in P \cup T$, x is *concurrent* with y, denoted $x\,co\,y$, if neither $x < y$ nor $y < x$ nor $x\#y$. These relations identify the way in which causality, conflict and concurrency are explicitly represented in this partial order view of semantics.

An *occurrence net* is a net $O = (B,E,F)$ satisfying the following conditions: (i) for every $b \in B$, $|{}^\bullet b| \leq 1$, (ii) O is acyclic (the directed graph corresponding to the net is acyclic, or equivalently, the causal precedence relation $<$ is a partial order) (iii) O is finitely preceded (every node in the net has finitely many causal predecessors) and (iv) no transition $t \in T$ is in self-conflict. The elements of B are referred to as *conditions*, and the elements of E are referred to as *events*. Occurrence nets define a partial order on the elements $B \cup E$. It can be shown that, for any two nodes x,y of the occurrence net, x and y are either causally related, concurrent, or in conflict[62]. In this

way, occurrence nets explicitly represent the causality, conflict and concurrency relations between the nodes B and E. Informally, when using occurrence nets as a basis for the representation of the execution behaviour of a net, the elements of B are interpreted as local state instances, and elements of E as transition firings.

Branching processes allow associating the execution behaviour represented by an occurrence net with a specific net system. Let $\Sigma = (P, T, W, M_0)$ be a net system. A ***branching process*** of Σ is a labeled occurrence net $(O, p) = (B, E, F, p)$ where the labeling function p satisfies the following conditions: (i) $p(B) \subseteq P$ and $p(E) \subseteq T$ (ii) for every $e \in E$, the restriction of p to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet p(e)$, and similarly for $e^\bullet$ and $p(e)^\bullet$ (iii) the restriction of p to $Min(O)$ is a bijection between $Min(O)$ and M_0 (iv) for every $e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $p(e_1) = p(e_2)$, then $e_1 = e_2$. Branching processes of Σ are occurrence nets whose conditions and events are labeled by the places and transitions of Σ. A branching process can be used to represent one or more executions of the net system, either in part or in full.

For the purposes of verification, we are interested in obtaining a branching process which represents all possible executions of the net. In order to ensure that all possible executions are represented, an ordering relation is defined on branching processes, describing when one branching process contains another. Let $\beta' = (O', p')$ and $\beta = (O, p)$ be two branching processes of a net system. β' is a *prefix* of β if O' is a subnet of O satisfying (i) $Min(O)$ belongs to O'(ii) if a condition b belongs to O', then its input event $e \in {}^\bullet b$ in O also belongs to $O'$and (iii) if an event e belongs to O', then its input and output conditions ${}^\bullet e \cup e^\bullet$ in O also belong to O' and p' is the restriction of p to O'. The *unfolding* of a net system is a branching process which is *maximal* with respect to the prefix ordering relation on branching processes. It is unique up to isomorphism, and represents the *complete* behaviour of a concurrent program, in terms of local states and events [32]. As noted earlier, the occurrence net associated with the branching process $(O, p) = (B, E, F, p)$ represents local state instances B and events E of the net system, and the causality, conflict and concurrency relations between them. In particular, the causal precedence relation defines a partial order. In this sense, the unfolding of a net system is a partial order representation of behaviour of the net system (concurrent system).

Given an unfolding of a net system, which represents all possible execution behaviours of the net system in a single structure, a means of representing individual concurrent executions is required. A *configuration* C of a branching process is a subset of events satisfying the conditions (i) $e \in C \Rightarrow \forall e' \leq e : e' \in C$ and (ii) $\forall e, e' \in C : \neg(e \# e')$; that is, a configuration of a branching process is a set of events which is closed under the causality relation, and contains no conflicting events. Configurations represent partial order views of prefixes of concurrent system executions. Maximal configurations correspond to runs (concurrent system executions).

3.2.4.1 Partial Order Reduction Methods

Partial order reduction methods are based upon using a modified search of the program state space, known as a *selective search*, to generate a *reduced state space* which has fewer states and transitions than the full program state space. In a selective search, for each state s reached during the search, instead of exploring the set of all transitions enabled at s, *enabled*(s), only a subset of transitions is explored. The subset of transitions explored is chosen carefully, so as to guarantee that the reduced state space satisfies the desired property if and only if the full state space satisfies

the property. The reduced state space is then used for verification of the property, using standard model checking algorithms.

Various treatments of this approach have been investigated in the literature, and vary in the way in which the subset of transitions is computed and the class of properties to which the method is applicable. These include the *persistent set* method [45, 46], the *stubborn set* method [112, 113], and the *ample set* method [86].

The approach is loosely based upon the notion of trace semantics; in particular, the observation that interleaving sequences corresponding to the same concurrent system execution (i.e. a trace) contain related information, and that when the property to be verified is insensitive to this equivalence, not all interleavings corresponding to a concurrent system execution need be explored. Although inspired by this notion, Godefroid notes in [46] that in some cases (e.g. the detection of deadlock states), the reduced state space need not contain at least one interleaving sequence for every possible concurrent execution of the program.

In the following sections, we review the key concepts and details of the approach.

Key details of the approach: We illustrate the approach with the basic persistent set selective search method of Godefroid, presented in [45], which aims to create a reduced state space sufficient for detecting deadlock states. The reduced state space search is based on a modified state space exploration algorithm in which only a persistent set of transitions, denoted by $ps(s)$, is explored. Informally, a set of transitions $ps(s) \subseteq enabled(s)$ is *persistent* at s if when firing any finite sequence of transitions from $T \backslash ps(s)$ to reach a state s', the transitions in $ps(s)$ will still be enabled in s'. More formally, a persistent set is defined as follows:

Definition 3.3. A set T of transitions enabled in a state s is *persistent* in s iff, for all non-empty sequences of transitions $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ from s in the full state space and including only transitions $t_i \notin T$, $1 \leq i < n$, t_n is independent in s_n with all transitions in T.

Figure 3.3 shows a modified state space search based on computing and exploring only a persistent set of transitions at each state reached.

```
1   Stack := empty;
2   H := empty;
3   push initial state onto Stack;

4   while (Stack is not empty)
5       pop s from Stack;
6       if (s is not in H) then
7           insert s in H;
8           T := ps(s);
9           foreach t in T do
10              s' := the state that results from executing t in s;
11              push s' onto Stack;
12          endforeach
13  end while
```

Figure 3.3: Algorithm for depth-first persistent set search

Using this fact, Godefroid showed that, by performing a reduced state space search based on

persistent sets, any deadlock state reachable in the full state space will also be reachable in the reduced state space. The proof is based on the observation that, on a path of length k from a state s to a deadlock state d, if only transitions from outside the persistent set $ps(s)$ are fired, then all transitions in $ps(s)$ will remain enabled in state d, and so d cannot be a deadlock. Therefore, at least one transition t from $ps(s)$ must be fired on the path from s to d. By suitable permutations of independent transitions in the path, an equivalent path of length k in which t is fired from s can be produced (and which can be shown to be explored by the persistent set selective search).

In order to implement the algorithm in Figure 3.3, a method is required to compute, for each state s reached in the search, a persistent set $ps(s)$ of transitions at s. Peled has shown [19, Chapter 10] that the complexity of checking that a given set of transitions T is persistent is equivalent to checking reachability of the full state space. For this reason, rather than checking arbitrary subsets of enabled transitions for the persistence property, algorithms have been developed which are guaranteed to produce persistent sets of transitions. Godefroid has summarized several approaches to computing persistent sets presented by several authors in [46]. Each of these approaches compute persistent sets based on the static structure (program text) of the system being verified.

Godefroid extended his theory in [46] to cater for general safety properties and properties expressed by linear-time temporal logic formulas, as well as deadlocks. These properties require more complex theoretical arguments for proving that properties are preserved. Valmari and Peled also considered the detection of general safety and liveness properties using partial order reduction, in the reports mentioned earlier.

In the case applying the partial order reduction approach to linear-time temporal logic formulas, special care must be taken to ensure that the temporal property is not sensitive to the order of independent transitions in the system (which the selective search algorithm may depend on permuting). In some approaches to applying the method to linear-time temporal logic formulas, this problem is solved by requiring that all *visible* transitions (those transitions which may affect the truth value of one or more of the propositions in the temporal formula) are considered as being dependent, and restricting the class of properties to be checked to those which are *stuttering invariant*. Informally, a linear-time temporal logic formula f is invariant under stuttering if whenever the formula f holds on a given interleaving sequence, the formula also holds on any sequence obtained from the original sequence by stuttering (repeating) any state of the sequence a finite number of times.

Examples from the literature: Godefroid developed a theory of partial order reduction based on persistent sets which caters for the detection of deadlocks [45], safety properties [48] and linear temporal logic properties. A summary of these works is presented in his PhD thesis [46]. Valmari developed a theory of partial order reduction based on stubborn sets, catering for the detection of deadlocks [111], and general linear-time temporal logic properties [112]. Finally, Peled developed a version of the theory based on the notion of ample sets, first presented in a proof context in [88], and re-presented in a model checking context in [86]. A very concise and informative summary of partial order reduction, presented from the point of view of ample sets, is presented in [19, Chapter 10].

The method has been combined with other approaches: partial order reduction and on-the-fly automata theoretic model checking were considered in [87]; partial order reduction and symbolic

model checking were considered in [4]. Partial order reduction has been perhaps most notably the key approach to combating state explosion in the model checker SPIN [55].

Complexity: In the case of partial order reduction, the improvement realized is difficult to quantify, as the reduction obtained from the method varies over a fixed input size (size of concurrent program + size of formula). The effectiveness of the approach depends upon (i) the degree of concurrency of the concurrent program, which determines the number of equivalent sequences in each equivalence class and (ii) the degree to which the property is sensitive to the interleavings of concurrent events. Thus, the same program can have very different reductions, depending on the property being verified. However, as reported in the literature, impressive results have been found in many cases.

3.2.4.2 Methods based on Unfoldings

The unfoldings approach to model checking is based on representing the behaviour of a concurrent program by its unfolding. Model checking based on unfoldings involves a two-step process: in the first step, a finite prefix of the unfolding is generated, which is guaranteed to contain all reachable states of the finite state program; in a second step, the finite prefix is used as input to a model checking algorithm, which implicitly checks if all execution sequences consistent with the finite prefix satisfy the property in question.

Key details of the approach:

The first use of unfoldings as an approach to combat state explosion in model checking was reported in McMillan [63]. The approach was based the idea of using the unfolding as a partial order representation of the behaviour of a concurrent program. The advantage of using the unfolding is that, compared with the Kripke structure or similar formalism based on interleaving semantics, it is an exponentially compact representation. The problem with the unfolding however is that it is generally infinite, for concurrent programs with non-terminating executions. McMillan proposed to avoid this problem by computing a finite prefix of the unfolding, with the property that it contains all reachable states of the concurrent program. This finite prefix could then be used as a basis for analysis. In [63], McMillan presented an algorithm for computing a finite prefix of the unfolding of a 1-safe Petri Net. The finite prefix is constructed by modifying the construction of the full unfolding, so that unfolding does not proceed beyond what are known as *cut-off events*. Cut-off events are events in the computation which, when fired, reach states already visited during the construction. McMillan proved that such an approach does result in a finite prefix containing all reachable states. In [62], Esparza noted that the size of McMillan's complete prefix can be larger than necessary: in some cases, the size of the minimal complete prefix is $O(n)$, where n is the size of the Petri Net, while McMillan's algorithm generates a prefix of size $O(2^n)$. An improved algorithm is presented in [62].

In his original paper, McMillan showed how the finite prefix could be used to answer questions about the behaviour of the concurrent system. In particular, he showed how to solve the reachability of a terminal state (or deadlock) problem in terms of the finite prefix of the unfolding, using an algorithm based on the branch and bound techniques. McMillan notes that although the problem of determining the existence of a terminal marking in an occurrence net is NP-complete. the problem is solved in practice for even very large unfoldings.

Examples from the literature: The unfoldings approach introduced by McMillan has been ex-

tended in various ways: to checking ever larger classes of properties (general reachability, temporal logic) and in applying different algorithmic techniques. In [34, 116, 33], the unfoldings approach has been combined with automata-theoretic model checking, in order to check general properties specified using linear-time temporal logic. In this approach, a synchronized product is defined between a Petri Net and Buchi automaton on observable places only. Melzer et al. [79] combined the unfoldings approach with linear programming in order to perform reachability analysis. In this approach, a marking equation is used to define the constraints which hold during reachability analysis. The marking equation is a set of inequalities which characterize the set of reachable markings of an acyclic net. This equation is then solved using linear programming techniques. Heljanko [54] used the finite prefix of the unfolding together with logic programming in order to perform reachability analysis. In this approach, the reachability problem is translated into a rule-based logic program. This effectively reduces the reachability problem to SAT. A SAT-checker is used to check for a stable model.

A comparison of several of these approaches, as applied to reachability analysis, appears in [96].

Complexity: In the case of unfoldings, there are some interesting results. Reachability analysis is $PSPACE$-complete in general (for systems represented by automata communicating through rendez-vous communication or bounded buffers, or as synchronous products of transition systems, or as 1-safe Petri Nets) [96]. In [96], Esparza shows that this complexity is reduced to NP-complete (in the size of the complete prefix) when carried out using finite prefixes.

3.2.5 Distribution-based Methods

The main computational task of the model checking problem is the exploration of the program state space, which is limited primarily by the amount of available memory on the machine on which model checking takes place. When the size of the validation model is such that the state space does not fit into main memory, virtual memory must be used to store parts of the model. Given that the state space exploration algorithm, whether depth first search or breadth first search, requires random access to the visited states of the state space, some of these which may reside in virtual memory pages on disk, this results in thrashing and a marked decrease in computational performance. As with other large computational problems, we can increase the size of the problems which can be handled by introducing additional processing power and random access memory.

The distribution-based approach to addressing state explosion in model checking is based upon distributing the model checking problem over a network of workstations. The immediate advantages of this approach are that (i) a virtually unlimited amount of random-access memory is made available to the solution of the problem, which is important as the amount of available memory is generally the limiting factor in the model checking of large validation models, and (ii) with the availability of multiple processors, parallel processing techniques may be used to achieve a potential speedup solving the problem.

Distribution of the problem is based upon a partitioning of the state space, with the state space being partitioned into as many regions as there are workstations. Each workstation node is responsible for exploring the portion of the state space belonging to its assigned partition.

The method has been applied successfully to the problem of determining reachability [67],

as well as the general LTL model checking problem [9, 7, 8], with very promising results. This success is dependent upon resolving some key issues involved in developing a parallel version of a sequential model checking algorithm.

In this section, we review the key concepts and details of the approach.

Key details of the approach: Distributed reachability analysis has been considered by several investigators in the literature [1, 102, 67]. To illustrate the issues of the distribution-based approach, we review the key issues of the distribution-based approach to reachability presented in [67]. Lerda et al. considered the problem of distributing the model checking problem over a network of workstations, in the case of checking reachability and safety properties only. Their work was based on the model checker SPIN, which uses an algorithm based upon the automata-theoretic approach to model checking, where both the concurrent system and the property to be checked are represented as automata. In this context, solving the model checking problem reduces to solving a reachability problem (via depth-first search) in a product state space, formed by the synchronous product of the automaton representing the concurrent system, and the automaton representing the negation of the property: that is, the task is to discover reachable acceptance states in the product space. In their approach, the depth-first search exploration of the state space is distributed over a network of N workstations. The distribution of the problem is based upon partitioning the state space into a set of N regions, one for each workstation. Each workstation is then responsible for managing its region of the state space, which entails exploration of successors of states in its partition, as well as storing visited states belonging to its partition in a local hash table, $V[i]$. Each workstation executes the same algorithm, $Visit()$, a modified version of a classical depth-first search algorithm. A local pending queue, $U[i]$, is used to store states belonging to partition i which have yet to be processed. According to the algorithm, each process proceeds by checking for a state to be processed in its local pending queue, $U[i]$. If a state is found, the algorithm checks if the state is already in its set of visited states, $V[i]$. If it is present, no further exploration takes place. If it is not present, the successors of the state are determined. For each successor, a partition function $partition(s)$ is used to determine which partition the successor state belongs to. If the successor state belongs to the local partition i, and has not been visited, it is explored locally; if the successor state belongs to the partition of another process j, the state is sent to the local pending queue $U[j]$ for processing by the remote workstation. When exploration of all local successors has completed, which may involve recursion, the algorithm checks the local pending queue $U[i]$ for the next state to process.

In addition to the N processes executing copies of the algorithm $Visit()$, a manager process is used to coordinate the initialization of the algorithms, as well as detect termination of the distributed algorithm. Initialization consists of determining which partition the initial state belongs to and sending that state to the designated local pending queue. Termination detection is required as each instantiation of the $Visit()$ algorithm will wait indefinitely for new states in its partition to arrive from remote processors in its local pending queue $U[i]$, and will continue to wait even though all queues $U[]$ may be empty and no messages are in transit.

Because exploration of states occurs in parallel, the depth-first search exploration order (postorder) is not preserved by the distributed algorithm. Lerda et al. note that, although this does not pose a problem for reachability analysis, where exploration order does not matter, it does mean that the algorithm is not adequate for checking general LTL properties, where instead of

searching for reachable acceptance states in the product automaton, the algorithm must discover reachable acceptance cycles. The algorithm of Courcoubetis et al. [26] used by SPIN to check general LTL properties depends on reachable acceptance states being processed in postorder.

A major issue in the success of the approach is how to partition the state space. Lerda et al. note that a partition function should satisfy three desirable properties: (i) determining the partition for state s should be a function of that state alone (ii) the partition function should balance the number of states in each partition and (iii) the partition function should minimize the number of cross transitions (those times when a successor state belongs to a different region and must be sent to a remote pending queue). The authors consider two strategies for partitioning: one strategy based on partitioning the state space according to hash function - this leads to balancing of the number of states in each partition, but does not do well in minimizing cross transitions; and another strategy based on partitioning according to the state values taken by a single component (the designated process) - this balances the workload, and at the same time generates a lower number of cross transitions, on average.

One of the important functions of a state space exploration approach is the ability to generate an error trace when an error is encountered. Generating an error trace is complicated by the fact that in the distributed version of the algorithm, we no longer have a stack containing a full path to the current state: each process stack only contains states in its partition. The authors solve this problem by sending to the remote pending queues not only the remote state, but also a path (sequences of transitions) to that state, relative to the previous state on that path from the same partition. Thus, the pending sets $U[i]$ contain relative paths to remote states, which can be used to reconstruct the full path to each state in $U[i]$. In this way, each process will contain a complete error trace in its stack, and can be used to produce the required error trace.

Finally, the authors note that it is important that any gains introduced through distribution of the problem should not rule out the use of other state space reduction methods. In particular, they show that their algorithm is compatible with such methods used by $SPIN$, such as bit state hashing, partial order reduction, and state compression.

Examples from the literature: The distribution-based approach to reachability has been explored by several researchers, and in the context of several different model checking environments [1, 102, 67]. Barnat et al. have investigated several approaches to the application of the method to general LTL model checking. In [9], the authors extend the method of Lerda et al. to general LTL model checking based on a nested depth-first search through the introduction of a *dependency structure*. This structure is used to maintain successor relationships between acceptance states and transfer states (those states involved in cross transitions) encountered in the search and allow the algorithm to enforce the required post order when reachable acceptance states are explored. Other approaches based on property-driven distribution of the state space [7], in which the property is used as a basis for partitioning the state space, and negative cycle detection [12], in which the problem of detecting accepting cycles is reduced to a problem of detecting negative length cycles, which admits a more efficient parallel solution, have also been explored. A review of these approaches is presented in [11].

Barnat et al. have also considered distributed LTL model checking based on breadth-first search in [8]. The authors note that this approach has two advantages over depth-first search based exploration: firstly, that breadth-first search is more amenable to parallelization than depth-

first search, due to the fact that exploration is based on discovering successors in a *frontier* of states and this can simplify parallelization of the algorithm; secondly, that unlike the algorithm of Courcoubetis et al., it does not require postorder exploration of reachable cycles from reachable acceptance states.

Limitations of the method: This is a very promising approach, as it caters to both reachability analysis, which is sufficient for checking reachability and general safety properties, as well as general LTL model checking. In particular, it places no restrictions on the class of problems which may be considered.

One area of difficulty is ensuring compatibility with other state space reduction methods - each method needs to be adjusted to the distributed approach, resulting in costly re-engineering of complex algorithms. As cited in the examples above, this re-engineering may or may not be successful.

Complexity: In the case of the distributed reachability algorithm of Lerda et al. they demonstrate that the partitioning function based on states of a designated process is superior to that of the hash function. In both cases, the space complexity per process is $O(|S|/N)$, where N is the number of processes. In the case of cross transitions, which can be viewed as message complexity of the distributed algorithm, the average fraction of cross transitions is $O(N - 1/N)$ in the case of the hash function partition approach, which tends to 1 as N increases; in the case of the designated process approach, the average fraction of cross transitions is $O(\phi_d * k/P)$, where P is the number of processes, k, the number of processes per transition, and ϕ_d, the fraction of cross transitions on the designated process. The time complexity per process is linear in the number of states considered per process, made up of $O(|S|/N)$ states from the region, together with cross states encountered.

In the case of distributed model checking of LTL formulae, as presented in [9], they note that the memory complexity is on average linear, in the size of the state space and the degree of non-determinism present.

3.3 Summary

Model checking is an automatic method for the verification of finite state concurrent programs. Given a concurrent program P and a specification of temporal behaviour φ, model checking allows determining if all executions of the concurrent program satisfy the temporal specification of behaviour.

The main limitation of the model checking technique is the state explosion problem; the fact that state space of a concurrent program can be exponential in the size of the program. The main cause of the state explosion problem is the representation of concurrency by interleaving.

Given that the model checking problem is $PSPACE$-complete, any attempt to produce an efficient (polynomial complexity) algorithm for model checking, in the general case, is unlikely to succeed. Thus, approaches to combating state explosion cannot *eliminate* the problem for model checking; rather, they *mitigate* the problem by allowing ever larger validation models to be validated.

In this section, we surveyed the broad classes of approaches to dealing with the state explosion found in the literature:

- automata-theoretic methods
- symbolic methods
- model extraction-based methods
- partial-order methods
- distribution-based methods

These approaches vary in their view of how the state explosion problem in model checking may be mitigated. The reduction achieved by many of the methods, for a given problem size, varies depending on the particular combination of concurrent program and temporal property considered.

In the next chapter, we move our attention to the problem of trace checking, and examine the origins of state explosion in the trace checking problem, and the techniques for dealing with state explosion which have been considered.

Chapter 4

Dynamic Property Detection and The State Explosion Problem

As we saw in Chapter 2, many important problems in distributed computing can be cast as executing some notification or reaction when a distributed system execution satisfies a desired temporal evolution of states, or dynamic property. In particular, we examined examples from the implementation of distributed algorithms, testing and debugging of asynchronous distributed computations, as well as providing fault-tolerance in asynchronous distributed computations.

The *dynamic property detection problem* involves checking that the execution of an asynchronous distributed program (distributed computation) satisfies a desired temporal evolution of states, or *dynamic property*. Unlike model checking, which considers *all* executions of a concurrent system, dynamic property detection is concerned with checking a *single* execution.

We consider the property detection problem in the context of *asynchronous distributed systems*. Unlike synchronous distributed systems, in which timeliness guarantees are placed on processor execution speed and message communication, and execution is arranged to proceed in lock-step, asynchronous distributed systems have relatively few guarantees and so exhibit a greater range of execution behaviours.

Any process intending to carry out property detection must have a means for constructing global states, and this involves somehow *monitoring* the execution in order to obtain information to be able to construct such states. The monitoring problem for asynchronous distributed systems is complicated by two key characteristics of asynchronous distributed systems: distribution, and the relativistic effect. Firstly, a global state of the system is distributed throughout the system: it is made up of the local states of the various processors, together with the states of communication channels. These local process states and channel states need somehow to be gathered together to produce a global state. The assembly of such states must be performed with care, as not all global states which may be constructed on the basis of monitoring are meaningful. Secondly, the relativistic effect, which describes the fact that, due to the absence of timeliness guarantees, different processes may see different orderings of events in the same distributed computation, may result in different processors coming to different conclusions as to the actual states or sequences of states through which a distributed computation may have passed. Both of these key factors make

property detection in asynchronous distributed systems challenging. Indeed, although synchronous distributed systems are also distributed, they do not exhibit the relativistic effect.

Given this dependence on monitoring, an important aspect of property detection is the monitoring strategy employed. When properties are stable, detection of dynamic properties can be based on the construction of individual global states of the distributed computation. By periodically testing the value of a stable property on a constructed global state of the computation, correct detection of the property can be assured. Such a construction of individual global states can be based on an active monitoring strategy, wherein the process or processes performing detection actively probe the distributed computation for local state information, and, using this information, construct a consistent (meaningful) global state of the distributed computation. The snapshot protocol of Chandy and Lamport is based on such a monitoring strategy.

For the detection of properties which are not stable, detection must be based on passive monitoring and the construction of observations. Observations are views of the distributed system execution which are in a sense complete: the effect of every event occurring in the distributed computation is represented. An observation represents one possible sequence of states through which the computation may have passed. In passive monitoring, processes participating in the distributed computation send local information corresponding to state transitions of interest to a monitor process, which passively receives them. The monitor process may then build up an observation of the computation based on the received information.

Unfortunately, due to the relativistic effect of asynchronous distributed computations, no individual process can ever know the actual sequence of states which the computation passed through. This necessitates the introduction of modal operators, such as Pos and Def, in order to make the problem of detecting unstable dynamic properties well defined.

Property detection in the general case, as based on a passive monitoring strategy, can be viewed as having four distinct phases:

Specification: Specification involves describing the desired temporal evolution of states of the distributed computation, or dynamic property, in some formalism. A variety of formalisms are used for describing properties, including predicates on global state, regular languages and automata, or temporal logics. Modal operators provide a variety of semantic interpretations required for different application purposes.

Modeling: In the modeling phase, the distributed program is instrumented with (redundant) code to generate event notifications, where each event notification represents a transition of interest of the system execution. The event notifications together comprise the verification model (execution trace) of the system execution. The verification model should contain enough information to allow verification of the property, but it will, in general, represent an abstract model of the execution. The resulting model is used in the verification/detection phase.

Execution and Monitoring: In this phase, the instrumented distributed program is executed and generates a verification model (execution trace) representing the execution. The elements of the verification model are collected by one or more monitors and stored for later analysis.

Verification/Detection: The verification model (execution trace) generated during the program

execution and monitoring phase is analyzed, in order to determine if the execution trace satisfies the specified property. As noted in Chapter 2, detection may take place concurrently with execution/monitoring, in as run-time trace checking, or after execution of has terminated, as in post-mortem trace checking. The different times at which detection takes place are a result of application-specific requirements.

The main limitation of dynamic property detection based on passive monitoring is the state explosion problem, which arises when the distributed computation being verified contains a high degree of concurrency, and so results in large number of possible observations. The state explosion problem effectively places an upper bound on the size of systems for which general dynamic properties can be detected.

In the rest of this section, we briefly present background relevant to the dynamic property detection problem, and examine how the state explosion problem arises in dynamic property detection. The remainder of the chapter is devoted to examining the approaches to combating state explosion in dynamic property detection.

4.1 Fundamentals of Dynamic Property Detection

In our investigation, we focus on the detection of dynamic properties in asynchronous distributed systems, for several reasons. Firstly, asynchronous distributed systems are generally considered a realistic model of distributed systems occurring in practice. Secondly, any results obtained for asynchronous distributed systems will be applicable to distributed systems in which stronger guarantees on process execution and message passing are in effect. In this way, results obtained for asynchronous distributed systems represent a lower bound on what is possible.

An *asynchronous distributed system* is a collection of n sequential *processes* $P_1, \ldots, P_n$ which communicate solely through the exchange of messages. Processes do not share memory, and do not have access to a global clock. Furthermore, no assumptions are made concerning the relative speeds of processes. Communication is achieved via one-way message channels χ_{ij} between pairs of processes P_i and P_j, $i \neq j$. Communication is assumed to be reliable, but it may be subject to arbitrary but finite delays. No assumptions are made about the order in which messages are received with respect to the order in which they are sent.

Monitoring Asynchronous Distributed Systems

A key aspect of solving the dynamic property detection problem in asynchronous distributed system concerns the physical limitations of monitoring the execution behaviour of such systems. In an asynchronous distributed system, due to finite but arbitrary delays in the transmission of messages between processes, as well as the lack of any guarantees on the relative speeds of processes, it is not possible for any process within the distributed system to determine the actual sequence in which events in the distributed computation have occurred.

As Lamport showed in [65], the best information available in a distributed system concerning the relative ordering of events (or local states) must be based on causality relationships between events in the distributed system. The *happened-before relation* [65] is a irreflexive, transitive

relation on events occurring in a distributed system, based on the causal relations between events induced by execution of events at processes and any message passing occurring between processes.

The happened-before relation defines a partial order on the set of events occurring in the distributed computation. Although the happened before relation does not determine an actual total ordering of events (or global states) through which the computation passed, it does limit the possibilities: any sequence of events which is consistent with the happened before relation represents a *possible* sequence of events through which the computation may have passed.

We define the happened-before relation on the events of a distributed computation formally below. One important advantage of the happened-before relation is that it can be implemented fairly efficiently in a distributed system through the use of *vector clocks* [74]. The vector clock encoding of causality in a distributed computation represent an important foundation for algorithms for property detection.

Modeling Executions of Asynchronous Distributed Systems

In this section, we present the formalisms used to formally model the executions of asynchronous distributed programs. We adopt the notation of [83].

The activity of each sequential process participating in the distributed computation is modeled as a sequence of *events*, where each event corresponds to an internal state change, the sending of a message, or the receipt of a message. In the case of communication events, we denote the sending of a message m by $send(m)$ and the corresponding receipt of the message m by $receive(m)$.

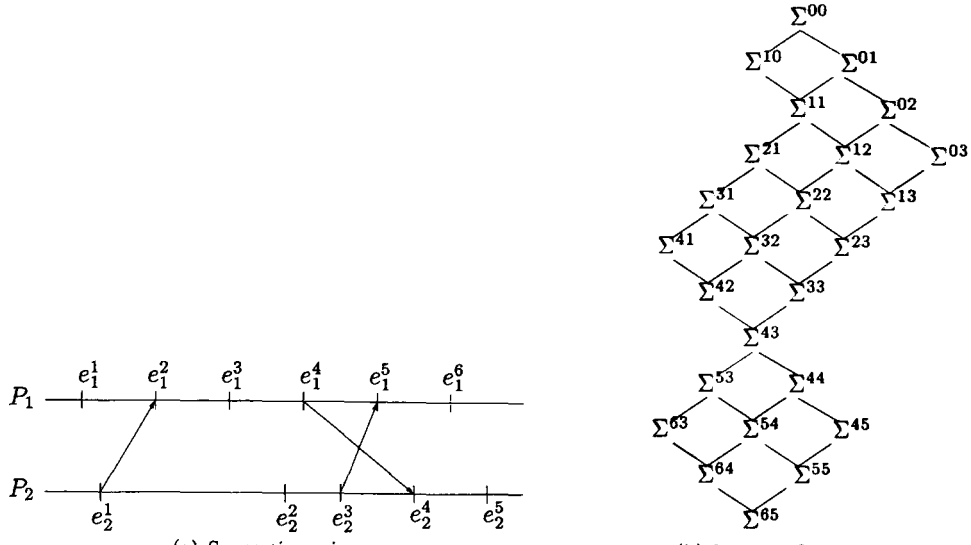
The *local history* of a process P_i participating in the computation is a sequence of events $h_i = e_i^1 e_i^2 \ldots$. where the superscript indicates the sequential order in which the events occur within process P_i. We shall use h_i^k to denote the prefix of length k of the local history h_i[1]. The *global history* of the computation is the set $H = h_1 \cup \ldots \cup h_n$ containing all events executed in the system.

As mentioned in the previous section, the best information we have concerning the relative order of occurrence of events is given by the *happened before* relation [65] $\rightarrow$:

1. if $e_i^j, e_i^k \in h_i$ and $j < k$ then $e_i^j \rightarrow e_i^k$
2. if $e_i = send(m)$ is a send event, and $e_j = receive(m)$ is the corresponding receive event, then $e_i \rightarrow e_j$
3. if $e_i \rightarrow e_j$ and $e_j \rightarrow e_k$, then $e_i \rightarrow e_k$

Additionally, it is assumed that for any event $e \in H$, $\neg(e \rightarrow e)$ holds true (Lamport notes that systems in which an event happens before itself are not physically meaningful). Under this assumption, the happened-before relation is an irreflexive partial order relation [65]. The happened-before relation reflects the causal precedence between events occurring in the distributed computation induced by sequential process execution and message passing. We therefore sometimes refer to this relation as the causal precedence relation.

[1]Although the variables h_i,h_i^k refer to sequences of events, we shall also require referring to the set of events making up those sequences. We use the same notation to refer to both the sequence of events and its corresponding set of events. The intended meaning should be clear from the context.



(a) Space time view

(b) Lattice of global states

Figure 4.1: Two views of a distributed computation.

A *distributed computation* is the partially ordered set $(H, \rightarrow)$, where $\rightarrow \subseteq H \times H$ is the binary causal precedence relation. Distributed computations are a partial order representation of the execution of an asynchronous distributed program. Distributed computations can be visually represented by a *space-time diagram*, where the execution of each process P_i in the computation is represented by a horizontal line, labeled with events from h_i in increasing order from left to right, and messages between processes are represented by directed arrows, again proceeding from left to right. Figure 4.1(a) shows a sample distributed computation involving two processes.

Although the distributed computation has thus far been described in terms of events and their ordering, we shall require a corresponding state-based view. Let σ_i^k denote the *local state* of the process P_i immediately after having executed event e_i^k. The local state of a process consists of a set of local variables (including the program counter) and the current values of those variables in that state. Let σ_i^0 denote the initial state of the process, before any events have been executed.

A *global state* of the distributed computation will be defined as an n-tuple of local states $\Sigma = (\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$, one for each process in the computation.

Not all global states which can be assembled as a tuple of local states are meaningful. Each global state Σ defines a subset $Cut(\Sigma) = h_1^{c_1} \cup \ldots \cup h_n^{c_n}$ of the global history called a *cut*. A cut of the global history can be thought of as a moment in time in a distributed computation: it divides the global history into events which came before the cut, and events which come after the cut. A global state is *consistent* if the cut associated with it is left-closed under the causal precedence relation: that is, Σ is consistent if $\forall e, e' \in H \ (e \in Cut(\Sigma)) \wedge (e' \rightarrow e) \Rightarrow e' \in Cut(\Sigma)$. Left closure and the causal precedence relation ensure that whenever a receive event belongs to a cut, so does its corresponding send event. Cuts which are not consistent violate this property and are not meaningful, in the sense that they do not correspond to physical reality. Define the *frontier* of a cut C , denoted $frontier(C)$, to be the maximal set of events of C, with respect to the sequential ordering of processes. Given a cut $C = h_1^{c_1} \cup \ldots \cup h_n^{c_n}$, the frontier of C is the set

of events $\{e_1^{c_1}, \ldots, e_n^{c_n}\}$.

Given a distributed computation $\gamma = (H, \rightarrow)$, let $\Omega = e^1 e^2 e^3 \ldots$ be any total ordering of the events in H which is consistent with the partial order defined by causal precedence relation $\rightarrow$ of γ. Ω is called a *sequential observation* of the distributed computation γ. A sequential observation can be thought of as a sequence of events which would arise if the distributed program were executed on a machine with a single processor. For each sequential observation $\Omega = e^1 e^2 e^3 \ldots$, there is a corresponding sequence of global states $\Sigma^0 \Sigma^1 \Sigma^2 \ldots$ where Σ^0 represents the initial state $(\sigma_1^0, \ldots, \sigma_n^0)$ and each global state Σ^i is obtained from the previous global state Σ^{i-1} by some process executing the event e^i in γ. Because Ω is an ordering of events consistent with the partial order relation $\rightarrow$ of H, each of the states Σ^i is consistent. In the sequel, we shall use the term sequential observation, or observation, for short, to refer either to a total ordering of events of the distributed computation, or its corresponding sequence of global states. The interpretation should be clear from the context. We denote the set of all observations of a distributed computation γ by Ω_γ.

For two adjacent global states Σ^{i-1}, Σ^i of the observation Ω, their respective cuts differ by the firing of a single event e^i; that is, $Cut(\Sigma^i) = Cut(\Sigma^{i-1}) \cup \{e^i\}$. In terms of states, we denote this fact using the notation $\Sigma^i = e^i(\Sigma^{i-1})$. We say that Σ^i is the *immediate successor* of Σ^{i-1} (or Σ^{i-1} is the *immediate predecessor* of Σ^i) in Ω, denoted by $\Sigma^{i-1} \prec_\Omega^{im} \Sigma^i$. The transitive closure of this relation, denoted $\prec_\Omega$, defines the *successor* (or, *predecessor*) relation between states in the observation Ω. The immediate successor and successor relations between states in an observation can be extended to the set of all states in all observations (and so to all possible states of the distributed computation). We say that Σ' is an *immediate successor* of Σ (or Σ is an *immediate predecessor* of Σ'), denoted by $\Sigma \prec_\gamma^{im} \Sigma'$, if and only if there exists an observation Ω such that $\Sigma \prec_\Omega^{im} \Sigma'$. The subscript γ indicates that the immediate successor relation is associated with the distributed computation γ. Similarly, we say that Σ' is a *successor* of Σ (or Σ is a *predecessor* of Σ'), denoted by $\Sigma \prec_\gamma \Sigma'$, if and only if there exists an observation $\Omega \in \Omega_\gamma$ such that $\Sigma \prec_\Omega \Sigma'$.

Consider now the set of all possible consistent global states of the distributed computation γ. We denote this set by Σ_γ. The successor relation $\prec_\gamma$ defines a partial order on the possible consistent global states of the distributed computation. The set Σ_γ, together with the *successor* relation $\prec_\gamma$ on global states, defines a *lattice* structure $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)^2$. The lattice can be imagined as having n axes, one for each process. Let $\Sigma^{c_1 \cdots c_n}$ denote the global state $(\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$ and $c_1 + \ldots + c_n$ its *level*. Then, the level of the global state $\Sigma^{c_1 \cdots c_n}$ represents the number of events necessary to produce it. This structure is referred to as *the lattice of consistent global states* or *lattice of global states*. This structure is important as it represents all possible sequential observations of the distributed computation: every sequential observation of the computation appears as some path of states through the lattice, and conversely, every path through the lattice is a possible sequential observation of the distributed computation. Figure 4.1(b) shows the lattice of global states associated with the distributed computation of Figure 4.1(a).

In what follows, given a distributed computation $\gamma = (H, \rightarrow)$, we will refer to the corresponding lattice of global states $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)$ as the *computation state space* of the distributed computation γ. It is the construction of this computation state space from an observed distributed computation

[2] Formally, a partially ordered set $(P, \rightarrow)$ is a lattice if any of its two element subsets has a greatest lower bound and a least upper bound in $(P, \rightarrow)$.

which gives rise to the state explosion problem.

In the sequel, we shall require interpreting the lattice of global states from the point of view of different formalisms. In particular, we shall require viewing the lattice of global states alternatively as a labeled directed acyclic graph, or labeled DAG, as well as a Kripke structure. In the definitions which follow, we assume that AP is a set of *atomic propositions* defined on global state.

A *directed acyclic graph*, or DAG, is a graph $G = (V, E)$ where V is the set of graph verticies, $E \subseteq V \times V$ is the set of (directed) graph edges, and (V, E) is acyclic. A *labeled DAG* G is a DAG G together with a finite alphabet of symbols A and labeling function $\lambda : V \rightarrow 2^A$ where $\lambda(v)$ represents the labeling of the vertex $v \in V$. The lattice of global states $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)$ quite naturally defines a directed acyclic graph if we set $G = (\Sigma_\gamma, \prec_\gamma^{im})$. This structure, in which graph edges represent immediate successor relationships between global states, is sometimes referred to in the literature as *the DAG of global states* [5]. An important labeling associated with the DAG of global states is the labeling $\lambda : \Sigma_\gamma \rightarrow 2^{AP}$ which assigns to each global state $\Sigma \in \Sigma_\gamma$ the subset of atomic propositions, $\lambda(\Sigma)$, of AP which hold true in Σ.

We met the definition of a Kripke structure in Section 3.1. A Kripke structure $M = (S, S_0, R, L)$ may be similarly associated with the lattice of global states, and models the lattice of global states as a labeled state transition system. We obtain the associated Kripke structure by setting $S = \Sigma_\gamma$, $S_0 = \Sigma_0$ (the unique minimal element of the lattice), $R : \Sigma_\gamma \times \Sigma_\gamma$, where $R(\Sigma, \Sigma')$ iff $\Sigma \prec_\gamma^{im} \Sigma'$, and $L : \Sigma_\gamma \rightarrow 2^{AP}$ the labeling function which assigns to each global state $\Sigma \in \Sigma_\gamma$ the subset of atomic propositions of AP which hold true in Σ. The structure defined above may not satisfy the totality requirement, in the case where the lattice of global states is finite. In this case, we may ensure totality by defining $R(\Sigma_{final}, \Sigma_{final})$ where Σ_{final} is the unique maximal element of the lattice.

Notice that the view of the lattice of global states as a labeled DAG and as a Kripke structure are essentially the same. We define them separately for notational convenience only.

Specification of Dynamic Properties

The dynamic property detection problem is motivated by the fact that a number of important applications depend on the ability to detect when a distributed system satisfies a desired (or undesired) temporal evolution of states, or dynamic property.

In Chapter 2, we reviewed a number of applications in which dynamic property detection is required, and examples of the types of dynamic properties which arise in those applications. For example, in the implementation of distributed algorithms, we saw that relevant dynamic properties included the property of reaching a global state on which certain conditions are satisfied, such as stability, deadlock, and loss of a token. In the case of testing of reactive distributed systems, we saw that relevant dynamic properties included the property of satisfying specific relationships over time between a reactive system and its environment.

Specifying dynamic properties requires the use of a specification formalism, some of which were mentioned in the examples presented in Chapter 2. A range of formalisms have appeared in the literature for specifying dynamic properties of distributed computations, including predicates (e.g. simple predicates [24], sequences of predicates [6]), temporal logics [30, 97], and various descriptions of formal languages, including regular expressions and finite state automata [60, 5].

Given a dynamic property specification, in some formalism, for a distributed system, the dynamic property can be identified with a *set* of temporal evolutions of the system: those evolutions of system states of the distributed system which satisfy the dynamic property specification, taken from all evolutions of system states which are possible for the distributed system. In this set-based view of dynamic properties, each temporal evolution in the set can be viewed as satisfying the dynamic property.

In this section, we shall look in more detail at some of the formalisms used in the specification of dynamic properties. In what follows, we shall find it convenient to differentiate between two versions of the dynamic property detection problem, corresponding to two important ways in which dynamic properties may be specified:

- the *global predicate evaluation problem*, in which dynamic properties are specified by simple global predicates
- the *temporal predicate evaluation problem*, in which dynamic properties are specified by sequence-based global predicates (or similar formalisms for specifying sets of sequences, such as finite state automata)

Although it will become apparent that the global predicate evaluation problem can be seen as a special case of the temporal predicate evaluation problem, we define the two problems separately, as they are treated separately in the literature.

The *global predicate evaluation problem* was the earliest formulation of the property detection problem. There, the aim is to detect whether or not the system passes through a global state satisfying a predicate defined on global state. In this view of the dynamic property detection problem, dynamic properties are specified in terms of a single Boolean valued predicate Φ defined on global state. In the sequel, we use Φ, Φ_i, Φ' to denote simple predicates defined on global state. For example, $\Phi_1 \equiv (x > 5) \wedge (y == 7)$ and $\Phi_2 \equiv (\chi_{23} == \emptyset)$ are examples of such global predicates.

In the general formulation of the global predicate evaluation problem, predicates are potentially unstable. Unlike stable predicates, they may become true in one observation of a distributed computation, yet remain false in another observation of the same distributed computation, which presents a problem for detection. In order to ensure that the predicate detection problem for distributed computations is well-defined, Cooper and Marzullo [24] introduced modal operators *Pos* and *Def* so that predicate detection becomes *observation-independent*:

Definition 4.1. A distributed computation $\gamma = (H, \rightarrow)$ satisfies $Pos\,\Phi$ if and only if, for *some* sequential observation Ω of γ, Φ holds true in some state of Ω.

Definition 4.2. A distributed computation $\gamma = (H, \rightarrow)$ satisfies $Def\,\Phi$ if and only if, for *all* sequential observations Ω of γ, Φ holds true in some state of Ω.

The modal operators *Pos* and *Def*, in addition to making the predicate detection problem well-defined, also have an application-specific interpretation. If the predicate Φ represents a desired temporal evolution of states, then detecting $Def\,\Phi$ ensures that all possible observations of the distributed computation satisfy this desired evolution, and so represents a fault-free system execution. On the other hand, if the predicate Φ represents an undesired temporal evolution of

states, then detecting $Pos\,\Phi$ checks if some possible observation of the distributed computation satisfies this undesired evolution, and so represents an erroneous system execution. The global predicate evaluation problem is surveyed in [83].

The *temporal predicate detection problem* is the general form of the property detection problem, in which properties are specified in terms of formalisms which allow constraining the relative ordering of states. In the sequel, we use $\varphi, \varphi_i, \varphi'$ to denote sequence-based predicates defined on global state.

With regard to the specification of temporal properties, a number of formalisms for specifying dynamic properties as sequences of predicates have been proposed. In [6], Babaoglu and Raynal presented a sequence-based predicate language for specifying properties as *simple predicates*, *simple sequences*, and *interval-constrained sequences*. A simple predicate is a simply a Boolean valued predicate SP defined on global state, as discussed earlier. A sequential observation satisfies a simple predicate SP if the predicate holds true in some state of the observation. A simple sequence is a sequence $SS = SP_1; \ldots; SP_m$ of simple predicates. A sequential observation satisfies a simple sequence SS if each simple predicate SP_k , for $1 \leq k \leq m$, holds true in a distinct state Σ_{i_k} of the observation, and the states Σ_{i_k} appear in the observation in the same order as their corresponding predicates SP_k appear in SS. Simple sequences permit specification of relative temporal ordering between states in an observation. An interval-constrained sequence is a sequence $ICS = [\theta_1]SP_1; \ldots; [\theta_m]SP_m$ of simple predicates SP_i and interval predicates θ_i. The interval predicates θ_i are also constrained to be simple predicates. An observation satisfies an interval-constrained predicate if it satisfies the simple sequence predicate formed by the SP_k, and, additionally, over every state in the interval between satisfying states $\Sigma_{i_{k-1}}$ and Σ_{i_k}, (or the initial state and Σ_{i_1}, in case $k = 1$), the interval predicate θ_k does not hold. Interval-constrained sequences permit describing conditions on intervals in sequential observations. This form of property specification is useful for applications such as debugging, where breakpoint specifications are often relatively simple and need to be easy to encode.

A more general approach to temporal property specification is based on formal language theory. In this approach, a temporal property φ is associated with a regular language $L(\varphi)$. Regular languages and the finite deterministic automata that recognize them were introduced in 3.2.1. A regular language can thus be specified by describing the finite state automaton which recognizes it. This feature of regular languages makes the specification of temporal properties very convenient.

For example, the regular language corresponding to the set of sequences which pass through a state satisfying the predicate Φ is specified by the finite state automaton described in Figure 4.2. In the example, the automaton $(Q, \Sigma, \delta, q_0, Q_F)$ is specified by $Q = \{q_0, q_1\}$, $\Sigma = \{\Phi, \neg\Phi\}$, $\delta(q_0, \Phi) = q_1$, $\delta(q_0, \neg\Phi) = q_0$, $Q_0 = \{q_0\}$ and $Q_F = \{q_1\}$.

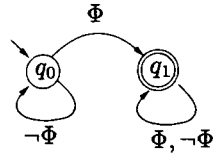


Figure 4.2: An example automaton.

Temporal properties φ are interpreted over observations. We need to introduce modal operators in temporal property detection, for the same reason as in global predicate evaluation: in order to make detection observation-independent.

In the work of Babaoglu and Raynal, where sequence-based predicates were used to specify dynamic properties, sequence-based predicates were quantified using the modal operators *Pos* and *Def*, in a manner analogous to the definition of satisfaction for global predicates. In this approach, the distributed computation is viewed as defining a set of possible sequential observations, and satisfaction is defined in terms of whether some or all of those sequential observations individually satisfy the property.

Definition 4.3. Distributed computation γ satisfies $Pos\,\varphi$, denoted $\gamma \models Pos\,\varphi$, if and only if there exists an observation Ω of γ such that $\Omega \models \varphi$

Definition 4.4. Distributed computation γ satisfies $Def\,\varphi$, denoted $\gamma \models Def\,\varphi$, if and only if for all observations Ω of γ it is the case that $\Omega \models \varphi$

Here, $\Omega \models \varphi$ denotes the satisfaction relation for φ over sequential observation Ω, which varies according to the class of properties considered. For example, if φ represents a property in the class SS, the satisfaction relation $\Omega \models \varphi$ will refer to the satisfaction relation particular to the class SS.

The above definition of satisfaction for the modal operators $Pos\,\varphi$ and $Def\,\varphi$ can be extended to the case where the dynamic property φ is specified by a regular language, $L(\varphi)$. In this case, labeled sequential observations are viewed as sequences of propositions. The states of the sequential observation are labeled with sets of predicates, from a set AP of predicates defined on global state. The labeling of a state is intended to reflect the global predicates which hold true in that state. With global states so labeled, sequential observations now induce sequences of such state labels, referred to as observation labellings. Interpreting predicate labels as atomic propositions, regular languages $L(\varphi)$ over the set of atomic propositions AP may be defined. Given this view of sequential observations as sequences of propositions, there are several views possible for the satisfaction of a property φ by a sequential observation Ω.

If the dynamic property φ is specified by a general regular language $L(\varphi)$ (i.e. a language of finite words over the alphabet AP), then one definition of satisfaction is the following:

- $\Omega \models \varphi$ if and only if some labeling of the sequential observation Ω is in the language $L(\varphi)$

In other words, some labeling of the *complete* sequential observation Ω belongs to the language. A limitation of this definition of satisfaction is that if the sequential observation is non-terminating, then this view of satisfaction is not well-defined (as $L(\varphi)$ is a language of *finite* words over the alphabet AP).

