The Dissertation Committee for Mehmet Alper Sen
certifies that this is the approved version of the following dissertation:

Techniques for Formal Verification of Concurrent and Distributed Program Traces

Committee:

Vijay K. Garg, Supervisor

Adnan Aziz

Craig M. Chase

Mohamed G. Gouda

Margarida F. Jacome

Techniques for Formal Verification of Concurrent and Distributed Program Traces

by

Mehmet Alper Sen, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2004

To my mother, Ferhunde,

and my brothers, İlker and Berat

Acknowledgments

I consider myself to have been an extremely fortunate Ph.D. student in having Vijay Garg as my Ph.D. supervisor. It is impossible to fully express the many ways in which he has helped to make my Ph.D. experience a fulfilling and enjoyable phase of my life. I learnt how to conduct and evaluate research from him. He has been a constant source of inspiration, encouragement, and guidance in my research work. Moreover, he has been a great friend and mentor, and I have enjoyed our discussions very much, on both technical and non-technical matters.

This dissertation has been shaped by discussions that I have had at various times with fellow students. I am indebted to Luay Nakhleh, Jeff Golden for discussions early in my Ph.D. Neeraj Mittal has helped me at various times by reviewing my papers and providing me with valuable criticism that has greatly improved my work. It has been my privilege to have been able to develop ideas with him. I am also grateful to Arindam Chakraborty, Anurag Agarwal, Selma Ikiz, and Vinit Ogale who have helped me by providing a critical ear during the final stages of my dissertation.

My committee members, Adnan Aziz, Craig M. Chase, Mohammed G. Gouda, Margarida F. Jacome, have also helped to shape my Ph.D. into its current form through their valuable comments and criticisms. Their varied expertise has provided me with different perspectives from which to re-evaluate my work.

My stay in Austin has been enjoyable largely due to my friends Selim Tasiran, Luay Nakhleh, Faik Gur, Sinan Goktepeli, Muammer Ozdemir, Murat Maga, and Selma Ikiz.

The biggest credit goes to my family for their love, support, and sacrifices. My mother, Ferhunde Şen, has kept me going by her never-ending confidence in me. Without my family, this dissertation would never have been written.

Mehmet Alper Sen

The University of Texas at Austin
May 2004

Techniques for Formal Verification of Concurrent and Distributed Program Traces

Publication No. ____________________

Mehmet Alper Sen, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Vijay K. Garg

Traditional approaches for eliminating errors in concurrent and distributed programs include formal methods and testing. This dissertation presents an approach toward combining formal methods and testing, while avoiding the complexity of model checking or theorem proving and the pitfalls of ad hoc testing. Our technique enables *efficient* formal verification of specifications on execution traces of *actual* scalable systems.

By allowing an observer to analyze a partial order trace rather than a total order trace, we get the benefit of properly dealing with concurrent events and especially of detecting errors from analyzing successful executions, errors which can occur under a different thread scheduling. Surprisingly, temporal logic model checking even on a finite partial order trace is NP-complete in the size of the trace description. We develop techniques to combat the state explosion problem. Our algorithms have polynomial-time complexity in the size of the trace description as opposed to exponential.

We suggest the use of an abstraction technique called slicing. Intuitively, the "slice" of a trace with respect to a predicate (specification) is a sub-trace that contains *all* the global states of the trace that satisfy the predicate such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states). We present off-line and on-line slicing algorithms with respect to predicates in a subset of temporal logic CTL. We also show that the slicing problem is equivalent to the problem of detecting whether there exists *at least one* global state of the trace that satisfies the predicate.

We develop efficient algorithms to detect several useful classes of predicates. In particular, we show how to use the slicing algorithms to detect predicates in a subset of temporal logic CTL. We also present efficient detection algorithms for predicates that do not allow efficient slicing.

We have developed a prototype system Partial Order Trace Analyzer (POTA), which implements our algorithms. We verify several scalable and industrial protocols including CORBA's GIOP, ISO's ATMR, cache coherence and mutual exclusion. Our experimental results indicate that slicing can lead to exponential reduction over existing techniques both in time and space.

Contents

Chapter 1

Introduction

Concurrent and distributed systems are commonplace. Examples of such systems include communication protocols, operating systems, microprocessors. The design process of such systems involves many stages and people. These together with non-determinism and the presence of multiple threads of control makes such systems prone to errors. In many systems, especially those employed in safety-critical environments, such as avionics and automobiles, errors are costly and can not be accepted. Thus, it is a challenging task to find ways of reducing the number of errors in safety-critical systems. Formal methods and testing have been widely used to improve the reliability of such systems.