Another view of satisfaction applies to the case in which the dynamic property φ represents a safety property. As noted in Alpern-Schneider [3], safety properties form a subclass of the the class of properties defined by regular languages: safety properties can be characterized by prefix-closed finite automata on finite words. Further, a (possibly non-terminating) sequential observation satisfies a safety property if and only if every finite prefix of the sequential observation satisfies the safety property. Thus, in the case of safety properties φ_{safe}, we have the following definition of satisfaction:

- $\Omega \models \varphi_{safe}$ if and only if for some labeling of the sequential observation Ω, and for each prefix of the labeling, the prefix is in the language $L(\varphi_{safe})$

The advantage of this definition is that it applies to sequential observations which are also non-terminating.

An alternative approach to making dynamic property detection observation-independent was proposed in [5], and is based on viewing the lattice of global states as a labeled, directed acyclic graph. In this approach, Babaoglu, Fromentin and Raynal defined satisfaction in terms of modal operators $SOME$ and ALL. The modal operators $SOME$ and ALL were introduced in order to unify many previous results concerning property detection which had appeared in the literature. The modal operators $SOME$ and ALL differ quite markedly from the modal operators Pos and Def, in which the distributed computation is viewed as a set of sequential observations, and the modalities reflect satisfaction for some or all of those sequential observations. In the case of $SOME$ and ALL, the lattice of global states corresponding to a distributed computation is viewed as a labeled, directed acyclic graph, where graph nodes correspond to global states and are labeled with sets of predicates from a set AP of predicates defined on global state. The labeling of a state is intended to reflect the set of global predicates (atomic propositions) which hold true in that state. With global states so labeled, sequential observations now induce sequences of such state labels, referred to as observation *labellings*. Viewing the lattice of global states of the distributed computation as a labeled, directed acyclic graph permits defining satisfaction of a property by a distributed computation in terms of global states of the lattice, as opposed to sequential observations.

Under the modal operators $SOME$ and ALL, the dynamic property detection problem for a dynamic property φ takes the following form:

Definition 4.5. Given an alphabet AP of global predicates, a lattice of global states $\mathcal{L}$, a labeling function λ, a state Σ of $\mathcal{L}$ and a property φ represented by the language $L(\varphi)$, $\Sigma \models SOME\,\varphi$ if and only if there exists at least one observation terminating in Σ whose labeling defines a word in $L(\varphi)$.

Definition 4.6. Given an alphabet AP of global predicates, a lattice of global states $\mathcal{L}$, a labeling function λ, a state Σ of $\mathcal{L}$ and a property φ represented by the language $L(\varphi)$, $\Sigma \models ALL\,\varphi$ if and only if all observations terminating in Σ have labellings which define words in $L(\varphi)$.

One point of comparison between these two sets of modal operators concerns their use with non-terminating distributed computations.

The modal operators Pos and Def are defined in terms of sequential observations and, as such, deciding the satisfaction of Pos (resp. Def) requires deciding whether or not a dynamic property holds on some (resp. all) *complete* sequential observation(s). In the case of a non-terminating distributed computation, only finite prefixes of its sequential observations may be observed, and it is generally not possible to decide whether a property holds on a sequential observation based solely on observing a finite prefix. For example, in the case of unstable properties specified by simple global predicates, the predicate may not hold on an observed prefix of a non-terminating sequential observation, but become true on the (unobserved) continuation of that prefix. Thus, in the case where distributed computations are non-terminating, it may not be possible, in general,

to decide the truth of Pos and Def. This proviso on the detection of properties defined in terms of Pos and Def will be reiterated in the sequel when necessary to qualify statements concerning the applicability of detection algorithms.

One the other hand, the modal operators $SOME$ and ALL are defined in terms of global states, and the prefixes of sequential observations leading to them. Whether the distributed computation is terminating or non-terminating, it is always possible to decide $SOME$ or ALL correctly for all states reached in the finite prefix, and so make definitive conclusions about satisfaction of the property on the prefix. Furthermore, in the case of terminating computations, and when satisfaction of a dynamic property φ by a sequential observation Ω is defined as above for general regular languages, it is possible to express the modal operators Pos and Def in terms of the modal operators $SOME$ and ALL, evaluated at the unique final state of the distributed computation: given a terminating distributed computation $\gamma = (H, \rightarrow)$ and a dynamic property φ

$$\begin{aligned} \gamma \models Pos\,\varphi \quad &\text{iff} \quad \Sigma_{final} \models SOME\,\varphi \\ \gamma \models Def\,\varphi \quad &\text{iff} \quad \Sigma_{final} \models ALL\,\varphi \end{aligned}$$

where Σ_{final} represents the unique maximal state of the lattice of global states $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)$ of γ. These relations are immediate from the definitions.

Explicit State Property Detection

In this section, we consider algorithms for the detection of the dynamic properties described above based on explicit state enumeration.

Cooper and Marzullo [24] presented an algorithm for explicit state detection of unstable properties defined by predicates on global state with modal operators Pos and Def. The detection algorithms for Pos and Def are both based on an underlying algorithm for constructing the lattice of global states. The algorithm for exploring the lattice of global states is presented in Figure 4.3.

The algorithm is based on a passive monitoring strategy. The monitor stores event notifications received from each process P_i in a queue Q_i. The monitor constructs the lattice of global states as a series of levels l_i, where level l_{i+1} is defined as the set of states which are successors of states in level l_i. Starting from the initial level l_0, the lattice construction algorithm constructs l_1, l_2, and so on. The lattice construction algorithm does not begin the construction of the next level until all events required have arrived.

The algorithm for detecting $Pos\,\Phi$ is superimposed on the lattice construction algorithm: the algorithm is based on constructing each level in the lattice and checking if some state in the level satisfies the predicate. Figure 4.4 shows the algorithm for detecting the predicate $Pos\,\Phi$. The algorithm for detecting $Def\,\Phi$ is similar.

In the case of the detection of dynamic properties quantified by $SOME$ and ALL, the de facto detection algorithm is that of Babaoglu-Fromentin-Raynal, presented in [5]. This algorithm is based on an automata-theoretic approach to the detection of dynamic properties, wherein the dynamic property is represented as a finite state automaton on finite sequences, and a particular product space is explored for reachable acceptance states. However, the exploration of the product space is still based on the lattice construction presented in Figure 4.3. We shall be discussing this algorithm in more detail in Chapter 6.

```
1   current : set of global states : init {initial state}
2   previous : set of global states : init ∅
3   level : integer : init 0

4   while current ≠ ∅
5       level = level + 1
6       previous = current
7       current = ∅
8       foreach Σ ∈ previous do
9           enabled(Σ) = compute_enabled(Σ)
10          foreach e ∈ enabled(Σ) do
11              Σ' = e(Σ)
12              current = current ∪ Σ'
13          od
14      od
15      % wait until we can compute the next level level
16  end while
```

Figure 4.3: Algorithm for constructing the lattice of global states.

We note that other algorithms exist for constructing the lattice of global states from a distributed computation. In [28], Diehl et al. present an algorithm for constructing the lattice of global states based on the close correspondence between a partial order and its lattice of ideals. In this construction, the lattice of global states is constructed, not in a level-based manner, as in the algorithm of Cooper and Marzullo, but based on any observed linearizations of the distributed computation. The algorithm requires maintaining the *cover* of the partial order $(H,\rightarrow)$, which represents the immediate predecessor/successor relationships of events in the partial order. For each event received from an observed linearization, a particular sub-lattice is added to the existing lattice structure. The advantage of this algorithm is a faster construction of the lattice, due to not having to wait until all events of a given level have arrived at the monitor. For details of the algorithm, see [28].

State Explosion Problem

The main problem in trace checking of general dynamic properties is in dealing with large computation state spaces. When a distributed computation exhibits a high degree of concurrency between processes, the number of possible observations of the distributed computation becomes unmanageably large: the size of the lattice of global states is $O(S^N)$, where N is the number of processes in the computation, and S is the maximum number of events on any process.

The algorithm of Cooper and Marzullo is exponential in both space and time: it is exponential in time as there are $O(S^N)$ possible global states in the lattice, and the algorithm executes in time proportional to the number of global states; it is exponential in space as the states for each level are stored, and the breadth of the state space grows exponentially (at each level k, there are $O(N^k)$ possible states).

This complexity estimate also applies to any algorithms based on the lattice construction. In particular, the algorithm for detecting dynamic properties presented in Babaoglu-Fromentin-

```
1   Pos(Φ):
2   begin
3       % Synchronize processes and distribute Φ
4       send Φ to all processes ;
5       current := S(0,0,...,0);
6       release processes ;
7       lvl := 0;
8       while no state in current satisfies Φ do
9           last := current;
10          lvl := lvl + 1;
11          current := states of level lvl reachable from a state in last ;
12      end while
13  end
14  report Pos Φ
```

Figure 4.4: Algorithm for detecting Pos Φ.

Raynal is based on the exhaustive exploration of the computation state space (using the level-based approach or a linearization-based approach), and so suffers at least the same complexity.

More generally, Chase and Garg [18] have shown that checking the property $Pos\,\Phi$ for arbitrary predicate Φ is NP-complete. Similarly, checking $Def\,\Phi$ for arbitrary predicate Φ is $coNP$-complete.

4.2 Techniques for Alleviating State Explosion

In this section, we consider the broad classes of approaches used in property detection to combat state explosion which have appeared in the literature. The approaches have been divided into the following categories:

- methods for stable properties: based on an active monitoring approach, in which a snapshot algorithm is used to periodically construct individual global states of the distributed computation, which are then tested for satisfaction. The stability characteristic of the property ensures that this approach ensures correct detection.

- filtering-based approaches: based on the observation that the number of global states in the lattice depends upon the number of events in the distributed computation. These methods reduce the size of the lattice by selectively removing events from the computation which have no effect on the satisfaction of the property.

- property-structural methods: these methods use information about the structure of the property (the predicate or predicates defining the property) in order to avoid exploring parts of the state space which are known not to contain satisfying global states.

- slicing methods: these methods are based on constructing a slice of the distributed computation (with respect to the property being detected) as an initial step. A slice is a distributed computation which is generally smaller than the original and whose state space is guaran-

teed to contain all the solutions to the property contained in the original computation state space.

- distribution-based methods: based on a divide and conquer approach to property detection, these methods distribute the detection computation across a set of processors, in order to achieve parallelism

- methods from model checking: these approaches are based on applying model checking techniques to the property detection problem

In what follows, we consider each approach according to the following criteria: the motivation behind the approach, or view of the state explosion problem, the details behind the approach, the resulting improvement in complexity, and the variations of the approach which appear in the literature.

4.2.1 Methods for Stable Properties

A *stable* property of a distributed system D is a property defined on a global state S such that once it becomes true, it remains true in all states S' reachable from S. As we saw in Chapter 2, a number of important problems in distributed systems can be formulated as an instance of the dynamic property detection problem in which the property to be detected is stable. Examples include end of computation phase detection, deadlock detection, and token loss detection in the provision of mutual exclusion. In these cases, the associated stable properties to be detected are "the computation has reached its end of phase", "the system is deadlocked", and "the token has been lost".

When a property is stable, the stable property detection problem can be solved by repeatedly constructing a consistent global state of the system execution, and testing to see if the predicate describing the property holds. The problem of determining a global state in a distributed computation is difficult: processes can only record their local state, and the messages sent and received by them. Further, all processes cannot record their local state at the same time, unless they have access to a common clock. Further, care must be taken to ensure that the global state constructed is meaningful, or consistent.

In [16], Chandy and Lamport aim to devise a distributed algorithm so that processes can record their own state, and the states of communication channels, in such a way that the set of local states and communication channels form a consistent global system state. We review the key details of that algorithm in the following sections.

Key details of approach: The snapshot algorithm, presented in [16], is a distributed algorithm which is superimposed on the distributed application. The snapshot algorithm and the distributed computation execute concurrently. The algorithm is based on each process being held responsible for recording its own local state, and the state of all incoming channels. The state of a channel at a given moment in time is the set of messages sent on that channel minus the set of messages received on that channel. As mentioned earlier, the key problem to be faced is in synchronizing the recording activities of the individual processes, so that a consistent global states is recorded. This is achieved through the use of 'marker' messages to synchronize the local snapshots and to

ensure correct recording of channel states. The algorithm consists of two rules: a marker sending rule, and a marker receiving rule:

- Marker sending rule (process P_i): for each channel c, incident on and directed away from P_i, P_i sends one marker along c after P_i records its state and before P_i sends further messages along c
- Marker receiving rule (process P_i): on receiving a marker along channel c: if P_i has not recorded its state, then P_i records its state and records the state of c as $\emptyset$; otherwise, P_i records the state of c as the sequence of messages received along c, after P_i's state was recorded, and before P_i received the marker along c

The snapshot algorithm may be spontaneously initiated by any single process. Chandy and Lamport show that, once initiated, the algorithm will terminate at all processes P_i, resulting in each process P_i holding (i) its snapshot of its own local state and (ii) the channel states of the channels χ_{ji} incident on the process P_i.

Chandy and Lamport note that the state constructed by the snapshot algorithm may not be a state that the distributed computation actually passed through. Let SEQ be a sequential observation of a distributed system execution in which the snapshot algorithm is executed, and suppose that the snapshot was initiated in a state S_i of SEQ and the algorithm terminated in a state S_e of SEQ. They prove that the global state constructed by the algorithm, $S*$, has the property that there is a permutation SEQ' of SEQ which differs only in states between S_i and S_e of SEQ, such that $S*$ occurs as a global state of the sequential observation SEQ'. Chandy and Lamport note that although the actual sequential observation SEQ which the distributed system passed through cannot be observed (one of the fundamental limitations cited earlier), this property can be used to deduce whether or not the property holds in SEQ: if the property holds in $S*$, then the property is guaranteed to hold in S_e of SEQ , and in all successor states, due to stability; if the property does not hold in $S*$, then the property is guaranteed never to have held in S_i of SEQ, nor in any predecessor states, again, due to stability.

In terms of the lattice of global states, given a path SEQ through the lattice and states S_i and S_e on the path where the snapshot algorithm is initiated and terminates, respectively, the global state constructed by the snapshot algorithm lies on some path SEQ' between states S_i and S_e in the lattice.

Limitations of the approach: The approach is limited to the detection of stable properties. Therefore it cannot be used to detect properties (global predicates) which are unstable. This holds even if we take repeated snapshots [10]. This approach is also restricted to checking properties which can be expressed as a simple predicate on global state. The snapshot algorithm for constructing global states is based on an active monitoring approach, in which individual global states are constructed (or a sequence of such) by actively probing the distributed program execution. Consequently, in this form, it applies only to solve the run-time version of the stable property detection problem. In cases where we need to detect stable properties in other application circumstances which require post-mortem detection(such as testing, or post-mortem debugging), other more general algorithms may need to be applied.

Complexity results: The snapshot algorithm explores at most one sequential observation of the distributed computation; in fact, it generally explores only a small part of any one sequential

observation, between the states when it is initiated and where it terminates. Therefore, its complexity in terms of the distributed computation is at worst linear in the number of events H of the distributed computation.

Examples of the approach used in the literature: Chandy and Lamport presented the first algorithm for detecting stable properties. In their formulation of the problem, they assumed that message delivery between processes satisfied a FIFO assumption. In [75], Mattern presented a version of the snapshot algorithm based on the underlying concept of vector clocks in which no FIFO assumption on message delivery is required. The snapshot algorithm has also been extended to the generation of sequences of global states. In [52], Helary presented an algorithm for obtaining a *sequence* of global snapshots, and gathering together the local states for each snapshot, through a combination of several algorithms: a marker algorithm, which generates pairwise consistent local states, together with a wave sequence algorithm, which, in each wave, collects the set of local states in such a way that the local states from each snapshot are collected together. Bougé [10] also studied the construction of sequences of global states through snapshots.

4.2.2 Filtering-based Methods

Filtering [71, 60, 61] refers to the process of selectively excluding event instances from the distributed computation $(H, \rightarrow)$ before analysis, in such a way that the truth value of the property to be detected is not affected. Filtering reduces the size of the distributed computation, and so reduces the size of the lattice which needs to be explored. In this sense, filtering is an important technique for combating state explosion in property detection. Filtering also results in less processing (intrusion) at the application processes, as well as more timely detections, due to the reduced size of the lattice to be explored.

Filtering is closely connected to the way in which causality is encoded in a distributed computation [61] : if events are to be removed from the distributed computation, it is important that the removal of such events does not invalidate or otherwise alter causality relationships between events.

In this section, we discuss the key concepts and approaches associated with filtering.

Key concepts: *Logical clocks.* In asynchronous distributed systems, the happened before relation defines a partial order on the events occurring in the distributed computation, based on the causal relations between events induced by the sequential order of processing at each process together with message passing between processes. Logical clocks aim to encode some or all of that causality information. A *logical clock* is a mechanism used to establish a relative order on the events occurring in an asynchronous distributed system. In this approach, every process maintains a local clock value, LC, which represents the view of time held by that process. Local clocks are updated using a set of clock update rules which are triggered by the occurrence of events. Each event e_i occurring in the distributed system may then be associated with a logical time of occurrence, or logical timestamp, $LC(e_i)$. Various formulations of logical clocks have appeared in the literature, including scalar clocks [65] and vector clocks [36, 73]. These logical clock formulations vary in the degree of information they represent concerning the relative ordering of events in an asynchronous distributed system.

Lamport introduced a formulation of logical clocks in [65] which is suitable for recording a total

ordering of the events in a distributed computation. In this approach, each process maintains a scalar value SC representing its view of global time. As each event e_i occurs, it is assigned a timestamp $SC(e_i)$ based on the current value of the scalar clock SC local to the process at which it occurs. Updates to clock values are triggered by event occurrences. Each clock variable SC is initialized to zero. When messages are sent, the clock value of the send event is piggybacked onto the message. The following rules are used to update clock values:

$$SC(e_i) = SC + 1 \qquad \text{if } e_i \text{ is an internal event or a send event}$$
$$SC(e_i) = max\{SC, TS(m)\} + 1 \qquad \text{if } e_i = \text{receive(m)}$$

where $TS(m)$ denotes the scalar clock value associated with the sending event of message m and max is the component-wise maximum. Given two events, e_1 and e_2 in the distributed computation, the scalar clock mechanism satisfies the *weak clock condition* $e_1 \rightarrow e_2 \Rightarrow SC(e_1) < SC(e_2)$, where $\rightarrow$ is the happened-before relation on events. The disadvantage of the logical clock mechanism based on scalar values is that the reverse implication $SC(e_1) < SC(e_2) \Rightarrow e_1 \rightarrow e_2$ does not necessarily hold for events on different processes. This prohibits, in particular, the use of logical clock values to determine when two events are concurrent.

Vector clocks are a mechanism which provide more information on causality than scalar clocks, and are the de facto method for encoding causality in distributed systems. Vector clocks were developed independently by Mattern [73] and Fidge [36]. Rather than each process maintaining simply a scalar value to represent logical clock values, each process maintains a *vector* VC of natural numbers. As before, in order to record the causal relationships between events, each event e_i is assigned a timestamp, $VC(e_i)$, based on the current vector clock value. As before, when messages are sent, the clock value of the send event is piggybacked onto the message. The following rules are used to update the values of vector clocks at each process:

$$VC(e_i)[i] = VC[i] + 1 \qquad \text{if } e_i \text{ is an internal event or a send event}$$
$$VC(e_i) = max\{VC, TS(m)\} \qquad \text{if } e_i = \text{receive(m)}$$
$$VC(e_i)[i] = VC[i] + 1$$

where $TS(m)$ denotes the vector clock value associated with the sending event of message m and, again, max is the component-wise maximum.

Vector clock values can be interpreted in the following way: given an event e_i^k occurring at process P_i,$VC(e_i^k)[i] = k$, the number of events which causally precede event e_i^k on process P_i, and for each $j \neq i$, $VC(e_i^k)[j]$ represents the number of events which causally precede event e_i^k on process P_j.

The importance of vector clocks lies in their ability to characterize three important conditions used in the construction of consistent global states.

Firstly, vector clocks satisfy the *strong clock condition* [83]: $e \rightarrow e' \equiv VC(e) < VC(e')$, where the ordering $<$ on vector clock values (which are vectors) is determined according to the relation $V < V' \equiv (V \neq V') \wedge V[k] \leq V'[k]$ for $1 \leq k \leq n$. This strong clock condition permits determining when two events are concurrent, based only on their logical vector timestamps. Secondly, they can be used to characterize when two events are *pairwise consistent*. Two events e_i^k and e_j^l (where $i \neq j$) are *pairwise inconsistent* when they cannot belong to the frontier of the same consistent cut[3]. This can only happen when the cut includes at least one receive event without including

[3]The focus on the frontier of the consistent cut arises from the fact that the lattice of global states is constructed

its corresponding send event (and so is not left-closed under the causality relation). The two possible cases in which this can happen (and which involve events e_i^k and e_j^l) are (i) ($e_i^k \rightarrow e_j^l$ and $\exists e_i^{k'} : e_i^k \rightarrow e_i^{k'} \rightarrow e_j^l$) or ($e_j^l \rightarrow e_i^k$ and $\exists e_j^{l'} : e_j^l \rightarrow e_j^{l'} \rightarrow e_i^k$). In terms of vector clocks, the events e_i^k and e_j^l are pairwise inconsistent when $VC(e_i^k)[i] < VC(e_j^l)[i]$ or $VC(e_j^l)[j] < VC(e_i^k)[j]$. Two events e_i^k and e_j^l are therefore pairwise consistent when this cannot arise; namely, when $VC(e_i^k)[i] \geq VC(e_j^l)[i]$ and $VC(e_j^l)[j] \geq VC(e_i^k)[j]$. Finally, vector clocks can characterize when the set of events $frontier(C) = \{e_1, \ldots, e_n\}$ in the frontier of a consistent cut C are pairwise consistent: $\forall i, j : V(e_i)[i] \geq V(e_j)[i]$. This condition is important as the global state associated with a cut is consistent if and only if the events in the frontier of the cut are pairwise consistent.

One of the disadvantages of vector clocks is that they require piggybacking $O(n)$ information to each message sent in the distributed computation: when distributed computations involve a large number of processes, this can result in significant overhead. The method of direct dependencies is another method of encoding causality in distributed systems which only requires piggybacking a scalar value onto messages. For a detailed survey of techniques used to encode causality in asynchronous distributed systems, see Raynal and Singhal [93].

Key details of the approach: Trace checking can be viewed as performing a *simulation*, which involves recording events and their causal dependencies immediately after execution, and then using those events and causality information to simulate all possible sequential observations. Program transitions (int, send, recv) are instrumented with software probes, in order to generate event notifications and send them to the monitor process for analysis. In filtering distributed computations, we selectively remove a subset of events from consideration: conceptually, starting from the complete distributed computation $(H, \rightarrow)$, which reflects all events occurring in the computation, we derive the reduced distributed computation $(H_{red}, \rightarrow_{red})$ through filtering. Filtering generally proceeds by determining, in some manner, a set of *relevant* events for a property: those which can affect the truth value of the property.

We can differentiate between static filtering and dynamic filtering. In *static* filtering, the decision as to which events are relevant is made at the time of program instrumentation, during the modeling phase of trace checking, and before event notifications are generated; in *dynamic* filtering, event notifications are selectively removed after they have been generated, either at the sending process or at the monitor.

Filtering, however it is performed, must ensure the following key requirement: the property will be detected in the full computation $(H, \rightarrow)$ iff it is detected in the reduced computation $(H_{red}, \rightarrow_{red})$.

A key decision is how to determine which events can (and should) be filtered. Events cannot be filtered (removed) arbitrarily. In removing events, we potentially remove:

1. transitions which affect the values of program variables computed in the simulation, and so potentially affect correct detection of the property

2. causality information between events which may be necessary to correctly simulate the executions (e.g. if we remove send or recv events, we will produce a different causality relation $(H_{red}, \rightarrow_{red})$, than in the actual computation $(H, \rightarrow)$, not a suborder)

incrementally, constructing successor cuts from predecessor cuts through the addition of a single event. Consistency conditions therefore need only be verified for events in the frontier of each new candidate cut.

Furthermore, depending on application requirements, it may not be desirable to filter all events not relevant to the property(see *Limitations*).

Examples of the approach: In this section, we provide two examples from the literature of how filtering has been used to generate reduced lattices.

Filtering based on weak vector clocks. In [71], Marzullo and Neiger present an approach to filtering based on *relevant events* and *weak vector clocks*. They note that for many properties, only a subset of so called relevant events need to be considered in order to check the property. Given a property defined by a predicate Φ on global state and a distributed computation $(H, \rightarrow)$, an event e_i^k is said to *potentially affirm* Φ if the execution of e_i^k in some global state can cause the value of Φ to change from false to true. Similarly, event e_i^k *potentially rejects* Φ if the execution of e_i^k in some global state can cause the value of Φ to change from true to false. Event e_i^k *potentially changes* Φ if it potentially affirms or potentially rejects Φ. Among all the events of a distributed computation $(H, \rightarrow)$, the *relevant events* (relative to a predicate Φ) are those events which potentially change Φ. For example, in the case of deadlock detection, the relevant events are the sending of a request for a resource, the sending of a grant of a resource, and the receiving of a grant of a resource. Marzullo and Neiger describe an approach in which only relevant events, those which potentially change the predicate Φ, are sent to the monitor process for analysis. The potential difficulty in adopting this filtering approach is that when send and receive events are not relevant, it is possible that important causal relations between relevant events may be affected. *Weak vector clocks* [71] (with respect to a fixed predicate Φ) are a logical clock mechanism, based on standard vector clocks, in which the local clock values VC_Φ need only be updated by (i) relevant events occurring at the local process and (ii) receive events which indicate that another process has potentially changed Φ. In this scheme, all events of the distributed computation are timestamped with the weak vector clock VC_Φ corresponding to the property. Unlike the case of vector clocks, now different events on the same process can have the same weak vector clock timestamp. Weak vector clocks are *weak* in the sense that they no longer satisfy the strong clock condition of vector clocks; however, as reported in [72], they satisfy variants of the strong clock condition, the pairwise consistency condition, and the consistent state condition, which are sufficient for the purposes of constructing the lattice of global states based on the level-based algorithm of [24]. They also note that a conservative approximation of the relevant events for a distributed computation for a predicate Φ can be obtained by determining the process variables involved in the predicate, and then identifying all events which read or write those variables as relevant. This approach to filtering based on weak vector clocks is a static filtering approach, in that the decision as to which events are relevant must be made before instrumenting the program with software probes containing the weak vector clock VC_Φ.

Filtering based on observable events. In [60], Jard et al. present a method of filtering which is specific to the linearization-based lattice construction of [28]. Their approach was applied to the case of checking temporal properties, specified as finite state automata on finite sequences of events. The set of events of the full computation $H = O \cup X \cup \bar{X}$ is modeled as the disjoint union of set of observable events O, the set of send events, X, and the set of corresponding receive events, $\bar{X}$. In their approach, the observable events O of a distributed computation are the set of events which are involved in the specification. The computation is filtered by assuming that all events in X and $\bar{X}$ are in principle unobservable. In the analysis, the partial order $(H, \rightarrow_H)$

is replaced by the partial order $(O, \rightarrow_O)$ where $\rightarrow_O$ is the partial order $\rightarrow_H$ restricted to the set O . Thus, $(O, \rightarrow_O)$ is a suborder of $(H, \rightarrow_H)$. The linearization-based construction of the lattice of global states corresponding to the partial order $(O, \rightarrow_O)$ uses the cover of the partial order (the immediate predecessor/successor causal relationships between elements of O). The authors present algorithms for retrieving the cover of the causality relation for $(O, \rightarrow_O)$, when causality is encoded in the usual way, for *all* events in H, using vector clocks, and when causality is encoded using direct dependencies. This covering relation is then used together with the linearization-based lattice construction algorithm to generate the reduced lattice for $(O, \rightarrow_O)$. This approach to filtering is dynamic, in the sense that the decision as to which events to include in O can be made at the monitor when the cover of the suborder is being generated.

In [61], Jard and Jourdan consider the relationship between filtering and encoding the causal dependency relation in detail. They present the technique of *adaptive filtering*, which permits encoding the causal dependency relation in a way more conducive to filtering requirements. However, this technique assumes the direct dependency encoding of causality.

Limitations of the approach: The lattice constructed from $(H_{red}, \rightarrow_{red})$ does not include all global states of original lattice. This means that although a property may be correctly detected in the reduced lattice, when the detection occurs, a trace of the full computation is not immediately available. This may be required for certain applications, such as debugging (where examination of the full state space is required) or fault-tolerance (where analysis of the exceptional execution may be required, as in forward exception handling).

We also note that filtering will not in general be compatible with the detection of temporal dynamic properties, unless those properties are stuttering-invariant. Stuttering-invariance was discussed in Section 3.2.4.1.

Complexity: Filtering ameliorates the state explosion problem by reducing the size of the partial order which needs to be considered. If S denotes the maximum number of events on any process in $(H, \rightarrow)$, and S_{red} denotes the maximum number of events on any process in $(H_{red}, \rightarrow_{red})$, then the complexity of the lattice construction is reduced from $O(S^N)$ to $O(S_{red}^N)$. The complexity is still exponential, but leads to smaller lattice structures being explored.

4.2.3 Property-structural Methods

It was noted in Section 4.1 that the global predicate evaluation problem is NP-complete for the Pos modality [18], and $coNP$-complete for the Def modality [105], and therefore complexity theory tells us that finding efficient (polynomial) solutions for the general case is very unlikely. In such a situation, one strategy for finding efficient solutions is to consider special cases of the general problem, for which efficient solutions can be found.

The property-structural approach is based on this point of view: it is an approach to combating state explosion is based on exploiting the *structure* of a property in order to avoid exploring the full lattice of global states. Structural aspects of the property include its *syntactic* structure, as well as its *semantic* structure. By syntactic structure, we refer to knowledge that, for example, a predicate is formed from a disjunction of local predicates. By semantic structure, we refer to knowledge that, for example, the global states on which a predicate evaluates to true is known to form a sub-lattice of the lattice of global states. In this approach, attention is focused on defining

classes of predicates for which efficient solutions can be found, in such a way that these classes represent many of the important predicates used in practice.

A considerable body of work based on this approach exists. Predicates classes, such as disjunctive [41], conjunctive [41, 58, 42], relational [109], generalized conjunctive [43], observer-independent [17, 18], linear [18], regular [39] have been defined. These predicate classes, when taken in combination with specific modal operators, have been shown to admit efficient (polynomial-time) detection algorithms.

The results apply in the main to properties specified as predicates on global state, but the approach has also been extended to consideration of temporal predicates. Further, a number of these algorithms additionally depend upon assumptions concerning termination of the distributed computation, and so apply only to the case of post-mortem analysis.

In this section, we discuss the key concepts and approaches associated with property-structural methods for combating state explosion in dynamic property detection.

Key concepts:

Property classes. We illustrate here the range of predicate classes considered. A *local* predicate is a predicate whose truth value depends only on the state of a single process. A predicate is *conjunctive* (resp. *disjunctive*) if it can be written as a conjunction (resp. disjunction) of local predicates. Examples of conjunctive predicates are mutual exclusion violation - describing when two processes are each in their critical section - and resource sharing violation - two processes simultaneously holding a read-lock and a write-lock on a shared data item. *Generalized conjunctive predicates* are conjunctive predicates which may additionally include predicates on channel states. Termination detection is an example of a generalized conjunctive predicate - termination occurs when all processes are passive and all channels are empty. *Relational* predicates are predicates of the form $\Phi = \sum_{i=1}^{n} x_i < k$, where each x_i represents a value or count on process P_i. Token loss detection or detection of excessive use of a resource may be achieved with relational predicates. Identification of predicates as being disjunctive, conjunctive, relational and generalized conjunctive represent knowledge of syntactic structure of a predicate.

A predicate Φ is said to be *observer-independent [17]* if $Pos\,\Phi \equiv Def\,\Phi$; that is, it holds in one observation if and only if it holds in all observations. For example, stable predicates are observer-independent. Also, locally-defined predicates are observer-independent, even when they are not stable.

Linear predicates [43] are the first of a class of predicates which may be characterized by the structure of their solution set, the set of consistent cuts in which they evaluate to true. In particular, a predicate is *linear* if the set of consistent cuts which satisfy the predicate form an inf-semi-lattice (definition to follow) of the lattice of consistent cuts. Linear predicates have an alternative characterization, in terms of *forbidden states*, which provide the basis for their efficient detection. Informally, a predicate is linear if, for each global state Σ in which it evaluates to false, there is a local state component σ_i of the global state which is forbidden, in the sense that no other global state containing σ_i will satisfy the predicate. Examples of linear predicates include conjunctive predicates under the modality *Pos* and monotone channel predicates (definition to follow).

Finally, a predicate is *regular* if the set of consistent cuts satisfying the predicate form a sub-lattice of the lattice of consistent cuts. Equivalently, a predicate is regular if, whenever consistent

cuts C, D satisfy the predicate, then so do $C \cup D$ and $C \cap D$. Examples of regular predicates include any local predicates, any linear predicate, and certain predicates on the states of channels, such as "there is no outstanding message in the channel", or "there is no token message in transit".

Identification of predicates as being stable, observer-independent, linear and regular represent semantic knowledge concerning a predicate.

Modal operators: In presenting efficient algorithms for the detection of property classes, modal operators play a significant role. The property-structural approach has considered modal operators *Pos*, *Def*, *Controllable* and *Invariant*. A distributed computation $(H, \rightarrow)$ satisfies *Controllable* Φ if and only if, for *some* sequential observation, Φ holds true for all states of the sequential observation. A distributed computation $(H, \rightarrow)$ satisfies *Invariant* Φ if and only if, for *all* sequential observations, Φ holds true for all states of the sequential observation. These two modalities relate to the problems of controllability of distributed computations, important in debugging [105] and fault-tolerance applications [106], and the testing of program invariants. Each combination of modal operator and property class effectively defines a separate problem, with its own algorithmic solution.

Structural aspects of the lattice of consistent cuts: Property-structural methods make reference to structural aspects of the lattice of consistent global states for efficiently detecting properties with known semantic structure. The identification between a consistent global state and a consistent cut permits viewing the semantics of distributed computations in terms of the lattice of consistent cuts, as opposed to the lattice of global states. In the case of reasoning about semantic properties of global predicates, property-structural approaches generally use this consistent cut-based view. In this view, the key lattice property takes the following form: for any two consistent cuts C and D in the lattice, the sets $C \cup D$ and $C \cap D$ are also consistent cuts. Three important related notions are inf-semi-lattice, sup-semi-lattice and sub-lattice. An inf-semi-lattice satisfies the property that, for any two cuts C and D in the inf-semi-lattice, the set $C \cap D$ is a cut in the inf-semi-lattice. An inf-semi-lattice is guaranteed to have a least consistent cut. Similarly, an sup-semi-lattice satisfies the property that, for any two cuts C and D in the sup-semi-lattice, the set $C \cup D$ is a cut in the sup-semi-lattice. A sup-semi-lattice is guaranteed to have a greatest consistent cut. A sub-lattice satisfies both the inf-semi-lattice property and the sup-semi-lattice property, and is guaranteed to have both a least consistent cut and a greatest consistent cut. Efficient algorithms for detecting linear and regular properties make use of the fact that efficient algorithms have been developed for computing the least and greatest cuts of a semi-lattice.

Details of the approach: In this section, we illustrate how the property structural approach is used to design efficient detection algorithms, for some of the classes of properties mentioned above. We present examples of how the approach is used in the case of syntactic knowledge (for disjunctive and conjunctive predicates), as well as the case of semantic knowledge (for linear and regular predicates). All examples involve only the modality *Pos*.

Disjunctive and conjunctive predicates. Disjunctive and conjunctive predicates under the *Pos* modality were considered by Garg et al. in [41]. Detection of disjunctive predicates $\Phi = LP_1 \vee \ldots \vee LP_k$, where $1 \leq k \leq n$, can be performed by each process detecting the predicates LP_i which are local to it, and announcing detection of Φ when one such local predicate is detected. In the case of conjunctive predicates, $\Phi = LP_1 \wedge \ldots \wedge LP_k$, Garg notes that there are two ways in which to check such a predicate: (i) construct all consistent global states and check if the

predicate is true in any of those consistent global states or (ii) identify local states σ_i where the LP_i hold true, and check if combinations of the σ_i are consistent. Garg focuses on the second approach (the first approach leads to a combinatorial explosion in the number of combinations to consider). A naive approach could be based upon (i) determining all local states σ_i in which the LP_i hold true and then (ii) checking all possible combinations of such states, but this would also lead to a combinatorial explosion of possible global states. However, Garg makes use of two key observations: (i) in order to check consistency of a global state, we only need use one σ_i between every two external events (ii) if, in exploring a combination of local states involving σ and σ', we find that $\sigma \rightarrow \sigma'$, then not only will σ and σ' be pairwise inconsistent, and so cannot form part of a consistent global state, but σ will causally precede any successor of σ' as well, and so no global state containing σ will exist. In this way, any global states involving σ may be eliminated from consideration. These observations permit developing an algorithm for detecting conjunctive predicates which (i) avoids sending all local states satisfying local predicates LP_i to the monitor and (ii) avoids checking all possible combinations of local states. The algorithm works as follows: non-checker processes (the processes participating in the distributed computation) send a local snapshot message to the checker process whenever the local predicate becomes true for the first time between external messages. The checker process builds queues Q_i of local snapshots and explores the queues using the reasoning described above, searching for a consistent global state which satisfies the predicate. The algorithm has complexity $O(mn)$, where m is the number of messages sent or received by any process and n is the number of processes. Thus, in the case of detecting disjunctive and conjunctive predicates under the modality *Pos* as describe above, efficient (polynomial) algorithms based on the above approaches exist.

Channel Predicates: We mention one extension to the above. Certain predicates must make statements about the states of channels. An example is termination detection, where a distributed program is deemed to have terminated iff all processes are passive and all channels are empty. In the case of weak conjunctive predicates (conjunctive predicates under the *Pos* modality), only local states are examined, and they are thus unable to describe such properties. Let χ_{ji} be a channel from process P_j to process P_i. The *state* of the channel χ_{ji} is defined to be the messages sent on the channel minus the messages received on the channel. A channel predicate on channel χ_{ji} is any Boolean-valued function defined on the state of the channel. A *generalized conjunctive predicate* is a predicate formed by the conjunction of predicates on local state and channel predicates. Termination detection is an example of a generalized conjunctive predicate. Garg and Waldecker define the notion of *linear* or *monotonic* channel predicates [43]. A channel predicate is linear if, when false, it remains false when either there are sends on the channel but no receives, or there are receives on the channel but no sends. For example, "the channel is empty" is linear (exclusively sending more messages to a channel which is not empty will not make the predicate true), whereas "The channel has an even number of messages" is not linear (both exclusively sending messages and exclusively receiving messages to a channel which has an odd number of messages will make the predicate true). Linearity is important for the following reason: when a linear predicate on a channel is false in a global state, there is at least one process which must make further progress before the channel predicate can become true. Using this characteristic of linear channel predicates to eliminate states from consideration, the algorithm described earlier for detecting conjunctive predicates was modified in [43] to produce an algorithm for detecting generalized conjunctive

predicates. The non-checker processes, as before, send a local snapshot message to the checker process whenever the local predicate becomes true for the first time between two external messages. But this time, it sends the local state plus the values *incsend* and *increcv* , where these represent the additional messages sent and received since the last detection. In addition to rejecting cuts which are not consistent as before, the checker process can now reject cuts which do not satisfy one of the channel predicates. By monotonicity, there will be at least one process which may be advanced before the channel predicate will become true. The algorithm has $O(m^2n + mn^2)$ time complexity, and $O(mn^2)$ space complexity, where m is the maximum number of application messages generated by a process, and n is the number of processes. The algorithms are based on an architecture of non-checker and checker(monitor) processes, and operate on-line.

Linear predicates: The ideas described in the previous section form the basis of the notion of linear predicates. Detection of linear predicates depends on the notion of a *forbidden state*: a local state $G[i]$ of a consistent cut G is forbidden, denoted $forbidden(G, i)$, for a predicate Φ if for all successor cuts H in the lattice of cuts, then $(G[i] \neq H[i])$ or $\neg\Phi(H)$ holds. Informally, a local state $G[i]$ is forbidden if its inclusion in any successor cut H of G implies that Φ is false in H. A predicate Φ is linear iff $\forall G : \neg\Phi(G) \Rightarrow \exists i : forbidden(G, i)$. Linearity implies that, if the predicate is false, there is at least one process state $G[i]$ which is forbidden. If we persist in keeping $G[i]$ in the consistent cuts we consider, the predicate will remain false. Conversely, we may always advance one step on P_i and not risk missing a consistent global state in which the predicate is satisfied. This forms the basis for detection of linear predicates. Informally, the algorithm operates as follows, starting from the initial state: if the predicate does not hold in the current state, we know from linearity that the current state contains a forbidden local state. By identifying this state $G[i]$ and the process it resides on, P_i, we may advance by one state on process P_i without missing a consistent cut which satisfies Φ. The process is then repeated. Examples of linear predicates include any local predicate (if the predicate is false in a local state, it will remain false until we advance from that state), conjunctive predicates, any linear channel predicate, or any generalized conjunctive predicate where channel predicates are linear. Garg shows that the set of cuts satisfying a linear predicate form an *inf-semi-lattice* within the lattice of consistent cuts. This guarantees, in particular, that the least cut satisfying Φ is well-defined. In [18], Garg presents an algorithm for detecting linear predicates with complexity $O(mn)$, where m is the maximum number of states on any process and n is the number of processes. The algorithm is guaranteed to detect the least cut satisfying Φ.

Regular predicates: Regular predicates are those whose solution set form a sub-lattice of the lattice of global states. Garg notes that regular predicates may equivalently be characterized as those which are both linear and post-linear. *Post-linear* predicates represent the dual of linear predicates: a predicate Φ is post-linear iff the states satisfying Φ form a sup-semi-lattice. Post-linear predicates are guaranteed to have a greatest cut which satisfies them. Garg notes that, in the case of terminating distributed computations, post-linear predicates may be detected using the same approach for detecting linear predicates, but beginning the detection from the terminal state of the distributed computation. Such a detection approach is guaranteed to detect the greatest cut satisfying Φ. The detection of regular predicates, whether the least cut satisfying the predicate, or the greatest cut satisfying the predicate, can be achieved using the detection algorithm for linear or post-linear predicates [18], and so the complexity of detection is the same

Predicate class	Structure/Defining Characteristic	Lattice Correspondence
disjunctive	$\Phi = LP_1 \vee \ldots \vee LP_m$	$\mathcal{N}$ A
conjunctive	$\Phi = LP_1 \wedge \ldots \wedge LP_m$	$\mathcal{N}$ A
relational	$\Phi = \sum_{i=1}^{m} < k$	$\mathcal{N}$ A
observer-independent	$Pos\,\Phi = Def\,\Phi$	N/A
linear	$\forall G : \neg\Phi(G) \Rightarrow \exists i : forbidden(G, i)$	solutions form inf-semi-lattice
post-linear	dual of linear	solutions form sup-semi-lattice
regular	$\Phi(G) \wedge \Phi(H) \Rightarrow \Phi(G \cap H) \wedge \Phi(G \cup H)$	solutions form sub-lattice

Table 4.1: Predicate Classes and Their Characteristics

as for linear predicates, $O(mn)$, where m is the maximum number of events on any process, and n is the number of processes.

These classes of predicates and their defining characteristics are summarized in Table 4.1.

Examples from the literature: In this section, we cite examples from the literature in which other modalities, in addition to Pos, are considered.

In the case of the modality Def, a number of results appear in the literature. Garg and Waldecker considered the detection of strong conjunctive predicates (conjunctive predicates under the Def modality) in [42]. Given a strong conjunctive predicate $Def\,\Phi = Def\,LP_1 \wedge \ldots \wedge LP_m$, the approach is based on identifying overlapping intervals I_i^k on which the LP_i are true. On each process, the contiguous states on which the LP_i are true define intervals $I_i^1, I_i^2, \ldots$ Garg shows that if such overlapping intervals exist, then $Def\,LP_1 \wedge \ldots \wedge LP_m$ will hold true, and conversely, that if $Def\,LP_1 \wedge \ldots \wedge LP_m$ holds true, there must be a set of overlapping intervals. This latter fact is proved by showing that, if none of the intervals overlap, a sequential observation may be constructed in which the predicate never holds. The complexity of their detection algorithm is $O(m^2p)$, where m is the number of event queues (equal to the number of local predicates) and p is the maximum length of any queue. The class of observer-independent predicates, whose truth value is the same whether under the Pos or Def modality, may be checked in $O(|H|)$ time, by checking if the predicate holds on an arbitrarily selected sequential observation. This was shown by Charron-Bost et al. in [17]. This result also covers the cases of stable and disjunctive predicates, as they are observer-independent. In the case of regular predicates, in [98], Sen and Garg present several ad hoc necessary and sufficient conditions to reduce the $co\mathcal{NP}$-complete complexity of detection under the modality Def to polynomial complexity, for certain classes of regular predicates.

The property-structural approach has also been applied to the detection of temporal predicates. In [97], Sen and Garg consider applying the property-structural approach to restricted classes of temporal predicates. Predicates are specified using a version of the branching-time temporal logic CTL, defined appropriately over the lattice of consistent cuts, where sequential observations replace paths. One advantage of the use of CTL as a specification languages is that it permits a unified presentation of the modal operators considered previously: given a predicate Φ, $Pos\,\Phi \equiv EF\,\Phi$, $Def\,\Phi \equiv AF\,\Phi$, $Controllable\,\Phi \equiv EG\,\Phi$, and $Invariant\,\Phi \equiv AG\,\Phi$. This paper unifies existing results for the modalities Pos, Def, $Controllable$, and $Invariant$, and introduces some new results. Examples of new results presented include polynomial complexity algorithms for $EG\,\Phi$ and $AG\,\Phi$, with Φ linear, as well as $EG[\Phi_1 U \Phi_2]$ and $AG[\Phi_1 U \Phi_2]$ where Φ_1 is conjunctive

Predicate class	$Pos\,\Phi$	$Def\,\Phi$
disjunctive	$O(\|H\|)$	$O(\|H\|)$
conjunctive	$O(n^2m)$	$O(m^2p)$
generalized conjunctive	$O(m^2n + mn^2)$	open
relational	open	open
observer-independent	$O(\|H\|)$	$O(\|H\|)$
linear	$O(mn)$	open
post-linear	$O(mn)$	open
regular	$O(mn)$	open

Table 4.2: Predicate Classes and Their Complexity of Detection

and Φ_2 is linear. These algorithms are suitable for off-line use only.

Limitations of the approach: The property-structural approach to combating state explosion in dynamic property detection is highly successful, in that efficient algorithms are obtained for many important classes of properties expressed as predicates on global state. This exemplifies the success in adopting the approach to combating state explosion outlined in the introduction, based on identifying classes of properties for which efficient algorithms may be found, as opposed to addressing the general property detection problem.

As seen in the above examples, one of the limitations of the property-structural approach is the fact that each modal operator, property class specification requires a different detection algorithm, resulting in a multiplicity of detection algorithms. Some of the algorithms also depend upon off-line assumptions, which restricts the applications to which these detection algorithms may be applied.

Finally, application of this approach has been less successful to date on combating state explosion in the case of temporal predicates, with the vast majority of the results applying to the case of properties specified by global predicates. The existing results place serious restrictions on the classes of temporal properties for which detection algorithms exist.

Complexity: The complexity results for key property classes and modal operators *Pos* and *Def* are summarized in the Table 4.2, where *open* refers to the fact that algorithms of polynomial complexity are not known for the general case.

4.2.4 Slicing Distributed Computations

Slicing [39] is an approach to combating state explosion in the analysis of distributed computations, based on the idea of dynamic program slicing for distributed programs, discussed in Section 3.2.3. Given a distributed computation $\gamma = (H, \rightarrow)$ and a Boolean predicate Φ, a *slice* of the computation γ with respect to Φ, denoted $slice(\gamma, \Phi)$, is a partial order defined on *subsets of events*, where each subset of events is to be interpreted as being executed atomically, and the set of consistent cuts of the slice is guaranteed to contain all solutions of the original computation $(H, \rightarrow)$ which satisfy the predicate. In general, slices of a distributed computation will be significantly smaller that the original computation. Polynomial time algorithms exist to create slices of distributed computations.

Distributed computation slicing can be used to mitigate the effects of state explosion by per-

forming predicate detection in a two stage process: in the first stage, a slice $slice(\gamma, \Phi)$ of the original distributed computation γ is created with respect to the predicate Φ defining the property; in the second stage, the slice is analyzed, using existing lattice exploration techniques to explore the lattice of consistent cuts associated with the slice, which is guaranteed to contain all consistent cuts of the original computation satisfying the predicate.

In addition to this important application of combating state explosion in property detection, distributed computation slicing is also an important ingredient in the solution to the *predicate control problem*. Given a distributed computation and a predicate defined on global state, the predicate control problem aims to add synchronization arrows to the distributed computation such that the predicate always stays true. In this context, computation slicing is used to determine when a distributed computation is controllable. Predicate control has applications in debugging [105] and software fault-tolerance [106].

In the following sections, we review the key concepts and details of distributed computation slicing.

Key concepts: Lattice Theory and Birkhoff's Representation Theorem for Finite Distributive Lattices. Slicing is based on a theorem by Birkhoff concerning the correspondence between partial orders and finite distributive lattices. We review the basic definitions associated with partial orders and distributive lattices, in order to be able to state the theorem.

A *partially ordered set* or *poset* $\tilde{P} = (P, \leq_{\tilde{P}})$ is a set of elements P together with a reflexive, antisymmetric, transitive binary relation $\leq_{\tilde{P}}$ on P which represents an ordering relation on the elements of P. For $x, y \in P$, we use $x <_{\tilde{P}} y$ to denote $x \leq_{\tilde{P}} y$ and $x \neq y$. If $x <_{\tilde{P}} y$ or $y <_{\tilde{P}} x$, we say that the elements x and y are comparable. Given a set of elements $P' \subseteq P$ and an element $x \in P$, we say that x is an *upper bound* for P' if $y \leq_{\tilde{P}} x$ for every $y \in P'$. The upper bound x is a *least upper bound* for P' if $x \leq_{\tilde{P}} x'$ for all upper bounds x' of P'. The notion of a lower bound and the greatest lower bound are defined similarly. Finally, given two elements $x, y \in P$, the greatest lower bound of the set $\{x, y\}$ is referred to as the *meet* of x and y, and denoted $x \sqcap_{\tilde{P}} y$. Similarly, the least upper bound of the set $\{x, y\}$ is referred to as the *join* of x and y, and denoted $x \sqcup_{\tilde{P}} y$.

A partially ordered set defines a *lattice* if, for every pair of elements $\{x, y\}$ in the set, the least upper bound of $\{x, y\}$ and the greatest lower bound of $\{x, y\}$ are likewise elements of the partially ordered set. A subset of a lattice is a *sub-lattice* if the subset is closed under the meet and join operations. A lattice is *distributive* if its meet operation distributes over its join operation.

Given a partial order $\tilde{P} = (P, \leq_{\tilde{P}})$, a subset $I \subseteq P$ is an *ideal* or *left-closed subset* of P if, for each element $x \in I$, the set of elements which precede x in the order on P are also in I. We denote the set of ideals of a partial order $\tilde{P}$ by $Ideals(\tilde{P})$. The set of ideals of $\tilde{P}$, together with the set inclusion relation $\subseteq$, form a distributive lattice $(Ideals(\tilde{P}), \subseteq)$. In this lattice, the meet operation $\sqcap$ corresponds to set intersection $\cap$, and the join operation $\sqcup$ corresponds to set union $\cup$.

Birkhoff's representation theorem makes explicit the relationship between partial orders and finite distributive lattices. Given a finite distributive lattice $\tilde{L} = (L, \leq_{\tilde{L}})$, an element $x \in L$ is *join-irreducible* if $x \neq 0$ (the zero element of L) and $\forall a, b \in L : x = a \sqcup_{\tilde{L}} b \Rightarrow (x = a) \vee (x = b)$. Let $JoinIrr(\tilde{L})$ be the set of join-irreducible elements of $\tilde{L}$. We now state Birkhoff's representation theorem for finite distributive lattices:

Theorem 4.1. Let $\tilde{L}$ be a finite distributive lattice. Then the map $f : L \rightarrow Ideals(JoinIrr(\tilde{L}))$ defined by $f(a) = \{x \in JoinIrr(\tilde{L}) | x \leq_{\tilde{L}} a\}$ is an isomorphism of L onto $Ideals(JoinIrr(\tilde{L}))$. In

a dual manner, let $\tilde{P}$ be a finite poset. Then the map $g : P \rightarrow JoinIrr(Ideals(\tilde{P}))$ defined by $g(a) = \{x \in P | x \leq_{\tilde{P}} a\}$ is an isomorphism of P onto $JoinIrr(Ideals(\tilde{P}))$.

Birkhoff's theorem states that there is a one-to-one correspondence between a finite poset and a finite distributive lattice. Given any poset, we can reconstruct the corresponding lattice by considering its left-closed subsets; conversely, given a lattice, we can reconstruct the poset by considering its join-irreducible elements.

This theorem will be seen to play a major role in the algorithm used to construct slices with respect to a predicate. As noted in [39], this theorem is important in a computational sense, as the set of join-irreducible elements is often much smaller than the size of the corresponding lattice. Thus, if a computation on $\tilde{L}$ can be performed instead on $JoinIrr(\tilde{L})$, then there is a significant computational advantage (c.f. model checking techniques on unfoldings).

We consider the above theory in the context of a particular partial order, the distributed computation $(H, \rightarrow)$. As we have seen in Section 4.1, the left-closed subsets of the partial order $(H, \rightarrow)$ are exactly the set of consistent cuts $Cuts(\tilde{H})$ of the distributed computation. There, we noted that the set of consistent cuts forms a lattice under set inclusion. We now know that this lattice is also a distributive lattice. The meet operation $\sqcap$ on the lattice corresponds to set intersection, and the join operation $\sqcup$ corresponds to set union. Birkhoff's theorem tells us that there is an isomorphism between the elements of the distributed computation and the join-irreducible elements of lattice, under the mapping $g(e) = \{e' \in H | e' \rightarrow e\}$. In other words, the consistent cuts of the form $\{e' \in H | e' \rightarrow e\}$ are the join-irreducible elements of the lattice of global states. These consistent cuts, one for each event $e \in H$, are referred to as the *local configurations* of the events e.

The idea behind slicing is as follows. A regular predicate is such that the set of consistent cuts on which it is true forms a sub-lattice of the lattice of global states. Using the Birkhoff representation theorem, if we are able to identify the join-irreducible elements of this sub-lattice, the representation theorem can be used to construct a partial order on subsets of events such that the lattice of ideals corresponding to the partial order is exactly the sub-lattice in question.

Representing slices. Slices consist of partial orders defined on subsets of events, where events in a subset are interpreted as being executed atomically. The slice describes (i) which events are to be executed atomically and (ii) which order those subsets should be executed in. Formally, given a distributed computation $(H, \rightarrow)$ and a global predicate Φ, the triple $(I_\Phi, F, \rightarrow_\Phi)$, where $F \subseteq 2^H$ and $\rightarrow_\Phi$ is a partial order on F, is a slice [39] if the following two conditions are satisfied:

1. F is a partition of some subset of H
2. $\forall G \in Cuts(\tilde{H}) : \Phi(G) \equiv (G \in Cuts(\tilde{F}))$

The first requirement guarantees that the elements of the slice are mutually disjoint subsets of H. The second requirement states that the slice must capture all and only the consistent cuts satisfying Φ.

Garg and Mittal prove in [39] that a slice for a distributed computation exists if and only if the predicate Φ is a regular predicate.

Details of the approach: Slicing regular predicates. In the case of a regular predicate, the slice of a distributed computation contains all and only the solutions (if any) to the predicate. As

noted in [39], the size of the lattice $Cuts(\tilde{H})$ and the size of the solution set $Cuts_\Phi(\tilde{H})$ may be exponential in the size of the problem, and so a naive algorithm which enumerated the consistent cuts in $Cuts_\Phi(\tilde{H})$ would be inefficient. Mittal and Garg provide an efficient algorithm. The slicing algorithm for regular predicates consists of four steps (we present only an outline here):

Step 1: compute the least cut, V, of $Cuts_\Phi(\tilde{H})$, using the algorithm for linear predicates

Step 2: compute the greatest cut, W, of $Cuts_\Phi(H)$ using the reverse algorithm for linear predicates

Step 3: the sub-lattice $W - V$ of $Cuts(\tilde{H})$ has now been identified. This sub-lattice contains all solutions of $Cuts_\Phi(\tilde{H})$, by regularity of Φ, as well as other global states which do not. The join-irreducible elements of the sub-lattice $Cuts_\Phi(\tilde{H})$ are found as follows: given an event $e \in H$, define the predicate $\Phi_e(X) = \Phi(X) \wedge (e \in X)$ and let $J(e)$ be the least cut in $W - V$ satisfying $\Phi_e(X)$. It is shown in [39] that these consistent cuts are the join-irreducible elements of $Cuts_\Phi(\tilde{H})$. The least consistent cut algorithm for linear predicates is used to determine all such elements, for each $e \in H$.

Step 4: the join-irreducible elements $J(e)$ may be such that some are equal. In this step, equivalence classes $C_1, \ldots, C_m$ on the set of events in $W - V$ are defined: two events e, f are *equivalent* if $J(e) \leq J(f)$ and $J(e) \leq J(e)$. This is achieved by constructing a graph representing containment, and identifying the strongly connected components. Each equivalence class C_i is represented by a join-irreducible element $J(C_i)$ in the lattice, where $J(C_i) = J(e)$ for one element e of the class C_i. Then the slice $(I_\Phi, F, \rightarrow_\Phi)$ is defined as follows: $I_\Phi = V$, $F = \{C_i | 1 \leq i \leq m\}$ and $\rightarrow_\Phi = \{(C_i, C_j) | J(C_i) \subseteq J(C_j) \wedge J(C_i) \neq J(C_j)\}$.

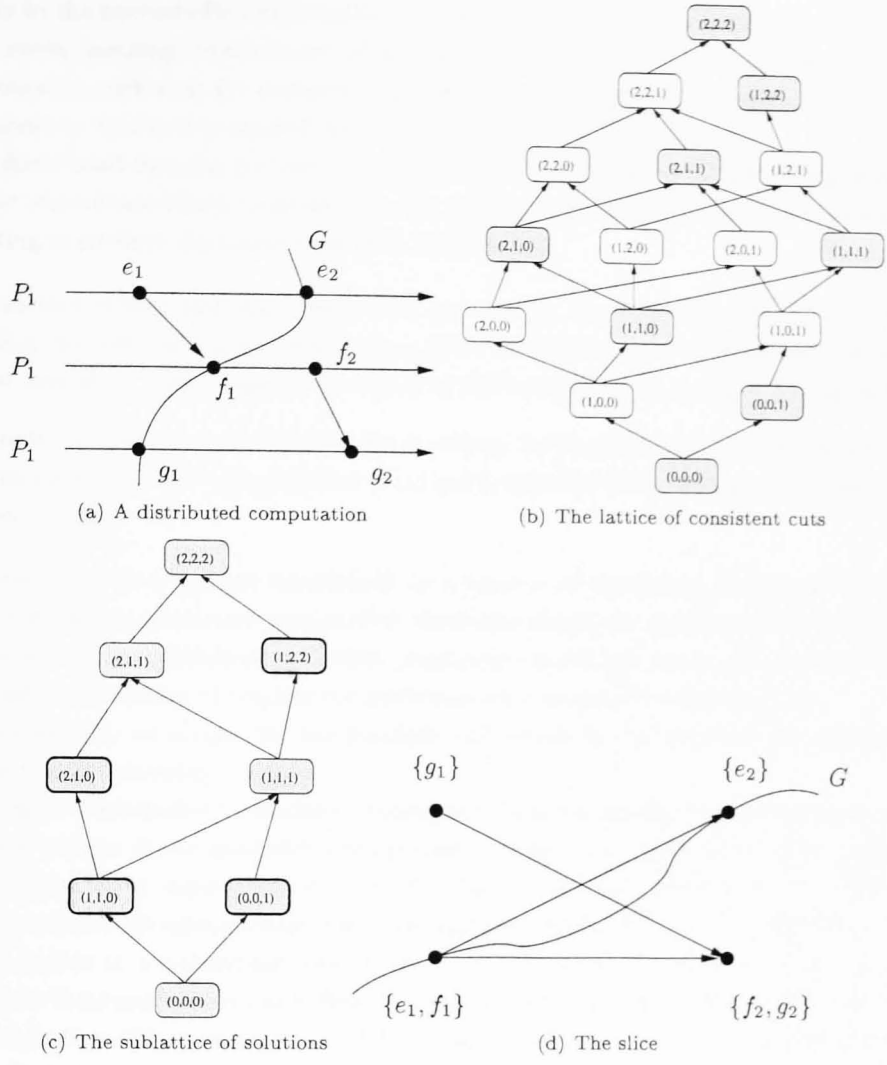
The algorithm for computing the slice runs in $O(N^2|H|)$, where N is the number of processes in the system, and $|H|$ is the number of events in the distributed computation.

Figure 4.5, reproduced from [39], shows the stages of construction of a slice of a distributed computation with respect to a predicate. Figure 4.5(a) shows the original distributed computation. Figure 4.5(b) shows the lattice of consistent cuts, where the cuts satisfying a given but unspecified predicate have been shaded. Steps 1 and 2 of the algorithm effectively identify this sub-lattice. Step 3 identifies the join-irreducible cuts of the sub-lattice of shaded cuts: in the example, $J(e_1) = J(f_1) = \{e_1, f_1\}$, $J(e_2) = \{e_2, e_1, f_1\}$, $J(g_1) = \{g_1\}$ and $J(f_2) = J(g_2) = \{e_1, f_2, f_1, g_2, g_1\}$. Figure 4.5(c) shows the sub-lattice of solutions, with join-irreducible elements indicated by a thick border: in the example, the equivalence classes of events are $C_1 = \{e_1, f_1\}$,$C_2 = \{g_1\}$,$C_3 = \{e_2\}$, $C_4 = \{f_2, g_2\}$. Figure 4.5(d) shows the slice corresponding to the original distributed computation.

Examples from the literature: This slicing approach has been extended to handle the case of general Boolean predicates [81] and temporal predicates [99].

In [81], Mittal and Garg considered developing an approach to slicing which would apply to general predicates, not necessarily regular. The approach is based on identifying a sub-lattice of the lattice of consistent cuts which contains all consistent cuts which satisfy the predicate, even if the latter do not form a sub-lattice. This can be achieved by adding in consistent cuts to complete the sub-lattice.

In [99], Sen and Garg extend the slicing approach to the detection of temporal predicates. In this work, a temporal logic called regular CTL, and denoted $RCTL^+$, is defined. $RCTL^+$ is a



(a) A distributed computation

(b) The lattice of consistent cuts

(c) The sublattice of solutions

(d) The slice

Figure 4.5: A computation (a), its lattice (b), a sub-lattice (c) and the corresponding slice (d).

restricted form of the temporal logic CTL in which atomic propositions must be regular predicates. Sen showed that slices can be defined and computed for temporal formulae in which the temporal operators are restricted to EF, AG, EG.

Limitations of the approach: The slicing algorithms presented in [39, 81, 99] require exploring the partial order from its maximal elements in reverse, in order to compute the greatest cut. Thus, slicing algorithms are generally limited to post-mortem analysis.

Complexity: Computing the slice of a regular predicate can be achieved in $O(N^2|H|)$ [39].

4.2.5 Distribution

In centralized dynamic property detection schemes, such as those considered thus far, the property detection task is performed at a single monitor process. The main computational task of the trace checking problem is the exploration of the computation state space, which is limited

primarily by the amount of available memory on the machine on which the monitor process resides. As with model checking, when the size of the validation model (now represented by the distributed computation) is such that the computation state space does not fit into main memory, paging of virtual memory results in a marked decrease in computational performance.

In a distributed dynamic property detection scheme, the property detection task is distributed over a set of processes which cooperate in order to detect the property. There are two key reasons for wanting to produce distributed detection algorithms:

1. reduction of time and space complexity per process: by using a distributed detection algorithm, the time and space complexity can be distributed over the processes cooperating in the detection. This mitigates the effects of the state explosion problem at each site.

2. speedup of detection: distribution also opens up the possibility of achieving faster detection. This speedup can be achieved if the processes involved in the detection are allowed to execute concurrently.

This method has been applied successfully to a number of algorithms for the on-line detection of weak conjunctive predicates (conjunctive predicates under the modality *Pos*) [38, 58], on-line detection of strong conjunctive predicates (conjunctive predicates under the modality *Pos*) [38], and the off-line detection of conjunctive predicates for a range of modalities [115].

In this section, we review the key concepts and details of this approach to combating state explosion in trace checking.

Key concepts: *Decentralized Detection Architecture.* In a centralized detection scheme, with each application process P_i, we associate a non-checker process NC_i. These non-checker processes can be conceived as being superimposed on the P_i. The non-checker processes do not perform detection: they receive information from the application processes and are responsible for conveying that information to a centralized checker process (or monitor), C. The centralized checker process receives local state information from the non-checker processes and uses this information to perform detection. The computation and data required for detection is centralized at the checker process. In a decentralized detection model, with each application process, we associate a distributed checker process DC_i. In place of a centralized checker process, the distributed checker processes collectively perform detection. In this scheme, two types of messages are exchanged between processes: *application messages*, and *control messages*. Application messages are messages exchanged between processes by the underlying distributed program. Control messages are messages exchanged between distributed checker processes in order to permit detection. Figure 4.6 illustrates these two architectures. In the diagram, arrows with solid lines represent application messages, and arrows with dotted lines represent control messages.

In some detection algorithms [58], control messages may be avoided by piggybacking control information onto application messages. The data required for detection is distributed amongst the distributed checker processes.

Decentralized token-based algorithms versus fully distributed algorithms. Distributed algorithms can vary in the degree of concurrency they provide. Token-based algorithms are distributed algorithms where processing is synchronized through the use of a token, passed between processes. Generally, only the process which has the token may be active. A token-based approach to

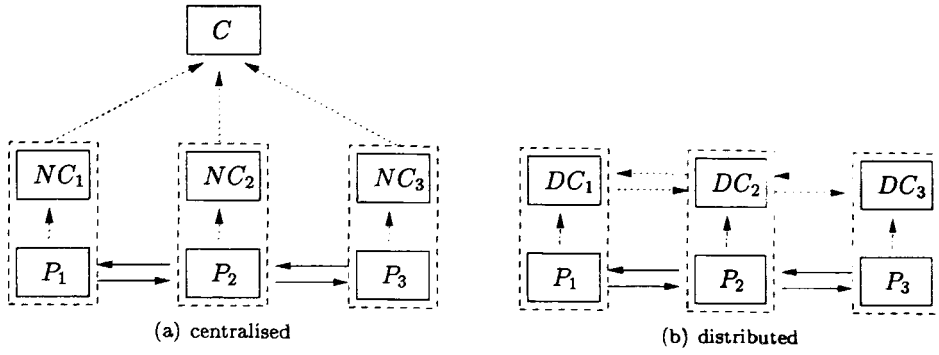


Figure 4.6: Centralised vs Distributed architectures.

providing distribution of detection allows decentralization of the algorithm (permitting distribution of time and space complexity) but results in no concurrency: only the process which currently holds the token may perform steps of the detection. This effectively eliminates any potential speedup due to parallelism. Non-token based distributed algorithms must synchronize any concurrent activity through the use of additional control messages. This results in an equitable distribution of space and time complexity, as well as achieving speedup.

Message complexity. When algorithms are distributed, in addition to *time complexity* and *space complexity*, we must consider the *message complexity* of the algorithm. The message complexity of a distributed algorithm is the total number of bits of information communicated in messages exchanged by the distributed algorithm. Message complexity is significant as each message sent or received incurs overhead (in the form of context switches between processes and execution of various protocol layers). It can happen that the positive advantages incurred by distributing processing can be significantly reduced by increased message complexity. Thus, message complexity is an important consideration in distributed detection algorithms.

Examples from literature: The first distributed algorithms for predicate detection appeared in Garg and Waldecker, for the case of on-line detection of weak conjunctive predicates [41] and strong conjunctive predicates [42]. The distributed version of the algorithm for weak conjunctive predicates was based on the approach taken in the centralized version of the algorithm (identifying a consistent set of local states, each of which satisfies the local predicate) together with the observation that the problem of determining when a set of local states $S = T \cup U$ is consistent can be subdivided into the problems of determining separately when the set of local states T and U are consistent, and checking an additional condition between sets of local states. This permits distribution of the algorithm by dividing the set of processes into groups, and having each group run the centralized detection algorithm, to identify a candidate consistent cut for the group. These candidate cuts for each group are then assembled by an overall checker process which checks if the remaining additional condition holds true. If it does, the predicate is detected. If not, one or more of the candidate cuts are returned to the groups, and the process repeated. A similar approach was used to provide a distributed algorithm for strong conjunctive predicates, only in this case a result for finding a concurrent set of intervals is used.

As cited in [38], this approach suffers from the fact that the group processes may have to send an exponential number of global states (exponential in the number of processes in the group)

to the overall checker process. Garg and Chase presented improved on-line distributed detection algorithms for weak conjunctive predicates in [38]. In that paper, the approach was based on circulating a token containing a candidate consistent cut $G[]$ and a vector of colors $color[]$, where each $color[i]$ is red or green. If a local state $G[i]$ in the candidate cut has $color[i] = red$, then the local state $G[i]$ happened before some other local state $G[j]$; if $color[i] = green$, then $G[i]$ does not happen before any other $G[j]$. The token is sent to processes whose local state $G[i]$ is red. Upon receiving the token, the monitor process waits for local state changes. At each local state change, it updates $G[i]$ with the new local state and recomputes the values in the color vector. When $color[i] = green$, it sends the token on to another process whose local state is marked *red*. Again, this algorithm operates in essentially the same way as the centralized algorithm for detecting weak conjunctive predicates, which is based on advancing past a forbidden state (one which causally precedes some other local state and can never be part of a consistent set of local states) in a candidate cut. The distributed version has time and space complexity $O(mp^2)$ in total, and $O(mp)$ per process, where p is the number of processes over which the predicate is defined, and m is the number of messages sent or received. The algorithm has message complexity $O(mp)$ messages sent, with $O(n)$ bits per message, and so overall $O(mp^2)$ bits. The disadvantage of a token-based algorithm is that there is no concurrency: only the monitor which has the token can be active. Garg and Chase introduce parallelism into the algorithm using a set of g tokens: the set of processes is divided up into g groups, where each group runs the single token algorithm, following eliminated states until a consistent group state is reached. These consistent group states are sent to an overall checker process, which assembles them into a global cut and checks for consistency. If they are consistent, the predicate has been detected. If they are not consistent, the overall checker process sends a token back to one or more groups, and the process is repeated.

The algorithm described above is based on the use of vector clocks, and only requires participation of the p processes in the system on which the local predicates are defined. Garg and Chase present a second distributed algorithm for detecting weak conjunctive predicate based on direct dependencies between states, in which it is necessary that all n processes in the system participate. This algorithm has has time and space complexity $O(mn)$ in total, with worst case for a process $O(m)$. The message complexity is $O(mn)$ messages sent with $O(1)$ bits per message and so overall $O(mn)$ bits.

Hurfin, Mizuno, Raynal and Singhal [58] provided an efficient distributed detection algorithms for the on-line detection of weak conjunctive predicates, which is not token-based. The algorithms are based on modeling distributed computations in terms of events, local states and intervals of local states (between communication events) and their causal dependencies. The conjunctive predicate detection problem is recast in terms of finding a set of intervals which are (i) concurrent and (ii) such that each interval contains a local state satisfying the local predicate. Two versions of the algorithm are presented. The first is based on each process maintaining sets of concurrent intervals, and, for each set, each process checking if its interval in the set satisfies the predicate; the second is based on each process keeping track of one set of intervals, each of which satisfies the local predicate, and checking whether the set is concurrent. The algorithms are fully distributed (each process runs the same algorithm) and do not depend on the passing of a token for synchronizing detection activity. The authors note that, although the volume of control information exchanged by these algorithms is the same as the vector clocks based distributed algorithm of Chase and

Garg, $O(mp^2)$ (where m is the number of messages sent by any process and p is the number of processes over which the predicate is defined), a key aspect of these algorithms is that they do not introduce any additional messages - control information is piggybacked on application messages. By contrast, the two algorithms of Chase and Garg may exchange up to $O(mp)$ and $O(mn)$ control messages respectively. This is a significant as message exchanges account for significant extra overhead.

Distributed detection algorithms have also been developed for post-mortem analysis. Ventkatesan and Dathan [115] provide a family of distributed detection algorithms for ***off-line*** detection of conjunctive predicates. Their application focus is testing and debugging via deterministic replay: the execution of the distributed program is recorded in such a way that it can be deterministically replayed, and during analysis, it is replayed one or more times. The authors consider the detection of conjunctive predicates Φ under a range of modalities useful for debugging applications: $POSSIBLY(\Phi)$, $ALWAYS(\Phi)(\equiv \neg POSSIBLY(\neg\Phi))$, $DEFINITELY(\Phi)$, $FIRST(\Phi)$ and $LAST(\Phi)$. The fact that detection proceeds off-line permits a wider range of modalities to be efficiently detected in a distributed manner. The algorithms are based on computing intersections of *projections of spectra*, where a spectrum is defined as a contiguous interval of states on a single process for which a local predicate is true. A *spectrum* S can be described as the interval $S = [first(S), last(S)]$, where $first(S)$ is the first event in the spectrum, and $last(S)$ is the last. Given an event e on process P_i, the authors define the *first consistent cut event* $FCCE_k(e)$ to be the first event on process P_k which can participate in a consistent cut with event e. The *last consistent cut event* $LCCE_k(e)$ is defined similarly. The *consistency set* for e is a set of spectra, one for each process, defined for process P_k by $[FCCE_k(e), LCCE_k(e)]$, and represents the possible events on other processes which may be involved in consistent cuts with e. Based on these ideas, *the projection of spectrum S_i onto process P_j*, denoted by $\pi(S_i \rightarrow P_j)$, is a spectrum on P_j defined by $[FCCE_j(first(S_i)), LCCE_j(last(S_i))]$ and represents those events on P_j which are concurrent with at least one event in S_i. Finally, given a set of spectra $S_1, \ldots, S_n$, the *ith intersection of projections of spectra* is defined as $I_i = \pi(S_1 \rightarrow P_i) \cap \ldots \cap \pi(S_n \rightarrow P_i)$. This defines a spectrum on P_i such that, for any event $e \in I_i$, e is in the consistency set of at least one event of S_1, one event of S_2, etc. Ventkatesan and Dathan present sequential and distributed algorithms for computing the $I_1, \ldots, I_n$, given a set of spectra $S_1, \ldots, S_n$. The algorithms are based on performing $n-1$ iterations of a two-phase process which starts with the $S_1, \ldots, S_n$and iteratively eliminates events which do not satisfy the conditions required for the intersections of spectra. The algorithms for detecting conjunctive form predicates for the various modalities are based on computing the intersections of spectra. For example, in the case of the modal operator $POSSIBLY(\Phi)$, it can be shown that if a set of spectra $S_1, \ldots, S_n$can be found such that each $S_i$satisfies the local conjunct, and the corresponding intersections $I_1, \ldots, I_n$ are all non-empty, then both $(first(I_1), \ldots, first(I_n))$ and $(last(I_1), \ldots, last(I_n))$ are consistent cuts which satisfy the conjunctive form predicate. In order to avoid having to examine all possible combinations of such spectra, the authors use a technique, first introduced in [41], for eliminating spectra which can never be part of a solution.

The authors note the message complexity of their distributed algorithms, for computing intersections of spectra as well as detecting conjunctive form predicates, but no other complexity estimates for the detection algorithms are provided.

Limitations of the approach: Despite their great potential for distributing the computational task more equitably amongst detection monitor processes and achieving speedup, distributed detection algorithms have yet had only limited applicability. The approaches cited only apply to the detection of weak conjunctive predicates and strong conjunctive predicates, in particular, and exploit the structure of the predicates in order to distribute the detection task. Providing distributed detection for more general classes of predicates is required. One reason for this state of affairs may be that that distributed algorithms are generally more complex to develop than centralized algorithms. This is particularly true if one wishes to avoid the introduction of additional message complexity due to communication required between monitor processes.
Complexity: The algorithms presented are effective in distributing the time and space complexity equitably amongst a set of detector processes. For example, in the case of weak conjunctive predicates, time and space complexity is reduced from $O(n^2m)$ for the centralized checker process to $O(mn)$ per process in the distributed case. However, some of this benefit is eliminated due to additional message overhead introduced by the required communication between monitor processes. For example, in the case of weak conjunctive predicates, this overhead can be as much as $O(mn^2)$ bits. However, as cited earlier, Hurfin et al. [58] showed that by careful algorithmic design, this additional message complexity can be avoided.

4.2.6 Model Checking Methods

The rationale behind the use of model checking techniques for combating state explosion in dynamic property detection is to exploit the use of techniques which have proved successful in combating state explosion in model checking in the problem of dealing with state explosion in the dynamic property detection.

In this section, we outline examples of existing research which has pursued this approach to the state explosion problem in trace checking. The examples to follow are limited in application to the global predicate detection problem (in which dynamic properties are specified by simple global predicates defined on global state) and in which only the modal operators Pos and Def are considered.

It should be noted that although these examples support the view of this thesis that techniques for combating state explosion in model checking have the potential to be used successfully in trace checking, they do not consider the more general issue of which of the many techniques for combating state explosion in model checking are well-suited for application to the dynamic property detection problem. This will be the subject of the next chapter.
Examples from the literature: There are two examples of the use of model checking techniques for combating stat explosion in trace checking appearing in the literature: partial order reduction and symbolic model checking.
Partial order reduction. In [104], Stoller et al. investigate applying the method of partial order reduction to combat the problem of state explosion in the detection of $Pos\,\Phi$ and $Def\,\Phi$ (where Φ is a simple predicate defined on global state) over a distributed computation $\gamma = (H, \rightarrow)$. The authors investigate the use of a *persistent set selective search* (discussed in Section 3.2.4) which performs a reduced search of the state space of a concurrent system, while at the same time detecting all deadlocks. The approach is based on being able to construct a concurrent system

$System_{Pos}$ (resp. $System_{Def}$) such that the concurrent system contains a deadlock iff the distributed computation γ satisfies $Pos\,\Phi$ (resp. $Def\,\Phi$). Once this system has been constructed, a persistent set selective search of $System_{Pos}$ (resp. $System_{Def}$) is used to discover if any deadlocks do, in fact, exist. The details of the approach (construction of the concurrent system and calculation of persistent sets) vary, depending on the modal operator under consideration.

The concurrent system representing γ is constructed based on a computational model presented in Godefroid [46]. We now recap briefly the main elements of this computational model. A concurrent system is described as a tuple $\langle \mathcal{P}, \mathcal{O}, \mathcal{T}, s_{init} \rangle$, consisting of processes $\mathcal{P}$, variables $\mathcal{O}$, transitions $\mathcal{T}$ and an initial state s_{init}. In this model, each process $P_i \in \mathcal{P}$ describes a set of control points, variables in $\mathcal{O}$ may be shared or non-shared variables, and transitions in $\mathcal{T}$ are of the form $t = \langle L_1, G, C, L_2 \rangle$, where L_1, L_2 are sets of control points (with at most one from each process), G is a Boolean-valued guard defined on variables, and C is a command, a sequence of operations defined on variables in $\mathcal{O}$. Global states of the system are tuples $s = \langle L, V \rangle$ where L is a set of control points (one per process) and V is a valuation of the variables in $\mathcal{O}$. A transition $t = \langle L_1, G, C, L_2 \rangle$ is enabled in global state $\langle L, V \rangle$ when $L_1 \subseteq L$ and G evaluates to true when using the values in V. If a transition $t = \langle L_1, G, C, L_2 \rangle$ is enabled in state $s = \langle L, V \rangle$, then it can be executed, leading to a state $s' = \langle (L \setminus L_1) \cup L_2, C(V) \rangle$, where $C(V)$ represents the new values obtained by using the operations in C to update the values in V. Such transitions are denoted by $s \xrightarrow{t} s'$. Executions of the system are finite or infinite sequences $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots$ such that $s_1 = s_{init}$ and for all i, $s_i \xrightarrow{t_i} s_{i+1}$.

Details of approach for $Pos\,\Phi$: The approach to applying partial order reduction to the solution of the problem involves the following three steps:

1. construct a concurrent system $System_{Pos}$ (see below) which has the same set of executions as the distributed computation $\gamma = (H, \rightarrow)$
2. modify $System_{Pos}$ so that the detection problem $\gamma \models Pos\,\Phi$ is expressed as a deadlock detection problem for the concurrent system: that is, $Pos\,\Phi$ holds in the original distributed computation γ iff the system $System_{Pos}$ contains a deadlock
3. use the persistent set selective search algorithm to detect if a deadlock exists in the system $System_{Pos}$ (and so decide if the property holds over the computation).

Constructing $System_{Pos}$: Using the model of computation outlined above, each process of the distributed computation γ is represented by a process $P_i \in \mathcal{P}$, $1 \leq i \leq n$, with one control point representing each local state σ_i^k. For each process P_i in $\mathcal{P}$, a shared variable p_i of $\mathcal{O}$ is defined, whose possible values are the vector clock values $VC(e_i^k)$ of the events $e_i^k \in h_i$. Transitions $t_{ik} \in \mathcal{T}$ are defined to model the execution of events e_i^k of γ: the transition t_{ik} is enabled in $s = \langle L, V \rangle$ when (i) the current control state of P_i corresponds the the local state σ_i^{k-1} in which e_i^k is enabled and (ii) the new value of p_i in $s' = \langle L', V' \rangle$ which would result from firing t_{ik} is such that the set of timestamps $\{p_1, \ldots, p_n\}$ are pairwise consistent (i.e. represent a consistent global state); the effect of executing transition t_{ik} is to (i) update the control state of P_i to correspond to the new local state σ_i^k and (ii) the variable p_i is updated with the timestamp $VC(e_i^k)$ of the event e_i^k. In this way, the distributed computation γ is modeled by a concurrent system $System_{Pos}$, in the sense that they have corresponding sets of execution sequences.

In order to express the detection of $Pos\,\Phi$ as deadlock, $System_{Pos}$ is modified as follows: (i) a new process P_0 is added to the system, having only two control points: an initial control point $l_{0,nd}$ (where the subscript nd represents the fact that $Pos\,\Phi$ is 'not detected') and a terminal control point $l_{0,d}$ (where the subscript d represents 'detected') (ii) a new transition $t_0 \in \mathcal{T}$ is added, defined in such a way that (a) it is enabled in a state $s = \langle L, V \rangle$ only when process P_0 is in control state $l_{0,nd}$ and Φ holds in $s = \langle L, V \rangle$, and (b) firing t_0 has the effect of changing the control state of P_0 to $l_{0,d}$ and disabling all transitions t_{ik}. This latter feature requires updating the enabling conditions of the t_{ik} to additionally require that process P_0 be in control state $l_{0,nd}$. The authors show that, in the modified system, $\gamma \models Pos\,\Phi$ holds iff the constructed system $System_{Pos}$ reaches a deadlock.

Computing persistent sets: An algorithm is required to compute persistent sets $ps(s)$ for each state s reached during the exploration of $System_{Pos}$ using the persistent set search. The authors show that most existing algorithms for computing persistent sets result in sets which are ineffective (i.e. result in all transitions in $enabled(s)$ being explored), due to the special structure of the constructed system. It is proven that if the predicate Φ is false in a state s, then a set of directions $ps(s)$ containing all enabled transitions which potentially make the predicate value change from false to true is a persistent set. Using this fact, the authors provide an algorithm for efficiently computing such a persistent set from state s.

Assuming that the Boolean predicate Φ can be written as a conjunction of Boolean predicates ϕ_i (that is, $\Phi = \phi_1 \wedge \ldots \wedge \phi_n$, where $n \geq 1$), the authors then show that the time complexity involved in computing persistent sets at each state s is $O(Nd)$, where N is the number of processes in the system, $d = max\{|supp(\phi_1)|, \ldots, |supp(\phi_n)|\}$, and $supp(\phi_i)$ is the *support* of the formula ϕ_i (the variables over which the formula is defined). The overall time complexity of the persistent set selective search is $O(NdN_e)$, where N_e is the number of states explored by the algorithm. The authors show that the algorithm, when applied to the case of conjunctive form predicates, has time complexity $O(N^2S)$, where S is the maximum number of steps taken by a single process, which matches the time complexity of the efficient algorithm by Garg and Waldecker for detection of conjunctive form predicates. Stoller et al. also note that their algorithm applies to a larger class of predicates than conjunctive form predicates. They show that their algorithm for detection of $Pos\,\Phi$ is superior to the algorithm of Cooper and Marzullo, as the latter algorithm does not exploit the structure of the predicate in order to determine which part of the state space to explore - it explores all states. Finally, the algorithm presented by Stoller et al. is designed for off-line detection of finite computations, based on constructing $System_{Pos}$ and using the depth-first search based persistent set search algorithm of Godefroid. They claim that the method can be adjusted for on-line use, in particular by changing the order in which the search is carried out, but no proof of this claim is given.

Details of the approach for $Def\,\Phi$: The details of the approach for detecting $Def\,\Phi$ are similar to that of $Pos\,\Phi$:

1. construct a concurrent system $System_{Def}$ which has the same set of executions as the distributed computation $\gamma = (H, \rightarrow)$
2. express the detection problem $\gamma \models Def\Phi$ as a deadlock detection problem for the concurrent system: that is, $Def\,\Phi$ holds in the original distributed computation iff the constructed

system $System_{Def}$ contains a deadlock

3. use the persistent set selective search algorithm to detect if a deadlock exists in the system $System_{Def}$ (and so decide if the property holds over the computation).

Constructing $System_{Def}$: The construction of the concurrent system $System_{Pos}$ to model the distributed computation γ is identical to the case of $System_{Pos}$. In order to express the detection of $Def\,\Phi$ as deadlock, changes are made to system $System_{Def}$. The key idea is to, during the state space search, keep track of whether the predicate Φ became true in any state on the path currently being explored. If the final state of the distributed computation is reached without Φ having become true, then $Def\,\Phi$ is violated. The system $System_{Def}$ is modified as follows: (i) a new process P_0 is added, having only two control points $l_{0,nv}$ (where the subscript nv represents the fact that $Def\,\Phi$ is 'not violated') and $l_{0,v}$ (where the subscript v represents the fact that $Def\,\Phi$ is 'violated') (ii) a new shared variable h is added, and is used to record the fact that, along the current path followed by the depth-first exploration, a state was encountered where the predicate Φ became true (this is achieved by adding an operation to the effect of each transition t_{ik} which sets h to true if the state reached by executing t_{ik} satisfies Φ) (iii) a new transition t_0 is added, defined in such a way that (a) t_0 is enabled only when process P_0 is in control state $l_{0,nv}$, the control states of the processes $P_1, \ldots, P_n$ are in the terminal control states of the computation, and $h = false$ and (b) when transition t_0 fires, it changes the control state of process P_0 to $l_{0,v}$. The transition t_0 will be enabled only when it is possible to reach the final state of the distributed computation by a path on which Φ never holds. Note that any state containing the control state $l_{0,v}$ represents a deadlock of $System_{Def}$, as no transitions in either P_0 or the processes $P_1, \ldots, P_n$ are enabled. The authors show that $System_{Def}$ will enter a deadlock *containing* $l_{0,v}$ iff some path leading to the final state of the computation has not encountered a state satisfying the predicate Φ.

Computing persistent sets: The algorithm for constructing persistent sets is largely based on the same idea as in the case of $Pos\,\Phi$: to identify a set of enabled transitions which must be fired in order to change the value of Φ from false to true. The authors present worst-case time complexity for the case of conjunctions of local predicates: the worst-case time complexity for calculating persistent sets is $O(N)$; the overall time complexity of the persistent set selective search is $O(NN_e)$, where N_e is the number of states explored by the algorithm. The authors remark that $PS_{Def}(s)$ sometimes returns $en(s)$, so that, in the worst-case, N_e is $\Theta(S^N)$. However, in many cases, $PS_{Def}(s)$ returns a proper subset, and so explores many fewer states than the algorithm of Cooper-Marzullo.

Symbolic model checking. In [103], Stoller et al. investigate the application of symbolic methods of model checking to the problem of predicate detection. The idea behind the approach is to encode the property detection problem as a Boolean formula in such a way that a distributed computation $\gamma = (H, \rightarrow)$ satisfies the property if and only if the Boolean formulae evaluates to the Boolean value *true*.

In this work, Stoller only considers properties specified by a predicate defined on global state with modal operator *Pos*. Under this modality, the property detection problem is equivalent to determining if there exists a consistent global state which satisfies Φ.

In this approach, Boolean formulae are used to encode the variables of the processes involved in

the distributed computation, as well as the vector clock values. The variables of the processes are represented as $x_0, \ldots, x_{N-1}$ and vector clock values as $vc_{1,1}, \ldots, vc_{1,N}, \ldots, vc_{N,N}$. The Boolean formula which encodes the detection problem as a satisfiability problem is

$$\Phi(x) \wedge globalState_\gamma(x, vc) \wedge consis_\gamma(vc)$$

where the predicate $globalState_\gamma(x, vc)$ represents the restrictions on what it means to be a global state for distributed computation $\gamma = (H, \rightarrow)$, and $consis_\gamma(vc)$ represents what it means for a set of vector clocks vc to represent a consistent set of local states. These restrictions involve the possible variable and possible vector clock values over the full distributed computation, and so the formulae become quite long.

The algorithm used to check satisfiability is shown in Figure 4.7. Detection does not involve the use of a satisfiability checker, but a simple comparison of BDD values for the Boolean predicate representing the problem, and $false_{bdd}$. The detection algorithm presented Figure 4.7 is also suitable only for off-line detection, in that the Boolean formulas $globalState_\gamma(x, vc)$ and $consis_\gamma(vc)$ represent the all possible values for variables and vector clocks over the entire computation. It could be adjusted for run-time property detection by periodically checking its truth value over intermediate finite prefixes of the computation.

```
1   procedure BDD-detection(γ, Φ̄)

2       b := true_bdd
3       b := b ∧_bdd globalState_γ(x, vc)‾
4       b := b ∧_bdd consis_γ(vc)‾
5       b := b ∧_bdd Φ̄(x)
6       if b = false_bdd then
7           return("γ ⊭ Pos(Φ))")
8       else
9           return("γ ⊨ Pos(Φ))")
10      fi
```

Figure 4.7: Pseudo-code for BDD detection

In terms of performance, Stoller notes that, on the basis of experiments in comparing his algorithm to a standard enumerative algorithm for detecting $Pos\ \Phi$ off-line, when the property is not violated, the new method is faster by a factor that increases exponentially with the number of process in the system. However, he also notes that the BDD-based detection uses significantly more memory than the enumerative approach, because the enumerative approach does not ever store any representation of the full set of consistent global states.

Stoller also notes that the detection of $Pos\ \Phi$ (as well as $Def\ \Phi$ and other temporal properties) may be reduced to CTL model checking, by encoding a computation as a transition system whose interleaving sequences are the same as the sequential observations of the distributed computation under consideration, and using a CTL model checker to check whether that transition system satisfies the relevant CTL formula (for example, $Pos\ \Phi$ may be encoded as $EF\ \Phi$). Such a solution would be suitable for off-line detection, but would require some adjustment in the case of run-time

detection, in which the whole of the computation is not generally available. This is particularly true as CTL model checking involves the use of fixpoints in checking temporal operators.
Limitations of the Approach: A key limitation of applying these model checking approaches to the problem of combating state explosion in trace checking is that it is necessary to find a suitable encoding of the problem for each modal operator considered, and (i) this may or may not be possible and (ii) if possible, may or may not result in an efficient algorithm.

4.3 Summary

Trace checking considers the problem of determining when the execution of an asynchronous distributed program satisfies a desired temporal evolution of states or dynamic property. Given the execution of an asynchronous distributed program, or distributed computation, $(H, \rightarrow)$ and a desired (or undesired temporal evolution of states, represented as a dynamic property φ, the dynamic property detection problem aims to determine if some or all of the sequential observations of the distributed computation satisfy φ. Due to the limitations on making observations of asynchronous distributed systems and the characteristics of unstable properties, the trace checking problem must be defined in such a way that the detection of general properties is observation-independent. This is achieved by defining satisfaction in terms of the set of all possible observations and application-specific modal operators, such as Pos and Def.

The main limitation of the trace checking technique is the *state explosion problem*: the fact that the computation state space of the distributed computation (which implicitly contains the set of all possible observations) can be exponential in the size of the distributed computation. Indeed, the trace checking problem for $Pos\,\varphi$ has been shown to be NP-complete, and the trace checking problem for $Def\,\varphi$ has been shown to be $coNP$-complete.

In the face of this computational complexity of the trace checking problem, a number of techniques have appeared in the literature, developed in order to mitigate the effects of state explosion, and we surveyed the key classes of these techniques in this chapter:

- *methods for stable properties*: based on an active monitoring approach in which consistent global states of the distributed computation are constructed periodically, this approach is well-suited to the on-line detection of stable properties, providing an efficient solution. However, it is not applicable to properties other than stable properties specified in terms of global predicates.

- *filtering-based methods*: based on eliminating or *filtering* events, statically or dynamically, from the observed distributed computation before detection takes place, resulting in an analyzed distributed computation which can be considerably smaller than the original. The degree of computation state space reduction achieved depends upon the proportion of relevant or observable events in the distributed computation. The approach depends on having a method for identifying which events are relevant or observable, as well as special techniques for encoding causality in the reduced model. This approach is potentially applicable to temporal properties. One disadvantage of the method is that it does not result in a complete trace being analyzed, which certain applications, such as debugging and fault-tolerance, may depend upon, as they have an application-specific need to engage in error trace analysis.

- *property-structural methods*: a technique based on identifying classes of properties, based on their syntactic or semantic structure, for which efficient (polynomial) detection algorithms may be found. In this sense, the technique is very successful, resulting in a large class of efficient algorithms. However, the disadvantage of the approach is that (i) the classes of properties considered are limited in scope, especially when it comes to temporal properties and (ii) the approach results in a multiplicity of detection algorithms, one for each modal operator, property class combination. Further, a number of the classes are restricted to off-line detection only.

- *distributed computation slicing*: a technique in which the detection/verification phase of trace checking becomes a two-step process, in which a slice of the distributed computation with respect to a property is created in the first step, followed by analysis. Slices can be created in polynomial time, this reducing the size of the lattice which needs to be explored. The degree of reduction achieved depends upon the size of the solution set of the property. Unlike filtering, which depends upon identifying relevant or observable events, the slicing approach depends upon identifying subsets of events which correspond to join-irreducible elements of the solution set. The approach has been applied primarily to non-temporal predicates, but results exist for specific classes of temporal predicates. This approach is also limited to off-line trace checking.

- *distribution-based methods*: based on a divide and conquer approach, where detection is distributed over a set of workstations, resulting in polynomial time and space complexity per process. At present, the approach is limited to the detection of conjunctive predicates, with Pos and Def modality. These methods do apply to on-line trace checking.

- *model checking-based methods*: this approach is based on applying techniques from model checking, partial order reduction and symbolic state space exploration, to the trace checking problem. In the case of partial order reduction, the approach leads to polynomial algorithms for detection of properties specified by global predicates, for the modalities Pos and Def, both for on-line and off-line trace checking. In the case of the symbolic approach, solutions have been found only for such predicates under the Pos modality, and is possibly only suitable for off-line checking.

The chief problem with the above methods is that many of them apply only to the case of checking dynamic properties in which properties are specified by predicates in global state. In the case where temporal predicates are covered, the temporal predicates are restricted to specific classes. As we have seen in Chapter 2, many important applications, such as testing, debugging, and provision of fault-tolerance, depend on solving the dynamic property detection in the case of general temporal properties. We shall look to model checking techniques surveyed in Chapter 3 as a source for techniques for solving the dynamic property detection problem in the case of general temporal properties.

In the next chapter, we shall look in more detail at the relationships between the model checking and trace checking problems, in an effort to determine which model checking techniques might be well suited to combating state explosion in the trace checking context.

Chapter 5

Comparative Analysis of Techniques

The original goals of this thesis (with the associated research tasks) were to answer the following questions :

- what are the broad classes of approaches used to address state explosion in model checking and in dynamic property detection? (task: survey existing approaches to combating state explosion in both model checking and trace checking)
- how do the contexts in which these two problems are carried out affect the feasibility of an approach to combating state explosion? (task: compare and contrast the contexts and identify promising candidates)
- is it possible to adapt techniques successful for combating state explosion in model checking to combat state explosion in dynamic property detection? (task: explore the application of one or more candidate techniques)

In previous chapters, we addressed the first question by surveying the range of techniques appearing in the literature for combating state explosion in model checking, presented in Chapter 3, and in dynamic property detection, presented in Chapter 4. There we saw that, in both model checking and dynamic property detection, a range of techniques are available for addressing the state explosion problem. In this chapter, we address the second question; that is, we begin to investigate how contexts in which model checking and trace checking are carried out affect the degree to which techniques in model checking can be used in combating state explosion in dynamic property detection and, through this, identify promising candidates for further detailed development.

Our approach will be based on a *comparative analysis* of the two problem areas, with respect to solutions to the problem of state explosion. By a comparative analysis, we understand the following:

- an initial comparison of the respective problem definitions and problem contexts in which these problems must be solved
- based on the identified problem definition and problem context differences, an investigation of which techniques for combating state explosion in model checking show the greatest promise for application in trace checking

In the introduction to this work, we argued that it was quite likely that techniques for combating state explosion in model checking could be used in the context of dynamic property detection, because dynamic property detection and model checking exhibited many apparent *similarities*, with respect to problem definition, semantics and algorithmic approach. We also noted that, despite these similarities, there were also significant *differences* between the two problems, and that these differences might actually prevent a technique from being applied in the new context. The initial stage of the comparative analysis aims to make these similarities and differences explicit.

Concerning the second task, let us note that determining *conclusively* whether or not a model checking technique can be successfully applied in the context of dynamic property detection really can only be achieved through a detailed attempt at development of algorithms based on the approach. Given the complexity of the techniques surveyed in Chapter 3, this is a challenging task, for any one of the individual techniques surveyed there. Therefore, our aim in this second task is not to make such conclusive determinations, but rather try to identify *promising candidates* for further investigation. Our approach will involve first identifying any existing use of the technique (which might be referred to as existing synergies) and then to consider the potential use of the technique (which might be referred to as potential synergies), taking into account the similarities and differences between the two problems, identified earlier. On the basis of this comparative analysis, we will select one or more such promising techniques and carry out the development and analysis - this will make up the remainder of the thesis.

In the first part of this chapter, we investigate the significant similarities and differences between the two problems which can potentially affect the application of model checking techniques. Using this information, in the second part of the chapter, we examine each technique for combating state explosion in model checking and its potential applicability to the dynamic property detection context.

5.1 Problem Definition and Problem Context: Similarities and Differences

Similarities between the definition and context of the two problems add to the potential of successfully applying model checking techniques for combating state explosion to the trace checking problem. Differences between the problem definition and problem context detract from this potential.

In considering similarities and differences of the two problems, we shall touch on the following aspects: the problem definition, underlying semantic structures, the fundamental algorithms involved, and the detection context.

5.1.1 Problem Definition and Problem Context: Similarities

We note the following similarities:

1. **specification of temporal properties:** both problem definitions involve specification of temporal properties via temporal logics (or equivalently expressive formalisms, such as finite state automata), based on the interleaving semantic model. In particular, this permits view-

ing the behaviours of both concurrent programs and distributed computations in terms of state transition systems or similar structures [60]. In the case of model checking, the program state space is represented by the Kripke structure associated with a concurrent program. In the case of trace checking, the computation state space is represented by the lattice of consistent global states (in the case of global predicate evaluation) or the Kripke structure associated with a distributed computation (in the case of temporal predicate evaluation). One important consequence of this is that both problems suffer from the combinatorial explosion brought on by exploring all interleavings (resp. sequential observations) consistent with a concurrent program (resp. distributed computation).

2. **notion of conformance**: in both problems, the notion of whether the behaviour of a system (concurrent program, or distributed computation) conforms to a property is based upon a language containment test: informally, the set of execution behaviours of the system must be contained in the set of execution behaviours defined by the property. This conformance notion is central to model checking: the notion that a finite state transition system is a *model* for the specification. This notion of conformance is distinguished from other notions of conformance, such as refinement orderings, or observational equivalence, which are used in verification frameworks where both specification and system are modeled as automata [21].