Formal methods are a collection of notations and mathematical techniques for modeling systems, for precisely and unambiguously describing the properties of systems, and for analyzing whether the specified properties are satisfied. There are two broad approaches in formal methods—model checking and theorem proving. *Model checking* [CE81, QS82, CGP00] uses finite state machines as a system model, and specifications are written in a specialized language called *temporal logic* [Pnu77]. These logics can express a wide variety of useful properties of concurrent and distributed programs. *Theorem proving* [BM79, ORR$^+$96, KMM00] expresses the system model and the specifications in a suitable logic, and constructs a proof using the inference rules in the logic that the system model implies the specifications.

Both of these methods enable an exhaustive analysis of the system by reasoning about all possible executions of the system model. However, they both have disadvantages that limit their applicability in practice. In model checking, modeling complex systems as finite state machines suffers from the *state explosion problem.* This is the exponential growth of the number of states in the model with the number of components which comprise the state. Thus model checking is not capable of dealing with systems that are made up of a large number of small finite state machines or systems that manipulate data. Theorem proving can involve generating and proving hundreds of lemmas and a lot of manual effort. Even if a system has been formally verified, we still cannot be sure of the correctness of a particular implementation. The reason is that formal methods, in general, work on an abstract model of a system and make assumptions on the environment. However, for safety-critical systems such as avionics or automobiles, it is crucial to reason about the particular implementation.

Testing (or simulation) is the most frequently used reliability assurance method. It is focused on sampling the executions of a system, and comparing the actual behavior with the behavior that is expected according to the specification.

The main reason for the popularity of testing is that it is simple, feasible and gives a high cost performance ratio. Testing has also the advantage of being able to check an actual implementation of a system rather than a model of it. Unfortunately since testing is applied directly to actual programs with a huge or infinite number of states, it is impossible or impractical to check all the executions of a program in a systematic way. Therefore, testing is not as exhaustive as model checking or theorem proving. Also, testing techniques are ad-hoc and do not allow for formal specification and verification of logical properties that a concurrent and distributed system needs to satisfy.

This dissertation presents an approach toward combining formal methods and testing in a technique called *Predicate Detection* (also called Runtime Verification). This technique enables *efficient* formal verification of specifications on executions of actual scalable systems. Predicate Detection uses mathematical techniques as in formal methods for analyzing whether the specified properties are satisfied, but at a cost same as testing. Predicate Detection is cheaper than formal methods because checking program correctness for particular executions is much easier than proving program correctness for all possible executions. Predicate Detection, similar to testing, has also the advantage of being able to check an actual implementation of a system rather than a model of it. A Predicate Detection environment can be *off-line* (assuming that the entire execution trace is available *a priori*) or *on-line* (while the program is executing).

The main goal of this dissertation is to present *efficient* Predicate Detection algorithms. We depict an overall view of a Predicate Detection environment in Figure 1.1. The contributions of this dissertation in addressing all of the aspects in our Predicate Detection environment are discussed in detail in the following sections.

1.1 Execution Trace Model: Partial Order

An execution of a concurrent and distributed program consists of multiple *processes.* We denote a set of events generated by processes during program execution by a *trace.*

A concurrent and distributed program, when executed, may generate several traces; therefore, a Predicate Detection tool should handle large numbers of traces efficiently. In order to increase the scalability, we model an execution trace of a program as a *partial order* between events (also called a computation) generated

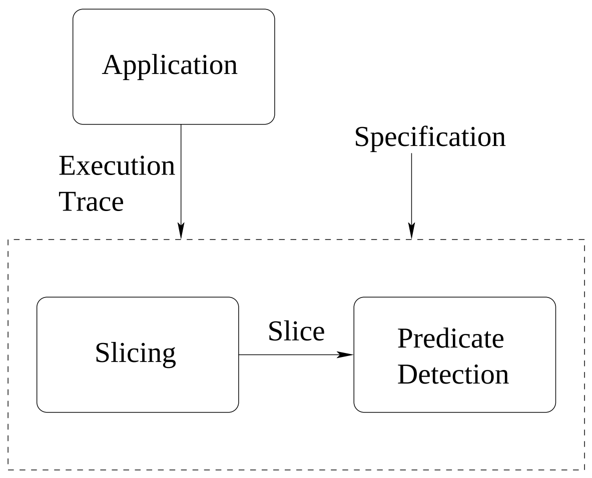


Figure 1.1: Predicate Detection environment overall view

during program execution as opposed to a total order between events. Every total order of events that respects the partial order relation corresponds to an order in which the events could have been executed.