3. **fundamental algorithms:** solutions to both problems are based on state space exploration, and make use of graph traversal algorithms in order to explore the program state space (resp. computation state space) of the concurrent program (resp. distributed computation). Therefore, state space reduction techniques are directed at the same basic algorithm.

4. **semantic structures:** in both problems, we can represent the executions of concurrent programs either as sets of partial orders, or as sets of interleaving sequences. Therefore, state space reduction methods in model checking which make use of particular semantic representations have the potential to be used in trace checking as well.

5.1.2 Problem Definition and Problem Context: Differences

We note the following differences:

1. **all executions versus one execution:** in model checking, all executions of the program are verified, as opposed to dynamic property detection, in which only the single *observed* execution of the program is verified or checked.

2. **finite state assumption:** in model checking, a finite state assumption is required in order to guarantee termination of the exploration of the program state space. In property detection, there is generally no such assumption that the program being monitored is finite state. This is due in part to the fact that dynamic property detection is concerned with analyzing either terminating distributed computations, or finite prefixes of non-terminating distributed computations. In this sense, termination of the exploration of the (complete) computation state space is not an issue. For applications of dynamic property detection such as controlling

distributed computations, detecting breakpoints in debugging, and providing fault-tolerance via exception detection and handling, this assumption matches application requirements.

3. **finite observations versus infinite execution paths:** model checking, when combined with the finite state assumption, has the potential to verify infinite execution paths of the concurrent program: infinite execution paths are represented within the finite Kripke structure, in a 'folded' form, via finite length cycles. On the other hand, dynamic property detection generally avoids the finite state assumption, and also avoids the use of a folded structure to represent the possible behaviours of a distributed computation (c.f. the lattice of global states, and the algorithm of Cooper-Marzullo). As mentioned earlier, this is due in large part to the applications of dynamic property detection. Our position here is that we take this approach as given, and make no attempt to justify it. In the absence of a finite state assumption, we can only hope to check terminating distributed computations, or finite prefixes of non-terminating distributed computations (due to the fact that observing an infinite execution would take infinite time). In this sense, model checking is generally concerned with verifying infinite execution paths, and property detection is concerned with checking properties on finite observations, either of terminating distributed computations or of finite prefixes of non-terminating distributed computations.
   An important consequence of this inability to observe infinite execution sequences (in the absence of a finite state assumption) means that we cannot generally check *liveness* properties, whose violation cannot be determined by examining finite prefixes of computations only [3]. However, finite execution sequences are sufficient for checking *safety* properties, which if violated on a program execution, are always violated on some finite prefix of that execution. Given that the application of state space reduction techniques can differ considerably, depending on whether one is checking safety or liveness properties, this difference is significant.

4. **monitoring and the probe effect:** unlike model checking, property detection depends upon the execution and monitoring of the distributed computation. Model extraction in property detection occurs at run-time and can result in the *probe effect* [77], wherein the attempt to extract information concerning the execution can itself modify the execution behaviour. In carrying out property detection and techniques for combating state explosion which involve instrumenting the program, this undesired effect needs to be considered. In particular, we may rate model checking techniques which necessitate less intrusion more highly than those which introduce more intrusion.

5. **systematic exploration of non-determinism and termination of exploration:** when concurrent programs involve non-deterministic choice between program transitions, during program execution, possible alternatives are non-deterministically resolved in favor of one choice. In model checking, all possible non-deterministic choices are systematically explored during the generation of the program state space. A consequence of this is that it is possible to determine when all reachable states of the program have been visited, and this fact is used to determine when program state space exploration may terminate. In dynamic property detection, such non-deterministic choices are resolved non-deterministically *by the*

program during execution, excluding the possibility of such systematic exploration of non-determinism. There are two important consequences of this lack of a systematic exploration of possible non-deterministic choices:

(a) *it is generally not possible to determine, based on observing a finite prefix of an execution, when all reachable states have been visited.* To see this, consider a program which contains non-deterministic choice between two transitions t_1 and t_2 from state s: given a finite prefix of a computation which passes through state s, it is possible that the non-deterministic choice was resolved in favour of t_1 each time state s was reached in the prefix, but that the non-deterministic choice may (or may not) be resolved in favour of t_2 in a continuation of the prefix which reaches s. This means that the state resulting from the firing of t_2 may or may not be reached, but we cannot determine this based on the finite prefix.

(b) *continuations of executions from a given state s are not necessarily the same*: In particular, it is no longer the case that we may avoid exploring execution paths from a visited state s, under the assumption that any executions explored from s upon revisiting will be equivalent (in some sense) to those explored when s was visited for the first time. This fact is used implicitly in the termination reasoning of model checking.

Model checking algorithms are based on the assumption that programs are finite state and that non-determinism is systematically explored, and consequently that the program state space exploration algorithm will eventually *terminate*, after exploring all possible reachable states. By contrast, the algorithms we have seen in property detection make no assumptions concerning termination: they do however implicitly assume that all events in the distributed computation are processed. This difference is significant as a number of approaches to combating state explosion in model checking depend on these two characteristics resulting from systematic exploration of non-determinism (e.g. for proofs of correctness)

6. **receipt of information:** in the run-time variant of dynamic property detection, the analysis algorithms have to contend with three characteristics concerning the receipt of information from the distributed computation:

   (a) *information is incomplete*: in run-time property detection, events of the distributed computation are conveyed to the monitor(s) concurrent with execution of the distributed program. Consequently, algorithms for run-time property detection must be able to carry out processing, despite not having full information concerning the distributed computation under analysis.

   (b) *information is received in breadth-first manner*: events are received by the monitor(s) in an order which is largely consistent with a breadth-first exploration of the computation state space. This ordering of events is due to the fact that executions of processes making up a distributed computation largely make progress in a uniform manner - this can be violated if processor speeds differ greatly and there is little communication between processes, however. This ordering can constrain the way in which the computation state space is explored, and consequently the suitability of a state explosion

technique. For example, a number of methods for combating state explosion in model checking are based on (and correctness proved for) depth-first explorations of the state space. Such methods may require significant adjustment in order to take into account the breadth-first receipt of information.

(c) *information can be delayed*: due to the asynchronous nature of communication, delays in the receipt of individual events can arise from delays in communication of events to the monitor process(es). Such delays may result in delays in exploration of the state space. Algorithms for run-time property detection (in particular, exploration of the computation state space [24, 28]) have been developed to overcome such delays, and any technique for combating state explosion will have to be compatible with such methods.

The issues concerning the receipt of information apply only to the case of run-time property detection. These delays are not present in post-mortem analysis of distributed computations, in which the distributed computation has terminated, and all information concerning the distributed computation has been collected.

7. **partial order observation:** due to the inability to observe the actual execution which a distributed computation passes through, distributed computations are quite naturally modeled as partial orders. This fact may enhance the suitability of techniques for combating state explosion based on partial order semantics.

8. **application requirements:** dynamic property detection can be subject to application-specific requirements on detection: two examples are *timeliness of detection* and *complete trace information*. Unlike model checking, which is used solely for the purposes of verification and always carried out off-line (not concurrent with program execution), the run-time variant of dynamic property detection has applications which require timeliness of detection. For example, when using dynamic property detection in the context of control of a distributed application, or the provision of fault-tolerance via exception detection and handling, *timeliness of detection* is an important consideration. This fact, for example, can lead to choices between competing computation state space exploration algorithms (e.g. breadth-first based on level-based algorithm [24] versus breadth-first based on linearizations [28]). Such choices impact on implementation of state space reduction techniques and the suitability of techniques for use with applications subject to such constraints. *Completeness of trace information* refers to the requirement that, after detection has occurred, the application-specific notification or reaction may depend upon a complete error trace being provided (complete in the sense that key events and state variables have not been removed from the execution trace), in order to support any related analysis. One example of this is in the case of fault-tolerance: once an exceptional execution is identified, full error trace information may be required in order to satisfy exception handling, when error handling is based on forward recovery techniques (which involve recovering from the exception by analyzing the error and taking corrective action). This fact, for example, may lead to choices between methods which vary in the trace information they are able provide.

9. **non-standard modal operators:** dynamic property detection makes use of a range of

application-specific modal operators, including Pos, Def, $SOME$, ALL, discussed in Chapter 4. Some of these modal operators are not considered in model checking (c.f. $SOME$, ALL). Any approach to combating state explosion in trace checking must be compatible with these modal operators.

The similarities and differences outlined above affect the suitability of the application of techniques for combating state explosion, as well as the design of algorithms to implement those techniques. In the next section, these observations concerning the similarities and differences between the two problems will be used in determining whether the state explosion techniques from model checking can be potentially applied in the new context of trace checking.

5.2 Analysis of Suitability of Techniques for Property Detection

In this section, we turn our attention to the problem of identifying which techniques for combating state explosion in model checking show potential for combating state explosion in property detection. The analysis carried out in this section will be used to determine those particular model checking techniques which deserve detailed investigation, which shall be carried out in subsequent chapters of the thesis.

In order to provide a systematic treatment of each technique, we consider them in turn. Our approach will involve focusing on the following two aspects:

- existing synergies: identifying any direct use of the technique or significant relationships with existing techniques used to combat state explosion in property detection
- potential synergies: identifying *potential* use of the technique in combating state explosion, in the light of the similarities and differences between the two problems, or similarly identifying potential limitations of the technique, given the new context

We examine the techniques in the order presented in the survey.

5.2.1 Automata-theoretic methods

As we saw in Section 3.2.1, the automata-theoretic approach to model checking views the model checking problem from the point of view of formal languages and automata. In this approach, both the temporal specification φ and the program P are viewed as automata. Model checking proceeds by translating the temporal specification φ into an automaton A_φ which accepts the same language as the specification, and checking the product $P \times A_{\neg\varphi}$ for non-emptiness.

In this basic formulation, the approach can be carried out successfully while *still* exploring the full state space of the program P. In the on-the-fly variant of the approach, the automaton $A_{\neg\varphi}$ is used to guide the exploration of the state space, by exploring the synchronous product of P and $A_{\neg\varphi}$. This results in only that part of the program state space necessary for detecting the property being explored, and so mitigating the state explosion problem.

In this section, we aim to determine the suitability of the on-the-fly variant of the automata-theoretic approach to combat state explosion in property detection.

Existing Synergies

We begin by considering existing use of the automata-theoretic approach in trace checking. In [60], Jard et al. put forward the automata-theoretic approach as an approach to trace checking of general temporal properties. This was based on the observation that the lattice of global states can be viewed as a labeled transition system and so this model checking technique can be applied. In that paper, temporal properties were specified as regular languages over program actions, and satisfaction was defined in terms of prefixes of sequential observations using the modal operators $SOME$ and ALL (there notated as $\models_\exists$ and $\models_\forall$). Detection is based on labeling each state in the lattice with a set of automaton states: for each state Σ, Σ is labeled with the set of automaton states reachable by considering all paths from the initial state to Σ. Jard et al. note that this may be performed during lattice construction in an inductive manner, based on the set of automaton states reached by predecessor states. Although the automata-theoretic approach forms the basis of this algorithm for detection of such general temporal properties, the product space of the lattice and the automaton is defined in such a way that the full computation state space of the distributed computation is always explored; that is, the automaton is not used to guide the search and so avoid exploring parts of the computation state space. The automata-theoretic approach has also been applied to the case of detection of state-based temporal properties [5], in a manner similar to that proposed in [60]. The automata-theoretic approach to the detection of temporal properties presented in [5] also explores fully the computation state space of the distributed computation.

Use of the on-the-fly automata-theoretic approach to combat state explosion is not without precedent in dynamic property detection. An approach based on a similar idea has been used by Cooper and Marzullo in the algorithm presented in [24] for the detection of $Def\,\Phi$, where Φ is a predicate on global state. In that algorithm, the negation of the property (in this case $\neg\Phi$) is used to guide the search through the lattice: successors of a global state are explored only if the global state satisfies $\neg\Phi$. Detection of $Def\,\Phi$ holds iff a level of the lattice is reached in which no states satisfying $\neg\Phi$ are present. The approach is based on the observation that, once a path is known to satisfy Φ, then there is no need to explore any continuation of that path. This condition on which successors are explored in the detection of $Def\,\Phi$ can be viewed as a form of synchronous composition between the lattice of global states of the distributed computation and an automaton representing the negation of the property.

Potential Synergies

We now turn our attention to the potential synergies between the on-the-fly automata-theoretic approach to combating state explosion in model checking, and trace checking.

The first and most obvious potential synergy is the fact that automata-theoretic model checking approach is already used to check temporal properties in dynamic property detection with the BFR algorithm, introduced in Section 4.1. This makes it likely that incorporating the on-the-fly variant of the approach to combating state explosion in property detection may be achieved with a straightforward modification of existing algorithms. The fact that the key existing (and well-known) algorithms for checking temporal properties in trace checking based on the automata-theoretic approach are widely used strengthens this argument.

Another key advantage concerns the assumptions underlying the on-the-fly approach. The

method makes no assumptions concerning the property, other than it can be expressed as a finite state automaton, which means that the approach does not unduly constrain the dynamic properties which may be specified. Further, exploration can proceed in any fashion, as the method only requires the reachable states of the product automaton are correctly labeled in order to determine satisfaction. This is important in the face of competing algorithms used to explore the lattice of global states.

We also note that the method does not depend upon assumptions concerning termination of the exploration for correctness. Because of the fact that dynamic properties are necessarily restricted to those which may be decided on finite prefixes of sequential observations, it is a method which is essentially based on reachability analysis, in that determination of satisfaction depends solely upon exploring all reachable accepting states in the product automaton (as opposed to exploring all reachable accepting cycles in the product automaton). This is compatible with the fact that dynamic property detection algorithms generally do not terminate.

Finally, this method appears to be compatible with the modal operators $SOME$ and ALL, as well as Pos and Def (subject to the restriction that Pos and Def cannot be decided in general for non-terminating computations). Although making such a claim requires formal proof, we note informally that the approach explores all paths which do satisfy the property, and so if a path exists from the initial state to a state Σ for which a predicate holds, then this path will be explored by the on-the-fly approach.

5.2.2 Symbolic Methods

Symbolic methods to combating state explosion in model checking were presented in Section 3.2.2.

The idea behind the symbolic state space exploration approach is to encode the state transition system representing the program state space to be verified as a Boolean formula, defined over a suitable set of Boolean valued variables and represented efficiently using binary decision diagrams ($BDDs$). The model checking algorithm for symbolic model checking then operates on the BDD representation of the system, instead of a Kripke structure. Satisfiability may then be decided through the use of model checking algorithms which operate on BDDs.

Existing Synergies

As we saw in Section 4.2.6, Stoller [103] applied the symbolic approach to model checking to the case of checking $Pos\ \Phi$, where Φ is a simple predicate defined on the global state of the system. The idea behind the approach is to encode the dynamic property detection problem as a Boolean formula in such a way that a distributed computation $\gamma = (H, \rightarrow)$ satisfies the property if and only if the Boolean formulae evaluates to *true*.

In the work presented there, Stoller only considers predicates Φ on global state qualified by the modal operator Pos. Under this modality, the dynamic property detection problem is equivalent to determining if there exists a consistent global state of the distributed computation which satisfies Φ. In this approach, Boolean formulae are used to encode the variables of the processes involved in the global predicate, represented as $x_0, \ldots, x_{N-1}$, as well as the vector clock values recorded during monitoring, represented as $vc_{1,1}, \ldots, vc_{1,N}, \ldots, vc_{N,N}$. The formulation of the solution

presented there was suitable for off-line detection, in which all possible variable values for the process variables and vector clock variables recorded during monitoring are represented in the formula $\Phi(x) \wedge globalState_\gamma(x, vc) \wedge consis_\gamma(vc)$ representing the problem, *before* evaluation of the formula.

Potential Synergies

We now consider the potential of applying the symbolic model checking approach to the general dynamic property detection problem.

We consider first the approach of Stoller et al.. One of the chief difficulties in adopting this approach is discovering a suitable Boolean formula representation for a given property detection problem, taking modal operators into account. The encoding of $Pos\,\Phi$ is relatively straightforward, as the problem is equivalent to finding a consistent global state which satisfies the predicate. This is encoded as a set of constraints representing what it means to be a global state, what it means to be a consistent global state and what it means to satisfy the predicate. This view of the problem does not require the need to model the possible sequential observations (paths through the lattice of global states), described in the definition of satisfaction for Pos. In the encoding of $Def\,\Phi$, the modeling of a sequential observation is not so easily avoided. A similar problem will arise for other modal operators considered, such as $SOME$ and ALL. In this sense, the symbolic model checking approach as proposed by Stoller et al. may place significant constraints on the type of dynamic properties which can be checked.

An approach which avoids the need to find a suitable representation of the problem, as described above, is to perform detection of dynamic properties using a CTL model checker, which could be used to detect dynamic properties which may be expressed as CTL temporal formulae. In particular, the formula $Def\,\Phi$ is expressible as the CTL formula $AF\,\Phi$. However, the problem of encoding $SOME\,\Phi$ and $ALL\,\Phi$ remains, as these modal operators do not seem to be expressible in CTL.

Another potential difficulty with this approach is that, in order to adapt the approach to run-time dynamic property detection, some means of evaluating the Boolean formula and checking the dynamic property *without* having access to the complete distributed computation is required; otherwise, detection will need to be postponed until the end of the computation is reached. In the case of detecting $Pos\,\Phi$ using the approach of Stoller et al., this can be achieved, for example, by periodically evaluating the predicate on a *prefix* of the distributed computation. It is unclear how such a solution might work in the case of a CTL model checker employed to check general temporal formulae, as suggested above. On the other hand, the symbolic approach does seem well-suited to the problem of post-mortem dynamic property detection, where the full distributed computation is available.

Finally, it was noted by Stoller et al. that, in the symbolic algorithm implementing the detection of $Pos\,\Phi$, the algorithm used a far greater amount of memory than the enumerative dynamic property detection algorithm of Cooper and Marzullo [24]. This was due to the fact that the symbolic algorithm represented the complete lattice of global states during detection (by way of encoding all possible vector clock values in the formula representing the problem), as opposed to the algorithm of Cooper and Marzullo which only maintains two levels of the lattice at any point in time. With large computations, this aspect of the approach would also need to be avoided.

which again may be difficult in the case of checking temporal properties.

We may summarize by saying that there are several challenges associated with this approach which raise doubts about its suitability as a technique for combating state explosion in dynamic property detection, at least in the case of run-time property detection.

5.2.3 Model Extraction-based Methods

Model extraction-based methods for combating state explosion in model checking were presented in Section 3.2.3.

The idea behind model extraction-based methods is to reduce the size of the program state space considered in checking the property through irrelevant component elimination, data abstraction and component restriction and thereby reduce the effect of state explosion problem. Irrelevant component elimination is achieved through program slicing, based on eliminating program variables, program transitions, and even processes of the concurrent program, which do not affect verification of the property. Identification of such program elements is based on a static analysis of control and data relationships. The result is a smaller program model, with a reduced number of states, transitions and processes. Data abstraction, on the other hand, is based on replacing concrete variable domains of the variables of the concurrent program with (smaller) abstract variable domains. The resulting program contains the same number of variables and transitions, but with a smaller *potential* program state space (represented by the Cartesian product of the abstract domains of the variables involved in the program).

Model extraction-based methods are applied to the representation of the concurrent program (code) during the modeling phase to generate a smaller validation model. Model extraction based-methods therefore still rely on an existing model checking algorithm to perform model checking on this smaller model.

Existing Synergies

Dynamic program slicing in model checking has a close connection with the approaches of filtering and computation slicing in trace checking. Both filtering and computation slicing aim to reduce the size of the considered computation state space, but they do it in differing ways.

Computation slicing is based on using the structure of the predicate and its solution set to eliminate event combinations which do not lead to states which satisfy the property, and therefore for the purposes of detection, may be ignored. In this approach, the number of variables considered in each local state and the number of events considered in the distributed computation is not affected, however fewer transitions in the computation state space are explored, as now sets of events are fired atomically, eliminating intermediate global states which do not satisfy the property. In this sense, computation slicing corresponds to dynamic program slicing in that the set of states and transitions to be explored is reduced and the remaining states and transitions are sufficient to check the dynamic property. Filtering, on the other hand, is based on identifying variables and events of the distributed computation which are *relevant* to the detection of the property, and eliminating non-relevant variables and events from the lattice exploration. Filtering can be applied to the detection of general temporal properties, as long as those temporal properties are stuttering-invariant. Like computation slicing, filtering corresponds to dynamic program slicing in that the

set of states and transitions to be explored is reduced and the remaining states and transitions are sufficient to check the dynamic property. However, because filtering actually removes variables and events from the distributed computation, it could be argued that filtering bears a slightly stronger connection to dynamic program slicing than does computation slicing. Unlike dynamic program slicing, filtering does not depend upon complex dependency analysis, and can be implemented as a conservative approach, based on identifying events of the computation which potentially affect detection of the property.

With respect to data abstraction, there is no seeming connection between the use of data abstraction in model checking, and any existing state explosion technique in trace checking.

Potential Synergies

We now turn our attention to the potential use of program slicing and data abstraction in a trace checking context.

One potential synergy between model extraction-based methods and trace checking is that dynamic program slicing has potential for use in conjunction with filtering to produce a better approximation to the set of variables and transitions of the program which are relevant for detection of the property. A dynamic program analysis could be carried out before the modeling phase of trace checking, in order to identify those variables and transitions which were required in order to correctly detect the property considered. As this analysis would be based on control and data dependencies, it would represent a closer approximation to the optimal set of relevant variables and transitions than the conservative analysis, based on events which *potentially* change the predicate, employed in filtering. This approach would apply to general properties, including temporal predicates. In other words, dynamic program slicing could be used to produce a *better* filtering by identifying a smaller set of relevant variables and events. This strategy may be limited by the finite state assumption of model checking, upon which dynamic filtering may be dependent. Further, it would not avoid the inherent disadvantages of filtering, such as its inability to provide a full trace for notification or reaction purposes. In this respect, computation slicing has the advantage.

Another potential synergy concerns data abstraction. This technique could be used, during the execution and monitoring phase of trace checking and in conjunction with event generation, to reduce the size of the potential computation state space of the distributed computation by replacing concrete data values of local states of the distributed computation with abstract values from a (smaller) abstract domain. Although the number of variables represented in local states and number of events of the distributed computation would remain the same, this would reduce the size of the potential computation state space, represented by the Cartesian product of the variable domains. This could result in a smaller number of global states possible in each level of the lattice exploration (in the case of a level-based exploration) and so mitigate the problem of state explosion.

5.2.4 Partial Order-based Methods

The idea behind partial order-based methods is to exploit the properties of a different semantic model (the partial order semantic model, as opposed to an interleaving semantic model) in

combating state explosion.

Partial order-based methods were surveyed in Section 3.2.4. Two key partial order-based methods for combating state explosion were considered there: *partial order reduction*, and *the method of unfoldings*. In the case of partial order reduction, the idea is to generate a reduced program state space, based on the partial order information implicit in a dependency relation between program transitions, which contains enough interleavings of the full program state space in order to correctly verify the property. In the case of unfoldings, the idea is to represent program behaviour using a partial order-based representation (the unfolding) which is (potentially) exponentially more compact, and develop algorithms which operate on that representation.

The key opportunity, when considering the applicability of partial order-based methods to the dynamic property detection problem, is the close semantic match between partial order semantics, on the one hand, which underly partial order-based methods, and the partial order based representation of an asynchronous distributed system execution, as represented by a distributed computation.

In the case of partial order reduction, distributed computations may be interpreted in terms of *traces*, which were covered in Section 3.2.4, in the following way: for each state Σ of the distributed computation, the set of all sequential observations from the initial state to the state Σ define a finite trace. Indeed, the distributed computation itself defines a (possibly infinite) trace of equivalent sequential observations. In the case of the method of unfoldings, the distributed computation may be interpreted in terms of *configurations of an unfolding*: the distributed computation itself represents a (possibly infinite) configuration of the unfolding of the asynchronous distributed program. This close correspondence between the underlying semantic representations of partial order reduction (traces) and the method of unfoldings (configurations) to distributed computations represents a promising starting point.

In this section, we aim to explore the suitability of partial order based methods for combating state explosion in dynamic property detection.

Existing Synergies

We consider here the existing synergies between partial order methods and existing methods for combating state explosion in dynamic property detection.

In Section 4.2.6, we saw that the method of partial order reduction has been applied to the detection of dynamic properties. Stoller et al. [104] considered the application of partial order methods to solving the problems $Pos\,\Phi$ and $Def\,\Phi$, in the case where Φ is a simple predicate defined on global state. In that approach, the property detection problem was encoded as a deadlock detection problem in a modified system, in such a way that a distributed computation satisfied the property if and only if a deadlock existed in the modified system. The modified system was then checked for deadlock using a *persistent set* selective search. This approach was applicable to both a run-time and post-mortem context. The authors identified extending the method to cater for the detection of temporal properties as one area of future work.

Unlike partial order reduction, the method of unfoldings has not been explicitly used as an approach to combating state explosion in dynamic property detection. However, there are some interesting connections between the method of unfoldings, and certain results of Sen and Garg. For example, Sen and Garg prove in [98] that $Def\,\Phi$ holds iff a certain associated property holds

on all join-irreducible elements of the lattice. It can be shown that the join-irreducible elements of the lattice of global states are exactly the local configurations of the distributed computation (where the distributed computation itself is viewed as a configuration of an unfolding): those cuts of the lattice of the form $LC(e) = \{e' \in H | e' \rightarrow e\}$. Local configurations similarly play a very important role in unfolding based-algorithms.

Potential Synergies

Considering the close connection between trace semantics and distributed computations, it is very likely that partial order reduction methods could be used with success in dynamic property detection in the checking of temporal properties. Indeed, as we saw in Section 3.2.4.1, a very rich theory has been developed in model checking based on partial order reduction, and this theory has been used with success in model checking tools, such as *SPIN* [55].

The development of such an approach presents a number of interesting challenges. Firstly, partial order reduction, as a model checking technique associated with *LTL* model checking, does not make use certain of the modal operators which are commonly encountered in dynamic property detection, such as *SOME* or *ALL*. For the approach to be generally applicable within trace checking, it should be compatible with these modal operators. Secondly, partial order reduction has a complex associated theory, involving complex correctness proofs, based on assumptions of a finite number of program states and termination of state space exploration. Adapting this theory to the trace checking context could prove to be a challenging task, given these assumptions, and the related differences between the model checking context and the trace checking context. One additional limitation of such an approach involves the restriction that partial order reduction imposes on temporal properties: that they be stuttering-invariant. This would restrict the class of dynamic properties which could be detected using the method, which could be significant for applications like debugging, where temporal predicates need not be stuttering-invariant. Thus, despite the very promising connection between trace semantics and distributed computations, developing an approach to combating state explosion in dynamic property detection based on partial order reduction could present several significant challenges.

Similar opportunities and challenges hold for applying the method of unfoldings to combating state explosion in dynamic property detection. Concerning opportunities, as cited earlier, a distributed computation itself may be viewed as a configuration of the unfolding of a distributed program. This configuration is generated through execution of the distributed program, and, as such, comes 'for free'. This contrasts with the situation in model checking where the unfolding and configurations of the unfolding must be generated in an initial step. Another important opportunity concerns the structure of the distributed computation, viewed as a configuration of an unfolding. Configurations are, by definition, *conflict-free* (the notion of conflict in nets was introduced in Section 3.2.4.2). A special class of conflict-free nets are known as *marked graphs*. Informally, a net is a marked graph if every place has at most one input transition and one output transition. Esparza notes that many interesting problems concerning marked graphs are solvable in polynomial time [35]. This raises the possibility of developing algorithms for dynamic property detection which have polynomial complexity, based on viewing a distributed computation as a marked graph. Finally, as cited in Section 3.2.1, the method of unfoldings has been combined with the automata-theoretic approach, which may permit the detection of temporal properties

based on such an approach. Given the correspondence between join-irreducible consistent cuts of the lattice and local configurations of the unfolding, this suggests that unfolding-based algorithms could be directly applied to the distributed computation, viewed as a configuration.

However, as in the case of partial order reduction, there are challenges. Although the distributed computation represents a configuration which is obtained 'for free', existing methods based on unfoldings are based on exploring a finite prefix of the unfolding which is guaranteed to contain all reachable states, which are furthermore assumed finite in number. In the case of off-line dynamic property detection, it would appear that this assumption does not present a problem: the configuration representing either a terminating distributed computation, or a finite prefix of a non-terminating distributed computation, would be finite and would contain all reachable states of the terminating computation (or prefix of the non-terminating computation). In the case of on-line dynamic property detection, determining when an observed finite prefix of the configuration contains all reachable states suffers from the problem of the fact that we do not explore non-deterministic choices systematically, one of the differences between the model checking context and trace checking context cited earlier. In such a case, new algorithms (or substantial modifications to existing algorithms) for property detection based on unfoldings may be required.

5.2.5 Distribution

The distribution-based methods of model checking were presented in Section 3.2.5. The idea behind these methods for combating state explosion in model checking is to employ a divide and conquer approach, by making use of a set of processes to carry out the detection effort, as opposed to a single centralized process. This not only makes more memory and processing power available, and so increases the size of problems which can be solved, but also introduces the possibility of speedup, due to parallel processing.

Existing Synergies

Distribution is an approach to combating state explosion in model checking which has also been independently considered in trace checking. Indeed, in Chapter 4, we reviewed several distributed detection algorithms [40, 41, 42, 38, 58] for conjunctive and generalized conjunctive predicates with using the modal operators Pos and Def.

These examples of the distribution approach were based on using the structure of the predicate and predicate detection algorithm (in this case, conjunctive predicates and algorithms for detecting conjunctive predicates) to organize distribution. The approach effectively distributed both the time and space complexity of conjunctive predicate detection across a set of checker processes. Such distributed solutions to the trace checking problem not only increase the size of distributed computations which may be checked, but also contribute to the timeliness requirement of run-time dynamic property detection through potential speedup.

The key disadvantage of these methods is that they are tailored to one very specific class of predicates. In this sense, these approaches to distribution differ from the approaches to distribution used in model checking, where distribution of the general reachability problem and the general model checking problem are considered.

Potential Synergies

The application of distribution to the general reachability problem as considered in model checking represents a state space reduction technique for trace checking with very good potential. There are several reasons for this.

Firstly, reachability analysis, when combined with the automata-theoretic approach to trace checking, is general enough to permit the detection of dynamic properties which can be expressed as regular languages, which includes general safety properties. In particular, the formulation of the problem, based on the algorithm of [5] and presented in Chapter 4 reduces the detection of temporal properties to detecting reachable acceptance states in the product. In this sense, implementing a distributed reachability analysis algorithm, along the lines of that presented in Lerda et al. [67], would represent a distributed solution to the *general* trace checking problem.

Secondly, the general reachability approach would be compatible with the modal operators Pos and Def (subject to the restriction that Pos and Def cannot be decided in general for non-terminating computations) and $SOME$ and ALL, as the conditions for satisfaction of these modal operators are expressed solely in terms of reachable acceptance states of the product. The distribution-based algorithm is guaranteed to explore all reachable states of the product, and so would not affect satisfaction of modal operators.

Thirdly, as noted in Barnat et al. [8], breadth-first search is more amenable to parallelization than depth-first search, due to the fact that exploration is based on discovering successors in a *frontier* of states and this can simplify parallelization of the algorithm. This provides a match between the natural breadth-first evolution of events in a distributed computation with an easily parallelizable breadth-first search algorithm for conducting distributed reachability analysis.

In trace checking, the effect of cross transitions will be more significant than in model checking (again, due to the timeliness requirement on trace checking) and so partitioning of the state space in such a way as to minimize cross transitions would be an important issue. As noted in the work of Lerda et al., such an algorithm would need to provide a complete error trace, to be reported upon detection and upon which notification or reaction could be based.

5.2.6 Summary

In the previous sections, we investigated which techniques for combating state explosion in model checking show the greatest promise for application in trace checking.

As mentioned in the introduction to this section, this investigation was based on identifying any *existing* use of model checking techniques for combating state explosion in trace checking (existing synergies) as well as the *potential* use of model checking techniques for combating state explosion in trace checking (potential synergies).

We summarize the results of the above discussion in Table 5.1. In the table, in the case of identifying existing synergies, 'yes' indicates existing use of the technique in trace checking, and 'no' indicates no prior use of the technique in trace checking. In the case of potential synergies, we differentiate between the potential synergies for run-time trace checking and off-line trace checking. Potential synergies are classified as 'weak', 'strong', and 'very strong', reflecting, respectively, weak, strong, and very strong degrees of compatibility together with absence of significant potential challenges to application of the method.

Model Checking Approach	Existing Synergies	Potential Synergies (run-time)	Potential Synergies (off-line)
Automata-theoretic	yes	very strong	very strong
Symbolic	yes	weak	strong
Model Extraction (slicing)	yes	strong	strong
Model Extraction (data abs.)	no	strong	strong
Partial Order (reduction)	yes	very strong	very strong
Partial Order (unfolding)	no	weak	strong
Distributed	yes	very strong	very strong

Table 5.1:

5.3 Selection of Candidates for Further Development

As mentioned in the introduction to this chapter, the only way to conclusively determine the whether or not a model checking technique can be successfully applied in the context of dynamic property detection can really only be achieved through a detailed attempt at development of algorithms based on the approach.

In Chapters 6 and 7 of this thesis, we shall do exactly that: we shall engage in a detailed attempt at the development of algorithms for combating state explosion in trace checking, based on two model checking methods: on-the-fly automata-theoretic model checking, and partial order reduction.

Before we move to that development, we should like to present our reasons for selection of these two specific approaches, which as based on the comparative analysis of the model checking methods carried out in Section 5.1 and Section 5.2.

5.3.1 The Case for An On-the-Fly Approach to Combating State Explosion

Based on the observations made in the comparative analysis, we note the following potential synergies between an on-the-fly automata-theoretic approach to combating state explosion and trace checking:

1. This method for combating state explosion is based on the automata-theoretic approach to model checking, which is the basis of the several well-known, existing algorithms for the

detection of temporal dynamic properties [60, 5]. It is also possible that implementation may be achieved with a straightforward modification of existing algorithms.

2. The method does not make any assumptions concerning the property to be detected, thus supports wide applicability in detection of dynamic properties.

3. The method is essentially based on reachability analysis, and does not depend upon assumptions concerning the way in which the computation state space is explored.

4. The method appears to be compatible with the modal operators $SOME$ and ALL. as well as Pos and Def (subject to the restriction that Pos and Def cannot be decided in general for non-terminating computations), thus supports wide applicability in detection of dynamic properties with application-specific modal requirements.

For these reasons, we feel that the on-the-fly approach has a good chance of being successfully applied to the task of combating state explosion in property detection. In Chapter 6, we investigate the development of such an approach.

5.3.2 The Case for A Partial Order Approach To Combating State Explosion

Based on the observations made in the comparative analysis, we note the following potential synergies between an partial order reduction approach to combating state explosion and trace checking:

1. the trace semantics connection: the evolving distributed computation is semantically a trace - or more specifically a partial order representing a trace, and so there is an excellent semantic match between the distributed computation and trace theory, upon which partial order reduction is based.

2. partial order reduction is known to be highly successful in model checking, and is known to combine well with other approaches to combating state explosion (e.g. on-the-fly automata-theoretic approach [87], symbolic approach [4])

3. although the partial order reduction approach imposes a restriction of stuttering-invariance on properties to be detected, for many applications, such as testing and exception detection and handling, dynamic properties represent specifications of distributed computations and are generally stuttering-invariant. On the other hand, properties for applications such as debugging, certain dynamic properties may not be next-free (e.g. halting the debugger in a particular state).

4. partial order reduction has already been investigated as a technique for combating state explosion in the case of properties described in terms of global states [104]. Thus, a further investigation of the technique for the case of temporal properties will provide a point of comparison.

For these reasons, we feel that partial order reduction has a good chance of being successfully applied to the task of combating state explosion in dynamic property detection. On the other hand, we noted certain challenges with this approach:

1. partial order reduction was developed for the case of LTL model checking, in which satisfaction is based on *all* interleaving sequences satisfying the desired temporal property (which corresponds to the Def modality of trace checking). Partial order reduction was not designed for use with certain modal operators found in trace checking, such as Pos, $SOME$, and ALL.

2. partial order reduction has a complex associated theory, involving complex correctness proofs, based on assumptions of finite program state space, which does not hold in a trace checking context

These challenges, although potentially making development of an algorithm more difficult, also have the potential advantage of illustrating the difficulties which may be encountered in adapting a technique from model checking to the trace checking context. Identifying such difficulties and attempting to resolve them are also an important part of this investigation. In Chapter 7, we investigate the development of such an approach.

5.4 Summary

In this chapter, we investigated the similarities and differences between model checking techniques and their suitability for use as techniques for combating state explosion in trace checking. We also identified two candidate techniques for further development: the on-the-fly automata theoretic approach, and the partial order approach. In the following two chapters, we present the development of detection algorithms for temporal properties which incorporate these techniques for combating state explosion.

Chapter 6

An On-the-Fly Automata-theoretic Approach to Combating State Explosion

6.1 Introduction

In the previous chapter, we outlined several reasons why the *on-the-fly* approach to combating state explosion in model checking is a promising candidate for combating state explosion in the detection of temporal properties in trace checking. In this chapter, we present the development of such an algorithm.

The essence of this approach is to use the automaton defining the dynamic property to guide the search and avoid exploring paths through the computation state space which are known not to satisfy the property.

In order to implement the on-the-fly approach to combating state explosion, we shall begin with an existing detection algorithm, based on the automata-theoretic approach, and adjust the algorithm to incorporate on-the-fly state space reduction. The algorithm of Babaoglu, Fromentin, and Raynal, presented in [5], is an ideal candidate: it is an algorithm for the detection of temporal properties (expressed as automata recognizing regular languages) in trace checking, is based on an automata-theoretic approach, and explores the computation state space exhaustively. Using this algorithm as a base, we adjust the exploration of the computation state space in such a way that continuations of sequential observations which are known not to satisfy the property are not explored.

As we shall show, the resulting on-the-fly version of the Babaoglu-Fromentin-Raynal algorithm will exhibit the following desirable features:

- supports detection of properties φ which can be described as regular languages
- can be used in an on-line or off-line context
- computations need not satisfy a finite state assumption

- is compatible with modal operators $SOME$ and ALL, and Pos (subject to the restriction that Pos cannot be decided in general for non-terminating computations)

The rest of the chapter is organized as follows. In Section 6.2, we present the necessary background on the existing automata-theoretic approach of Babaoglu-Fromentin-Raynal and certain definitions concerning deterministic finite state automata, used to represent regular properties. In Section 6.3, we consider the design issues to be considered in developing such an algorithm and, in Section 6.4, present a description of the on-the-fly algorithm. In Section 6.5, we demonstrate the correctness of the algorithm, focusing on an invariance property which ensures invariance of the modal operators. Section 6.6. examines the complexity of the algorithm. In Section 6.7, we summarize and present conclusions.

6.2 Background

In this section, we review the automata-theoretic approach to dynamic property detection of Babaoglu, Fromentin and Raynal [5], discussed briefly in Chapter 4. In the interests of brevity, we shall sometimes refer to this algorithm as the BFR algorithm.

The approach is based on viewing the lattice of global states as a labeled DAG and defining satisfaction through the use of the modal operators $SOME$ and ALL. As mentioned in Section 4.1, when defining the temporal predicate detection problem, modal operators may be defined based on two views of the distributed computation: viewing the distributed computation as a set of sequential observations and defining satisfaction in terms of modal operators Pos and Def; or viewing the computation as a DAG of global states, and defining satisfaction in terms of global states instead, using the modal operators $SOME$ and ALL.

6.2.1 Detection of temporal properties

In this section, we summarize the concepts required in order to understand the modifications to the algorithm we shall introduce later.

The original paper [5] presented the development of the algorithm in terms of a general graph $G = (V, E)$ and an alphabet A, later instantiated as the lattice of global states and a set of atomic propositions defined on global states, respectively. In Section 4.1, we showed how the lattice of global states can be viewed as a labeled DAG. In this summary of the algorithm, we shall refer directly to the lattice and the set of global propositions. In order to describe the algorithm, we need to establish some notation related to the view of the lattice of global states as a labeled DAG.

Graph Labeling and Languages

In what follows, we assume that $\gamma = (H, \rightarrow)$ is a distributed computation, $\mathcal{L} = (\Sigma_\gamma, \prec_\gamma)$ is the associated lattice of global states. Let AP be a set of atomic propositions defined on global states, and $\lambda : \Sigma_\gamma \rightarrow 2^{AP}$ be an associated labeling function.

Let $DAG_\gamma = (\Sigma_\gamma, \prec_\gamma^{im})$ be the directed acyclic graph associated with the lattice of global states $\mathcal{L} = (\Sigma_\gamma, \prec_\gamma)$. Given a global state $\Sigma \in \Sigma_\gamma$, let DAG_Σ denote the subgraph of DAG_γ consisting

of the global state Σ and its predecessors. The subgraph DAG_Σ is itself a directed acyclic graph. A *directed path* of DAG_Σ is a sequence of global states $\pi_\Sigma = \Sigma_0\Sigma_1\Sigma_2\ldots\Sigma_k$ starting at the initial state and ending at Σ. Let Π_Σ denote the set of all directed paths in DAG_Σ. Directed paths in the DAG_Σ correspond to sequential observations in the lattice starting from the initial state and terminating at Σ. The set of all directed paths Π_Σ in DAG_Σ corresponds to a trace of equivalent sequential observations.

Given a directed path $\pi_\Sigma = \Sigma_0\Sigma_1\Sigma_2\ldots\Sigma_k$, a *labeling* of the path π_Σ is defined to be a word $w_0w_1\ldots w_k$ such that, $\forall i : 0 \leq i \leq k$, $w_i \in \lambda(\Sigma_i)$. Each directed path π_Σ may have several possible labellings. The labeling of a directed path, denoted by $\tilde{\lambda}(\pi_\Sigma)$, defines a *set* of words, representing all possible labellings. The set of labellings for all directed paths in Π_Σ defines a language $L^\lambda(\Sigma)$ associated with each global state Σ. The language is defined as $L^\lambda(\Sigma) = \bigcup_{\pi_\Sigma\in\Pi_\Sigma}\tilde{\lambda}(\pi_\Sigma)$. Path labellings in DAG_Σ correspond to propositional sequences, obtained from sequential observations in the lattice starting from the initial state and terminating at Σ which have had their states labeled with atomic propositions.

In this way, with each state Σ of the lattice $\mathcal{L}$, is associated a language of propositional sequences $L^\lambda(\Sigma)$.

Dynamic Properties

Dynamic properties aim to characterize the temporal evolution of states in the distributed computation. In the framework of Babaoglu-Fromentin-Raynal, a dynamic property is defined by a language of words over the alphabet AP. The convention is to differentiate between a dynamic property φ and the set of words $L(\varphi)$ which characterize it. In this view, each word of the language is viewed as satisfying the property.

Dynamic properties are assumed to be such that they define regular languages. *Regular languages* are those which can be specified by regular expressions, or equivalently, be recognized by a finite automaton [57]. This feature of regular languages makes the specification of temporal properties very convenient. We assume in the sequel that the automaton describing the regular language is a deterministic finite automaton (DFA), with a possible dead (or trap) state defined. The definition of a deterministic finite automaton was presented in Section 3.2.1.

Given a property φ represented by the language $L(\varphi)$, satisfaction of the property φ by a distributed computation in this framework is defined in terms of the associated labeled DAG of global states, as follows:

Definition 6.1. Given an alphabet AP of global predicates, a lattice of global states $\mathcal{L}$, a state labeling function λ, a state Σ of $\mathcal{L}$ and a property φ represented by the language $L(\varphi)$, $\Sigma \models SOME\,\varphi$ if and only if there exists some labeling of at least one directed path terminating in Σ which defines a word in $L(\varphi)$.

Definition 6.2. Given an alphabet AP of global predicates, a lattice of global states $\mathcal{L}$, a state labeling function λ, a state Σ of $\mathcal{L}$ and a property φ represented by the language $L(\varphi)$, $\Sigma \models ALL\,\varphi$ if and only if all labellings of all directed paths terminating in Σ define words in $L(\varphi)$.

In the next section, we review the algorithms for determining satisfaction of temporal properties under these modalities.

Detection of dynamic properties

In the spirit of the automata-theoretic approach, the satisfaction relations defined above for the modal operators $SOME$ and ALL can be expressed in terms of the languages associated with the dynamic property. In particular, when expressed in terms of languages:

$$\begin{aligned} \Sigma \models SOME\,\varphi \quad &\text{iff} \quad L^{\lambda}(\Sigma) \cap L(\varphi) \neq \emptyset \\ \Sigma \models ALL\,\varphi \quad &\text{iff} \quad L^{\lambda}(\Sigma) \subseteq L(\varphi) \end{aligned}$$

Thus, deciding satisfaction of a property over a distributed computation can be achieved by determining if the language relations described above hold true. These relations in turn can be expressed in terms of reachable states of the automaton which recognizes the property, in the following way.

Given a property φ, a lattice $\mathcal{L}_\gamma$ and a state $\Sigma \in \Sigma_\gamma$, suppose that the property φ is recognized by the deterministic finite automaton $A_\varphi = (Q, \Sigma, \delta, Q_0, Q_F)$. Let $R^\varphi(\Sigma)$ denote the set of all states of A_φ which are reached after processing all words in $L^\lambda(\Sigma)$. The relations become

$$\begin{aligned} L^{\lambda}(\Sigma) \cap L(\varphi) \neq \emptyset \quad &\text{iff} \quad R^{\varphi}(\Sigma) \cap Q_F \neq \emptyset \\ L^{\lambda}(\Sigma) \subseteq L(\varphi) \quad &\text{iff} \quad R^{\varphi}(\Sigma) \subseteq Q_F \end{aligned}$$

Thus, deciding satisfaction reduces to deciding if the relations above hold. The authors show that the sets $R^\varphi(\Sigma)$ can be calculated inductively, based on the following relation:

$$R^{\varphi}(\Sigma) = \bigcup_{\alpha \in \lambda(\Sigma), q \in R^{\varphi}_{pred(\Sigma)}} \delta(q, \alpha)$$

where

$$R^{\varphi}_{pred(\Sigma)} = \bigcup_{u \in pred(\Sigma)} R^{\varphi}(u)$$

and $pred(\Sigma)$ denotes the immediate predecessors of the state Σ in the lattice.

The inductive calculation of the sets $R^\varphi(\Sigma)$ above can in turn be performed by a breadth-first search of the lattice. The algorithm is shown in Figure 6.1. The algorithm works in conjunction with an algorithm for exploring the lattice of global states of the distributed computation. The exploration is exhaustive, in the sense that all states in lattice of global states are explored.

For each global state Σ, the associated set of reachable automaton states $R^\varphi(\Sigma)$ is represented in the algorithm by the Boolean array $B_\Sigma[Q]$. $B_\Sigma[q]$ evaluates to *true* iff there is a labeling of a path leading from the initial state Σ^0 to Σ, whose corresponding run of the automaton leads to state q. A fictitious state Σ^{-1} is introduced as the unique predecessor of the initial state Σ^0. The corresponding Boolean array $B_{\Sigma^{-1}}$ is such that $B_{\Sigma^{-1}}[q_0]$ is the only array element which is initially *true*.

Algorithm 6.1 makes use of two additional Boolean arrays $B_{pred}[Q]$ and $B^\alpha[Q]$ to represent intermediate sets of reached automaton states. Given a state Σ, $B_{pred}[Q]$ represents the set of

```
1   previous := {Σ^{-1}}
2   current := {Σ^0}

3   while (current ≠ ∅)
4       foreach Σ ∈ current do
5           foreach q ∈ Q do B_pred[q] := ⋁_{Σ'∈pred(Σ)} B_Σ'[q] od
6           foreach α ∈ λ(Σ) do
7               B^α[Q] := (false, ..., false)
8               foreach (q ∈ Q : B_pred[q] = true) do
9                   foreach r ∈ δ(q, α) do B^α[r] := true od
10              od
11          od
12          foreach q ∈ Q do B_Σ[q] := ⋁_{α∈λ(Σ)} B^α[q] od
13      od
14      previous := current
15      current := {global states directly reachable from those in previous}
16  end while
```

Figure 6.1: Algorithm to compute the set of automaton states $R^{\varphi}(\Sigma)$.

automaton states reached by considering all paths leading from the initial state to immediate predecessors Σ' of Σ; that is, $B_{pred}[q] = true$ if and only if there is a labeling of a path leading from the initial state to some predecessor Σ' of Σ, whose corresponding run of the automaton leads to state q. The array $B_{pred}[Q]$ is calculated in line 5 of the algorithm. $B^{\alpha}[Q]$ represents the set of automaton states reached by runs corresponding to path labellings of paths leading from the initial state to Σ which have a final label of α. Given $\alpha \in \lambda(\Sigma)$, the array $B^{\alpha}[Q]$ is calculated in lines 7-11. Finally, the arrays $B^{\alpha}[Q]$ are used to calculate $B_{\Sigma}[Q]$ in line 12.

This exploration may be carried out using the level-based approach of [24] or the linearization-based approach of [28]. In the development of our algorithm for combating state explosion which follows, we shall assume that computation state space exploration is achieved using the level-based algorithm of Cooper-Marzullo.

Using the above algorithm for computing the sets $R^{\varphi}(\Sigma)$ for each global state of the lattice, deciding satisfaction of a temporal property φ in a global state Σ is achieved simply by checking the satisfaction relations defined earlier for $SOME\,\varphi$ and $ALL\,\varphi$ at state Σ.

6.2.2 The on-the-fly approach to automata theoretic model checking

We review the key ideas behind the on-the-fly version of automata-theoretic model checking.

The automata-theoretic approach views the model checking problem in terms of formal languages: checking that $L_P \subseteq L_{\varphi}$, where L_P is the language of the program, and L_{φ} is the language corresponding to the property φ. Checking this relation holds is equivalent to checking that $L_P \cap L_{\neg\varphi} \neq \emptyset$ does not hold. Given automata A_P and $A_{\neg\varphi}$ recognizing L_P and $L_{\neg\varphi}$, respectively, this in turn is equivalent to checking if the automaton which accepts the intersection $L_P \cap L_{\neg\varphi}$ is non-empty. This automaton is denoted by $A_P \cap A_{\neg\varphi}$.

In the on-the-fly version, rather than constructing the automaton A_P in a separate step and

using A_P and $A_{\neg\varphi}$ to compute the automaton accepting the intersection, $A_P \cap A_{\neg\varphi}$. the intersection is viewed as a synchronous product $A_P \times A_{\neg\varphi}$ between the two automata and it is this product space which is explored. This approach can be viewed as using the automaton to guide the exploration of the state space: in the synchronous product, only interleaving sequences which allow *both* the program automaton and the property automaton (which, in this case, represents the *negation* of the desired temporal property) to successfully transition are explored. More formally. if (s, a) is a state of the product state space $A_P \times A_{\neg\varphi}$, then a transition to a possible successor state (s', a') will only occur if the following conditions hold: (i) (s, t, s') is a transition of the automaton A_P, for some t (ii) (a, l, a') is a transition of $A_{\neg\varphi}$, for some set of possible input symbols l, and (iii) $L(s') \in l$, where $L(s')$ is the labeling of the successor state s'.

In this way, we need only keep as much of the automaton A_P in memory as is needed to explore the product. In the case where φ is represented by a finite deterministic automaton on finite words (which includes the class of safety properties), checking non-emptiness of the intersection is equivalent to finding reachable acceptance states of the product. In the case where φ is represented by a Buchi automaton on infinite words (which includes the class of liveness properties), checking non-emptiness of the intersection is equivalent to finding reachable acceptance cycles of the product.

In on-the-fly automata theoretic model checking, there is only a single notion of satisfaction: language containment (i.e. $L_P \subseteq L_\varphi$). This is due to the fact that in LTL model checking, it is implicitly assumed that the concurrent program satisfies the LTL property only when *all* interleaving sequences satisfy the property. Further, the languages associated with the property and the program are defined over the alphabet 2^{AP}, due to the fact that automata representations for LTL properties are defined over the alphabet 2^{AP} [114].

In the remainder of this chapter, we shall adapt the BFR algorithm to incorporate on-the-fly model checking, by using the automaton representing the property to guide the exploration, and so avoid exhaustive exploration of the lattice of global states.

6.3 Design Issues

In this section, we consider the general issues involved in introducing the on-the-fly automata-theoretic approach from model checking into the context of detection of dynamic properties. In particular, we consider:

1. assumptions underlying the approach of on-the-fly automata-theoretic model checking
2. any issues which arise due to the difference between assumptions underlying the on-the-fly automata-theoretic approach to model checking, on the one hand, and the new *context* for the proposed algorithm and any related requirements, application or otherwise, on the other.

The issues were considered briefly in the survey section of the thesis. We reconsider them here in more detail, as they potentially impact on the detailed design of our algorithm. Given that an algorithm for detection of dynamic properties based on the automata-theoretic approach already exists, our consideration lies in how these issues relate to the incorporation of the on-the-fly state space reduction.

Assumptions underlying the approach

In this section, we consider the on-the-fly automata theoretic method and its suitability at the level of algorithmic design assumptions. In particular, we discuss the key assumptions underlying the method, and consider the possibility of conflicts with the new trace checking context.

- **finite state assumption:** in the property detection problem, there is generally no assumption that the program being monitored is finite state, whereas in model checking there is. In model checking, the finite state assumption is used simply to ensure that the exploration of the product state space will always terminate. As mentioned in Section 5.1, we can only ever analyze a finite prefix of any distributed computation (even when the computation is infinite), and so a corresponding analysis of the distributed computation will also always terminate (either by reaching the end of a terminating distributed computation, or by reaching the end of the finite prefix of the non-terminating distributed computation, when termination is initiated by the user). Further, as the automata-theoretic approach effectively translates the trace checking problem into a reachability problem (see below), given the above, such a reachability problem will also terminate in the property detection case, even when the program being analyzed is not finite state.

- **class of properties:** model checking generally considers the checking of general temporal properties (safety and liveness), but in dynamic property detection, we can only check properties which can be decided based on observing at most finite prefixes of distributed computations. The on-the-fly approach for model checking was first introduced by Jard et al. [59], where it was shown that the model checking problem for properties which define regular languages (which includes the class of safety properties) translates into a reachability problem (reachability of product states in a specific synchronous product, as opposed to reachability of accepting cycles). Thus, in translating the on-the-fly method to check properties which define regular languages (and so the class of safety properties), we need only perform a standard reachability analysis of a product, locating reachable acceptance states, as opposed to the case of checking liveness properties, where we must locate reachable acceptance cycles.

- **exploration order:** algorithms in the literature for on-the-fly automata-theoretic model checking are presented in terms of a depth-first exploration, whereas property detection is generally based on breadth-first exploration. Unless algorithms are proved correct in an exploration-independent manner, this becomes a correctness issue. In this case, because the problem for safety is effectively a reachability problem in a product space, the correctness of the algorithm does not depend upon the exploration order. Note that this would not be the case for checking liveness properties, which equates to checking for reachable accepting cycles in the product space - there, exploration order does effect the detection of cycles.

- **modal operators and invariance:** in on-the-fly model checking of LTL properties, there is an implicit assumption that all interleaving sequences must satisfy the property (equivalent to applying a single modal operator, similar to Def), whereas, in property detection, a variety of application-dependent modal operators are used, such as Pos, Def, $SOME$, and ALL. An important issue in applying on-the-fly reduction is whether these modal operators

are preserved under the transformation we introduce. In other words, we need to ensure that the reduction does not invalidate the correctness of detection of properties quantified by these operators.

These issues arise when we consider adapting the on-the-fly approach to the context of dynamic property detection. Of these, we shall carefully address the invariance of modal operators in Section 6.5.

6.4 Design

As noted in Section 6.2, the Babaoglu-Fromentin-Raynal algorithm for detection of dynamic properties is an existing algorithm which is based on the automata-theoretic approach. As we shall demonstrate, this algorithm can be modified in order to obtain an implementation of the on-the-fly automata theoretic approach.

6.4.1 On-the-fly automata-theoretic approach

The Babaoglu-Fromentin-Raynal algorithm already exhibits several features of the on-the-fly method. It is based on the automata-theoretic approach, in that properties and computations are represented as languages, and satisfaction is represented in terms of relations between those languages. It also explores the computation state space and automaton state space simultaneously, constructing a form of 'product' state space, as in the on-the-fly method. However, the 'product' state space explored does not correspond to a synchronous product as in on-the-fly model checking: the entire computation state space is explored, and the corresponding automaton states generated. This results in the BFR algorithm exhaustively exploring the computation state space.

The key idea in on-the-fly automata-theoretic model checking is to use the automaton to guide the exploration of the program state space, by not following continuations of interleaving sequences which do not allow both the program automaton and the property automaton to transition successfully. This approach allows the algorithm to keep only that part of the program state space in memory which is needed to discover the reachable states of the product which potentially satisfy the property. This is achieved through the definition of the synchronous product.

In trace checking, the same principle can be applied, in that when checking for satisfaction of a dynamic property φ over the computation state space, we use the automaton A_φ representing the property to guide the search in such a way that sequential observations which are known *not* to satisfy the property are not explored. In order to do this, we need to simulate the synchronous product used in on-the-fly model checking.

6.4.2 Adjustments required

We simulate a synchronous product in the BFR algorithm by introducing a mechanism corresponding to that used in the on-the-fly synchronous product: a transition of the product state space is enabled iff (i) each of the component transitions are enabled and (ii) the input symbols contained in the label of the computation state component reached in the product state are accepted by the automaton.

In the case of a deterministic finite automaton A_φ representing the dynamic property φ, a propositional sequence corresponding to a labeled sequential observation is guaranteed not to satisfy the property if the run of the automaton corresponding to that propositional sequence leads to a dead state of the automaton. This condition can be decided in a state Σ on the sequential observation. In other words: in order to avoid this exhaustive exploration, we note that if, in exploring a path leading to Σ, we find we reach a state (Σ, a) of the product where a is a dead state, then all continuations of that path will leave the automaton at that dead state. Since a dead state can never lead to an accepting state, all continuations of this path need not be explored. Not exploring such continuations will not affect correct detection, but will result in fewer paths being explored.

We can adjust the existing detection algorithm to incorporate this reduction. Remember that the detection algorithm consists of two algorithms operating in interleaved fashion: the Cooper-Marzullo algorithm used to generate the states Σ of the computation state space, and the Babaoglu-Fromentin-Raynal algorithm, used to calculate the reachable automaton states $R^\varphi(\Sigma)$.

Let (Σ, a) be a state of the product space along a path, and let (Σ, e, Σ') be a transition in the computation state space. According to the BFR definition of satisfaction (based on languages over AP), this transition in the computation state space generates one or more corresponding transitions $a \rightarrow a'$ in the automaton state space: $a \rightarrow a'$ if for some label α in $\lambda(s')$ there is a transition (a, β, a') where $\alpha = \beta$. We need not follow a transition in the computation state space if we know that *all* corresponding automaton transitions lead to the dead state. We still want to push all automaton states through for the transitions we do explore, as we know by construction that at least one of them has a corresponding automaton transition which does not lead solely to trap.

This is the essence of the modification to the automata-theoretic approach which we will make.

6.4.3 Description of the Algorithm

In this section, we describe our on-the-fly algorithm for detecting temporal properties in a trace checking context.

The key change to the operation of the detection algorithm of Babaoglu-Fromentin-Raynal is to introduce the function *guided*() to determine, given a reachable state Σ of the computation state space, a subset of enabled transitions which should be explored, in the computation state space exploration part of the algorithm. That is, in the Cooper-Marzullo component of the algorithm, the function *guided*(Σ) is called to determine the set of enabled transitions to be explored, instead of the usual function *enabled*(). Figure 6.2 shows the function *guided*(Σ).

The function *guided*(Σ) returns a subset of the set *enabled*(Σ) of transitions enabled at state Σ such that, for each event e in the subset, the successor state $e(\Sigma)$ will have a set of associated automaton states $B_\Sigma[Q]$ which include a non-trap state.

The function works as follows. For each event $e \in$ *enabled*(Σ), the function determines if there is an automaton state in $B_\Sigma[Q]$ (lines 6-12), representing the current set of reachable states $R^\varphi(\Sigma)$. which can cause the automaton to transition on some symbol $\alpha \in \lambda(\Sigma')$ and end up at a non-trap state (line 8), where Σ' is the successor of Σ under the event e. In this case, the flag *syncpossible* is set for the event e (line 9). Otherwise, all transitions of the automaton possible from states in

```
1   procedure guided(Σ)

2   GT := {}
3   foreach (e ∈ enabled(Σ)) do
4       syncpossible := false
5       Σ' := e(Σ)
6       foreach (q ∈ Q : B_Σ[q] = true) do
7           foreach (α ∈ λ(Σ')) do
8               if (δ(q,α) is defined ∧ δ(q,α) ≠ trap) then
9                   syncpossible := true
10              fi
11          od
12      od
13      if (syncpossible = true) then
14          GT := GT ∪ {e}
15      fi
16  od
17  return GT

18  end
```

Figure 6.2: Algorithm for computing the set of guided transitions

$B_\Sigma[Q]$ will lead to trap, and the flag *syncpossible* will remain false. This represents the case in which enabling and firing the event e would lead to a state Σ' for which the only automaton state reached was trap. Thus, the flag *syncpossible* represents the fact that the successor state Σ' of an event e will have a set of reachable states $B_{\Sigma'}[Q]$ in which some non-trap state is included.

6.5 Correctness

In this section, we consider the correctness of the algorithm modification. Specifically, we need to show that, given a lattice and a property φ, if $\Sigma \models SOME\,\varphi$ holds in the full computation state space, then it will be detected by the modified algorithm.

This will involve proving the following invariant, informally stated as follows: if, for some state Σ, $\Sigma \models SOME\,\varphi$ in the full computation state space, then the state Σ will be reached in the reduced state space and the set $R^\lambda(\Sigma)$ will be the same, up to non-*trap* states. On the other hand, if the property is not satisfied, the state will not be explored.

In this section, we shall also comment on why the approach does not work for the modal operator ALL.

An Invariant

The proof of correctness is based on the following invariant: let Σ be a state in the full state space for which $R^\varphi(\Sigma) \neq \{trap\}$. Then:

1. Σ is reached in the reduced state space and

2. Σ has the set of a-states $R^{\varphi}_{red}(\Sigma)$ in the reduced state space, and which satisfy $R^{\varphi}_{red}(\Sigma) = R^{\varphi}(\Sigma) \vee R^{\varphi}(\Sigma) \setminus \{trap\}$

This invariant needs to be proved by induction, using the idea of the construction embodied in *guided*(). The proof involves identifying the sets $R^{\varphi}(\Sigma)$ and showing that the equalities hold. We now prove the invariant. Induction is based on the length of the path needed to reach a state.

Base Case: in the development of the algorithm in [5], the authors define a fictitious initial state Σ_{fic} which is an immediate predecessor of all states in the lattice which have no predecessor (i.e. the unique minimal element of the lattice) and define $R^{\varphi}(\Sigma_{fic}) = \{q_0\}$. For this initial state Σ_{fic}, it will be reached in the reduced state space and $R^{\varphi}_{red}(\Sigma_{fic}) = \{q_0\} = R^{\varphi}(\Sigma_{fic})$. Thus, the induction hypothesis holds in the initial state.

Induction Step: Now suppose that Σ is a state in the full state space for which $R^{\varphi}(\Sigma) \neq \{trap\}$. We need to show that:

1. Σ is reached in the reduced state space and
2. $R^{\varphi}_{red}(\Sigma) = R^{\varphi}(\Sigma) \vee R^{\varphi}(\Sigma) \setminus \{trap\}$

We first show that Σ is reached in the reduced state space. $R^{\varphi}(\Sigma) \neq \{trap\}$, so there is a state $q \in R^{\varphi}(\Sigma)$ such that $q \neq trap$ and a path $\pi = \Sigma_0\Sigma_1 \ldots \Sigma_k$ and associated labeling in the full computation state space such that $\Sigma_0 = \Sigma_{fic}$ $\Sigma_k = \Sigma$ and for each state Σ_i, the automaton state associated with Σ_i is not a trap state (these must be non-trap states, otherwise the final state q would be a trap state). In particular, there is an immediate predecessor state Σ_{pred} of Σ, a state $q_{pred} \in R^{\varphi}(\Sigma_{pred})$ such that $q_{pred} \neq trap$, and a label $\alpha \in \lambda(\Sigma)$ such that $\delta(q_{pred}, \alpha) = q$. Because the induction hypothesis holds, the state Σ_{pred} is reached in the reduced state space, and $R^{\varphi}_{red}(\Sigma_{pred}) = R^{\varphi}(\Sigma_{pred}) \vee R^{\varphi}(\Sigma_{pred}) \setminus \{trap\}$. By the construction of *guided*(), in the reduced state space, the transition from Σ_{pred} to Σ will be selected in *guided*(Σ_{pred}) and $q \in R^{\varphi}_{red}(\Sigma)$. Thus, Σ is reached in the reduced state space.

We now show that $R^{\varphi}_{red}(\Sigma) = R^{\varphi}(\Sigma) \vee R^{\varphi}(\Sigma) \setminus \{trap\}$. Because the non-trap state $q \in R^{\varphi}(\Sigma)$ was arbitrary, the above argument also shows that $q \in R^{\varphi}_{red}(\Sigma)$. Now we consider the fate of a trap state in $R^{\varphi}(\Sigma)$. If $q \in R^{\varphi}(\Sigma)$ and $q = trap$, then there is a path $\pi = \Sigma_0\Sigma_1 \ldots \Sigma_k$ and associated labeling in the full computation state space such that $\Sigma_0 = \Sigma_{fic}$ $\Sigma_k = \Sigma$ and for some state Σ_i, $1 \leq i \leq k$, the automaton state associated with Σ_i becomes a trap state and remains so until Σ_k. Let Σ_{pred} be the predecessor state of Σ in that path. By the induction hypothesis, $R^{\varphi}_{red}(\Sigma_{pred}) = R^{\varphi}(\Sigma_{pred}) \vee R^{\varphi}(\Sigma_{pred}) \setminus \{trap\}$.

The case $R^{\varphi}_{red}(\Sigma_{pred}) = R^{\varphi}(\Sigma_{pred}) \setminus \{trap\}$ is not consistent with the path being explored up to state Σ_{pred} in the reduced state space, and corresponds to the case where the exploration of that path was not continued, by an earlier transition in the path not being included in *guided*(). In this case, the *trap* state of the path will not be added to $R^{\varphi}_{red}(\Sigma)$.

The case $R^{\varphi}_{red}(\Sigma_{pred}) = R^{\varphi}(\Sigma_{pred})$ leads to two possibilities: the trap state may be added to the set $R^{\varphi}_{red}(\Sigma)$ if there is another non-trap state in $R^{\varphi}_{red}(\Sigma_{pred})$ and a label in $\lambda(\Sigma)$ which causes the transition Σ_{pred} to Σ to be selected in *guided*(Σ). If there is no such state-label pair, *guided*() will not explore the transition and the trap state will not be added to the set $R^{\varphi}_{red}(\Sigma)$.

The path was arbitrary, and for each such path, the trap state may or may not be added to $R^{\varphi}_{red}(\Sigma)$. Thus, $R^{\varphi}_{red}(\Sigma) = R^{\varphi}(\Sigma) \vee R^{\varphi}(\Sigma) \setminus \{trap\}$.

QED

Invariance of modal operator

From the above invariant, we may conclude that the modal operators $SOME$ is invariant under the reduction introduced by the *on-the-fly* transformation. The proof follows.

Theorem 6.1. $\Sigma \models SOME\,\varphi$ in the full computation state space if and only if $\Sigma \models SOME\,\varphi$ in the reduced computation state space.

Proof. If $\Sigma \models SOME\,\varphi$ in the full computation state space, then $R^{\varphi}(\Sigma) \cap Q_F \neq \emptyset$. Since Q_F does not contain *trap*, then, by the invariant, state Σ is reached in the reduced state space and $R^{\varphi}_{red}(\Sigma) \cap Q_F \neq \emptyset$. Thus, $\Sigma \models SOME\,\varphi$ in the reduced computation state space.

Conversely, if $\Sigma \models SOME\,\varphi$ in the reduced computation state space, then $R^{\varphi}_{red}(\Sigma) \cap Q_F \neq \emptyset$. Since, by the invariant, $R^{\varphi}_{red}(\Sigma) \subseteq R^{\varphi}(\Sigma)$, then this implies that $R^{\varphi}(\Sigma) \cap Q_F \neq \emptyset$. Thus, $\Sigma \models SOME\,\varphi$ in the full computation state space.

□

We have shown that the method is compatible with the modal operator $SOME$. One immediate consequence of this fact is that the method may also be used with modal operator *Pos* in the case of *terminating* distributed computations, given the relations:

$$\gamma \models Pos\,\varphi \quad \text{iff} \quad \Sigma_{final} \models SOME\,\varphi$$

where Σ_{final} represents the maximal state of the lattice of global states $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)$. In the case of non-terminating distributed computations, correct conclusions concerning the satisfiability of *Pos* cannot be made in general based on a finite prefix.

We mention here also why the method does not work for the modal operator ALL. It is possible to prove one direction of the required proof:

Theorem 6.2. If $\Sigma \models ALL\,\varphi$ in the full computation state space, then $\Sigma \models ALL\,\varphi$ in the reduced computation state space.

Proof. If $\Sigma \models ALL\,\varphi$ in the full computation state space, then $R^{\varphi}(\Sigma) \subseteq Q_F$. By the invariant, state Σ is reached in the reduced computation state space and $R^{\varphi}_{red}(\Sigma) \subseteq R^{\varphi}(\Sigma) \subseteq Q_F$. Thus, $\Sigma \models ALL\,\varphi$ in the reduced computation state space.

□

The converse of the above theorem does not hold. The invariant only guarantees that $R^{\varphi}_{red}(\Sigma)$ will be equal to $R^{\varphi}(\Sigma)$ up to non-*trap* states. If $R^{\varphi}(\Sigma) = R^{\varphi}_{red}(\Sigma) \cup \{trap\}$, it is possible that $R^{\varphi}_{red}(\Sigma) \subseteq Q_F$, but that $R^{\varphi}(\Sigma) \not\subseteq Q_F$. This means that $\Sigma \models ALL\,\varphi$ will hold in the reduced computation state space, when in fact $\Sigma \not\models ALL\,\varphi$ holds in the full computation state space.

In short, the invariant is not strong enough to ensure invariance of the modal operator ALL under the reduction produced by the on-the-fly approach presented in this chapter.

6.6 Complexity

We now consider the complexity of the on-the-fly approach.

The only change to the algorithm is the introduction of the function *guided*() which is used to compute, for each state Σ, the set of computation state space events which are (i) enabled from Σ, and (ii) which do not result in all corresponding automaton states leading to the dead state.

For each enabled event e, the function *guided*() must process at most $|A_\varphi|$ automaton states and, for each automaton state, must process at most $|AP|$ state labels. Thus, the processing of each event has complexity $O(|A_\varphi| * |AP|)$. Given that there are at most N events enabled at each state Σ, the overall complexity of *guided*() is $O(N * |A_\varphi| * |AP|)$.

It is difficult to gauge the reduction achieved by this approach in general, as the degree of reduction depends upon the dynamic property and the distributed computation in question (as in the case of on-the-fly automata theoretic model checking). However, as the method is based on the on-the-fly automata theoretic approach, we expect to obtain reductions in the size of the state space explored which are consistent with those encountered when the approach is used in the context of model checking.

Because the method avoids exploration of continuations of sequential observations which are guaranteed *not* to satisfy a dynamic property, it will perform best when sequential observations may be ruled out based on violation of the property over a finite prefix, which is the defining characteristic of safety properties. When dynamic properties place strict requirements on the relative ordering of global states, the degree of reduction increases, due to the fact that the chance of a sequential observation violating the ordering on a finite prefix increases and, by the method, continuations of this prefix will not be explored.

However, when the dynamic properties does not place strict requirements on temporal ordering, reduction can be non-existent. One such case is global predicate evaluation. Indeed, detection of properties such as $Pos\,\Phi$ and $Def\,\Phi$ exhibit a liveness character: on any finite prefix of a sequential observation which does not satisfy the predicate Φ, it is possible that Φ will be satisfied on some continuation of the prefix. In such cases, such continuations will not be ruled out by the method.

Therefore, this method appears to best suited to temporal specifications representing safety properties, which place strict requirements on the relative ordering of states in a sequential observation.

6.7 Implementation

We have implemented a version of the BFR algorithm which incorporates the on-the-fly modification presented in this chapter. In this section, we present an example of a reduced state space generated by the algorithm.

The distributed computation $\gamma = (H, \rightarrow)$ we consider is made up of two processes, P_1 and P_2. The lattice of global states corresponding to the distributed computation is shown in Figure 6.3. State Σ^{ij} denotes the state of the distributed computation reached after exploring the first i events of process P_1 and the first j events of process P_2.

In the example, the propositions defined on global state are taken from the set $AP = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \epsilon\}$. The proposition ϵ is used to represent the fact that none of the propositions $\varphi_i, i = 1, 2, 3, 4$ hold true in a global state. Global states are labeled with the propositions which hold true in those states. An exception to this rule are states whose labeling is the set $\{\epsilon\}$. In the interests of visual clarity, these state labellings have not been indicated in the figure. Any state with no labeling is

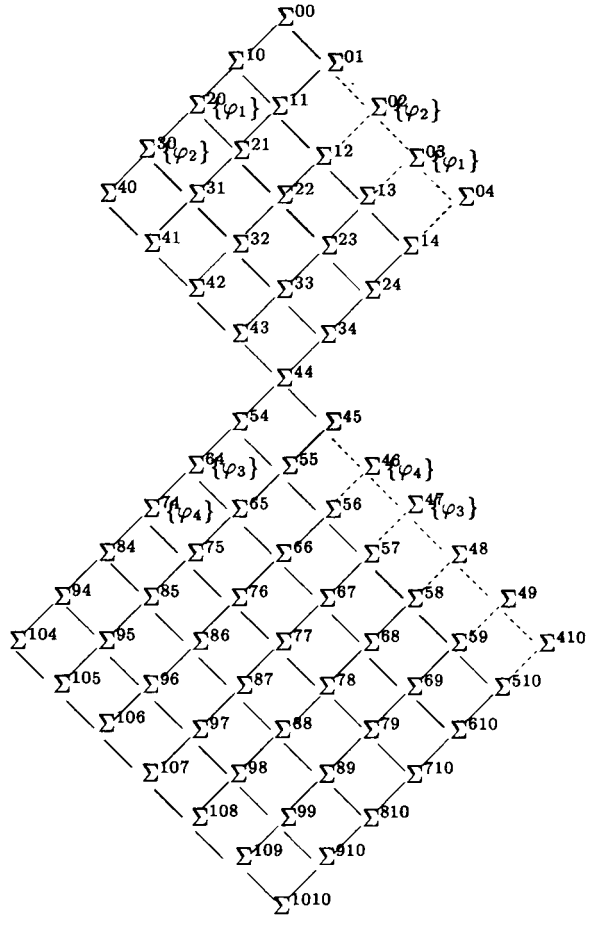


Figure 6.3: A distributed computation and on-the-fly reduction

therefore assumed to be labeled by the set $\{\epsilon\}$.

The property we consider in the example is one which can be described informally as requiring that the propositions φ_1, φ_2, φ_3 and φ_4 appear only once in the path labeling and that they appear in that order. It is described formally by the deterministic finite state automaton $A = (Q, \Sigma, Q_0, \delta, Q_F)$, where $Q = \{q_0, q_1, q_2, q_3, q_4, q_t\}$, $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \epsilon\}$, $Q_0 = \{q_0\}$ and $Q_F = \{q_4\}$. The transition relation δ is defined by the following transition table:

δ	φ_1	φ_2	φ_3	φ_4	ϵ
q_0	q_1	q_t	q_t	q_t	q_0
q_1	q_t	q_2	q_t	q_t	q_1
q_2	q_t	q_t	q_3	q_t	q_2
q_3	q_t	q_t	q_t	q_4	q_3
q_4	q_4	q_4	q_4	q_4	q_4
q_t	q_t	q_t	q_t	q_t	q_t

Note that the property is stable: once the acceptance state q_4 is reached, there are no further constraints on the labels that may appear.

In the Figure 6.3, transitions which are shown in solid lines are those which are explored in the reduced state space, and transitions shown in dotted lines are not explored in the reduced state space.

The distributed computation $\gamma = (H, \rightarrow)$ of this example satisfies $\Sigma^{1010} \models SOME\,\varphi$ and $\gamma \models Pos\,\varphi$, as there exists a sequential observation leading to the final state Σ^{1010} passing through states satisfying φ_1, φ_2,φ_3, φ_4, in that order. It can be seen that this satisfaction relation also holds for the reduced computation state space.

6.8 Conclusions

In this chapter, we explored the development of an algorithm for the detection of temporal properties which incorporates the on-the-fly approach to combating state explosion. The algorithm was shown to have the following advantages:

- easy to implement in the context of an existing detection algorithm based on the automata-theoretic approach
- is compatible with the modal operators $SOME$ and Pos used in dynamic property detection applications
- works at run-time or in a post-mortem context
- works in the case of terminating and non-terminating, infinite state programs (subject to the restriction that Pos cannot be decided in general for non-terminating computations)

Future work in this area could consider the compatibility of other methods for combating state explosion in trace checking with the automata-theoretic approach.

Chapter 7

A Partial Order Reduction Approach To Combating State Explosion

7.1 Introduction

In Chapter 5, we outlined several reasons why the *partial order reduction* approach to combating state explosion in model checking is a promising candidate for combating state explosion in the detection of temporal properties in trace checking. In this chapter, we present the development of a second approach to combating state explosion in dynamic property detection, this time based on the partial order reduction approach to model checking.

In model checking, the partial order reduction approach is based on performing a selective search of the state transition system representing the program state space, resulting in a reduced program state space. The basis of our approach will be to view the lattice of global states (the computation state space) as a state transition system (where events are considered as individual transitions of the system), and apply the partial order theory to that transition system.

Specifically, our aim is begin with an approach to detecting temporal properties in trace checking based on exhaustive analysis, and using the theory of partial order reduction from model checking, develop a corresponding approach to detecting temporal properties in trace checking in which the state explosion problem is mitigated.

In the case of deciding upon a formulation of the property detection problem, we use the formulation of the detection of temporal properties in trace checking of the previous chapter, wherein properties are specified by regular languages of finite words over an alphabet, quantified by modal operators $SOME$ and ALL, and property detection is based on the algorithm of Babaoglu-Fromentin-Raynal. This algorithm can also be used to detect Pos and Def in the case of terminating computations, using the equalities involving final states presented in Section 4.1. We consider this version of the property detection problem as (i) it is a well-established formulation of the problem with great generality (ii) it is based upon an exhaustive analysis of the computation state space and (iii) it will permit comparison with the results developed in the previous chapter.

In the case of making use of the theory of partial order reduction from model checking, it is important to realize that that although the central idea behind partial order reduction is perform-

ing a reduced state space search, there is no single theory of partial order reduction. Rather there are a family of theories, all based around the same central idea, which vary according to:

1. the means by which properties are specified (i.e. sequences of transitions, sequences of labeled states)
2. classes of properties considered (i.e. deadlock, safety, general safety and liveness)
3. the exploration methods the theory is based on (depth-first search, breadth-first search, etc.)

That is, partial order reduction theories are sensitive to these three problem parameters, in the sense that the theories can vary depending on the particular set of problem parameters we consider. We shall require choosing a formulation of the partial order theory which is suitable to the problem context we consider.

In addition, we shall see that there are several approaches possible for applying the technique of partial order reduction to the trace checking problem, namely the "modified system" approach and the "equivalent state space" approach. Both of these approaches will be described in Section 7.3, devoted to consideration of preliminary design issues. Depending on the approach chosen, there are further approach-specific design issues to be resolved, before the development of an approach to adapting partial order reduction in a way which is useful for trace checking can begin. In this sense, we shall see that applying partial order reduction to the problem of combating state explosion in trace checking is *significantly* more involved than applying the on-the-fly approach, as explored in the previous chapter.

As we shall demonstrate, for the resulting partial order approach to combating state explosion in the detection of dynamic properties which we do explore, it will exhibit the following desirable features:

- the approach is suitable for checking temporal properties which are stuttering-invariant and which define regular languages (which includes the class of safety properties)
- the approach is compatible with some (Pos, Def) but not all ($SOME$, ALL) of the standard modal operators used in dynamic property detection applications
- it is suitable for run-time or off-line trace checking
- works in the case of terminating, infinite-state programs

The rest of the chapter is structured as follows. In Section 7.2, we present background work on partial order reduction which will be necessary for the development of the reduced state space algorithm based on the partial order reduction approach. This background involves the development of a theory of partial order reduction which is appropriate for the trace checking context. In Section 7.3, we consider the design issues which underly the development of an algorithm for producing an equivalent state space, suitable for checking stuttering-invariant temporal properties in trace checking. In Section 7.4, we present our algorithm for generating an equivalent state space - a modification of the Cooper-Marzullo algorithm which generates a stuttering-equivalent lattice structure. Section 7.5 considers the correctness issues involved in the theory. Section 7.6 the complexity of the resulting state space exploration algorithm. In Section 7.7, we present our conclusions and consider further work in this area.

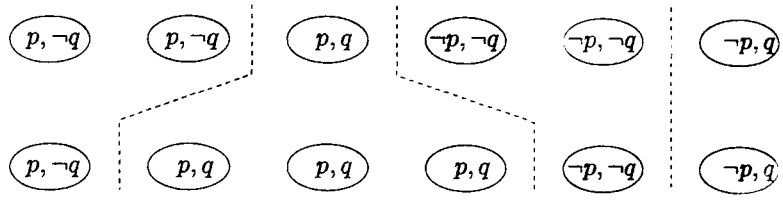


Figure 7.1: Two stuttering equivalent paths.

7.2 Background on Partial Order Reduction

In the presentation of the concepts required for partial order reduction, we must refer in detail to semantic aspects of the program being considered. The general notion of a finite state transition system is used to represent the behaviour of concurrent programs, in order to avoid being tied down to any particular representation of concurrent programs.

A *labeled finite state transition system* is a tuple $< S, S_0, T, L >$ where S is a finite set of states, S_0 is a set of initial states, T is a finite set of transitions such that each transition $\alpha \in T$ is a deterministic partial function $\alpha : S \to S$, AP is a finite set of propositions and L is a labeling function $L : S \to 2^{AP}$. Given a transition $\alpha \in T$, α is *enabled* at state $s \in S$ if $\alpha(s)$ is defined. The set of transitions enabled at a state s is denoted as *enabled*(s). When α is enabled at state s, then the state $\alpha(s)$ is the *successor* of s under α. A *path* through the transition system is defined as a sequence of states interleaved by transitions, i.e. a sequence $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ such that $s_0 \in S_0$ and for each $i \geq 0$, $s_{i+1} = \alpha_i(s_i)$.

Two transitions $\alpha, \beta \in T$, $\alpha \neq \beta$, are *independent* if, for each state $s \in S$ such that $\alpha, \beta \in$ *enabled*(s), the following two properties hold:

1. α and β do not disable each other: $\alpha \in$ *enabled*$(\beta(s))$ and $\beta \in$ *enabled*$(\alpha(s))$
2. α and β are commutative: $\alpha(\beta(s)) = \beta(\alpha(s))$

Independence defines an anti-reflexive, symmetric relation I on the set of transitions T. Two transitions $\alpha, \beta \in T$, $\alpha \neq \beta$, are *dependent* if they are not independent. The dependence relation D is the complement of the independence relation, defined by $D = (T \times T) \setminus I$, and defines a reflexive, symmetric relation on the set of transitions T.

A transition $\alpha \in T$ is *invisible* with respect to a set of propositions $AP' \subseteq AP$ if for each pair of states s, s' such that $s' = \alpha(s)$, we have $L(s) \cap AP' = L(s') \cap AP'$.

Two infinite paths $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ and $\rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$ are *stuttering equivalent*, denoted by $\sigma \sim_{st} \rho$, if there are two infinite sequences of positive integers $0 = i_0 < i_1 < i_2 < \ldots$ and $0 = j_0 < j_1 < j_2 < \ldots$ such that for every $k \geq 0$, $L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_{k+1}-1})$.

A finite sequence of identically labeled states within a sequence is called a block. Two paths are stuttering equivalent when they can be partitioned into finitely many blocks, such that the k-th block of one are labeled in the same way as the k-th block of the other. Corresponding blocks can have differing lengths. Figure 7.1 shows two stuttering equivalent paths.

It is possible to define in a similar manner the idea of stuttering equivalence for finite paths. Two finite paths $\sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots s_{|\sigma|-1} \xrightarrow{\alpha_{|\sigma|}} s_{|\sigma|}$ and $\rho = r_0 \xrightarrow{\beta_1} r_1 \xrightarrow{\beta_2} \ldots r_{|\rho|-1} \xrightarrow{\beta_{|\rho|}} r_{|\rho|}$

are *stuttering equivalent*, denoted by $\sigma \sim_{st} \rho$, if there are two finite sequences of positive integers $0 = i_0 < i_1 < i_2 < ... < i_n = |\sigma|$ and $0 = j_0 < j_1 < j_2 < ... < j_n = |\rho|$ such that for every $0 \leq k < n$, $L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1}) = L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_{k+1}-1})$.

The notion of stuttering equivalence for infinite paths is extended to labeled transition systems. Two labeled transition systems M and M' are *stuttering equivalent* if and only if

- M and M' have the same set of initial states
- for each path σ of M that starts from an initial state s of M, there exists a path σ' of M' that starts from the same initial state s such that σ is stuttering equivalent to σ'
- for each path σ' of M' that starts from an initial state s of M', there exists a path σ of M that starts from the same initial state s such that σ' is stuttering equivalent to σ

An LTL formula f is *invariant under stuttering* if and only if for each pair of stuttering equivalent paths π and π', we have $\pi \models f$ if and only if $\pi' \models f$. It can be shown that an LTL formula which is stuttering equivalent does not distinguish between labeled transition systems which are stuttering equivalent [19].

The development of the partial order theory to be considered in the sequel will involve discussion of certain results related to graph traversal algorithms, which form the basis of state space exploration. We introduce the key concepts and notation here. Let $G = (V, E)$ be a directed graph consisting of a set of verticies V and a set of edges $E \subseteq V \times V$. Given $v, w \in V$, a *path* from v to w is a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ where $v = v_1$ and $w = v_n$. Paths will be represented by listing the sequence of verticies $v_1, v_2, \ldots, v_n$ on the path. The *length* of a path is the number of edges contained in the path. A path is *simple* if all the edges and verticies on the path are distinct, except possibly the first and last verticies. A *cycle* is a simple path of length at least one which begins and ends at the same vertex.

A graph $G = (V, E)$ is *strongly connected* if, for any pair of verticies $v, w \in V$, there is a path from v to w and a path from w to v. A directed graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. G' is a *strongly connected subgraph* of G if G' is a subgraph of G and G' is strongly connected. Following [2], given a graph $G = (V, E)$, the verticies of G can be partitioned into equivalence classes $V_i, 1 \leq i \leq r$, such that v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let $E_i, 1 \leq i \leq r$, be the set of edges connecting the pairs of verticies in V_i. The subgraphs $G_i = (V_i, E_i)$ are called the *strongly connected components* of G. These represent the maximal strongly connected subgraphs of G. Every strongly connected subgraph of G is a subgraph of some strongly connected component of G.

The strongly connected components of a directed graph form a directed acyclic graph(DAG). This DAG can be represented as a tree, where the nodes of the tree are strongly connected components, and an edge appears between two nodes if a vertex in one component is reachable from a vertex in another component. A strongly connected component with no incoming edges is called a *source* strongly connected component. A strongly connected component with no outgoing edges is called a *sink* strongly connected component.

7.2.1 Reduced state space search and ample sets

Partial order reduction methods are based upon using a modified search of the program state space, known as a *selective search*, to generate a reduced program state space which has potentially fewer states and transitions than the full program state space, and yet which contains enough states and transitions in order to correctly verify the property in question. Partial order reduction methods were reviewed in Chapter 3. There, the *persistent set* approach of Godefroid was discussed as an example of the partial order reduction approach. It was noted that the persistent set reduction was suitable for checking properties which could be characterized in terms of deadlock or reachability of local states. For the development to be presented in this chapter, we shall use an alternative version of the partial order theory.

The version of the partial order theory which we present here and use in the sequel is presented in [19, Chapter 10]. We shall at times refer to this as the Peled theory, as the presentation is due to Doron Peled. As in other versions of the partial order theory, the method is based upon using a selective search to generate a reduced state space. In this version of the theory, however, specifications are assumed to be in the form of stuttering-invariant LTL formulae. The reduction is based on the fact that stuttering-equivalent LTL formulae do not distinguish between labeled transition systems which are stuttering-equivalent. Given a stuttering-equivalent formula φ, defined on a set of propositions $AP' \subseteq AP$, the aim is to generate a reduced state space which is stuttering-equivalent (with respect to the propositions in AP') and use that reduced state space, instead of the full state space, in the model checking exercise.

The reduced state space exploration is based on exploring only a subset $ample(s)$ of the transitions in $enabled(s)$ at each state s reached during the search of the state space. The subsets $ample(s)$ are constructed in such a way that the reduced state space search will provably produce a state graph M' which is stuttering-equivalent (with respect to the propositions in AP') to the full state space graph M. This allows using M' in place of M when verifying the formula φ.

A key ingredient of the approach concerns how to calculate ample sets. In most treatments of partial order reduction, the calculation of the set of directions to be followed in the selective search (here called ample sets) is dependent on the particular state space exploration algorithm (e.g. depth-first search, breadth-first search) used to explore the state space. However, in the treatment of the theory presented by Peled, ample sets are characterized in an exploration-independent manner. Peled presents a set of four conditions characterizing ample sets. These conditions are expressed solely in terms of the formula to be verified, the full state graph and the reduced state graph, and are independent of the exploration method used to generate the state graphs. The conditions are defined as follows, where s represents a state reached during the selective search, and we assume that the formula φ to be verified is defined on the set of propositions $AP' \subseteq AP$:

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$

C1 Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first

C2 if s is not fully expanded (i.e. $ample(s) \subset enabled(s)$), then every $\alpha \in ample(s)$ is invisible (with respect to the propositions in AP')

C3 a cycle (in the reduced state space) is not allowed if it contains a state in which some transition α is enabled, but never included in $ample(s)$ for any state s on the cycle

Using the conditions above [19, Chapter 10], it is possible to prove that the reduced state space search produces a state transition system which is stuttering-equivalent to the state transition system produced by the conventional state space search search, or full state transition graph.

7.2.2 Methods for constructing ample sets

Although a characterization of ample sets has been presented, we still need a practical means of calculating them during the state space exploration. For each state s encountered during the search, it is theoretically possible to construct ample sets $ample(s)$ by selecting arbitrary subsets of $enabled(s)$ and checking to see that each of the conditions **C0**, **C1** ,**C2** and **C3** hold. However, checking certain of these conditions has been shown to be computationally complex [19]. For example, it has been shown that checking condition **C1** is as complex as checking general reachability, and checking condition **C3** is equivalent to checking that a state is reachable from itself. Therefore, heuristics are used, which represent sufficiency conditions which are easier to check in practice.

The computation of sets of transitions satisfying conditions **C0** and **C1** has been studied extensively [112, 47, 86] in the context of early research on partial order reduction, appearing variously as stubborn sets, persistent sets and ample sets. The methods are based on using information concerning the dependency relation between transitions in the program, gained through various combinations of static and dynamic analysis, and taking into account the semantics of access to data by program transitions (i.e. shared versus non-shared objects, read- versus write-access). The chief difficulty in these methods is dealing with potential dependencies in future states (implicit in the description of condition **C1**) between transitions which touch shared objects, such as global variables and communication channels. Furthermore, ample sets must be computed at every global state reached during the search, creating considerable run-time overhead in the state space exploration.

Unlike conditions **C0**, **C1** and **C2**, which are independent of the method used to explore the state space, condition **C3**, which involves the detection of cycles, is exploration-dependent. For example, in a depth-first search, the detection of transitions which create cycles can be easily determined, as a prefix of the computation is maintained in the search stack. Thus, condition **C3** may be replaced by the sufficient condition [19, Chapter 10]:

C3-dfs If s is not fully expanded, then no transition in $ample(s)$ must reach a state that is on the search stack of the depth first search

Checking for the existence of a cycle in a breadth-first search is more difficult, as breadth-first search explores many interleavings simultaneously, and there is no notion of a current interleaving being explored. As noted in [19, Chapter 10], a necessary condition for the creation of cycles in a breadth-first search is that we revisit a state. This leads to the sufficient condition:

C3-bfs If s is not fully expanded, then no transition applied to state s in the current level results in a state in the current level or a previous level

This sufficiency condition for **C3** results in many more nodes being fully expanded than necessary, as repeated states are very common in concurrent programs.

7.2.3 Static analysis

A simplified approach to the identification of ample sets of transitions was pursued in [56]. The motivation behind the approach was to reduce as far as possible the run-time overhead incurred in calculating ample sets by performing the majority of the calculation in a static analysis phase. The approach depends on identifying statically which transitions in the program text are *globally independent* and *visible*, and then using that information at run-time to identify a set of *safe* transitions.

Two transitions $t, t' \in T$, $t \neq t'$, are said to be *globally independent* if and only if they are independent in every possible state where they are simultaneously enabled:

$$\forall s \in S \; t, t' \in \mathit{enabled}(s) \Rightarrow (t, t') \in I$$

where $I \subset T \times T$ is the independence relation on T. For example, two transitions on different processes which access only variables (not communication channels) local to their processes will be globally independent. What makes global independence so significant for computing ample sets is that if the set of transitions enabled in state s on process P_i, denoted by $T_i(s)$, are globally independent with all transitions on other processes P_j, $i \neq j$, then they satisfy conditions **C0** and **C1**.

Given a stuttering-invariant linear temporal logic formula φ, let $\mathit{Props}(\varphi) \subseteq AP$ denote the set of propositions on which the formula is defined. A transition t on P_i is said to be *safe* with respect to φ if:

- for all transitions t' on processes P_j such that $j \neq i$, transitions t, t' are globally independent, and
- t is invisible with respect to the propositions in $\mathit{Props}(\varphi)$

In the exposition presented in [56], Holzmann considered concurrent program models based on message passing communication, although the method applies equally well to communication based on shared memory. For example, any transition which is local to a process and is not a communication action (send or recv) is globally independent with all transitions on other processes P_j, $i \neq j$. However, communication actions may not be globally independent with all transitions on other processes P_j, $i \neq j$. For example, a recv operation which is enabled in a state may be disabled by the execution of a recv operation on another process, if the two operations share a communication channel.

It is shown in [56] that both global independence and visibility of transitions can be approximated statically and encoded in a table.

How can this static analysis phase lead to the efficient calculation of ample sets? During state space exploration (in the case of exploration based on depth-first search), ample sets can be identified by locating a set of transitions $T_i(s)$ enabled in state s on some process P_i which satisfy the following conditions:

- the transitions in $T_i(s)$ are safe, where safety is determined by a simple table lookup
- for each transition in $T_i(s)$, no transition leads to a state on the search stack

If no such set of transitions $T_i(s)$ exists, then the set of transitions *enabled*(s) is used as the ample set.

A cost penalty arises with this method if no set $T_i(s)$ containing only safe transitions can be found: then, the node must be fully expanded, even though a smaller ample set could possibly have been found using more elaborate algorithms for computing ample sets. This approach trades off run-time efficiency against accuracy.

7.3 Preliminary Design Issues

In this section, we consider some preliminary design issues which need to be resolved in applying partial order reduction to the problem of trace checking.

7.3.1 Applying Partial Order Reduction

The first such design issue concerns the way in which partial order reduction is applied to the detection problem. There are two quite different approaches to applying partial order reduction in the verification of properties of concurrent systems. The approaches differ by the way in which different classes of properties are verified:

- the "modified system" approach - in Godefroid's PhD thesis [46], a notion of equivalence (trace equivalence) between the full and reduced state space was shown to preserve transitions fired between the full and reduced state space, and so also the local states reached between the full and reduced state space. Thus, properties described in terms of transitions fired or the reachability of local states are preserved by trace equivalence, and so may be checked directly on the reduced state space. Godefroid showed how more general properties which were not preserved under trace equivalence (such as invariance or general safety properties) could be checked by creating a modified system in which the properties of interest were encoded as determining the reachability of a local state. In this approach, each class of property so considered potentially has a different modified system representation. Verification of the property is coupled with the exploration of the modified system using the partial order reduction.
- the "equivalent state space" approach - in this approach, an example of which is presented in [19], properties considered (e.g. the stuttering-invariant properties) are assumed to be invariant with respect to a certain equivalence (e.g. stuttering-equivalence) between state spaces, and the approach focuses on generating a reduced state space which preserves this equivalence. Unlike the "modified system" approach, the "equivalent state space" approach depends on properties being invariant under the equivalence in question. Peled considered this approach as the class of stuttering-invariant properties includes many important classes of properties. Verification proceeds by using standard verification algorithms on the reduced state space. In this approach, verification of the property is decoupled from the exploration of the system using partial order reduction.

Both of these approaches would be consistent with viewing the lattice of global states (the computation state space) as a state transition system (where events are considered as individual transitions of the system), and applying the partial order theory to that transition system. However, both are significantly different from one another, and we do not propose to consider both approaches here. How do we choose between approaches?

In a "modified system" approach, the property detection problem would be encoded as an equivalent problem of detecting deadlock or reachability of a local state in a modified system. This approach was used by Stoller et al. [104] to apply partial order reduction to the problem of global predicate detection, where $Pos\,\Phi$ and $Def\,\Phi$ were encoded as the problem of detecting deadlock in a modified system. In the case of checking temporal properties, such an approach may either be based on encoding $SOME\,\varphi$, $ALL\,\varphi$, $Pos\,\varphi$ and $Def\,\varphi$ as the problem of detecting deadlock, or reachability of a local state in a modified system. A potential advantage of this approach is that if a modified system can be found for a property (and this is not guaranteed!), it may be possible to avoid the general restriction of stuttering-invariance. The disadvantage of this approach is that, for each modal operator considered, a different modified system, a different definition of persistent sets and a different correctness proof will probably be required. This state of affairs was evident in the work of Stoller.

In an "equivalent state space" approach, a reduced state space would be generated which is equivalent in some sense to the original computation state space, and the equivalence chosen in such a way that the properties we want to check are invariant under the equivalence. The advantage of such an approach would be (i) a single algorithm can be used to generate a reduced state space for properties which are invariant under the equivalence. This considerably reduces the work involved in developing proofs and correct formulations of persistent sets (ii) detection algorithms for such properties which operate on the full state space (as generated in an exhaustive analysis) may be reused on the reduced state space. The disadvantage of such an approach would be the approach cannot be applied to properties which are not invariant under the equivalence.

Both of the approaches described above have advantages, and both approaches deserve consideration. However, for the purposes of this work, which is to highlight the issues involved in applying model checking techniques to the trace checking context, we shall choose only one of these approaches for further development, and our choice shall be fairly ad hoc. We choose the "equivalent state space" approach, simply due to the potential gains which may be realized in terms of algorithmic generality.

7.3.2 The Impact of Modal Operators

The "equivalent state space" approach to applying partial order reduction involves two important elements:

- defining an equivalence between program state spaces, under which properties of interest are invariant
- developing an algorithm for generating a reduced program state space which is equivalent (in the sense above) to the full state space

For example, in model checking, for the purposes of checking stuttering-invariant LTL properties, Peled showed that such properties were invariant under stuttering-equivalence between state spaces, and that an algorithm based on partial order reduction could be developed to generate a stuttering-equivalent reduced state space. In the definition of satisfaction for LTL properties, although no modal operator is specified, there is an implicit modal operator (equivalent to Def) which requires that all possible execution sequences must satisfy the given LTL property. Such modalities, as we will see, do impact greatly on the equivalence required. In the case of dynamic property detection, this task is complicated by the fact that we have two sets of semantically different modal operators to consider.

Given the task of applying this approach to the trace checking context, one way to proceed is to:

1. examine the basic requirements of the modal operators Pos, Def, $SOME$, and ALL for invariance
2. use these basic requirements to determine a suitable equivalence between state spaces under which properties would be invariant
3. identify a candidate reduction algorithm, based on partial order reduction, upon which an algorithm to generate a reduced state space (satisfying the desired equivalence)j could be based

In the rest of this section, we shall aim to identify candidate equivalence notions, and candidate algorithms from partial order reduction which could be used as a basis for generating such equivalences in a trace checking context.

7.3.2.1 Characteristics of Modals and Candidate Equivalences

In this section, we shall examine the basic characteristics of the modal operators used in trace checking, Pos, Def, $SOME$, and ALL, in order to determine the basic requirements of equivalences which will preserve these modal operators. Based on these requirements, we shall also propose candidate equivalences for these modal operators, and prove that these equivalences are sufficient to preserve invariance of the modal operators.

Stuttering-invariance assumption

We require that dynamic properties φ are stuttering-invariant when using the "equivalent state space" approach. Informally, this is required because the partial order reduction methods we consider are dependent upon freely permuting adjacent, independent transitions in order to avoid having to explore equivalent interleavings. Although such permutations can result in the corresponding labeled sequences being unequal, these labeled sequences are always guaranteed to be stuttering-equivalent [19, Chapter 10].

Modal Operators Pos and Def

These modal operators are *path-based*, in the sense that they are defined over the set of sequential observations of the distributed computation. Combined with the fact that dynamic properties

are now assumed to be stuttering-invariant, the basic requirements for an equivalence between the full and reduced computation state spaces which preserves Pos and Def will be an equivalence which:

1. is path-based, in the sense that the equivalence is defined in terms of the sequential observations of the distributed computation
2. preserves stuttering-equivalence of sequential observations in the distributed computation
3. preserves in some manner the set of sequential observations of the distributed computation

It is easy to see that the notion of stuttering-equivalence of computation state spaces is an equivalence between computation state spaces which satisfies these basic requirements, and so is a candidate equivalence for preserving stuttering-invariant properties quantified with the modal operators Pos and Def. In particular, we aim to show that given dynamic property φ defined over a set of propositions AP, a distributed computation $(H, \rightarrow)$ and its associated lattice of global states $\mathcal{L} = (\Sigma, \prec)$, that if $\mathcal{L}' = (\Sigma', \prec')$ is another lattice which is stuttering-equivalent to $\mathcal{L}$ with respect to the propositions in AP, then the modal operators Pos and Def are invariant under this equivalence between lattices. Before proving this assertion, we need to make some preliminary remarks.

Firstly, in Section 4.1, we defined satisfaction of the modal operators Pos and Def in terms of the distributed computation $\gamma = (H, \rightarrow)$ and its sequential observations. Because the lattice of global states $\mathcal{L} = (\Sigma_\gamma, \prec_\gamma)$ contains all and only the sequential observations of the distributed computation, we extend the definition of Pos and Def to lattices in the obvious way (i.e. a lattice $\mathcal{L}$ satisfies $Pos\,\varphi$ if and only if some sequential observation Ω of the lattice satisfies $\Omega \models \varphi$, and similarly for the modal operator Def).

Secondly, the notion of stuttering-equivalence has been defined in the context of labeled state transition systems. In order to prove the above assertion, we need to make use of the notion of stuttering-equivalence for sequential observations and lattices of global states. Given a lattice of global states $\mathcal{L} = (\Sigma_\gamma, \prec_\gamma)$ and a set of atomic propositions AP, we may view this lattice a labeled state transition system (S, S_0, T, L), where $S = \Sigma_\gamma$, $S_0 = \Sigma_\gamma^0$ where Σ_γ^0 is the unique minimal element of the lattice, and $L = \lambda$ is a labeling function which labels states in Σ_γ with propositions from AP. The transitions T of this state transition system are represented by elements of the immediate predecessor relation $\prec_\gamma^{im}$ on the global states: each pair of states Σ, Σ' for which $\Sigma \prec_\gamma^{im} \Sigma'$ is viewed as a transition in T. In this way, viewing the lattice of global states as a labeled state transition system, the definitions of stuttering-equivalence of paths and stuttering-equivalence of state transition systems carry over to stuttering-equivalence of sequential observations and stuttering-equivalence of lattices.

We now prove the desired assertion:

Theorem 7.1. Suppose that φ is a stuttering-invariant temporal property, defined over a set of propositions AP. Let $(H, \rightarrow)$ be a distributed computation with lattice of global states $\mathcal{L} = (\Sigma, \prec)$, whose states are suitably labeled with propositions from AP. Let $\mathcal{L}' = (\Sigma', \prec')$ be a lattice of global states, whose states are also labeled with propositions from AP, and which is stuttering-equivalent to $\mathcal{L} = (\Sigma, \prec)$ with respect to the propositions in AP. Then $\mathcal{L} \models Pos\,\varphi$ if and only if $\mathcal{L}' \models Pos\,\varphi$.

Proof. Suppose that $\mathcal{L} \models Pos\,\varphi$. We need to show that $\mathcal{L}' \models Pos\,\varphi$. Because $\mathcal{L} \models Pos\,\varphi$, there is a sequential observation Ω of $\mathcal{L}$ which satisfies $\Omega \models \varphi$. By assumption, $\mathcal{L}$ and $\mathcal{L}'$ are stuttering-equivalent with respect to the propositions in AP, so there exists a sequential observation Ω' of $\mathcal{L}'$ which is stuttering-equivalent to Ω with respect to the propositions in AP. Because φ is stuttering-invariant, and because $\Omega \models \varphi$, the sequential observation Ω' of $\mathcal{L}'$ satisfies $\Omega' \models \varphi$. Thus $\mathcal{L}' \models Pos\,\varphi$. (The converse holds by symmetry).

□

Theorem 7.2. Suppose that φ is a stuttering-invariant temporal property, defined over a set of propositions AP. Let $(H, \rightarrow)$ be a distributed computation with lattice of global states $\mathcal{L} = (\Sigma, \prec)$, whose states are suitably labeled with propositions from AP. Let $\mathcal{L}' = (\Sigma', \prec')$ be a lattice of global states, whose states are also labeled with propositions from AP, which is stuttering-equivalent to $\mathcal{L} = (\Sigma, \prec)$ with respect to the propositions in AP. Then $\mathcal{L} \models Def\,\varphi$ if and only if $\mathcal{L}' \models Def\,\varphi$.

Proof. Suppose that $\mathcal{L} \models Def\,\varphi$. Because $\mathcal{L} \models Def\,\varphi$, for each sequential observation Ω of $\mathcal{L}$, we have $\Omega \models \varphi$. We need to show that $\mathcal{L}' \models Def\,\varphi$. Suppose not; that is, suppose that there is a sequential observation Ω' of $\mathcal{L}'$ such that $\Omega' \not\models \varphi$. Because $\mathcal{L}$ and $\mathcal{L}'$ are stuttering-equivalent with respect to the propositions in AP, there is a sequential observation Ω of $\mathcal{L}$ which is stuttering-equivalent to Ω' with respect to the propositions in AP. Because φ is stuttering-invariant and $\Omega' \not\models \varphi$, we have $\Omega \not\models \varphi$. This contradicts the fact that $\mathcal{L} \models Def\,\varphi$. Therefore, $\mathcal{L}' \models Def\,\varphi$. (The converse holds by symmetry)

□

Modal Operators $SOME$ and ALL

An important distinction between the modal operators Pos and Def, on the one hand, and the modal operators $SOME$ and ALL, on the other, is that, unlike Pos and Def, which quantify over the set of sequential observations of a distributed computation, $SOME$ and ALL quantify over the set of paths leading to a pre-determined reachable global state Σ of the distributed computation.

Unlike Pos and Def which are *path-based*, these modal operators are *prefix-based*, in the sense that for a fixed, reachable global state Σ, they are defined over the set of prefixes of sequential observations of the distributed computation which lead from the unique initial state to Σ. In the definition of the modal operators $SOME$ and ALL presented in Section 6.2, we recall that given a dynamic property φ defined over propositions in AP and a distributed computation $\gamma = (H, \rightarrow)$, the lattice of global states $(\Sigma_\gamma, \prec_\gamma)$ of γ was viewed as a labeled directed acyclic graph $DAG_\gamma = (\Sigma_\gamma, \prec_\gamma^{im})$, with the labeling function $\lambda : \Sigma_\gamma \rightarrow 2^{AP}$. For each state Σ in DAG_γ, the state Σ and its predecessors form a labeled directed acyclic graph DAG_Σ, and the set of directed paths of DAG_Σ leading from the unique initial state to Σ were denoted by Π_Σ. Satisfaction of φ under modal operators $SOME$ and ALL was defined with respect to labellings of paths in Π_Σ.

Given the fact that the modal operators define satisfaction in terms of prefixes of sequential observations leading to a fixed state Σ, any meaningful correspondence between full and reduced computation state spaces must be on a state by state basis; that is, in order to be able to detect the property, for example, $\Sigma \models SOME\,\varphi$, we need to have Σ itself somehow represented in the reduced state space. In the simplest case, we would require that all states reachable in the full state space are also reachable in the reduced state space. Failing this, we would need to have a means

of establishing a correspondence between the state Σ in the full state space, and its counterpart state in the reduced state space; furthermore, this counterpart state should be unique. In the equivalence we consider here, we assume that all states of the full state space are represented in the reduced state space.

Based on the above discussion, the basic requirements for an equivalence which preserves $SOME$ and ALL will be an equivalence which:

1. is prefix-based, in that the equivalence is defined in terms of the structure of the state space represented by the sets Π_Σ
2. preserves stuttering-invariance of paths in the distributed computation
3. preserves in some manner the set of paths in each set Π_Σ

We now propose to define a notion of equivalence between computation state spaces (viewed as labeled DAGs) which we call *prefix-based stuttering-equivalence*. Informally, it is an equivalence between state spaces in which there is a correspondence between the sets Π_Σ in the full and reduced state space, under which stuttering-equivalence is preserved.

We wish to define this equivalence on labeled DAGs, and then use the equivalence to show that prefix-based stuttering equivalent lattices (when viewed as labeled DAGs) are invariant under the modal operators $SOME$ and ALL. Rather than define the equivalence for a general directed acyclic graph and labeling function, we use the notation presented in Chapter 6 and restrict the development to those DAGs which represent lattices of global states, as described above. Note that, unlike general DAGs, the DAGs we consider here have a unique minimal element (element with no predecessors).

Let DAG_1 and DAG_2 be two such directed acyclic graphs, with $DAG_i = (\Sigma_i, \prec_i^{im})$ with labeling function $\lambda_i : \Sigma_i \rightarrow 2^{AP}$, for $i = 1, 2$.

We first defined stuttering-equivalence between the sets of paths Π_Σ in a labeled DAG. For each $\Sigma \in \Sigma_i$, the sets Π_Σ are well-defined. Two sets of paths Π_Σ and $\Pi'_{\Sigma'}$ are *stuttering-equivalent* if and only if (i) Π_Σ and $\Pi'_{\Sigma'}$ have the same final state (i.e. $\Sigma = \Sigma'$) and (ii) for every path π_Σ in Π_Σ, there exists a stuttering-equivalent path $\pi'_{\Sigma'}$ in $\Pi'_{\Sigma'}$, and (iii) for every path $\pi'_{\Sigma'}$ in $\Pi'_{\Sigma'}$, there exists a stuttering-equivalent path π_Σ in Π_Σ. In this case, the notion of stuttering-equivalence of *finite* paths is used.

Two labeled directed acyclic graphs DAG_1 and DAG_2 are *prefix-based stuttering-equivalent* if and only if

- DAG_1 and DAG_2 have the same set of states (i.e. $\Sigma_1 = \Sigma_2$) and the same unique minimal element
- for each set of paths Π_Σ leading to state Σ in DAG_1, there exists a set of paths Π'_Σ of DAG_2 leading to Σ that such that Π_Σ is stuttering-equivalent to Π'_Σ
- for each set of paths Π'_Σ leading to state Σ in DAG_2, there exists a set of paths Π_Σ of DAG_1 leading to Σ that such that Π'_Σ is stuttering-equivalent to Π_Σ

We now show that prefix-based stuttering-equivalence is an appropriate equivalence for preserving the modal operators $SOME$ and ALL.

In what follows, we shall have to differentiate between the satisfaction of the modal operators $SOME$ or ALL with respect to the states Σ of a given lattice $\mathcal{L}$. Therefore, we use the notation $\mathcal{L}, \Sigma \models \varphi$ to denote the fact that $\Sigma \models \varphi$, for state Σ in lattice $\mathcal{L}$.

Theorem 7.3. Suppose that φ is a stuttering-invariant temporal property, defined over a set of propositions AP. Let $(H, \rightarrow)$ be a distributed computation with lattice of global states $\mathcal{L} = (\Sigma, \prec)$, whose states are suitably labeled with propositions from AP. Let $\mathcal{L}' = (\Sigma', \prec')$ be a lattice of global states, whose states are also labeled with propositions from AP, and which is prefix-based stuttering-equivalent to $\mathcal{L} = (\Sigma, \prec)$ with respect to the propositions in AP. Let Σ be a reachable state of $\mathcal{L}$. Then $\mathcal{L}, \Sigma \models SOME\,\varphi$ if and only if $\mathcal{L}', \Sigma \models SOME\,\varphi$.

Proof. Suppose that $\mathcal{L}, \Sigma \models SOME\,\varphi$. We want to show that $\mathcal{L}', \Sigma \models SOME\,\varphi$. Because $\mathcal{L}$ and $\mathcal{L}'$ are prefix-based stuttering-equivalent, the state Σ is a reachable state of $\mathcal{L}'$ and so the formula $\mathcal{L}', \Sigma \models SOME\,\varphi$ is well-defined. Because $\mathcal{L}, \Sigma \models SOME\,\varphi$, there is a path π through the lattice $\mathcal{L}$ with final state Σ such that $\pi \models \varphi$. Because $\mathcal{L}$ and $\mathcal{L}'$ are prefix-based stuttering-equivalent, there is a path π' through the lattice $\mathcal{L}'$ with final state Σ such that $\pi' \models \varphi$. Therefore, $\mathcal{L}', \Sigma \models SOME\,\varphi$. (the converse holds by symmetry)

□

Theorem 7.4. Suppose that φ is a stuttering-invariant temporal property, defined over a set of propositions AP. Let $(H, \rightarrow)$ be a distributed computation with lattice of global states $\mathcal{L} = (\Sigma, \prec)$, whose states are suitably labeled with propositions from AP. Let $\mathcal{L}' = (\Sigma', \prec')$ be a lattice of global states, whose states are also labeled with propositions from AP, and which is prefix-based stuttering-equivalent to $\mathcal{L} = (\Sigma, \prec)$ with respect to the propositions in AP. Let Σ be a reachable state of $\mathcal{L}$. Then $\mathcal{L}, \Sigma \models ALL\,\varphi$ if and only if $\mathcal{L}', \Sigma \models ALL\,\varphi$.

Proof. Suppose that $\mathcal{L}, \Sigma \models ALL\,\varphi$. Because $\mathcal{L}$ and $\mathcal{L}'$ are prefix-based stuttering-equivalent, the state Σ is a reachable state of $\mathcal{L}'$ and so the formula $\mathcal{L}', \Sigma \models ALL\,\varphi$ is well-defined. We want to show that $\mathcal{L}', \Sigma \models ALL\,\varphi$. Suppose not; that is, suppose that $\mathcal{L}', \Sigma \not\models ALL\,\varphi$. Then for each path π' through the lattice $\mathcal{L}'$ with final state Σ, we have that $\pi' \not\models \varphi$. Because $\mathcal{L}$ and $\mathcal{L}'$ are prefix-based stuttering-equivalent, for each path π' through the lattice $\mathcal{L}'$ with final state Σ, there is a path π through the lattice $\mathcal{L}$ with final state Σ which is stuttering-equivalent, and so $\pi \not\models \varphi$. This contradicts the fact that $\mathcal{L}, \Sigma \models ALL\,\varphi$. Therefore, $\mathcal{L}', \Sigma \models ALL\,\varphi$. (the converse holds by symmetry) □

7.3.2.2 Identifying Candidate Algorithms

In this section, we aim to identify candidate algorithms, based on partial order reduction, which can be used to generate reduced state spaces for the equivalences defined in the previous section. Thus far, we have determined that:

- for stuttering-invariant properties quantified by the modal operators Pos and Def, stuttering-equivalence is an equivalence between labeled lattices of global states which preserves these properties
- for stuttering-invariant properties quantified by the modal operators $SOME$ and ALL, we need a stronger form of equivalence, and have considered prefix-based stuttering-equivalence between labeled lattices of global states as a suitable equivalence

With this in mind, we now begin the search for candidate algorithms, based on partial order reduction, for generating these equivalences.

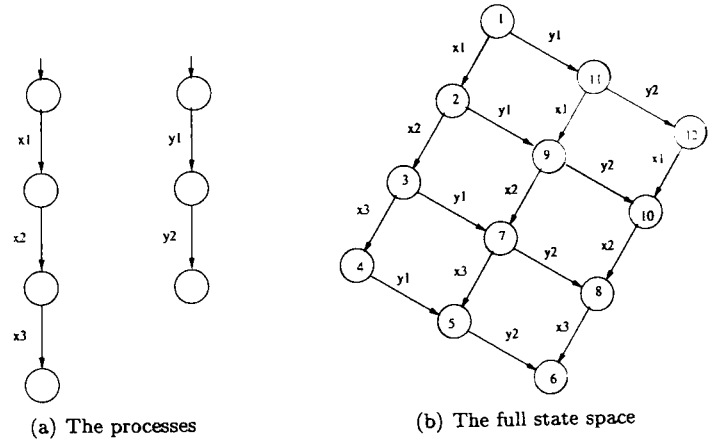
Ample sets based algorithm

Clearly, a candidate algorithm for generating a reduced state space which is stuttering-equivalent is the algorithm based on ample sets presented in [19, Chapter 10]. This model checking algorithm provides a good starting point for developing an algorithm to generate a reduced state space suitable for checking *Pos* and *Def* in a trace checking context. It is based on performing a selective search, in which only a subset (an ample set) of enabled transitions is explored from each state reached. Ample sets were defined in an exploration-independent way in [19, Chapter 10] in terms of four conditions relating the full and reduced state spaces: **C0**/**C1**/**C2** and **C3**.

Unfortunately, although the algorithm based on ample sets preserves stuttering-equivalence between full and reduced state spaces, it does not preserve the stronger prefix-based stuttering equivalence between the full and reduced state spaces. In particular, concerning the correspondence between the full and reduced state spaces:

- the reduced state space generated by the algorithm need not reach all states reachable in the full state space
- a representative for a path, although stuttering-equivalent to the original path, may include a finite number of additional (independent) transitions, resulting in the final state reached by the representative of a path differing from the final state of the original path

We illustrate these points with an example. Figure 7.2 shows a concurrent system of two processes and the full program state space. Figure 7.2(a) shows two processes, P_1 and P_2, and the resulting full program state space. Process P_1 executes transitions $x1$, $x2$ and $x3$ in sequential order. Process P_2 executes transitions $y1$ and $y2$ also in sequential order. The transitions in P_1 are independent of those in P_2, and vice versa. All transitions are treated as invisible. The resulting program state space is shown in Figure 7.2(b). States are numbered by the order in which they would be explored in a depth first exploration of the program state space.



(a) The processes (b) The full state space

Figure 7.2: Two concurrent processes and the full state space

Figure 7.3 shows one reduced state space which could be generated by an ample set exploration of the full program state space of Figure 7.2(b). States and transitions which are not explored are shown in dotted lines. This reduced state space does not explore all states (states 7, 8, 9, 10, 11 and 12 are not explored). Furthermore, the path $\pi = y1$ of the full state space is represented in the reduced state space by the stuttering-equivalent path $\pi' = x1\,x2\,x3\,y1$, where the representative path includes the additional transitions $x1$, $x2$ and $x3$.

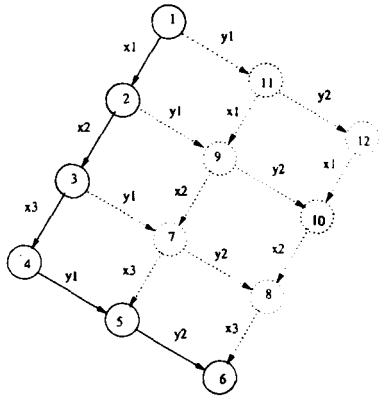


Figure 7.3: Ample set exploration

This latter point arises as the condition **C1** of ample sets permits the introduction of independent transitions when constructing representatives for paths in the full state space, and the underlying trace equivalence only guarantees that every transition in a path will be fired in its representative. These features of the reduced state space mean that the ample sets based algorithm (in its present form) is not a suitable candidate on which to base an algorithm to generate a reduced state space suitable for checking properties involving $SOME$ and ALL.

Sleep sets based algorithm

There is another algorithm for generating a reduced state space which is a possible candidate for generating a prefix-based stuttering-equivalent state space. In [45, 47], Godefroid introduced a method of partial order reduction based on *sleep sets*. Sleeps sets are based on the idea of avoiding the exploration of trace-equivalent interleavings by not exploring transitions which may be enabled in a state but whose exploration would result in exploring a path equivalent to (in the sense of trace equivalence) a path already explored in the state space exploration. This is achieved by entering selected transitions into *sleep* sets, and exploring only transitions in $enabled(s) \setminus sleep(s)$. As Godefroid has shown, state space reduction based on sleep sets results in a reduced state space with the same set of reachable states as the full state space, but with (potentially) fewer paths to those states [46]. Sleep sets avoid the wasteful exploration of multiple interleavings of independent transitions. The sleep sets algorithm presented in Godefroid could possibly be modified to take into account visible transitions to obtain an algorithm for generating a prefix-based stuttering-equivalent state space.

Figure 7.4 shows the reduced state space resulting from a sleep sets-based exploration of the full program state space shown in Figure 7.2(b). As before, dotted lines indicate states and transitions

which are not explored in the reduced state space. Note that although all states in the full state space are explored, not all transitions in the full state space are explored.

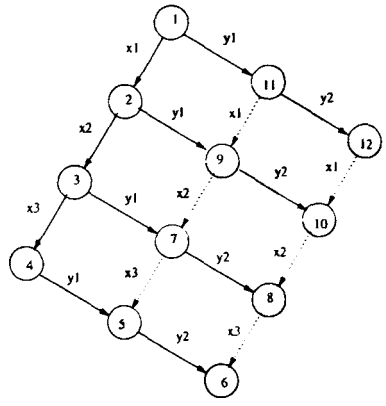


Figure 7.4: Sleep set exploration

7.3.2.3 Candidate Algorithms for Generating Reduced State Spaces

Based on the above discussion, we arrive at two candidate solutions for generating a reduced state space suitable for dynamic property checking:

- reduced state space generation based on ample sets, to generate a stuttering-equivalent state space, which is suitable for modal operators *Pos* and *Def* quantifying stuttering-invariant temporal properties
- reduced state space generation based on sleep sets, to generate a prefix-based stuttering-equivalent state space, which is suitable for modal operators $SOME$ and ALL quantifying stuttering-invariant properties

In what follows, we pursue the development of only one of these algorithms. Although both methods represent valid lines of investigation in an attempt to apply partial order reduction to the problem of trace checking, based on an "equivalent state space" approach, it would appear that the algorithm based on sleeps sets is particularly well-suited to the detection of dynamic properties for use with the algorithm of Babaoglu, Fromentin and Raynal. Despite this fact, we have chosen to pursue the development of an algorithm for generating a stuttering-equivalent reduced state space based on ample sets.

The reasons for this choice are based on two facts: (i) in the early stages of this research, the stuttering-equivalent approach to partial order reduction seemed a highly promising candidate, due to its success in model checking, and (ii) the impact of modal operators on the suitability of a state explosion technique for combating state explosion in trace checking was underestimated. As a consequence, a decision was made early on to adopt this approach. It was only after the approach was explored in detail did it become apparent that there were problems with this reduction approach for certain modal operators.

From one point of view, this represents a missed opportunity. We believe that an algorithm based on the sleep sets reduction is a very promising approach for the modal operators $SOME$

and ALL, and would result in an extremely simple and effective method. On the other hand, the development of the stuttering-equivalent approach, which appears in the sequel, illustrates very clearly the complex issues which can arise in adapting a state space reduction approach to the trace checking context, and in this sense, it successfully illustrates how the differences in context can affect techniques for combating state explosion.

As this thesis aims to investigate the issues involved in addressing state explosion in trace checking via model checking techniques, this early choice perhaps is fortuitous.

7.3.3 Summary

In this section, we have seen that partial order reduction can be applied using the "modified system" approach, or the "equivalent state space" approach, and that these two approaches differ considerably in their development.

We also saw that the success of applying the equivalent state space approach is dependent upon the invariance of modal operators under the chosen equivalence. By examining the requirements placed upon an equivalence which preserves these modal operators, we brought forward two candidate algorithms for generating reduced state spaces based on a selective search of the computation state space: one based on ample sets, and one based on sleep sets. As an illustration of the issues raised in adapting such a model checking algorithm to the trace checking context, we chose to develop the algorithm based on ample sets.

We review some facts concerning the problem we shall now try to solve:

1. Our approach is based on viewing the lattice of global states as a state transition system, and we attempt to apply the existing theory (and algorithm) of selective search based on ample sets from model checking to the problem of generating a reduced, stuttering-equivalent computation state space based on selective search for trace checking.

2. In particular, we aim to adapt this *existing* model checking algorithm to the trace checking context; that is, we do not design the algorithm from scratch.

3. The theory we start from consists of (i) a selective search algorithm based on ample sets, (ii) conditions for defining ample sets (iii) methods for calculating ample sets and (iv) an underlying theory of correctness

We now continue with the development of an algorithm for generating a reduced state space based on ample sets, which is suitable for trace checking.

7.4 Design Issues

In this section, we consider the design issues surrounding the design of an algorithm to generate a stuttering-equivalent reduced state space suitable for trace checking, given the fact that we know stuttering-equivalence is a suitable equivalence for checking properties based on the modal operators Pos and Def.

As mentioned previously, the partial order reduction approach views the full program state space as a transition system, and performs an exploration of that state space which explores only

enough states and transitions in order to guarantee that the resulting reduced state space preserves the properties of interest. The reduced state space is then used in the validation exercise to check whether or not properties hold. What we aim to do here is to apply the same technique to the computation state space, viewing the lattice of global states as a transition system, where each event corresponds to a transition. Although both the program state space and the computation state space can both be viewed as state transition systems, differences in the two contexts in which the technique is carried out may cause problems in porting the partial order reduction method to the new context.

As we aim to use an existing model checking algorithm (selective search based on ample sets) as a basis for development, we want to look at the differences in context between the existing model checking algorithm, and the trace checking context in which the algorithm to be developed will need to operate. Therefore, in this section, we consider the key assumptions underlying the partial order reduction method, and consider the possibility of conflicts with the new trace checking context.

The following key differences between the two contexts have been identified:

- **exploration order:** Algorithms for partial order reduction in the literature are generally based on depth-first search exploration of the program state space, whereas property detection is generally based on breadth-first search exploration of the computation state space. As our aim is to use partial order reduction in the context of breadth-first exploration of the lattice of global states, changes in exploration order potentially impact both the correctness and the effectiveness of the partial order reduction approach. Concerning correctness, depth-first search is the exploration method of choice in many treatments of partial order reduction, due to (i) its facility for conducting inductive proofs and (ii) its very concise characterization (via necessary and sufficient conditions) of how cycles may be created during an exploration. Consequently, many treatments of correctness in partial order reduction are cast in terms of depth-first exploration. Adapting the partial order approach to a context in which breadth-first search is a commonly-used exploration order may therefore require correctness proofs and other elements (such as ignoring provisos) to be redeveloped. Concerning effectiveness of the approach, the concise characterization of cycles in depth-first search again leads to relatively efficient methods for detecting such cycles and implementing provisos for avoiding ignoring, which in turn results in effective (in the sense of non-full expansion) computation of ample sets. Characterization of cycles in breadth-first search is much less concise (via sufficient conditions only), resulting in conservative provisos for guaranteeing the absence of ignoring, which in turn results in less effective computation of ample sets. This fact impacts the effectiveness of the method in a breadth-first search context.

- **safety properties only:** in a trace checking context, due to the fact that we may only observe finite prefixes of non-terminating distributed computations, we are only able to check properties whose satisfaction or violation may be demonstrated based on finite prefixes alone (e.g. safety properties). On the other hand, many formulations of the partial order reduction theory (c.f. [19, 86, 113]) present conditions for ample sets which are suitable for checking general safety *and* liveness properties. As will be shown in Section 7.5.2.4, a weaker notion of *finite stuttering-equivalence* can be used in place of stuttering-equivalence when checking

safety properties only. This opens up the possibility of producing a smaller reduced state space for checking safety properties in a trace checking context. However, adopting such an approach requires a corresponding theory of partial order reduction based on preserving finite stuttering-equivalence only. As in the case of exploration order, this will therefore require correctness proofs and other elements (such as ignoring provisos) to be redeveloped.

- **finite state assumption:** in model checking, the finite state assumption is used in order to guarantee that the exploration of the program state space will always terminate; in the dynamic property detection problem, there is generally no assumption that the program being monitored is finite state. As will be demonstrated, correctness proofs for the partial order theory depend on the finite state assumption (for guaranteeing absence of ignoring) and so, unless we are prepared to restrict the application of property detection to finite state programs, this issue needs careful consideration.

- **algorithm-initiated termination:** Model checking and trace checking also differ with respect to termination of the exploration and conditions holding at termination (such as the whether or not all reachable states have been explored at termination). In model checking, which bases state space exploration on graph traversal algorithms, termination of the state space exploration is guaranteed by the finite state assumption, together with the convention that states which are re-visited (after having already been explored) during the search are not re-explored. This convention 'works', as state space exploration algorithms systematically explore all possible successors of a state (and so explore all possible non-deterministic choices from each state), and so there is no need to re-explore a previously visited state: *all continuations from that re-visited state will have already been explored.* This guarantees that all states reachable from the set of initial states will be explored at termination. In trace checking, the situation is subtly different. In trace checking, it is in general impossible to detect at run-time when all reachable states of a distributed computation have been visited, and so impossible to terminate the exploration with the guarantee that all reachable states have been visited. This is due to the fact that, when a program contains states where non-deterministic choice between enabled transitions is possible, such possible non-deterministic choices in the program are resolved non-deterministically by the program execution itself, before the computation state space exploration begins. Thus, if a state is re-visited by the program, a different set of non-deterministic choices may be made from that re-visited state. Therefore, *the assumption that the possible continuations of a re-visited state will not only be the same but will also already have been visited does not hold in the trace checking context.* This means that, in general, if we wish to visit all reachable states of the distributed computation, we cannot terminate the exploration of the computation state space before reaching a terminal state, if such a state exists. Thus, the issue of termination in the algorithm for generating an equivalent state space will need to be considered carefully. Even if were possible to determine at run-time when all reachable states have been visited, there are application-dependent reasons which may influence the decision to terminate the exploration of the state space. For example, in some applications, such as testing, where the aim is to discover logical errors in the design of software, termination of the exploration of a distributed computation once all reachable states have been visited may be appropriate.

On the other hand, in applications such as fault-tolerance, where the failure of software to satisfy its specification may be caused by factors other than logical design errors (e.g. failure of other components upon which the software depends, such as hardware) termination after having visited all possible reachable states may not be appropriate.

- **user-initiated termination:** in certain applications of trace checking, such as testing, we shall require the ability for the user to terminate the exploration of the computation state space. The reason is that, in the case of non-terminating executions, or even terminating computations which are excessively long, the user may not wish to pursue the checking of the full computation, and may be satisfied with checking a prefix of the computation instead. In such a case, some arrangement needs to be made for ensuring that the resulting state space (now corresponding to the prefix explored) is still suitable for property detection, in some sense.

Having examined the issues arising in the design of such an algorithm, each of these issues will need to be addressed in the development of an algorithm to generate an equivalent reduced state space suitable for trace checking. We address the issues in the next section.

7.5 Design

In designing an algorithm for generating a stuttering-equivalent reduced computation state space, we need to start out with the existing algorithm for generating a stuttering-equivalent reduced program state space (and related theory) and determine how the design issues identified above may be resolved.

Our approach will be to group the design issues into two types: issues which have been considered within a model checking context previously (i.e. the issue of exploration order, and the issue of checking safety properties), and issues which are not relevant to the model checking context (i.e. the finite state assumption, the issue of algorithm-initiated termination, and the need for a user-initiated termination protocol).

Concerning the first set of design issues, although the issues of exploration order and safety properties have been considered in the literature, we did not find a version of the theory of partial order reduction which considers them together. Therefore, we found it necessary to develop such a theory. Once such a version of the theory has been developed, designing an algorithm to work in the trace checking context would require adjusting the theory to deal with the remaining three design issues: the finite state assumption, the issue of non-termination, and the need for a user-termination protocol.

We shall therefore adopt a two step development approach:

- in the first step, we develop a theory of partial order reduction which incorporates stuttering-equivalence, safety only properties, and exploration independence, resulting in a partial order theory which more closely matches the requirements of trace checking context
- in the second step, we adjust that theory to deal with the remaining issues of a differing finite state assumption, algorithm-initiated termination of the exploration, and the need for a user-termination protocol

The rest of the design section is organized as follows.

In Section 7.5.1, we consider the design issues involved in developing a version of the theory of ample sets which is suitable for checking safety properties in an exploration-independent manner in a model checking context. We shall show that this can be achieved by tailoring the Peled theory for safety properties only and requires introducing a new concept of equivalence, suitable for safety properties only, and a new concept of ignoring. The result is a theory of partial order reduction for model checking which is suitable for checking safety properties in an exploration-independent manner.

In Section 7.5.2, we go on to consider the further design issues which need to be considered in order to adjust this theory for the trace checking context. We shall show that issues such as the lack of a finite state assumption , the inability to detect termination of the execution, and the need for user-initiated termination necessitate significant adjustments to the theory in order to adapt it to the trace checking context.

7.5.1 Model Checking Design Issues

This section concerns adjusting the theory and algorithm behind selective search based on ample sets to the issues of stuttering-equivalence, exploration order and safety properties only.

7.5.1.1 The need for an exploration-independent characterization of ample sets for checking safety properties

The theory of partial order reduction is based on generating a reduced state space which contains fewer states and transitions than the full state space, and which can be substituted for the full state space in the model checking exercise. The reduction algorithm ensures that the reduced state space is equivalent, in a particular sense, to the full state space, and the substitution can be made when checking properties which are insensitive to that equivalence.

The reduced state space is generated by a modified state space exploration algorithm, which explores only a subset of directions, called an ample set, enabled at each state encountered during the search. Informally, ample sets explore just enough transitions from each state of the full state space in order to correctly check the property in question.

Early research on partial order reduction defined ample sets in an exploration-specific manner, in terms of conditions on the particular state space exploration algorithm being used (e.g. depth-first search, breadth-first search). In Section 7.2.1, it was shown that ample sets can be defined in an exploration-independent manner by a set of conditions relating the full state space and the corresponding reduced state space.

Exploration-independent formulations of ample sets are becoming increasingly important. Firstly, state space exploration methods other than depth-first search are now being more widely used in validation exercises. Alur et al. [4] show that symbolic state space exploration, based on breadth-first search, can benefit from being combined with partial order reduction, in order to give greater reduction in time and space required to perform validation exercises. Lluch-lafuente et al. [69] demonstrate that partial order reduction can be used in combination with directed model checking, where heuristics are used to perform a goal-directed exploration of the state space, based on exploration methods such as best-first search, $A*$, and others. For each such exploration method,

exploration-dependent conditions defining ample sets must be formulated and associated proofs of correctness must be developed. An exploration-independent characterization of ample sets can greatly simplify such proofs of correctness.

The presentation in [19, Chapter 10] focused on the conditions characterizing the ample sets needed to generate reduced state spaces sufficient for checking safety and liveness properties. In particular, the resulting reduced state space is guaranteed to be *stuttering-equivalent* to the full state space: informally, every infinite sequence in the full state space is guaranteed to have a representative in the reduced state space which is stuttering-equivalent. This form of correspondence between the full state space and the reduced state space is required when checking general safety and liveness properties.

When checking safety properties only, the requirements on the reduced state space are not as strong as in the case of general safety and liveness properties. In particular, we need only a reduced state space which is finitely stuttering-equivalent: informally, every *finite* sequence in the full state graph has a finite representative in the reduced state graph which is stuttering-equivalent. Such reduced state spaces generally contain fewer states and transitions than the corresponding stuttering-equivalent counterparts, and so represent an important reduction in state explosion which can be achieved when checking safety properties only.

This difference in reduced state spaces arises due to the different characteristics of safety and liveness properties [3]. Informally, safety properties can only be violated on finite prefixes of executions, whereas liveness properties can only be violated on infinite suffixes of executions. Therefore, the reduced state spaces necessary for checking each class of property differ in the number and type of representatives they must contain.

In the sequel, we shall extend the exploration-independent characterization of ample sets presented in [19, Chapter 10] to the case required for checking safety properties, wherein the reduced state space need only be finitely stuttering-equivalent to the full state space. Of the four exploration-independent conditions used to define ample sets, only one of these is dependent upon the class of property being checked, and accounts completely for the differences in the reduced state spaces required. This condition, referred to in [19, Chapter 10] as the *cycle condition*, guarantees the absence of ignoring cycles in the reduced state space graph. Ignoring is a problem which affects methods for combating state explosion based on generating a reduced state space (e.g. *stubborn set* method of [112], *persistent set* method of [46], *ample set* method of [86]) where only a subset of enabled transitions are explored from each state. We show that the cycle condition has a corresponding, weaker, safety-only counterpart, guaranteeing the absence of *ignoring strongly connected subgraphs with no progress direction* in the reduced state space graph. We show that this weaker condition, defined on strongly connected subgraphs of the reduced state space, is sufficient to prove finite stuttering-equivalence between the full and reduced state spaces. We also show that this condition is consistent with existing exploration-dependent conditions for checking safety properties, based on depth-first search and breadth-first search, by way of a spanning forest analysis.

Although these results shall be required for the development of our algorithm for applying partial order reduction to trace checking, the details of the development are purely theoretical matter, firmly within the theory of partial order reduction. Rather than present them here, we instead summarize the key results required and relegate the details of the development of these

results to an Appendix, so that they may be referred to without disrupting the flow of discussion. In the next section, we summarize the key results concerning this "safety properties only" version of the partial order reduction theory based on ample sets.

7.5.1.2 An exploration-independent characterization of ample sets for checking safety properties

The exploration-independent theory of partial order reduction for the case of safety properties is based on the notion of finite stuttering-equivalence. Two labeled transition systems M and M' are *finitely stuttering-equivalent* if and only if

- M and M' have the same set of initial states
- for each finite path σ of M that starts from an initial state s of M, there exists a finite path σ' of M' that starts from the same initial state s such that σ is stuttering-equivalent to σ'
- for each finite path σ' of M' that starts from an initial state s' of M', there exists a finite path σ of M that starts from the same initial state s' such that σ' is stuttering-equivalent to σ

Notice that the definition of finite stuttering-equivalence of labeled transition systems involves only finite paths through the labeled transition systems.

The four exploration-independent conditions used to define ample sets in the case of finite stuttering-equivalence are as follows, where s represents a state reached during the selective search, and we assume that the formula φ to be verified is defined on the set of propositions $AP' \subseteq AP$:

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$

C1 Along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first

C2 if s is not fully expanded (i.e. $ample(s) \subset enabled(s)$), then every $\alpha \in ample(s)$ is invisible (with respect to the propositions in AP')

C3-fin A strongly connected subgraph is not allowed in the reduced state graph if (i) it contains a state in which some transition α is enabled, but never included in $ample(s)$ for any state s in the subgraph, and (ii) there is no state s' in the subgraph for which $ample(s')$ contains a direction leading to state not contained in the subgraph

The condition **C3-fin** requires that the reduced state space cannot contain strongly connected subgraphs which are (i) ignoring and (ii) which do not contain directions which permit leaving the strongly connected subgraph. Given that a cycle in the reduced state space can be viewed as a strongly connected subgraph, it is easy to see that the condition **C3-fin** is strictly weaker than the condition **C3**. In the Appendix (Section A.2.2), it is proved that these conditions on ample sets are sufficient to guarantee that the reduced state space will be finitely stuttering equivalent to the full state space.

The exploration-independent description of the ignoring condition is a useful characterization of the properties of ample sets required to generate a finitely stuttering-equivalent state space, but in developing a real algorithm, we need provisos which can be efficiently computed and which guarantee the condition **C3-fin**. In the Appendix, Section A.3, we prove that this new exploration-independent condition is consistent with the following existing exploration-dependent provisos for ensuring the absence of ignoring in the case of safety properties:

C3-dfs' If s is not fully expanded, then at least one transition in $ample(s)$ must not reach a state that is on the search stack

C3-bfs' If s is not fully expanded, then at least one transition in $ample(s)$ must not reach a state that is in the current level or a previous level of the breadth-first search

In other words, if the proviso **C3-dfs'** (resp. **C3-bfs'**) holds during depth-first search-based (resp. breadth-first search-based) ample set exploration of the program state space, then the condition **C3-fin** holds in the reduced state space generated.

The exploration-dependent provisos for guaranteeing **C3** differ from the exploration-dependent provisos for **C3-fin** in terms of the creation of cycles in the reduced state space. For example, proviso **C3-dfs** requires that *no* transition in $ample(s)$ reach a state which is on the search stack (i.e. create a cycle) when $ample(s)$ is not fully expanded, as this potentially introduces an ignoring cycle into the reduced state space, and is enough to violate condition **C3**. On the other hand, proviso **C3-dfs'** requires that *at least one* transition in $ample(s)$ *not* reach a state which is on the search stack (i.e. create a cycle) when $ample(s)$ is not fully expanded. In other words, possible ignoring cycles *may* be created in the reduced state space, but not in such a way that condition **C3-fin** is violated. In the same way that a single transition (from state s) reaching a state on the search stack can potentially form an ignoring cycle, if all transitions (from state s) reach states on the search stack, it is possible to construct a strongly connected subgraph (informally, by taking all the individual cycles created and forming their union - each cycle is strongly connected, and all cycles have the state s in common) which is potentially ignoring and potentially such that there are no transitions in the reduced state space leading out of the subgraph. Such a subgraph would violate **C3-fin**.

7.5.1.3 Summary

The development presented in this section, together with the detailed proofs in the Appendix, provides a theory and associated algorithm, suitable for a model checking context, for generating a reduced state space which is finitely stuttering-equivalent to the full program state space.

In the sequel, we shall simply refer to the results of this theory as and when we need them. This will simplify the presentation of the required adjustments to make partial order reduction work at run time.

7.5.2 Trace Checking Design Issues

In this section, we consider the problem of adjusting the theory of selective search based on ample sets developed in the previous section to take account of the remaining design issues presented by the trace checking context:

- inability to terminate the execution, based on seeing all reachable states
- the need for user-initiated termination
- finite state assumption

We consider each of these issues in turn.

7.5.2.1 No Termination (Based On All Reachable States) Problem

As discussed in the design issues section, we noted that, unlike model checking, it is not possible to assume that the continuations of re-visited states will be isomorphic to the continuations of the states the first time they were encountered. This was due to the fact that, in model checking, possible non-deterministic alternatives from a state are systematically and exhaustively explored, whereas in trace checking, program execution non-deterministically selects one such alternative only, and so continuations may vary each time a given state is visited. This makes it impossible in general to terminate the exploration of the computation state space based on having seen all possible reachable states.

Several consequences arise due to this state of affairs:

1. The only way in which the exploration of the computation state space may terminate is by: (i) the computation reaching a terminal state or (ii) the user terminating the exploration through user-initiated termination (see Section 7.4.2.2)
2. The exploration algorithm may re-explore states of the computation state space which have already been visited

This has an impact on the operation of the algorithm. The **C3-fin** condition requires that, for ample set selections from state s in which $ample(s)$ is not fully expanded, at least one transition from $ample(s)$ leads to a new state which has not yet been explored. However, when re-exploring visited states, this will tend to result in full expansion. Why? Because for any transitions in $enabled(s)$ explored on previous visits to state s, we cannot explore the same ample sets - as these will not contain directions leading to new states. Thus, in order to satisfy condition **C3-fin**, the ample set construction algorithm must find ample sets containing new transitions, and so explore more and more transitions from the same state each time we visit it. Thus, exploring re-visited states will tend to cause ample set selection to result in full expansion.

Therefore, this inability to avoid exploring re-visited states, combined with ignoring proviso **C3-fin**, results in inefficient exploration of the computation state space. This is a significant impediment to porting the algorithm.

7.5.2.2 Need for User Termination

As pointed out in the design issues section, for certain applications (e.g. testing), we need to be able to user-terminate the exploration safely, and draw conclusions about the prefix of the computation observed, for two reasons:

1. the distributed computation being observed is a non-terminating computation

2. the distributed computation being observed is a terminating computation which, although terminating, is too long to check in its entirety

User-initiated termination of the exploration at level *level* will result in an incomplete reduced state space, in the sense that finite stuttering-equivalence may not hold between the (incomplete) reduced computation state space and the full computation state space. For example, if the full computation state space contains paths of length greater than *level*, and user-initiated termination occurs at a level *level'* such that $level' < level$, then the reduced computation state space generated to that point will contain paths of length at most *level'*, and so paths of length greater than *level* will not be represented.

In order to make use of such an incomplete reduced computation state space for the purposes of verification, some form of guarantee (in the form of an invariant) on the nature of the resulting equivalence between the incomplete reduced state space and the full state space would be required. One reasonable invariant is to require that, if the user-initiated termination were initiated at level *level*, all paths of length less than or equal to *level* have representatives in the (incomplete) reduced computation state space. But, for a given path, the length of a representative for that path depends upon the degree of ignoring present in the representative (where the degree of ignoring is represented by the number of invisible, independent transitions introduced by the selective search algorithm when constructing the representative) (see Appendix, Section A.2.2). In order to deal with possible ignoring present in paths, it is possible to introduce a *safe termination phase*, by performing an exhaustive exploration of the computation state space from the level *level* at which user termination was initiated, for a number of levels equal to the maximal amount of ignoring possible in any path of length less than or equal to *level*. This could be used to guarantee that all paths of length less than or equal to *level* were represented.

The problem with this solution is that the ample sets algorithm based on the ignoring proviso **C3-fin** can potentially introduce up to $|S|$ invisible, independent transitions in a representative, where $|S|$ is the size of the state space being explored. So, for example, if termination were initiated in level *level*, ensuring the invariant holds would require a safe termination phase of length *level* levels, which clearly negates any benefit obtained by the reduction.

Therefore, the ignoring proviso **C3-fin** makes implementing a termination protocol difficult.

7.5.2.3 Finite State Assumption

In the model checking algorithm, the finite state assumption plays a key role in the proof of finite stuttering equivalence (see Appendix, Section A.2.2) and, in particular, in the proof that ignoring does not occur in the reduced state space.

The assumption that the program state space is finite is essentially an assumption that the number of reachable states of the state space is finite and bounded. Given that in any exploration of the computation state space, the exploration will only ever examine a finite number of reachable states, we ask if it is possible to replace this assumption by the assumption that the number of reachable states is finite but unbounded.

As we saw earlier, there are only two ways in which the exploration may terminate:

1. the distributed computation is terminating, and is explored through to termination

2. the distributed computation is non-terminating, and the exploration is terminated by the user at level *level*

In the case of termination by the user, a termination protocol is required, and this protocol will need to ensure that ignoring does not occur for paths of length less than or equal to *level*. Whether the distributed computation contains finitely many states or infinitely many states, this does not affect the way in which any ignoring present in the (incomplete) reduced state space is adjusted.

7.5.2.4 Summary

The problems discussed in this section, namely

- the inability to terminate the execution, based on seeing all reachable states
- the need for user-initiated termination
- the finite state assumption

result in the ample sets algorithm in its current form (based on the development presented in Section , which resulted in the exploration-independent conditions **C0**, **C1**, **C2**, **C3-fin**) becoming inefficient when applied to the case of trace checking.

Each of these problems involve the ignoring proviso **C3-fin** and the issue of ignoring. As we shall show, it is possible to overcome these problems in the trace checking context by considering an alternative approach to dealing with ignoring.

In the next section, we consider an alternative approach to ignoring which avoids these problems.

7.5.3 An Alternative Approach To Ignoring

In [84], Nalumasu et al. investigated the use of an alternative approach to dealing with the problem of ignoring in partial order reduction. In this section, we describe this alternative approach and show how it can be used to overcome the problems cited earlier.

The authors noted that partial order reduction algorithms which involve the ignoring proviso can result in more states than necessary being explored, and the source of the inefficiency is the ignoring proviso itself. They introduce a partial order reduction method which does not involve an ignoring proviso. The method is based on a *two phase* approach to exploration of the program state space, in which the exploration of the set of paths leaving a state in the program state space consists of a repeated alternation of two phases: a first *deterministic, partial order* phase, followed by a second *full expansion* phase. The first phase is deterministic, in the sense that, during this phase, from any state s, program transitions are selected for exploration from state s only if they belong to a single process and that process has only one enabled transition; the first phase is partial order, in the sense that, during this phase, only one interleaving is used to explore a portion of the state space containing many equivalent interleavings. We shall refer to the algorithm as the Two Phase algorithm for program state space exploration.

The Two Phase algorithm is based on a version of partial order reduction presented in Holzmann-Peled [56], summarized in Section 7.2.3, which uses static analysis to assist in the selection of ample

sets. Static analysis is used to identify program transitions which are *safe* (globally independent with respect to transitions on other processes and invisible). Ample sets are then identified by locating processes P_i whose set of enabled transitions $T_i(s)$ are such that each transition in the set is safe. Following this approach, the Two Phase algorithm introduces a first, deterministic partial order phase by requiring that any ample sets (sets of safe transitions $T_i(s)$ from a process P_i) selected in the first phase are such that $|T_i(s)| = 1$. The algorithm is then structured to ensure that a second, full expansion phase always follows the first.

```
procedure model_check()
begin
    V_r := ∅; E_r = ∅;
    Twophase(InitialState);
end

procedure Twophase(s)
begin
    /* Phase 1 */
    (path, s) := phase1(s);

    /* Phase 2 */
    if (s ∉ V_r) then
        V_r := V_r + all states in path;
        E_r := E_r + path;
        foreach (t ∈ enabled(s)) do ;
            E_r := E_r + (s, t, t(s));
            if t(s) ∉ V_r then
                Twophase(t(s));
            fi
        endforeach ;
    else
        V_r := V_r + all states in path;
        E_r := E_r + path;
    fi
end
```

(a) Two Phase procedure

```
procedure phase1(in)
begin
    s := in; list := {s}; path := {};
    foreach process P do
        while (deterministic(s, P)) do
            /* t, the only transition enabled */
            olds := s;
            s := t(olds);
            olds := s;
            path := path + {(s, t, t(s))};
            if (s ∈ list) then
                goto NEXTPROC;
            fi
            list := list + {s}
        end while
        NEXTPROC :
    endforeach
    return (path, s);
end
```

(b) phase1() procedure

Figure 7.5: Algorithm for depth-first two phase

The algorithm is presented in Figure 7.5. The main algorithm consists of a recursive, depth-first search-based procedure *TwoPhase*(), pictured in Figure 7.5(a), which explores the paths from a state s in two phase fashion, and generates a reduced program state space. The reduced program state space is stored in a graph, $G_r = (V_r, E_r)$, with vertices V_r representing states reached in the exploration, and edges E_r representing transitions between states.

The procedure *TwoPhase*() begins by calling procedure *phase1*(), presented in Figure 7.5(b), to carry out the first phase processing. In the first phase, the algorithm visits each process once, in turn. At each process P_i, the algorithm calls the function *deterministic*(s, P) which returns *true* if the set of enabled transitions $T_i(s)$ at process P_i are each safe and such that $|T_i(s)| = 1$. If true, the transition is explored, and the relevant data structures are updated (the variable *path* holds the path of states and transitions explored in the current instance of phase 1 processing, and *list* holds the states visited in the current instance). Such deterministic execution continues

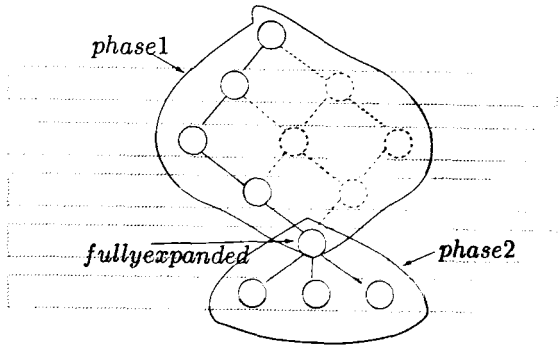


Figure 7.6: Depth-first two phase algorithm

at process P_i until $deterministic(s, P)$ returns $false$, or it reaches a state already visited in the first phase. At this point, phase 1 processing switches to the next process. At the end of the first phase, which is guaranteed to terminate as the program is finite state, only a sequence of deterministic, safe transitions (possibly empty) have been executed. At this stage, transitions which were enabled on the path but not explored, due to being involved in non-deterministic choice, may have accumulated. The procedure *phase*1() returns the final state reached by deterministic exploration, together with the path of states and transitions fired.

The procedure *TwoPhase*() then executes the code following the call to *phase*1(), which performs second phase processing. This processing phase is based on a classical recursive, depth-first search exploration. The graph (V_r, E_r) is first updated with the graph edges explored in the first phase. If the state s reached at the end of the first phase has not been visited, *TwoPhase*() performs a full expansion of the state s; otherwise, if the state has already been visited, the algorithm backtracks. Any ignoring on the path explored in phase 1 is eliminated by performing a full expansion in phase 2. This algorithm avoids ignoring by construction: on any path in the reduced state space leaving an expanded state, phase 1 always terminates, and is always followed by phase 2.

Figure 7.6 illustrates the way in which the algorithm works. States and transitions which are dashed represent states and transitions in the full program state space which are not explored.

The algorithm was presented in the context of SPIN [55]. Nalumasu proved that his provisoless approach can be used to model check stutter-invariant LTL properties, as well as stuttering-invariant CTL* properties.

We shall use these ideas in our algorithm to generate a reduced computation state space suitable for a trace checking context. We use the basis of the *two phase* approach to generate a finitely stuttering-equivalent state space in a trace checking context:

1. to permit the efficient calculation of ample sets, based on identification of a set of *safe* transitions from a process

2. to avoid the problems caused by using the ignoring proviso in a trace checking context

Further, as we shall see, the two phase approach to state space exploration is well-suited to the trace checking case, for two reasons:

1. in a distributed computation, and identifying events with their underlying transitions. the sets $T_i(s)$ of enabled transitions in state s at each process P_i, although they may not contain only safe transitions, will be deterministic, and so the deterministic phase is well-suited to the structure of distributed computations

2. the approach can be adapted to control the degree of ignoring in the state space when used with a termination protocol, as well as to work with distributed computations generated by programs with infinite state spaces

This new proviso will permit the following features:

1. it avoids the problem associated with revisiting parts of the state space and the negative effect on ample set selection resulting from ignoring proviso **C3-fin**

2. it aids the development of a user termination algorithm, as we can now limit the degree of ignoring which occurs to a user selectable bound, and so control the inefficiency resulting from invoking the user termination protocol

3. it allows exploration of infinite state programs, as the level of ignoring present in any incomplete prefix of the distributed computation can be bounded

We describe the algorithm in the next section.

7.5.4 The Algorithm

Our algorithm for generating a reduced finitely stuttering-equivalent computation state space based on partial order reduction based on ample sets is presented in this section. The key elements of the algorithm are: (i) ample set selection is based on the static analysis method of Holzmann-Peled where we look for a process P_i with $T_i(s)$ containing only safe transitions and (ii) exploration of paths is structured into two phases: a deterministic first phase in which no non-determinism is explored from any state reached, followed by a second phase of full expansion from a state.

Our algorithm differs from the depth-first algorithm presented in [84] in the following ways:

1. the algorithm is breadth-first, and so several paths are explored simultaneously; therefore, data structures necessary for managing the exploration need to be associated with each state

2. it is not required to record states visited during the exploration (in a hash table or a graph representing the explored state space), as we do not terminate the exploration based on having seen all reachable states

3. the way in which phase 1 is terminated, which depends in part on a differing view of the impact of cycles in model checking and trace checking

The last point deserves some explanation. In the algorithm presented in [84], a means of ensuring that phase1 terminates is required. This is achieved by maintaining a 'list' of the states reached in phase 1, and switching deterministic exploration at the current process to the next process, if any, when a process reaches a state already visited in phase1 (i.e. in the 'list'). This procedure, combined with the maintenance of the list, serves three functions: (i) it is used to guarantee that

phase 1 terminates (the program is assumed finite state, and so each process will eventually see a repeated state, and avoid a potential infinite loop) (ii) the 'list' is used to keep a record of the states and transitions traversed in phase1, so that the state graph can be updated after phase1 completes (iii) use of the list avoids one process exploring the same deterministic cycle in phase 1, which in model checking is wasteful, as it would not contribute to new states and transitions being added to the state graph.

In our algorithm, we also require a means to ensure that phase1 terminates. We could impose a finite state assumption on our computations and use the 'list' method to ensure that phase1 terminates. But such a method unduly restricts the distributed computations the resulting method can be applied to. Further, allowing a process to deterministically explore a cycle is not a problem in trace checking - in fact, the events making up the cycle must be explored as they form part of the computation. Because the exploration is organized in breadth-first manner, we also have no need for keeping a separate record of the states explored in phase1 to maintain the state graph(represented by previous and current). But more importantly, using the 'list' approach does not guarantee how much independent progress a process can make, leading potentially unbounded ignoring. We instead guarantee termination of phase1 simply by placing a numerical bound $MAXPROCSTEPS$ on process steps. This allows us to guarantee termination of phase1 (phase1 is guaranteed to terminate in at most $MAXPROCSTEPS * NPROCS$ steps, where $NPROCS$ is the number of processes), has the advantage of not requiring a finite state assumption, and also allows us to place a bound on ignoring present in the reduced state space, which aids development of a termination protocol. The question arises as to how we might set the bound. Setting the bound small raises the possibility of making less 'partial order progress' than might otherwise be achievable. Setting the bound high will allow more partial order progress, but will result in a long termination protocol. The choice of bound will depend upon whether termination is required.

We now describe the algorithm, which is presented in Figures 7.7, 7.8, 7.9, and 7.10.

Figure 7.7 shows the main algorithm, which is essentially a modification of the Cooper-Marzullo algorithm which incorporates the partial order reduction approaches described earlier, where the two phase approach is adjusted for breadth-first search. The data structures required for execution of the algorithm are as follows. The sets *previous* and *current* represent sets of global states, reached in the previous level and current level of exploration, respectively. The current level of exploration is held in *level*. The constant $MAXPROCSTPS$ holds the user-defined bound on the number of process steps which may be performed by any process in phase 1. The constant $NPROCS$ is set to the number of processes. Associated with each global state Σ, we maintain the following information:

- $\Sigma.phase$ - the current phase of processing associated with Σ
- $\Sigma.pid$ - the process id of the process from which execution is currently being explored, if processing is in phase 1; undefined otherwise
- $\Sigma.stepCount$ - the current number of events already processed at the current process $\Sigma.pid$, if processing is in phase 1; undefined otherwise

The initial state Σ_{init} is initialized so that $\Sigma_{init}.phase = first$, $\Sigma_{init}.pid = 1$ and $\Sigma_{init}.stepCount =$

```
1   current : set of global states : init {initial state}
2   previous : set of global states : init ∅
3   level : integer : init 0
4   MAXPROCSTEPS : const integer : init {bound on number of process steps}
5   NPROCS : const integer : init {number of processes}

5   while current ≠ ∅
6       level = level + 1;
7       previous = current;
8       current = ∅;
9       foreach Σ ∈ previous do
10          if (Σ.phase == first)
11              % perform first phase processing
12              phase1(Σ);
13          else
14              % perform second phase processing
15              phase2(Σ);
16          fi
17      od
18      % wait until we can compute next level level

19  end while
```

Figure 7.7: Algorithm for computing reduced state space.

0.

The algorithm operates on a level-by-level basis, as in the Cooper-Marzullo algorithm. Each state Σ in *previous* is processed as before, however, processing is now dependent upon the current phase $\Sigma.phase$ associated with the state: at each level, phase1 processing can result in only one deterministic successor being explored, whereas phase 2 processing explores all possible successors. Once all states in *previous* have been processed, the algorithm waits until enough events have arrived to begin processing of the next level, as in the original Cooper-Marzullo algorithm.

Figure 7.8 shows the algorithm for performing phase 1. Procedure *phase*1() performs the deterministic phase of the exploration, and operates analogously to the phase 1 processing of the algorithm *TwoPhase*. The predicate $enabled(\Sigma, pid)$ returns the set of events which are enabled in state Σ, and which belong to process *pid*: this set can be empty, or contain a single event. Given an event $e \in H$, we denote by $e.t$ the transition associated with e (the transition t of which e is an instance) and denote by $e.pid$ the process id of the process to which $e.t$ belongs. The predicate $safe(e)$ is true when the transition underlying event e is *safe*: when $e.t$ is ***invisible*** with respect to the set of propositions $Props(\varphi)$ associated with the dynamic property φ, and $e.t$ is *globally independent* with respect to all transitions on processes P_j, $j \neq e.pid$. We assume that these attributes of events have been identified through a static analysis phase of the distributed program and included in the information associated with events at generation.

Phase 1 proceeds as follows:

- if there is a transition e enabled at state Σ satisfying $safe(e)$ and associated with the current process $P_{\Sigma.pid}$, then that transition is explored from Σ resulting in state Σ'. If processing

```
1   procedure phase1(Σ)

2   if (∃e ∈ enabled(Σ, Σ.pid) s.t. safe(e))
3       % compute the successor state
4       Σ' = e(Σ);
5       % check if we have performed max steps
6       if (Σ.stepCount < MAXPROCSTEPS)
7           % continue processing process pid in next level
8           Σ.stepCount + +;
9       else
10          % switch processing to next process in next level
11          updatepid(Σ);
12      fi
13      Σ'.phase = Σ.phase; Σ'.pid = Σ.pid;
14      Σ'.stepCount = Σ.stepCount;
15      % add successor to next level to be processed
16      current = current + {Σ'}
17  else
18      % no deterministic transition - change process
19      updatepid(Σ)
20      if (Σ.phase == first)
21          phase1(Σ)
22      else
23          phase2(Σ)
24      fi
25  fi

26  end
```

Figure 7.8: Algorithm for phase 1

at $P_{\Sigma.pid}$ has not reached the maximum level of steps, $MAXPROCSTEPS$, then the step count is incremented and processing will continue at $P_{\Sigma.pid}$ in the next level; otherwise, the procedure *updatepid*() is called to update the process id and phase of processing which should be applied at the next level. The data structures associated with Σ' are initialized to reflect these decisions, and Σ' is added to the set *current* for processing in the next level.

- if there is no transition e enabled at state Σ satisfying $safe(e)$ and associated with the current process $P_{\Sigma.pid}$, then the procedure *updatepid*() is called to update the process and/or phase of processing which should be applied to Σ. The state Σ is then considered again for processing, based on the new values of its associated data structures. Note that this step may occur several times in the processing of a state, but that it will always result in either phase 1 processing at some process, or phase 2 processing being performed on Σ.

Note that the number of transitions which may be executed in any instantiation of phase 1 is bounded by $MAXPROCSTEPS * NPROCS$, as each process is limited to exploring at most $MAXPROCSTEPS$ transitions, and $NPROCS$ processes may participate in phase 1.

Figure 7.9 shows the algorithm for phase 2. In procedure *phase2*(), the algorithm simply

```
1  procedure phase2(Σ)

2  foreach (e ∈ enabled(Σ)) do
3      Σ' = e(Σ);
4      Σ'.phase = 1; Σ'.pid = 1;
5      Σ'.stepCount = 0;
6      % add successor to next level to be processed
7      current = current + {Σ'};
8  od

9  end
```

Figure 7.9: Algorithm for phase 2

```
1  procedure updatepid(Σ)

2  Σ.pid = Σ.pid + 1;
3  Σ.stepCount = 0;
4  % if all processes examined in phase1, switch to phase2
5  if (Σ.pid > NPROCS)
6      Σ.phase = second;
7  fi

8  end
```

Figure 7.10: Algorithm for updating process id

explores all possible successors Σ' of the state Σ. As part of the processing for each successor, it initializes the data structures to initiate phase 1 processing in the next level, and adds each successor to the set *current*.

Finally, Figure 7.10 shows the algorithm *updatepid*() which updates the data structures appropriately for Σ, causing the next process within phase 1, if any, to be considered or to switch to phase 2, if required.

Figure 7.11 illustrates the way in which our algorithm works.

7.5.5 Related Issues

In the following sections, we consider several issues which affect the use of the algorithm in checking dynamic properties:

- the determination of safe transitions
- the use of the algorithm with detection algorithms for detecting dynamic properties
- the compatibility of the approach with a user-initiated termination protocol

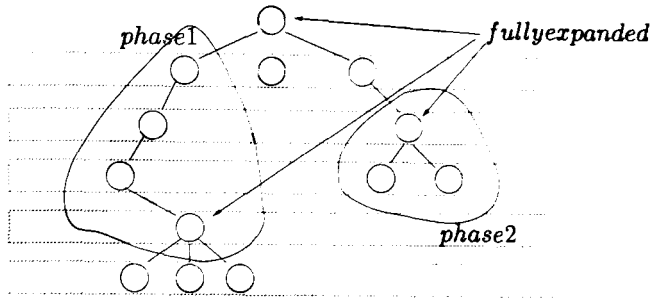


Figure 7.11: Breadth-first two phase algorithm

7.5.5.1 Determination of safe transitions

The algorithm is based on a selection of ample sets involving the static analysis approach described in [56]. In that paper, the selection of ample sets was based upon finding a set of enabled transitions which were safe with respect to the dynamic property φ being detected: globally independent (with respect to all transitions on other processes) and invisible with respect to the propositions in $Props(\varphi)$. In this section, we briefly discuss the identification of globally independent events in the trace checking context.

Dynamic property detection depends upon the modeling phase in order to instrument the distributed program with redundant, event generation code to produce the data making up the observation of the distributed computation. These event generators describe, for each event, for example:

- event type (internal, send, recv)
- vector clock timestamp of the event
- process id of the event
- local variables modified by the event

Such instrumentation requires some static analysis (pre-processing) of the program text, before the program is executed. Therefore, we can add in further static analysis to identify, for the transitions of the program (which underly the generated events):

- global independence of the transitions (with respect to transitions on other processes)
- visibility of transitions with respect to the propositions $Props(\varphi)$ in the dynamic property φ being detected

The approach requires consideration of the type of program statements and their semantics. In [56], the method of static analysis was applied within the context of the model checker SPIN, and its associated validation modeling language PROMELA [55]. A number of classes of program statements were identified as being safe. To decide global independence or visibility exactly would require examining all reachable states. Instead, heuristics are used to identify those transitions that are, with certainty, globally independent or visible, and assume that the other transitions are not.

In the model of a distributed program presented in Section 4.1, distributed program statements were classified as internal, send or receive. No programming language was described, and this precludes a detailed description of how to precisely define heuristics for deciding global independence and visibility. Therefore, the following discussion is somewhat informal.

The transitions in asynchronous distributed programs do not share memory, in the sense of sharing program variables local to a process, but they do share communication channels χ_{ij}. These shared memory objects have a bearing on the determination of global independence. The global state of a distributed program may require consideration of channel states in addition to the local states of processes involved in the computation. We now consider the three classes of program statements in turn:

Internal transitions int_i located on process P_i are globally independent with respect to transitions on other processes P_j, $j \neq i$, as these transitions share no memory and thus cannot be dependent. Internal transitions may or may not be visible, depending on their access to variables involved in atomic propositions involved in the property being checked.

In the case of $send_i(m)$ transitions on channel χ_{ij}, these transitions can exhibit dependencies with other communication transitions through channel states of χ_{ij}: channels act as shared variables in a distributed program, and so computing global independence of send transitions is, in general, as complex as checking reachability. As described in [56], the situation is greatly simplified if it can be shown that only one process has *exclusive send access* to a channel χ_{ij} (which must also include use of predicates which test the contents of the channel or whether the channel is full). Under the assumption of exclusive send access to the channel χ_{ij}, the transition $send_i(m)$ will be globally independent. Transitions $send_i(m)$ modify the channel state χ_{ij} upon execution, and may therefore be visible to any channel predicates defined on χ_{ij} and used in the property.

In the case of $recv_j(m)$ transitions on channel χ_{ij}, these transitions can also exhibit dependencies with other communication transitions through channel states of χ_{ij} and so computing global independence of receive transitions is, in general, as complex as checking reachability. As in the case of send transitions, the situation is greatly simplified if it can be shown that only one process has *exclusive receive access* to a channel χ_{ij}. Under the assumption of exclusive receive access to the channel χ_{ij}, the transition $recv_j(m)$ will be globally independent. Transitions $recv_j(m)$ modify local states of P_j as well as the channel state χ_{ij} upon execution, and may therefore be visible to any channel predicates defined on χ_{ij} and used in the property.

In summary, transitions which are *not safe* include:

- internal transitions int_i which are visible to propositions defined in terms of variables on P_i
- $send_i(m)$ transitions on channel χ_{ij} which do not have exclusive send access to the channel χ_{ij}, or which are visible to channel predicates defined on χ_{ij}
- $recv_j(m)$ transitions on channel χ_{ij} which do not have exclusive read access to channel χ_{ij}, or which are visible to channel predicates defined on χ_{ij} or local predicates defined on P_j

Thus, we assume that program transitions are identified as being safe or not, and visible or not, according to considerations of the type described above, and the information recorded in a table. During the execution and monitoring phase, this information is added to the usual information included with the event.

7.5.5.2 Detection of temporal properties

In order to make use of the algorithm developed in this chapter to detect stuttering-invariant temporal properties φ qualified by the modal operators Pos and Def, we require a detection algorithm to detect such temporal properties in combination with the modalities Pos and Def.

In the case of terminating distributed computations, we may use the algorithm developed in this chapter to conduct the exploration of the reduced computation state space. This algorithm may then be combined with the Babaoglu, Fromentin, Raynal algorithm used in the detection of $SOME$ and ALL to decorate the states of the computation states space with the automaton states reachable at each global state. Then, based on the relations:

$$\gamma \models Pos\,\varphi \quad \text{iff} \quad \Sigma_{final} \models SOME\,\varphi$$
$$\gamma \models Def\,\varphi \quad \text{iff} \quad \Sigma_{final} \models ALL\,\varphi$$

where Σ_{final} represents the maximal state of the lattice of global states $\mathcal{L}_\gamma = (\Sigma_\gamma, \prec_\gamma)$, we may decide $Pos\,\varphi$ and $Def\,\varphi$, based on the reduced state space generated.

In the case of non-terminating distributed computations, correct conclusions concerning the satisfiability of Pos and Def cannot be made in general based on a finite prefix. However, examination of the above relations at the final state of the finite prefix can provide useful information concerning satisfaction or violation of the dynamic property on the prefix.

7.5.5.3 Termination Protocol

In Section 7.5.2.2, it was noted that the algorithm for partial order reduction based on the ignoring proviso **C3-fin** can introduce up to $|S|$ invisible, independent transitions in the representative for a path and this can defeat any attempt at implementing a user-termination protocol which guarantees that, for paths of length *level*, stuttering-equivalent representatives will be present. In this section, we indicate how our algorithm can make implementing a user-termination protocol feasible.

One advantage of our algorithm is that it permits us to bound the degree of ignoring present in the reduced state space, for any given transition. As will be shown in the course of the proof of correctness of the algorithm in Section 7.6.2, the maximum degree of ignoring for any given transition is $MAXPROCSTEPS * NPROCS$. This bound on the degree of ignoring present allows the development of a user-termination protocol with the property that, if user-termination is initiated at level *level* and the user-termination protocol initiated, the reduced state space generated will contain stuttering-equivalent representatives for all paths in the full computation state space of length at most *level*.

In the case where user termination of the protocol is required, termination can be initiated at an arbitrary level *level*, and the termination protocol needs to guarantee that all finite paths of length *level* or less have stuttering-equivalent representatives in the (incomplete) reduced computation state space. This can be achieved by switching the mode of exploration from selective search at *level*, to exhaustive search in subsequent levels, and continuing the exploration for at least $level * MAXPROCSTEPS * NPROCS$ levels. This will guarantee that all paths of length at most *level* will have stuttering-equivalent representatives included.

We do not describe the algorithm here.

7.6 Correctness

In this Section, we prove that the *TwoPhase* algorithm, presented in Figures 7.7, 7.8, 7.9, and 7.10, does indeed generate a finitely stuttering-equivalent computation state space. Our advantage in this regard is that we have already proved (in the Appendix, Section A.2.2) that ample set selections satisfying conditions **C0**, **C1**, **C2**, **C3-fin** generate a finitely stuttering-equivalent computation state space. There, the strategy of the proof was to take a sample finite path in the full computation state space, and show exactly how the stuttering-equivalent representative(s) for that path are constructed. Much of the proof in the Appendix can be reused if we can show:

1. The set of transitions selected at each state s in the *TwoPhase* algorithm, denoted by $2P(s)$, satisfy conditions **C0**, **C1**, and **C2**.

2. These 'ample' set selections, when combined with the two phase exploration regimen of the *TwoPhase* algorithm, guarantee the absence of ignoring (i.e. all transitions in the finite path are eventually fired in the representative)

Then, by the details of the proof presented in the Appendix, we can conclude that for any given finite path σ in the full computation state space, the reduced computation state space generated by the *TwoPhase* algorithm contains a finite path which is stuttering-equivalent to σ, and therefore that the full computation state space is finitely stuttering-equivalent to the reduced state space.

In the proofs that follow, we need to make the following qualifying remarks:

1. Although the *TwoPhase* algorithm operates entirely on events in the distributed computation $(H, \rightarrow)$ and the information contained in those events, the selection of ample sets of events is based on the transitions T underlying those events. Indeed, although events and their causal dependency relation are used to determine the enabledness of events in a global state of the lattice, it is the global independence and visibility of the transitions underlying the events which is used to decide which collection of events forms an ample set, and drives the exploration of the lattice. Reasoning about the correctness of such set selections is cast in terms of the transitions underlying the events. Therefore, in the interest of clarity, we present the proofs of correctness in terms of transitions underlying the events of the distributed computation, as opposed to events themselves. This distinction between events and the transitions underlying them should be clear from the context.

2. The proofs of correctness of the *TwoPhase* algorithm are dependent on the proofs of the results introduced in Section 7.5.1.2 concerning exploration-independent conditions for generating a finitely stuttering-equivalent state space. The details of these proofs were presented in the Appendix. We shall at times need to make reference to the details of those proofs here.

7.6.1 Proving set selection satisfies C0, C1 and C2

We first show that the selection of ample sets in the *TwoPhase* algorithm satisfies **C0**, **C1** and **C2**.

In what follows, we shall need to associate with a transition t the process to which it belongs (this view of transitions belonging to a single process is consistent with our model of asynchronous distributed programs). Given a transition $t \in T$, $t.pid$ indicates the process id of the process to which the transition t belongs.

Lemma 7.1. Let $2P(s)$ represent a set of transitions selected by the algorithm of Figure 7.7. Then $2P(s)$ satisfies conditions **C0**, **C1** and **C2**.

Proof. The set selection depends upon which phase of processing the state s is undergoing.

In the case where the state s is undergoing phase 1 processing, the only set $2P(s)$ of transitions which can possibly be selected and explored from s consists of a single deterministic transition $t \in enabled(s)$ such that $safe(t)$ holds and $t.pid = s.pid$.

C0 is satisfied as if $2P(s)$ is non-empty, then it contains a transition enabled at s, and if $enabled(s)$ is empty, then no such deterministic transition exists and $2P(s)$ is empty. To show that **C1** is satisfied, we need to show that along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $2P(s)$ cannot be executed without a transition in $2P(s)$ occurring first. Suppose not; that is, suppose that for some path $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_n \xrightarrow{t_n} s_{n+1}$ in the full state space, t and t_n are dependent and $t \neq t_n$. We assume without loss of generality that t_n is the first such dependent transition. Since $t \neq t_n$, then t_n cannot be such that $t_n.pid = t.pid$; that is, they must be on different processes. But we know that t is globally independent with all transitions t' such that $t'.pid \neq t.pid$, and so, for all states s' where t and t' are simultaneously enabled, we know that t and t' are independent. Because t is independent of the transitions $t_1, \ldots, t_{n-1}$, we know that t is enabled in s_n. Therefore t and t_n cannot be dependent, which presents a contradiction. Thus, condition **C1** is satisfied by $2P(s)$. Finally, **C2** is also satisfied, as in the case when the set $2P(s)$ is not fully expanded at s, the single deterministic transition in $2P(s)$ is guaranteed to be safe, and therefore invisible.

In the case where the state s is undergoing phase 2 processing, $2P(s) = enabled(s)$, and this set selection obviously satisfies all three conditions.

□

It is instructive to consider, given the new way in which sets of transitions are selected by the *TwoPhase* algorithm, the way in which the representative iterations $\pi_k = \eta_k \circ \theta_k$ are constructed from σ. Given $\pi_k = \eta_k \circ \theta_k$, we select the next transition to fire from θ_k at state s_k based on whether s_k is undergoing phase 1 processing or phase 2 processing:

- if s_k is undergoing phase 1 processing, then there are three cases:
  - the transition selected is α, because we happen to be processing $\alpha.pid$, and α is deterministic (case **A**)
  - the transition selected is not α, but some other deterministic transition βon process $\beta.pid$ with $\beta.pid \neq \alpha.pid$. Note that such a transition selection satisfies **C1**, by the above lemma. Consequently, as in the proof of finite stuttering-equivalence, there are two cases:
    - the deterministic transition β is in θ_k(case **B1**)
    - the deterministic transition β is not in θ_k and is independent of all transitions in θ_k(case **B2**)
  - there are no deterministic transitions enabled at s_k and we switch to phase 2 processing to process s_k

- if s_k is undergoing phase 2 processing, then the transition selected is α(case **A**)

In this way, we may carry out exactly the same proof steps as was done for the proof of finite stuttering-equivalence, as every transition explored by the *TwoPhase* reduced state space construction appears as one of the cases in the ample set reduced state space construction.

7.6.2 Proving absence of ignoring

A key difference in the proofs of finite stuttering-equivalence between **C3-fin** and the *TwoPhase* algorithm lies in how ignoring is dealt with. In the proof of finite stuttering-equivalence presented in the Appendix, in which ample sets satisfied the conditions **C0**, **C1**, **C2** and **C3-fin** , we defined the following property of representatives constructed by the algorithm:

CMPLT for all $i \geq 0$, if α denotes the first transition of θ_i, then there exists $j > i$ such that α is the last transition of η_j and, for $i \leq k < j$, $\alpha$is the first transition of θ_k

This property, when satisfied, guarantees that a representative minimally fires all transitions in the path σ. It was shown in [19, Chapter 10], that this property holds for all representatives which satisfy condition **C3**. However, when **C3** is replaced with **C3-fin**, not all representatives have this property (i.e. some representatives can ignore α by entering an ignoring cycle and remaining in that ignoring cycle by selecting B2 transitions infinitely often). This led to differentiating between $reps(\sigma)$, the set of all representatives of σ, and $completereps(\sigma)$, those representatives which satisfy the completeness property.

In the *TwoPhase* algorithm, this state of affairs can never happen: representatives may not ignore enabled transitions indefinitely. Informally, representatives constructed by the *TwoPhase* algorithm may contain several *traversals* of an ignoring cycle (in terms of the construction, **B1** and **B2** may be fired in phase 1, which may traverse part of an ignoring cycle), but they will never be allowed to pursue the transitions making up those cycles indefinitely (**B1** and **B2** will only be selected in phase 1, and phase 2 will always be triggered in finite time, whether by maintaining a list of states visited in phase 1, or by placing a bound $MAXPROCSTEPS$ on the number of transitions a process may execute in phase 1). Therefore, the completeness property holds for *every* representative of σ constructed and so $reps(\sigma) = completereps(\sigma)$ for the *TwoPhase* algorithm.

We now demonstrate formally that, given an arbitrary representative, it belongs to $completereps(\sigma)$.

Lemma 7.2. Let α be the first transition on θ_i. Then there exists $j > i$ such that α is the last transition of η_j and, for $i \leq k < j$, $\alpha$is the first transition of θ_k.

Proof. According to the construction, if α is the first transition of θ_k, then either it is the first transition of θ_{k+1} (case **B**), or it will become the last transition of η_{k+1} (case **A**). We need to show that the first case cannot hold for every $k \geq i$. Suppose that this is the case. Consider the sequence of states $s_i, s_{i+1}, \ldots$. Let $s_k = first(\theta_k)$. According to the construction, $s_{k+1} = \gamma_k(s_k)$ for some $\gamma_k \in 2P(s_k)$. Moreover, because α is the first transition of θ_k and was not selected in case **A** to be moved to η_{k+1}, α must be in $enabled(s_k) \setminus 2P(s_k)$. Because the end of phase 1 will be reached in at most $MAXPROCSTEPS * NPROCS$ steps from s_i, there exists l satisfying $0 \leq l < MAXPROCSTEPS * NPROCS$ such that phase 1 terminates in state s_{i+l} and phase 2 initiated in state s_{i+l+1}. Then one of the following two cases holds:

- for some j, $0 \leq j \leq l$, $2P(s_{i+j}) = \{\alpha\}$, and α is fired in s_{i+j} (i.e. α is fired in phase 1) or
- phase 2 is initiated in s_{i+l+1} where full expansion occurs, and α is fired in s_{i-l-1} (i.e. α is fired in phase 2)

In either case, α will be fired, contradicting the assumption that case **A** never holds.

□

The rest of the proof of the finite stuttering-equivalence between the full computation state space and the reduced computation state space follows from the proof steps already given in the appendix.

7.7 Complexity

In this section, we consider the complexity of the approach. The algorithm presented in this chapter is essentially the level-based, computation state space exploration algorithm of Cooper-Marzullo, adjusted so that exploration of states in the lattice proceeds in a sequence of two phases: a deterministic phase, followed by a full exploration phase. The reduction potential of this algorithm lies in phase 1 processing, in which only a subset of enabled transitions from each state reached in phase 1 are explored.

We consider the time complexity of the procedures introduced. The procedure *phase*1() represents the deterministic phase of exploration. Examining the algorithm in Figure 7.8, we see that the main algorithmic complexity introduced is in (i) determining if a safe, enabled event e exists on process $\Sigma.pid$ and (ii) computing the successor state $e(\Sigma)$ and updating the data structure elements associated with the successor state Σ'. If no such enabled event exists, this processing may be repeated at the next process through a call to *updatepid*(). Therefore, each call to *phase*1() involves $O(N)$ time complexity. The procedure *phase*2() implements a simple computation of all possible the successors of the state Σ, and so has time complexity of $O(N)$. The procedure *updatepid*() has $O(1)$ time complexity.

We now consider the potential reduction of the method, which involves considering the degree to which phase 1 processing will be carried out during the state space exploration. The reduction achieved will vary depending on (i) how many safe events(transitions of the computation state space) are available in phase 1, and (ii) the particular choice of the variable $MAXPROCSTEPS$.

The availability of safe events depends upon which proportion of events in the distributed computation have been identified as being both globally independent and invisible. Global independence of an event corresponding to a program transition depends upon the degree of concurrency in the asynchronous distributed program. Visibility of an event depends upon the particular dynamic property which is being checked. The discussion on the determination of safe transitions, in Section 7.5.5.1, has a bearing on this. These considerations are the roughly same as those which apply to considering the reduction potential of the method in model checking.

The choice of the value for the variable $MAXPROCSTEPS$ was discussed earlier and involves a trade-off between the desire to allow phase 1 to continue as long as is possible, and the need to reduce the length of time of a termination phase, if any.

In general, we can expect similar degrees of reduction in this approach as are encountered in applications of partial order reduction to model checking, taking into account the fact that we now consider only one execution instead of all executions.

7.8 Implementation

We have implemented a version of the BFR algorithm which incorporates the partial order reduction described in this chapter. In this section, we present an example of a reduced state space generated by the algorithm.

The distributed computation $\gamma = (H, \rightarrow)$ we consider is made up of two processes, P_1 and P_2. The lattice of global states corresponding to the distributed computation is shown in Figure 7.12. State Σ^{ij} denotes the state of the distributed computation reached after exploring the first i events of process P_1 and the first j events of process P_2.

In the example, the propositions defined on global state are taken from the set $AP = \{\varphi_7, \varphi_{18}, \epsilon\}$. The proposition ϵ is used to represent the fact that none of the propositions $\varphi_i, i = 7, 18$ hold true in a global state. Global states are labeled with the propositions which hold true in those states. An exception to this rule are states whose labeling is the set $\{\epsilon\}$. In the interests of visual clarity, these state labellings have not been indicated in the figure. Any state with no labeling is therefore assumed to be labeled by the set $\{\epsilon\}$.

The property φ we consider in this example is one which can be described informally as requiring that the propositions φ_7 and φ_{18} appear one or more times in a path labeling, but that they appear in that order. It is described formally by the deterministic finite state automaton $A = (Q, \Sigma, Q_0, \delta, Q_F)$, where $Q = \{q_0, q_1, q_2, q_t\}$, $\Sigma = \{\varphi_7, \varphi_{18}, \epsilon\}$, $Q_0 = \{q_0\}$ and $Q_F = \{q_2\}$. The transition relation δ is defined by the following transition table:

δ	φ_7	φ_{18}	ϵ
q_0	q_1	q_t	q_0
q_1	q_1	q_2	q_1
q_2	q_2	q_2	q_2
q_t	q_t	q_t	q_t

Note that the property is stable: once the acceptance state q_2 is reached, there are no further constraints on the labels that may appear.

In the Figure 7.12, transitions which are shown in solid lines are those which are explored in the reduced state space, and transitions shown in dotted lines are not explored in the reduced state space.

The distributed computation $\gamma = (H, \rightarrow)$ of this example satisfies $\gamma \models Pos\,\varphi$ and $\gamma \models Def\,\varphi$, as all sequential observations leading to the final state Σ^{1010} pass through states satisfying φ_7 and φ_{18}, in that order. It can be seen that this satisfaction relation also holds for the reduced computation state space.

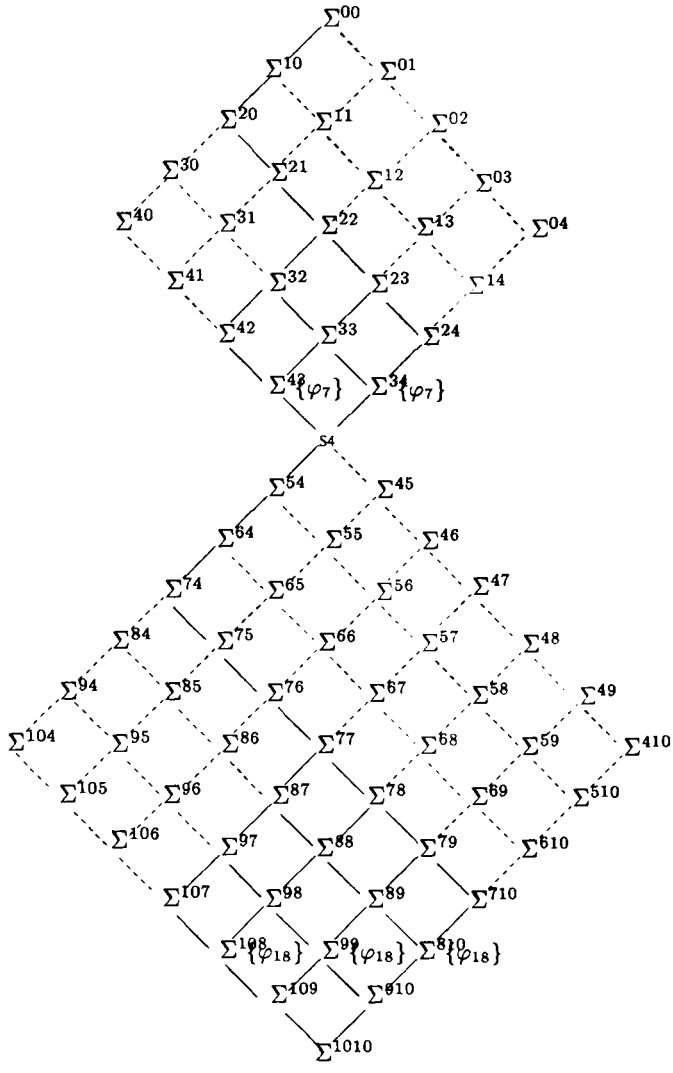


Figure 7.12: A distributed computation and partial order reduction

7.9 Conclusions

The partial order approach to state space reduction in trace checking of temporal properties of distributed computations was presented in this chapter.

The algorithm developed and presented in this chapter has the following advantages:

- suitable for checking temporal properties which are stuttering-invariant and quantified by the modal operators Pos and Def, standard modal operators used in property detection applications
- works at run-time or off-line
- works for terminating and non-terminating, infinite state programs (subject to the restriction that properties quantified by Pos and Def cannot be decided in general for non-terminating computations)

Unfortunately, there are disadvantages:

- reduction achieved depends upon the property (as in the case of partial order reduction model checking)
- not compatible for checking stuttering-invariant temporal properties quantified by the modal operators $SOME$ or ALL
- not suitable for some applications whose properties are not stuttering-invariant

Chapter 8

Conclusions

In this chapter, we aim to review the investigation carried out in the thesis, and draw the key conclusions and lessons learned. In doing so, we shall also point the way forward for further research along the lines drawn out in the thesis.

8.1 Summary of Results

The success of this research endeavor lies in the degree to which the following questions have been answered:

1. what can we conclude about the similarities and differences between the two problems, and, from that, which model checking techniques are viable candidates for use in a trace checking context

2. what can we conclude about the practical viability of attempts to transfer model checking method(s) to the run-time case?

In the case of similarities and differences between the two problems, these issues were investigated in Chapter 5, where we performed a comparative analysis of techniques used in model checking and trace checking for combating state explosion. There, the investigation was based on, first of all, comparing the two problem contexts, and then examining how the new trace checking context may or may not influence the success of using a model checking technique in the trace checking context. We considered each of the key model checking techniques in turn, identifying:

- the *existing* synergies between the technique and the existing trace checking techniques which incorporate approaches for mitigating the effect of state explosion. For example, we found that certain model checking techniques, namely, model extraction-based techniques, partial order reduction, symbolic and automata-theoretic approaches, were used in some limited form.

- the *potential* synergies between model checking and trace checking for incorporating approaches to combating state explosion. For example, we identified several candidates which

seemed well matched to the trace checking context, taking the context differences into account. These were described in the summary section, and formed the basis for our selection of two promising candidates for more detailed investigation.

Concerning the viability of an approach based on transferring techniques for combating state explosion from the model checking context to the trace checking context, we draw conclusions based upon our experience in attempting to port two existing model checking methods: an on-the-fly automata-theoretic based approach, and a partial order reduction approach.

In the case of the automata-theoretic approach, the results were promising. We presented an algorithm which mitigated the effect of state explosion and was at the same time compatible with certain important context differences: existing well-known techniques for performing dynamic property detection, the description of dynamic properties by (not necessarily stuttering-invariant) regular languages, and requirements for both on-line and post-mortem detection. The development of the algorithm was very straight forward, due in part to the existence of an algorithm for trace checking based on the automata-theoretic approach. Overall, this approach led to a working algorithm for dealing with the state explosion problem in trace checking, but only for certain modal operators.

In the case of the partial order approach, the situation was very different. The development of the algorithm was long and protracted, due mainly to context differences (such as exploration order, finite state assumption, the variety of modal operators, and termination) which required adjustment of the partial order reduction theory. The situation was also complicated by the fact that the partial order theory can be applied to the problem in several ways, and as part of the design of the approach, we had to decide between what would otherwise be suitable candidates for investigation. Overall, the approach led to a working algorithm for mitigating the effects of state explosion through partial order reduction, but only for certain modal operators.

Overall, on the basis of our development of these two approaches, we conclude that model checking techniques can be used in a trace checking context, but due to the differences in context, significant changes to the algorithms and the theory underlying the algorithms may be required. This is a hopeful result.

8.2 Future Work

Returning to the existing synergies and potential synergies for each of the broad classes of model checking methods identified in Chapter 5, the key conclusion to be drawn is that there are a range of important areas which would benefit from further investigation. We review here a few of the significant candidates.

In the case of partial order-based techniques, we feel that the following represent promising areas of investigation:

- developing an algorithm for partial order reduction for the modal operators $SOME$ and ALL, based on the technique of generating equivalent state spaces based on sleep sets
- further exploration of techniques based on the modified system approach to partial order reduction in the case of temporal properties, which have yet to be explored

- an exploration of approaches based upon the method of unfoldings, which could propose a theory for the detection of general temporal dynamic properties based on join-irreducible elements of the lattice of global states

The area of distribution is also a highly promising area of investigation, for several reasons, in addition to the fundamental reasons of mitigating the state explosion problem and introducing the possibility of speedup:

- being based on distribution of the reachability problem, the approach is general enough to permit the detection of dynamic properties which can be expressed as regular languages
- approach would be fully compatible with the modal operators Pos, Def, $SOME$ and ALL, as the conditions for satisfaction of these modal operators are expressed solely in terms of reachable acceptance states of the product
- there is a good match between the natural breadth-first evolution of events in a distributed computation with an easily parallelizable breadth-first search algorithm for conducting distributed reachability analysis

Bibliography

[1] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In P. Varaiya and H. Kurzhanski, editors, *Discrete Event Systems: Models and Applications*, volume 103 of *LNCIS*, pages 40–56. Springer-Verlag, August 1987.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[3] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[4] R. Alur, R.K.Brayton, T.A.Henzinger, S.Quadeer, and S. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification (CAV)*, Lecture Notes In Computer Science, pages 340–351. Springer-Verlag, 1997.

[5] Ozalp Babaoglu, Eddy Fromentin, and Michel Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *The Journal of Systems and Software*, 33(3):287–298, 1996.

[6] Özalp Babaoglu and Michel Raynal. Specification and verification of dynamic properties in distributed computations. Technical Report UBLCS-93-11, Department of Computer Science, University of Bologna, Laboratory for Computer Science, University of Bologna, Piazza di Porta S. Donato, 5, Bologna, Italy, 1993. Revised May 1994.

[7] J. Barnat, L. Brim, and I. Ćerná. Property driven distribution of nested DFS. In *3rd International Workshop on Verification and Computational Logic (VCL'02)*, pages 1–10. University of Southhampton, UK, 2002.

[8] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search ltl model checking. In *18th Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, 2003.

[9] J. Barnat, L. Brim, and J. Stríbrná. Distributed LTL model checking in SPIN. In M.B. Dwyer, editor, *8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 217–234. Springer-Verlag, 2001.

[10] L. Bougé. Repeated snapshots in distributed systems with synchronous communication. *Theoretical Computer Science*, 49:145–169, 1987.

[11] L. Brim and J. Barnat. Distribution of explicit-state LTL model checking. In T. Arts and W. Fokking, editors, *8th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2003.

[12] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. In R. Hariharan, M. Mukund, and V. Vinay, editors, *FST TCS 2001*, volume 2245 of *Lecture Notes in Computer Science*, pages 96–107. Springer-Verlag. 2001.

[13] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transcations on Computers*, C-35(8):677–691, August 1986.

[14] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{120} states and beyond. *Information and Computation*, 98:142–170, 1992.

[15] K. L. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):143–156, May 1983.

[16] K. Mani Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[17] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and temporal predicates in distributed systems. *ACM Transactions on Programming Languages and Systems*, 17(1):157–179, Jan. 1995.

[18] C. Chase and V.K. Garg. Detection of global predicates techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[19] E. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *10th Annual ACM Symposium on Principles of Programming Languages*, January 1983.

[21] Edmund M. Clarke and Jeanette M. Wing et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[22] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, volume 131 of *Lecture Notes In Computer Science*, pages 52–71. Springer-Verlag, 1981.

[23] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[24] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):167–174, 1991.

[25] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[26] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[27] Flaviu Cristian. Exception handling and software fault-tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, 1982.

[28] Claire Diehl, Claude Jard, and Jean-Xavier Rampon. Reachability analysis on distributed executions. Technical Report 1720, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France, July 1992.

[29] Laura K. Dillon and Qing Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 140–153, 1994.

[30] Omar Drissi-Kaitouni and Claude Jard. Compiling temporal logic specifications into observers. Technical Report 881, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes, France, July 1988.

[31] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *ACM SIGMOD Record*, 15(3), September 1986.

[32] Joost Engelfriet. Branching processes of petri nets. *Acta Informatica*, 28:575–591, 1991.

[33] J. Esparza and K. Heljanko. A new unfolding approach to LTL model checking. Technical Report HUT-TCS-A60, Helsinki University of Technology, 2000.

[34] Javier Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994.

[35] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer-Verlag, 1998.

[36] C. J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, Australia, 1988. University of Queensland.

[37] Vijay K. Garg. Observation of global properties in distributed systems. In *SEKE*, pages 418–425, 1996.

[38] Vijay K. Garg and Craig Chase. Distributed algorithms for detecting conjunctive predicates. In *IEEE International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, June 1995.

[39] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *IEEE International Conference on Distributed Computing Systems*, pages 322–329, Phoenix, May 2000.

[40] Vijay K. Garg and Brian Waldecker. Detection of unstable predicates in distributed programs. In *12th International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes In Computer Science*, pages 253–264, New Dehli, India, December 1992. Springer-Verlag.

[41] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299-307, March 1994.

[42] Vijay K. Garg and Brian Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.

[43] V.K. Garg, C. Chase, R. Kilgore, and J.R. Mitchell. Detecting conjunctive channel predicates in a distributed programming environment. In *International Conference on System Sciences*, volume 2, pages 232–241, January 1995.

[44] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[45] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes In Computer Science*, pages 176–185, Rutgers, June 1990. Springer-Verlag. Extended version in ACM/AMS DIMACS Series, Volume 3, pages 321-340, 1991.

[46] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., New York, NY, USA, 1996.

[47] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *Logic in Computer Science*, pages 406–415, 1991.

[48] Patrice Godefroid and Pierre Wolper. Using partial orders for efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2:149–164, 1993.

[49] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics in Models of Concurrent Programs*. NATO ASI, pages 477–498. Springer-Verlag, 1985.

[50] J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, Sept. 1999.

[51] J. Hatcliff, M.B. Dwyer, and S. Laubach. Staging static analysis using abstraction-based program specialization. In *Principles of Declarative Programming 10th International Symposium*, volume 1940 of *LNCS*, September 1998.

[52] Jean-Michel Helary. Observing global states of asynchronous distributed applications. In *3rd International Workshop on Distributed Algorithms*, volume 392 of *Lecture Notes in Computer Science*, pages 124–135, 1989.

[53] Jean-Michel Helary, Claude Jard, Noel Plouzeau, and Michel Raynal. Detection of stable properties in distributed applications. In *6th ACM SIGACT-SIGOPS*, Symposium on Principles of Distrubued Computing, pages 125–136, Vancouver, Canada, August 1987.

[54] K. Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. In *TACAS'99*, number 1579 in LNCS, pages 240–254. Springer-Verlag, 1999.

[55] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[56] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.

[57] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2nd edition, 2001.

[58] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunction of local predicates. Technical Report 2731, IRISA, Rennes, France, Nov. 1995.

[59] Claude Jard and T. Jeron. On-line model checking for finite linear temporal logic specifications. In *Proc. Intl. Workshop on Automatic Verification Methods for Finite State Systems*, pages 189–196, 1989.

[60] Claude Jard, Thierry Jeron, Guy-Vincent Jourdan, and Jean-Xavier Rampon. A general approach to trace-checking in distributed computing systems. Technical report, IRISA, Campus Universitaire de Beaulieu, Rennes, France, March 1994.

[61] Claude Jard and Guy-Vincent Jourdan. Dependency tracking and filtering in distributed computations. Technical Report PI-851, IRISA, Campus Universitaire de Beaulieu, Rennes, France, 1994.

[62] J.Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In *TCAS'96*, volume 1055 of *LNCS*, pages 87–106. Springer-Verlag, 1996.

[63] K.L.McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of the 4th Workshop on Computer Aided Verification*, pages 164–174, Montreal, 1992.

[64] M.Z. Kwiatkowska. Fairness for non-interleaving concurrency. Ph.d thesis, Faculty of Science, University of Leicester, 1989.

[65] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[66] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[67] F. Lerda and R. Sisto. Distributed memory model checking with SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massinik, editors, *6th SPIN Workshop on Model Checking of Software (SPIN99)*, volume 1680 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[68] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[69] A. Lluch-lafuente, S. Leue, and S. Edelkamp. Partial order reduction in directed model checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking Software*, pages 164–174, Grenoble, April 2002. Springer LNCS.

[70] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive Systems: Specification*. Springer-Verlag, Berlin, 1991.

[71] Keith Marzullo and Gil Neiger. Detection of global state predicates. In *Procedings of the Fifth International Workshop on Distributed Algorithms (WDAG-91)*, Lecture Notes In Computer Science, Delphi, Greece, October 1991. Springer–Verlag.

[72] Keith Marzullo and Laura Sabel. Using consistent subcuts for detecting stable properties. In *Procedings of the Fifth International Workshop on Distributed Algorithms (WDAG-91)*, Lecture Notes In Computer Science, Delphi, Greece, October 1991. Springer–Verlag.

[73] F. Mattern. Time and global states of distributed systems. In M. Cosnard et al., editor, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[74] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989. Reprinted in Global States and Time In Distributed Systems, IEEE, 1994.

[75] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 1993.

[76] A. Mazurkiewicz. Trace semantics. In *Advances in Petri Nets 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag, 1987.

[77] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[78] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[79] S. Melzer and S. Römer. Deadlock checking using net unfoldings. In *CAV'97*, number 1254 in LNCS, pages 352–363. Springer-Verlag, 1997.

[80] B. P. Miller and J. D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 316–325, Washington, DC, 1988. IEEE Computer Society.

[81] N. Mittal and V.K. Garg. Computation slicing: Techniques and theory. In *Symposium on Distributed Computing(DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[82] Sape J. Mullender, editor. *Distributed Systems*, chapter 5. ACM Press, second edition, 1993.

[83] Sape J. Mullender, editor. *Distributed Systems*, chapter 4. ACM Press, second edition, 1993.

[84] Ratan Nalumasu and Ganesh Gopalakrishnan. An new partial order reduction algorithm for concurrent system verification. In *Computer Hardware Description Languages (CHDL)*, pages 305–314. Chapman Hall, April 1997.

[85] Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part i. *Theoretical Computer Science*, 13(1):85–108, 1981.

[86] Doron Peled. All from one and one for all: Model checking using representatives. In *5th International Conference on Computer Aided Verification*, Lecture Notes In Computer Science, Greece, 1993. Springer-Verlag.

[87] Doron Peled. Combining partial order reductions with on-the-fly model checking. In *6th International Conference on Computer Aided Verification*, Lecture Notes In Computer Science, California, 1994. Springer.

[88] Doron Peled and Amir Pnueli. Proving partial order liveness properties. In *17th ICALP*, volume 443 of *Lecture Notes in Computer Science*, pages 553–571. Springer-Verlag, 1990.

[89] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[90] V.R. Pratt. Modeling concurrency with partial orders. *Int. J. of Parallel Programming*, 15(1):33–71, February 1986.

[91] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in cesar. In *5th International Workshop on Programming*, volume 137 of *Lecture Notes In Computer Science*, pages 337–351. Springer-Verlag, 1981.

[92] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report 595, University of Newcastle upon Tyne, 1997.

[93] M. Raynal and M. Singhal. Capturing causality in distributed systems. *IEEE Computer*, pages 49–56, February 1996.

[94] W. Reisig. Temporal logic and causality in concurrent systems. *Lecture Notes in Computer Science; Concurrency 88*, 335:121–139, 1988. NewsletterInfo: 31.

[95] D.J. Richardson, S.L. Aha, and T.O. O'Malley. Specification-based test oracles for reactive systems. In *14th International Conference on Software Engineering*, pages 105–118, Melbourne, Austrlia, 1992.

[96] C. Schröter and J. Esparza. Reachability analysis using net unfoldings. In H.D. Burkhard, L. Czaja, A. Skowron, and P. Starke, editors, *Workshop of Concurrency, Specification & Programming*, volume II of *Informatik-Bericht 140*, pages 255–270. Humboldt-Universität zu Berlin, 2000.

[97] A. Sen and V.K. Garg. Detecting temporal logic predicates on the happened-before model. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, Florida, 2002.

[98] A. Sen and V.K. Garg. On checking whether a predicate definitely holds. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, Montreal, Quebec, Canada, 2003.

[99] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *7th International Conference on Principles of Distributed Systems*, La Martinique, France, December 2003.

[100] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217-237, 1987.

[101] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the ACM*, 32:733–749, 1985.

[102] U. Stern and D. L. Dill. Paralleizing the murphi verifier. In *9th International Conference on Computer Aided Verification (CAV'97)*, 1997.

[103] Scott Stoller and Yanhong A. Liu. Efficient symbolic detection of global properties in distributed systems. In *10th International Confenerence on Computer Aided Verivication (CAV)*, volume 1447 of *LNCS*, pages 357–368, 1998.

[104] Scott Stoller, Leena Unnikrishnan, and Yanhong A. Liu. Efficient detection of global properties in distributed systems using partial order methods. In *12th International Confenerence on Computer Aided Verivication (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279, 2000.

[105] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, Florida, 1998.

[106] A. Tarafdar and V. K. Garg. Software fault-tolerance of concurrent programs using controlled re-execution. In *13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[107] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, pages 146–160, January 1972.

[108] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[109] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. In *Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, California, May 1993. ACM–ONR.

[110] Wilfredo Torres-Pomales. Software fault tolerance: A tutorial. Technical Report TM-2000-210616, NASA, Langley Research Center, Hampton, Virginia, October 2000.

[111] Antti Valmari. Error detection by reduced reachability graph generation. In *9th European Workshop on Application and Theory of Petri Nets*, pages 95–112, 1988.

[112] Antti Valmari. A stubborn attack on state explosion. In *2nd International Conference on Computer Aided Verification*, volume 531 of *Lecture Notes In Computer Science*, pages 25–41. Springer-Verlag, 1990.

[113] Antti Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes In Computer Science*. Springer-Verlag, 1991.

[114] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic In Computer Science*, pages 322–331, June 1986.

[115] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–176, February 1995.

[116] F. Wallner. Model checking ltl using net unfoldings. In *10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *LNCS*, pages 207–218, 1998.

[117] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[118] Mark D. Wood. Fault-tolerant management of distributed applications using the reactive system architecture. Technical Report TR91-1252, Department of Computing Science, Cornell University, 1991.

Appendix A

Exploration-independent Proofs

A.1 Exploration-dependent provisos for avoiding ignoring

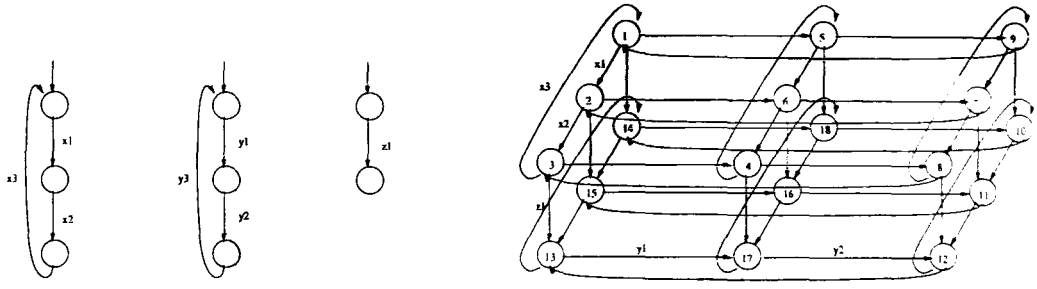
In this section, we informally review the provisos which have been developed in the context of partial order reduction based on depth-first search, both in the case of stuttering-equivalence and finite stuttering-equivalence. Depth-first search was the exploration method of choice in the early incarnations of partial order reduction theory, as it permits efficient detection of cycles and exhibits certain convenient properties with respect to inductive proofs. Our aim is to gain some insight into the form of exploration-independent condition required, by examining differences in the resulting reduced state spaces.

In what follows, we use the following example concurrent program to illustrate the variations in the reduced state state spaces generated. The program is composed of the three processes, illustrated in Figure A.1(a). Transitions from each process are independent of transitions on the other processes. The transitions x_i are interdependent, as are the transitions y_i. As we are focusing on condition **C3**, we assume that all transitions in the program are invisible. The full state space is shown in Figure A.1(b). For convenience, states in the state space are labeled by the order in which they were visited during a depth-first exploration generation of the state space.

A.1.1 Avoiding the ignoring problem: stuttering equivalence

Consider the case of constructing a reduced state space which contains one representative per infinite execution (stuttering equivalence) of the full state space. If any direction from a state s leads to a previously visited state on the execution currently being explored (i.e. on the search stack), that direction will create a cycle in reduced state space graph. Any path leading from an initial state to that cycle, together with the cycle itself, represents an infinite execution in the reduced state space. If the state s is not fully expanded (i.e. $\mathit{enabled}(s) \setminus \mathit{ample}(s)$ is not empty), then, *potentially*, transitions not selected from s can be ignored on that infinite execution. Therefore, when generating reduced state spaces which must contain representatives of infinite runs, *any* ample direction which creates a cycle can potentially result in ignoring.

In [86, 87], Peled provided a proviso for depth-first search which guaranteed generating a reduced state space containing one representative for every infinite execution in the full state



(a) Three concurrent processes

(b) The full state space

Figure A.1: Three concurrent processes and the full state space

space. In this work, a slightly different form of equivalence (based on the notion of *equivalence robustness* of a checked property) was required between the full and reduced state space. However, the treatment of ignoring is largely unaffected. The proviso was stated as follows:

C3-dfs If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is on the search stack

In this work, proof of correctness was based on induction on the order in which nodes backtrack during the depth-first search exploration. The proviso **C3-dfs** is used to ensure that the exploration from a state s can always reach a state where the induction hypothesis holds.

Figure A.2 shows one possible reduced state space for the concurrent program of Figure A.1(a), constructed by a modified depth-first search satisfying proviso **C3-dfs**. Dotted lines indicate transitions of the full state space which are not included in the reduced state space. Note that none of the cycles present in the graph are ignoring.

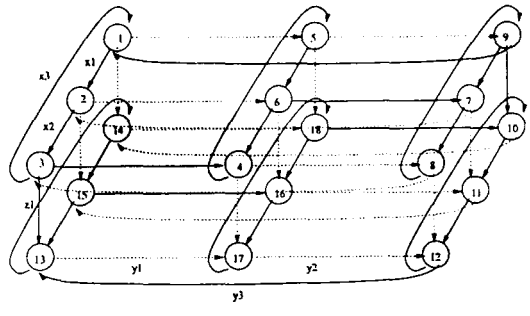


Figure A.2: The reduced state space with proviso C3-dfs

A.1.2 Avoiding the ignoring problem: finite stuttering equivalence

Consider now the case of constructing a reduced state space which contains one representative per finite trace (finite stuttering-equivalence). In such a scenario, ignoring can arise when all directions from s lead to cycles, and the state s is not fully expanded. Informally, each direction leaving s represents a possible continuation of a path reaching s. Therefore, if all directions lead to ignoring cycles, this means that, ***potentially***, from s, there is no path from s which fires one of the transitions in $enabled(s) \setminus ample(s)$.

In his PhD thesis [46], Godefroid presented a theory of partial order reduction which focused on generating reduced state spaces which contained one representative for every finite trace (starting from the initial state) in the full state space, which is sufficient for checking safety properties. In his thesis, Godefroid considered the use of partial order reduction in a slightly different context, defining the notion of a *trace automaton*, which did not explicitly involve the notion of (finite) stuttering-equivalence of state transition systems. However, the problem of ignoring and its treatment in the case of reduced state spaces where only representatives for finite sequences are required is largely unaffected. In that work, the proviso used to avoid ignoring was stated as follows:

C3-dfs' If s is not fully expanded, then at least one transition in $ample(s)$ must not reach a state that is on the search stack

This proviso requires that *at least one* direction from s must not create a cycle in the reduced state space. As in the work cited in the previous section, proof of correctness was based on induction on the order in which nodes backtrack during the depth-first search exploration. The proviso **C3-dfs'** is used to ensure that the exploration can always reach a state where the induction hypothesis holds.

Figure A.3 shows a reduced state space constructed by a modified depth-first search satisfying the proviso **C3-dfs'**. Ample sets have been chosen in such a way as to be consistent with the choices made in the previous example, to aid comparison of the state spaces.

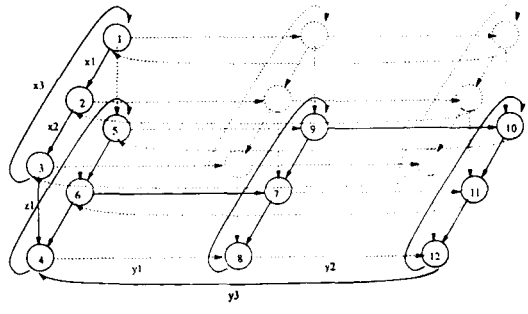


Figure A.3: The reduced state space with proviso C3-dfs'

A.1.3 Comparison of the reduced state spaces

It is instructive to compare the reduced state spaces associated with the two provisos.

First, note that the reduced state space satisfying **C3-dfs'** in Figure A.3 is a subgraph of the reduced state space satisfying **C3-dfs** in Figure A.2. This is partly a consequence of the fact that **C3-dfs** is a strictly stronger condition on ample sets than **C3-dfs'**, and partly a consequence of the fact that choices are ample sets were made consistently in each.

Note also that ignoring cycles *do* exist in the reduced state space generated using **C3-dfs'**. For example, the cycle represented by the states $\{1, 2, 3\}$ ignores transition $y1$. This illustrates well how what constitutes ignoring depends upon the equivalence required between the full and reduced state spaces. Under finite stuttering equivalence, such cycles no longer represent infinite executions which can lead to incorrect representatives which ignore transitions.

Although ignoring cycles do exist in the reduced state space generated using **C3-dfs'**, each ignoring cycle has a state s such that some direction in $ample(s)$ leads to a state outside the cycle. For example, in Figure A.3, the cycle represented by the states $\{1, 2, 3\}$, which ignores $y1$, contains the state 3, for which $ample(3)$ contains the direction $z1$, with successor state 4 lying outside the cycle. By contrast, it is easy to check that the much larger cycle, represented by the states $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, has no such direction, but it is not ignoring.

This leads us to surmise that, in the reduced state space satisfying proviso **C3-dfs'**, ignoring cycles are permitted as long as we can 'escape' from each ignoring cycle by some ample direction which leads to a state not on the cycle. We shall refer to such a direction as a *progress direction*, as they allows us to make progress in the exploration of the state space.

A.2 An exploration-independent condition for finite stuttering equivalence

In this section, we develop and prove the correctness of an exploration-independent condition for avoiding ignoring in the case of finite stuttering equivalence.

A.2.1 Motivating the condition

Based on the above observations made in Section A.1.3, we might propose the following condition for avoiding ignoring in the case of finite stuttering equivalence:

C3-fin(provisional) a cycle is not allowed in the reduced state graph if it contains a state in which some transition α is enabled, but never included in $ample(s)$ for any state s in the cycle, and there is no state s' in the cycle for which $ample(s')$ contains a direction leading to a state not in the cycle

This condition could be described informally by saying that every ignoring cycle within the reduced state space must have an ample progress direction leading to a state outside the cycle. It allows ignoring cycles to exist in the reduced state space, but we must have a means of escaping them to continue the exploration. Notice that the condition is satisfied for the reduced state space of Figure A.3. As condition **C3**, this condition applies to cycles appearing in the reduced state graph.

The following example in Figure A.4 shows that such a cycle-based condition is not adequate.

We suppose that the transitions are as in the previous example, but now the transitions $x1, x2, x3$ and $y1, y2, y3$ are adjusted so that they execute in a mutually exclusive manner. The resulting reduced state graph is shown, where states are labeled by the order in which they are visited, and dotted circles and lines represent states and transitions in the full state space which are not explored in the reduced state space. The cycle $\{1, 2, 3\}$ satisfies the proviso with progress direction $y1$; similarly, the cycle $\{1, 4, 5\}$ satisfies the proviso with progress direction $x1$. However, the reduced state graph is incorrect as it ignores transition $z1$. Note however, that the strongly connected subgraph $\{1, 2, 3, 4, 5\}$, made up by joining together the two simple cycles, is ignoring and does not have a progress direction leading to a state outside the strongly connected subgraph.

We therefore adjust the definition of the new condition to place a restriction on ignoring strongly connected subgraphs, as opposed to ignoring cycles. This condition is more general as

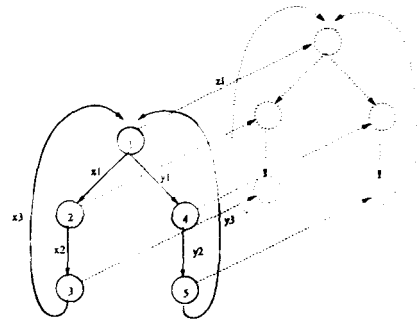


Figure A.4: A reduced state space violating the cycle condition

every ignoring cycle is also an ignoring strongly connected subgraph. The new cycle condition allows ignoring strongly connected subgraphs within the reduced state space graph, as long as there is a progress direction:

C3-fin a strongly connected subgraph is not allowed in the reduced state graph if it contains a state in which some transition α is enabled, but never included in $ample(s)$ for any state s in the subgraph, and there is no state s' in the subgraph for which $ample(s')$ contains a direction leading to a state not in the subgraph

Returning to Figure A.3, note that the strongly connected subgraphs are $\{1,2,3\}$,$\{4,5,6\}$, $\{7,8,9\}$, $\{10,11,12\}$ and $\{4,5,6,7,8,9,10,11,12\}$ and that the condition **C3-fin** is satisfied for them. Note that the last strongly connected subgraph is maximal (i.e. a strongly connected component).

Another interpretation of this condition arises by considering which strongly connected subgraphs are not allowed in the reduced state space. It is easy to show that if G' is a strongly connected subgraph of a directed graph G which has no edge leading to a vertex outside G', then G' must be a strongly connected component (i.e. maximal) and further, a sink strongly connected component (i.e. no outgoing edges). Thus, the condition **C3-fin** can be interpreted as a requirement that the reduced state space contain no ignoring sink strongly connected components.

A.2.2 Correctness

In this section, we need to show that the condition **C3-fin** is sufficient to guarantee that the reduced state space M' generated by a state space search which explores only an ample set of directions at each state reached will be finitely stuttering-equivalent to the full state space M.

In order to prove finite stuttering equivalence, we follow *closely* the proof of stuttering equivalence presented in [19, Chapter 10, Section 6]. In order to avoid being tied to a particular state space exploration algorithm, the proof is organized around an exploration-independent construction, which describes how a representative for a path σ is constructed through a sequence of construction steps. In the next section, we describe this exploration-independent construction.

In what follows, we assume that ample sets are constructed in such a way that they satisfy conditions **C0**, **C1**, **C2** and **C3-fin**.

A.2.2.1 Definition of construction

Let M be the full state graph of some system, and M' be a reduced state graph constructed using the ample set algorithm.

In the proof that follows, we shall make use of the notation introduced in Chapter 7 concerning labeled state transition systems, stuttering equivalence of paths, and finite stuttering-equivalence of labeled state transition systems. We introduce some additional notation required for the construction, taken from [19, Chapter 10, Section 6]. A *string* is a sequence of transitions from T. Let T^* be the set of all strings over T. Given a finite string $v \in T$, denote by $vis(v)$ the projection of v onto the visible transitions. Given a finite path $\sigma = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots s_{|\sigma|-1} \xrightarrow{\alpha_{|\sigma|}} s_{|\sigma|}$ through M, let $tr(\sigma)$ be the sequence of transitions $\alpha_1\alpha_2\ldots\alpha_{|\sigma|}$ on the path. Let v, w be two finite strings. The relation $v \sqsubset w$ holds if v can be obtained from w by erasing one or more transitions. The relation $v \sqsubseteq w$ holds if $v \sqsubset w$ or $v = w$.

Let $\sigma \circ \eta$ denote the concatenation of the finite paths σ and η of M, where the last state $last(\sigma)$ of σ is equal to the first state $first(\eta)$ of η. The length of a path σ is equal to the number of edges of σ and is denoted by $|\sigma|$.

Let σ be some *finite* path in the full state graph M, starting with some initial state. We will construct a (possibly infinite) sequence of paths $\pi_0, \pi_1, \ldots$, where $\pi_0 = \sigma$. Each path π_i will be decomposed into a concatenation of two paths, $\eta_i \circ \theta_i$, where η_i is of length i. Assuming that we have constructed the paths $\pi_0, \ldots, \pi_i$, we describe how to construct $\pi_{i+1} = \eta_{i+1} \circ \theta_{i+1}$. Let $s_0 = last(\eta_i) = first(\theta_i)$ and α the transition labeling the first edge of θ_i. Denote

$$\theta_i = s_0 \xrightarrow{\alpha_0=\alpha} s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$$

There are two cases:

A. $\alpha \in ample(s_0)$. Then select $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\alpha} \alpha(s_0))$. θ_{i+1} is $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots$; that is, θ_i without its first edge.

B. $\alpha \notin ample(s_0)$. By **C2**, all of the transitions in $ample(s_0)$ must be invisible, since s_0 is not fully expanded. Here again, there are two cases, **B1** and **B2**:

B1. Some $\beta \in ample(s_0)$ appears on θ_i after some sequence of transitions $\alpha_0\alpha_1\alpha_2\ldots\alpha_{k-1}$ independent with β; that is, $\beta = \alpha_k$. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-1}} \beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2} \xrightarrow{\alpha_{k+2}} \ldots$ in M. That is, β is moved to appear before $\alpha_0\alpha_1\alpha_2\ldots\alpha_{k-1}$. Note that $\beta(s_k) = s_{k+1}$. Therefore, $\beta(s_k) \xrightarrow{\alpha_{k+1}} s_{k+2}$.

B2. Some $\beta \in ample(s_0)$ is independent of all the transitions that appear on θ_i. Then there is a path $\xi = s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0=\alpha} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$ in M. That is, β is executed from s_0 and then applied to each state of θ_i.

In both cases, $\eta_{i+1} = \eta_i \circ (s_0 \xrightarrow{\beta} \beta(s_0))$ and θ_{i+1} is the path that is obtained from ξ by removing the first transition $s_0 \xrightarrow{\beta} \beta(s_0)$.

As the path σ is finite, the construction sequence may construct a path π_i such that θ_i has length zero. In such a case, the construction process will stop.

Let η be the path such that the prefix of η of length i is defined by η_i. We refer to η as a *representative* of the path σ. The path η is well defined in that η_i is constructed from η_{i-1} by

appending a single transition. In the sequel, for a given σ, we represent distinct representatives by η, η', η^l and their associated construction sequences of paths by $\{\pi_i\}_{i\geq 0}$, $\{\pi'_i\}_{i\geq 0}$, $\{\pi^l_i\}_{i\geq 0}$.

A.2.2.2 Key lemmas

The proof consists of a number of key lemmas which establish properties of the constructed representative in order to prove the main theorem (Theorem A.1) stating the finite stuttering equivalence between M and M'. Lemmas which hold true for both the case of finite stuttering equivalence and stuttering equivalence are reproduced here, without change. Their original names from [19, Chapter 10, Section 6] are quoted in brackets.

Lemma A.1. (Lemma 26, [19]) The following hold for all i, j such that $i \geq j \geq 0$:

1. $\pi_i \sim_{st} \pi_j$
2. $vis(tr(\pi_i)) = vis(tr(\pi_j))$
3. Let ξ_i be a prefix of π_i and ξ_j be a prefix of π_j such that $vis(tr(\xi_i)) = vis(tr(\xi_j))$. Then $L(last(\xi_i)) = L(last(\xi_j))$

Proof. It is sufficient to consider the case where $j = i+1$. Consider the three ways of constructing π_{i+1} from π_i. In case **A**, $\pi_i = \pi_{i+1}$, and all three parts of the lemma hold trivially.

Next, consider case **B1** of the construction, in which π_{i+1} is obtained from π_i by executing some invisible transition β in π_{i+1} earlier than it is executed in π_i. In this case, we replace the sequence $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} s_{k-1} \xrightarrow{\beta} s_k$ by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{k-2}} \beta(s_{k-1})$. Because β is invisible, corresponding states have the same label, that is, for each $0 < l \leq k$, $L(s_l) = L(\beta(s_l))$. Also, the order of visible transitions remains unchanged. Parts 1, 2 and 3 follow immediately.

Finally, in case **B2** of the construction, the difference between π_i and π_{i+1} is that π_{i+1} includes an additional invisible transition β. Thus, we replace some suffix $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of π_i by $s_0 \xrightarrow{\beta} \beta(s_0) \xrightarrow{\alpha_0} \beta(s_1) \xrightarrow{\alpha_1} \beta(s_2) \xrightarrow{\alpha_2} \ldots$. So, $L(s_l) = L(\beta(s_l))$ for $l \geq 0$. Again, the order of visible transitions remains unchanged. As in the previous case, parts 1, 2 and 3 follow immediately. □

Lemma A.2. (Lemma 27, [19]) Let η be the path constructed as the limit of the finite paths η_i. Then η belongs to the reduced state graph M'.

Proof. By induction on the length of the prefixes η_i of η. The base case is that η_0 is a single node, which is an initial state in S. According to the reduction algorithms, all the initial states are included in S' as well. For the inductive step, assume that η_i is in M'. Then notice that η_{i+1} is obtained from η_i by appending a transition from $ample(last(\eta_i))$.

□

Until now, it has not been necessary to distinguish between the different possible representatives η which may result from the construction. For any single path σ, distinct representatives may be created when there are two or more choices available for case **B**. We shall now require to differentiate between the various possible representatives for a given sequence σ.

In the construction of a representative η using ample sets satisfying **C0**, **C1**, **C2** and **C3**, all representatives constructed satisfy the following property:

CMPLT for all $i \geq 0$, if α denotes the first transition of θ_i, then there exists $j > i$ such that α is the last transition of η_j and, for $i \leq k < j$, α is the first transition of θ_k

This *completeness* property was proved in Lemma 28 [19, Chapter 10, Section 6] and ensures that every representative minimally fires all transitions of the path σ. However, when **C3-fin** replaces **C3** in the definition of ample sets, not all representatives generated by the construction have this property. We therefore need to differentiate between representatives for a given sequence σ.

Let $reps(\sigma)$ be the set of all representatives for σ produced by the construction. Let $completereps(\sigma)$ be the subset of $reps(\sigma)$ whose associated sequences satisfy the property **CMPLT**.

In order to prove finite stuttering equivalence, we need to show that, for any given finite sequence σ, there is some constructed representative η in $reps(\sigma)$ which satisfies the completeness property for σ. We prove this fact using the following two lemmas.

Lemma A.3. Let $\eta \in reps(\sigma)$ be a representative constructed, with associated construction sequence $\{\pi_i\}_{i \geq 0}$, and suppose that, for some $i \geq 0$, α remains the first transition of θ_k, for all $k \geq i$. Then there exists $\eta' \in reps(\sigma)$ with associated construction sequence $\{\pi'_i\}_{i \geq 0}$ and $j > i$ such that

1. for all $0 \leq k \leq i$, $\eta_k = \eta'_k$,
2. α is the last transition of η'_j and
3. for $i \leq k < j$, α is the first transition of θ'_k

Proof. The proof proceeds by contradiction. We assume that the result of the lemma does not hold. This implies, in particular, that for each $\eta' \in reps(\sigma)$ which does match η up to i, there does not exists a $j > i$ such that conditions 2 and 3 hold. We wish to derive a contradiction to show that this assumption cannot hold simultaneously with condition **C3-fin**. Condition **C3-fin** concerns the ignoring strongly connected subgraphs in the reduced state space. Therefore, we aim to use this negative assumption to produce an example of an ignoring strongly connected subgraph which contradicts **C3-fin**.

To demonstrate the existence of such an ignoring strongly connected subgraph, we consider the set of all paths $\eta' \in reps(\sigma)$ which match η up to construction step i: that is, all possible continuations η' of η_i which are consistent with the construction, and the suffix θ_i. Each such path η' is a path through M', by Lemma A.2. Further, we know that α is ignored on these paths, for $k \geq i$, by assumption. We wish to show that these paths naturally form a subgraph M'' of M' on which α is ignored, and that this subgraph can be used to demonstrate the existence of an ignoring strongly connected subgraph with no progress direction.

Let M'' be the directed graph whose nodes represent the states reached by the paths η' and whose edges represent the transitions between states encountered on the paths η'. M'' is a subgraph of the directed graph M', as every path η' which is used to define M'' is also a path of M'. Furthermore, α is ignored at all states of M'': that is, α is enabled at all states of M'', and α never appears as a transition between states in M''. This follows from the fact that, from construction step i on, α is enabled and ignored on every path used to define M''.

The subgraph M'', as a directed graph in its own right, can be factored into its strongly connected components. Any strongly connected component of M'' will be a strongly connected subgraph of M', but not necessarily maximal. Any sink strongly connected component of M'' will also be a sink strongly connected component of M'.

From graph theory, we know that the strongly connected components of M'' form a directed acyclic graph and, because of this fact, at least one of the components of M'' must be a sink strongly connected component. The directed acyclic graph can be represented as a tree, and the sink strongly connected component is a leaf node of that tree.

This sink strongly connected component of M'' will also a be a sink strongly connected component of M'. Because α is ignored at all nodes of M'' and in particular, on all strongly connected components of M'', we have demonstrated the existence of a strongly connected component of M' on which α is ignored. Thus, **C3-fin** does not hold for the reduced state space M'.

□

The next lemma shows that the reduction algorithm always produces at least one representative which consumes all transitions of a finite path σ of the full state space.

Lemma A.4. The set $completereps(\sigma)$ is not empty.

Proof. Select a representative $\eta \in reps(\sigma)$ with associated construction $\{\pi_i\}_{i\geq 0}$. If this construction satisfies the completeness property **CMPLT**, then we are done. Otherwise, there exists an integer $i_1 \geq 0$ and a transition α_{i_1} such that for all $k \geq i_1$, α_{i_1} is the first transition of θ_k. By the previous lemma, we can find another representative $\eta' \in reps(\sigma)$ and an $j_1 > i_1$ such that

1. for all $0 \leq k \leq i_1$, $\eta_k = \eta'_k$,
2. α is the last transition of η'_{j_1} and
3. for $i_1 \leq k < j_1$, α is the first transition of θ'_k

If this representative η' satisfies the completeness property, then we are done. Otherwise there exists an integer $i_2 \geq j_1$ and a transition α_{i_2} such that, for all $k \geq i_2$, α_{i_2} is the first transition of θ_k. By the same argument, we can find another representative η'' which has the same prefix as η' up to i_2 and fires α_{i_2}.

Since the length of σ is finite, this process need only be repeated finitely many times, and thus there is a representative in $reps(\sigma)$ which satisfies the completeness property for σ.

□

Thus, we have shown that at least one representative in $reps(\sigma)$ satisfies the completeness property. Note that representatives in $completereps(\sigma)$ are finite in length.

The next lemma holds for all representatives in $reps(\sigma)$.

Lemma A.5. (Lemma 29, [19]) Let γ be the first visible transition on θ_i and $prefix_\gamma(\theta_i)$ be the maximal prefix of $tr(\theta_i)$ that does not contain γ. Then one of the following holds:

- γ is the first transition of θ_i and the last transition of η_{i+1}, or
- γ is the first visible transition of θ_{i+1}, the last transition of η_{i+1} is invisible, and $prefix_\gamma(\theta_{i+1}) \sqsubseteq prefix_\gamma(\theta_i)$

Proof. The first case of the lemma holds when γ is selected from $ample(s_i)$ and becomes the last transition of η_{i+1}, according to case **A** of the construction. If this does not happen, there exists another transition β that is appended to η_i to form η_{i+1}. The transition β cannot be visible. Otherwise, according to **C2**, $ample(s_i) = enabled(s_i)$. By case **B1** of the construction, β must be the first transition of θ_i. But then β is a visible transition that precedes γ in θ_i, a contradiction.

There are three possibilities:

1. β appears on θ_i before γ (case **B1** in the construction)
2. β appears on θ_i after γ (case **B1** in the construction)
3. β is independent of all the transitions of θ_i (case **B2** in the construction)

According to the above construction, in (1), $prefix_\gamma(\theta_{i+1}) \sqsubset prefix_\gamma(\theta_i)$ since β is removed from the prefix of θ_i before γ when constructing θ_{i-1}. In (2) and (3), $prefix_\gamma(\theta_{i+1}) = prefix_\gamma(\theta_i)$ since the prefix of θ_{i-1}that precedes the transition γ has the same transition as the corresponding prefix of θ_i.

□

Lemma A.6. (Lemma 30, [19]) Let η be a representative from $completereps(\sigma)$. Let v be a prefix of $vis(tr(\sigma))$. Then there exists a prefix η_i of η such that $v = vis(tr(\eta_i))$.

Proof. By induction on the length of v. The base case holds trivially for $|v| = 0$. In the inductive step, we must prove that if $v\gamma$ is a prefix of $vis(tr(\sigma))$ and there is a path η_i such that $vis(tr(\eta_i)) = v$, then there is a path η_j with $j > i$ such that $vis(tr(\eta_j)) = v\gamma$. Thus, we need to show that γ will be eventually added to η_j for some $j > i$, and that no other visible transition will be added to η_k for $i < k < j$. According to case **A** of the construction, we may add a visible transition to the end of η_k to form η_{k+1} only if it appears as the first transition of θ_k. Lemma A.5 shows that γ remains the first visible transition in successive paths θ_k after θ_i unless it is being added to some η_i. Moreover, the sequence of transitions before γ can only shrink. Lemma A.4 shows that the first transition in each θ_k is eventually removed and added to the end of some θ_l for $l > k$. Thus, γ as well is eventually added to some sequence η_j.

□

Theorem A.1. (slight modification of Theorem 12, [19]) The structures M and M' are finitely stuttering equivalent.

Proof. Each finite path of M' that begins from an initial state must also be a path of M, for it is constructed by repeatedly applying transitions from the initial state. We need to show that for each path $\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-2}} s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ in M, where s_0 is an initial state, there exists a path $\eta = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots \xrightarrow{\beta_{m-2}} r_{m-1} \xrightarrow{\beta_{m-1}} r_m$ in M' such that $\sigma \sim_{st} \eta$. We will show that any path η in $completereps(\sigma)$ that is constructed above for σ is indeed stuttering equivalent to σ.

First, we show that σ and η have the same sequence of visible transitions: that is, $vis(tr(\sigma)) = vis(tr(\eta))$. According to Lemma A.6, η contains the visible transitions of σ in the same order, because for any prefix of σ with m visible transitions, there is a prefix η_i of η with the same m visible transitions. On the other hand, σ must contain the visible transitions of η in the same order.

Take any prefix η_i of η. According to Lemma A.1, $\pi_i = \eta_i \circ \theta_i$ has the same visible transitions as $\pi_0 = \sigma$. Thus, σ has a prefix with the same sequence of visible transitions as η_i.

We construct two finite sequences of indexes $0 = i_0 < i_1 < \ldots < i_{sb}$ and $0 = j_0 < j_1 < \ldots < j_{sb}$ that define corresponding stuttering blocks of σ and η, as required in the definition of stuttering. Assume that both $\sigma = \pi_0$ and η have sb visible transitions. Let i_n be the length of the smallest prefix σ_{i_n} of σ that contains exactly n visible transitions. Let j_n be the length of the smallest prefix η_{j_n} of η that contains the same sequence of visible transitions as σ_{i_n}. Recall that η_{j_n} is a prefix of π_{j_n}. Then by part 3 of Lemma A.1, $L(s_{i_n}) = L(r_{j_n})$. By the definition of visible transitions, we also know that if $n > 0$, for $i_{n-1} \leq k < i_n - 1$, $L(s_k) = L(s_{i_{n-1}})$. This is because i_{n-1} is the length of the smallest prefix $\sigma_{i_{n-1}}$ of σ that contains exactly $n-1$ visible transitions. Thus, there is no visible transition between i_{n-1} and $i_n - 1$. Similarly, for $j_{n-1} \leq l < j_n - 1$, $L(r_l) = L(r_{j_{n-1}})$. Thus, $\sigma \sim_{st} \eta$.

□

A.3 Relationship with exploration-dependent provisos

Although we established in the previous section the sufficiency of condition **C3-fin** as an exploration-independent condition for avoiding ignoring in the defining ample sets in the case of finite stuttering equivalence, it is possible that the condition is unnecessarily strong. In this Section, we shall demonstrate that existing exploration-dependent provisos for avoiding ignoring in the case of finite stuttering equivalence *are* sufficient conditions for ensuring condition **C3-fin** holds in the reduced state space. We consider both the cases of depth-first search and breadth-first search.

In proving the sufficiency relationship between existing provisos and condition **C3-fin**, we will need a way to relate the provisos (which are exploration-dependent) to the conditions (which are exploration-independent). We achieve this by considering the spanning forest induced by the exploration-dependent algorithm, which represents each strongly connected component of the reduced state space as a subtree in the spanning forest. We also show how this spanning forest analysis provides an effective way to develop new, computationally more efficient provisos which are sufficient for **C3-fin**. We use breadth-first search as an example, but the ideas apply equally well to other state space exploration methods.

A.3.1 Depth-first search

In this section, our aim is to show that a modified state space exploration algorithm based on depth-first search for which proviso **C3-dfs'** holds is sufficient to guarantee that the property **C3-fin** holds in the resulting reduced state graph. In order to relate the condition on cycles of **C3-dfs'** to the condition on strongly connected subgraphs of **C3-fin**, we consider the spanning forest induced during depth first-traversal.

Depth-first traversal of a graph $G = (V, E)$ can be used to create a spanning forest $S = (V, T)$, composed of an ordered sequence $T_1, T_2, \ldots, T_n$ of spanning trees which contain all the verticies of the graph G [2] and where graph edges are classified as tree edges or non-tree edges. Non-tree edges are further classified as *forward edges*, *back edges* or *cross edges*. A non-tree edge $< v, w >$

is: a forward edge if w is a non-son descendant of v ; a back edge if w is an ancestor of v : a cross edge if w is neither an ancestor nor a descendant of v . The strongly connected components $G_i = (V_i, E_i)$ of the graph G appear as subtrees $ST_{G_i} = (V_i, E_i \cap T)$ of the spanning forest, where all the verticies V_i are represented in each subtree ST_{G_i}, but only the edges in $E_i \cap T$ are represented as tree edges (the remaining edges in E_i are represented as non-tree edges). In this way, a depth-first traversal of a graph will visit all verticies of each strongly connected component, and will also provide a classification of the edges leaving each vertex. With this classification of edges in hand, we can now attempt to relate the two conditions.

Firstly, within the context of a depth-first search exploration of the state space, a direction creates a cycle (leads to a state on the stack, or in other words an ancestor in the current interleaving) if and only if it is a back edge. Condition **C3-dfs'** can therefore equivalently be seen as a condition on back edges: no vertex reached during the search, which corresponds to a state s in which $ample(s)$ is not fully expanded, may have outgoing edges which consist only of back edges.

Secondly, with respect to condition **C3-fin**, it can be shown that a necessary condition for a strongly connected subgraph not to have a progress direction is that it must contain a vertex whose outgoing edges consist solely of back edges. We shall need this result in order to prove sufficiency. However, rather than prove the result for arbitrary strongly connected subgraphs, we first prove a similar result for strongly connected components in the following lemma.

Lemma A.7. Let $G = (V, E)$ be a graph, with strongly connected components $G_i = (V_i, E_i)$ and let $S = (V, T)$ be the spanning forest generated in a depth-first traversal of the graph. Let $ST_{G_i} = (V_i, E_i \cap T)$ be the subtree representing G_i in the spanning forest. For each $G_i = (V_i, E_i)$, if G_i does not have an edge leaving G_i in G, then there is a vertex $v \in V_i$ such that the only outgoing edges of v in ST_{G_i} are back edges.

Proof. Assume that G_i has no edge leaving G_i in G. Consider the tree ST_{G_i} in S corresponding to G_i. We want to show that there must exist a vertex $v \in V_i$ for which the only outgoing edges of v in ST_{G_i} are back edges.

If the strongly connected component G_i is trivial (consists of a single vertex), then either it has no outgoing edge, or an outgoing edge to itself (a self-loop), which is necessarily a back edge. In both cases, the lemma holds. Therefore, we now assume that the component is non-trivial, and consists of at least a root node together with one or more child vertices.

Because, by assumption, no edge in G_i leaves G_i in G, then no outgoing edge of a vertex in ST_{G_i} leaves the tree. Thus, for any vertex $v \in V_i$, and any edge $< v, w > \in E_i$, $w \in V_i$. Furthermore, because G_i is strongly connected, for any two verticies $v, w \in V_i$, there is a path from v to w which remains in G_i. Consider now the leaf nodes of ST_{G_i}. Because all outgoing edges lead to verticies in the tree, these leaf nodes can only have outgoing back edges or cross edges (tree edges or forward edges are not possible outgoing edges from a leaf node). Any outgoing cross edges from a leaf node go from right to left: if $< v, w >$ is a cross edge, then $w < v$, where nodes are ordered by their dfs number. This is a property of depth-first search spanning trees. Consider now the left most leaf node *firstback* (the leaf node with the smallest dfs number). The only outgoing edges possible from here are back edges or cross edges. We cannot have a cross edge leaving here, otherwise the *firstback* would not be the left most: there would be a node with a dfs number smaller than *firstback*, as cross edges move from right to left, and to verticies which

are neither ancestors nor descendants. On the other hand, because the component is strongly connected, there must be an outgoing edge to the root node. Therefore, *firstback* is a vertex which consists only of back edges.

□

We illustrate the above Lemma with an example. Figure A.5 shows the spanning tree representation of the reduced state graph of Figure A.3, generated using proviso **C3-dfs'**. Tree edges are represented by solid lines, and non-tree edges by dotted lines. It identifies in particular the strongly connected components (verticies only listed) $SCC_1 = \{1.2,3\}$ and $SCC_2 = \{4,5,6,7,8,9,10,11,12\}$, as well as the additional non-maximal strongly connected subgraphs $\{4,5,6\}$, $\{7,8,9\}$, and $\{10,11,12\}$. The strongly connected component SCC_2 does not have an edge leaving SCC_2 in the reduced state graph. By the Lemma, it should contain a vertex consisting only of back edges. Vertex 12 is such a vertex.

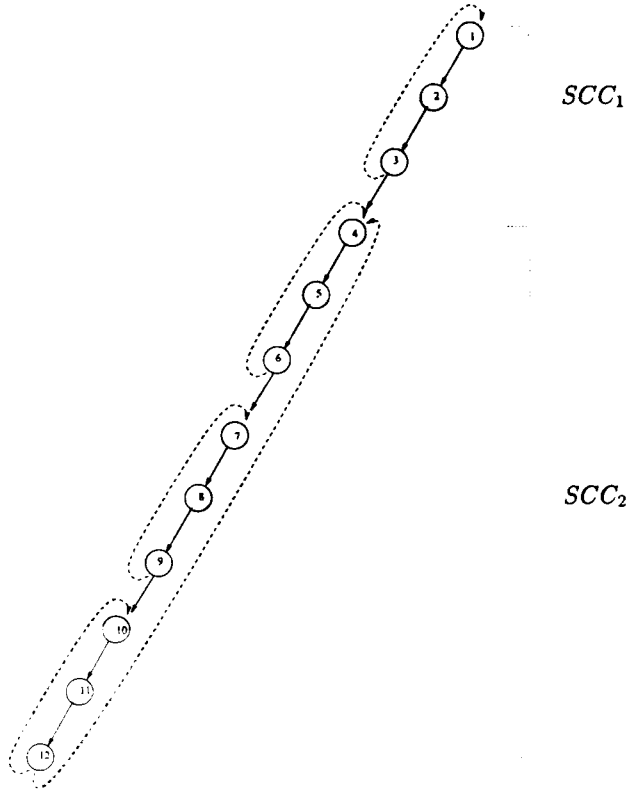


Figure A.5: The reduced state space of Figure A.3 in spanning tree form

With Lemma A.7 completed, we may now prove the following:

Corollary A.1. For the modified depth-first search generation of the reduced state space, the proviso **C3-dfs'** is a sufficient condition for **C3-fin**.

Proof. We prove the corollary by proving the contrapositive: assuming that condition **C3-fin** does not hold, we show that condition **C3-dfs'** does not hold. So suppose that condition **C3-fin**

does not hold for the reduced state space; that is, assume that there exists an ignoring strongly connected subgraph $SCSG$ which has no progress direction leaving the subgraph.

The subgraph $SCSG$ is ignoring, so there exists a state s' and a transition $\alpha \in T$ such that $\alpha \in enabled(s')$ but α does not appear in $ample(s)$ for any state s in the subgraph. Thus, $\alpha \in enabled(s') \setminus ample(s')$ and the state s' is not fully expanded. It is easy to show that for all successor states of s' (and so for all states of the subgraph) the same properties hold.

Any strongly connected subgraph which has no edge leaving the subgraph must be maximal, and so $SCSG$ is a strongly connected component. Therefore, by Lemma A.7, $SCSG$ must contain a state whose outgoing edges consist only of backedges. Furthermore, by the above, this state is not fully expanded.

Therefore, condition **C3-dfs'** does not hold in the reduced state space.

□

A.3.2 Breadth-first search

In [19, Chapter 10], a proviso was described which avoids ignoring in breadth-first search in the case of stuttering equivalence. We state it here for completeness:

C3-bfs If s is not fully expanded, then no transition in $ample(s)$ may reach a state that is in the current level or a previous level of the breadth-first search

This proviso is based on the fact that a necessary condition for closing a cycle in breadth-first search is that the state closing the cycle has already been visited during the search. Note that the proviso is based on information concerning the current level of the search or past levels. We now state a similar condition for the case of finite stuttering equivalence:

C3-bfs' If s is not fully expanded, then at least one transition in $ample(s)$ must not reach a state that is in the current level or a previous level of the breadth-first search

In this section, our aim is to show that a modified state space exploration algorithm based on breadth-first search for which proviso **C3-bfs'** holds is sufficient to guarantee that the property **C3-fin** holds in the resulting reduced state graph. As before, in order to relate the condition on cycles of **C3-bfs'** to the condition on strongly connected subgraphs of **C3-fin**, we consider the spanning forest induced during breadth-first traversal.

As in the case of depth-first search, breadth-first traversal of a graph generates a spanning forest and classifies edges in the graph, in the same way as depth-first traversal. However, in breadth-first search, there are only three types of edges: tree edges, back edges and cross edges. Forward edges no longer exist due to the fact that all immediate successors of a node s are visited before any successors of those successors (and so non-son descendants cannot arise).

As with the provisos to ensure the absence of ignoring for depth-first search, provisos for breadth-first search represent *sufficient* conditions to guarantee that the reduced state space does not contain ignoring strongly connected subgraphs with no progress direction (ignoring sink strongly connected components). This is because it is generally too complex to determine exactly the minimum conditions under which the edges explored from a vertex will lead to the creation of a sink strongly connected component.

In avoiding the creation of such subgraphs, we must pay particular attention to cross edges and back edges leading back into the subgraph. Informally, these edges have the potential to close a sink strongly connected component. Other possible edges encountered during a breadth-first search will only either lead to existing states outside the subgraph, or tree edges leading to new states of the subgraph.

Formally, we have the following lemma.

Lemma A.8. Let $G = (V, E)$ be a graph, with strongly connected components $G_i = (V_i, E_i)$ and let $S = (V, T)$ be the spanning forest generated in a breadth-first traversal of the graph. Let $ST_{G_i} = (V_i, E_i \cap T)$ be the subtree representing G_i in the spanning forest. For each $G_i = (V_i, E_i)$, if G_i does not have an edge leaving G_i in G, then, for all nodes v of ST_{G_i}, the only outgoing edges of v are tree edges leading into V_i, back edges leading into V_i or cross edges leading into V_i.

Proof. Consider the subtree $ST_{G_i}$of the spanning tree containing G_i, and consider a node $v \in V_i$ of that subtree. In the breadth-first exploration of G_i, all outgoing edges $< v, w >$ of verticies v in V_i are classified as one of the following:

- a tree edge, or back edge, or cross edge such that $w \in V_i$
- a tree edge or cross edge such that $w \notin V_i$

By assumption, G_i does not have an edge leaving G_i in G, and so the last two cases cannot hold. □

Tree edges represent the only transitions which reach new states in the breadth-first exploration of the state space. Thus, the proviso **C3-bfs'** effectively requires that if a state is not fully expanded, at least one direction in $ample(s)$ must correspond to a tree edge in the breadth-first search exploration of the state space. Tree edges are 'safe' in the sense that they either lead us into a new component, or stay in the same component but do not close the subgraph. We now prove that this proviso is sufficient to guarantee finite stuttering-equivalence.

Lemma A.9. For breadth-first search, the proviso **C3-bfs'** is a sufficient condition for **C3-fin.**

Proof. We prove the corollary by proving the contrapositive: assuming that condition **C3-fin** does not hold, we show that condition **C3-bfs'** does not hold. So suppose that the condition **C3-fin** does not hold. We need to show that the condition **C3-bfs'** does not hold. By assumption, there exists an ignoring strongly connected subgraph $SCSG$ in the reduced state space which has no progress direction leaving the subgraph.

The subgraph $SCSG$ is ignoring, so there exists a state s' and a transition $\alpha \in T$ such that $\alpha \in enabled(s')$ but α does not appear in $ample(s)$ for any state s in the subgraph. Thus, $\alpha \in enabled(s') \setminus ample(s')$ and the state s' is not fully expanded. It is easy to show that for all successor states of s' (and so for all states of the subgraph) the same properties hold.

Any strongly connected subgraph which has no edge leaving the subgraph must be maximal, and so $SCSG$ is a strongly connected component. Consider the greatest level of the spanning tree containing the subtree ST_{SCSG} and let v be any vertex in that level belonging to $SCSG$. By Lemma A.8, we know that the only outgoing edges of v are tree edges, back edges or cross edges leading back into $SCSG$. An outgoing tree edge from v is not possible, otherwise this would

contradict the maximality of the component. Since tree edges are those edges which visit new states in breadth-first search, all edges in v revisit states in the current level or previous levels of the search. Therefore the proviso **C3-bfs'** does not hold. □

The above proviso is particularly easy to enforce as tree edges are easily identified during a breadth-first search: they correspond to directions leading to states which do not exist in the hash table. However, because repeated states are encountered often in state space searches of concurrent programs, there will be many states in the state space which contain only non-tree edges. Thus, this proviso will result in a great number of states which are fully expanded, defeating attempts at partial order reduction.