Using a partial order model, we can capture exponential number of possible total order traces succinctly. By allowing an observer to analyze a partial order trace rather than a total order trace, we also get the benefit of properly dealing with concurrent events and especially of detecting errors from analyzing successful executions, errors which can occur under a different thread scheduling. We now illustrate detection of errors with an example. In Figure 1.2(a), a total order trace generated by a program with two processes is depicted. The events on process P_1 are generated in the order e_0, e_1, e_2 and the events on process P_2 are generated in the order f_0, f_1, and f_2. Each event is also labeled by CS if the corresponding process is in the critical section. For example, P_2 enters critical section at event f_1. Let f_1 be the send of a message from P_2 and e_1 be the receive of that message by P_2. The partial order trace corresponding to the total order trace in Figure 1.2(a) is depicted in Figure 1.2(b). Observe that the only dependencies in the partial order trace are

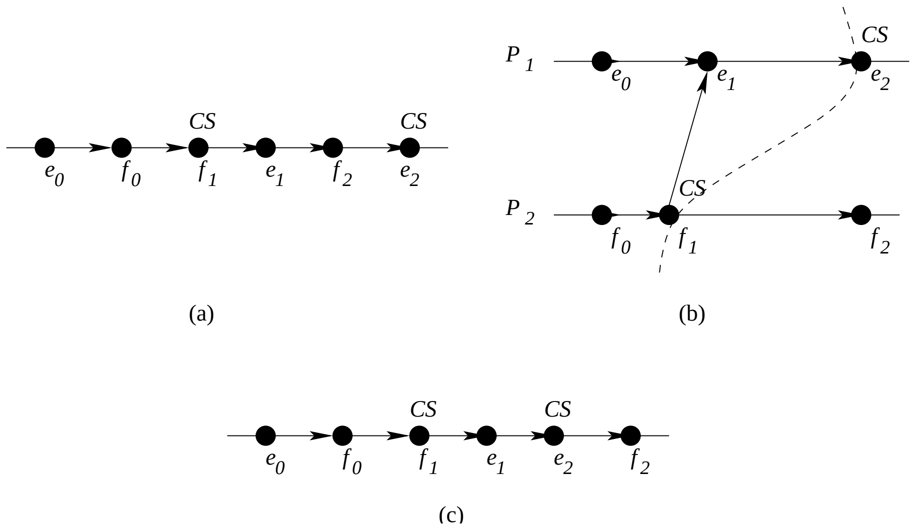


Figure 1.2: (a) A total order trace, (b) the corresponding partial order trace, (c) a total order trace generated from the partial order trace

the order of events on each process and the message send/receive dependencies. Suppose the specification is the mutual exclusion property which requires that there is no global state where two processes are in the critical section at the same time. The state of a system, referred to as a *global state*, is given by the set of events that have been executed so far (on all processes). The specification is satisfied for the total order trace in Figure 1.2(a). From the partial order trace we observe that events e_2 and f_2 can be executed in either order since there is no dependency between them. However, one of the orderings satisfies the specification, whereas the other results in a violation of the specification. We depict the ordering that respects the partial order relation and violates the specification in Figure 1.2(c).

Furthermore, a partial order model is a more faithful representation of concurrency and this model enables us to apply our theory to both concurrent and distributed systems.

1.2 Specification Language: Temporal Logic

While traditional assertions (pre and post conditions, invariants) can be used to express several properties of a program, there are many logical properties that cannot be checked with them. Temporal logics [Pnu77] can express a wide variety of useful properties of concurrent and distributed programs such as safety and liveness properties [Lam77, AS85]. Intuitively, a safety property specifies that "something bad will not happen" during system execution and a liveness property specifies that "something good will eventually happen" during system execution. An example of a safety property is mutual exclusion which states that at no time more than one process is in its critical section. An example of a liveness property is starvation freedom which states that every process that requests a resource eventually acquires the resource. With the use of temporal logic, we can both *unambiguously* describe the properties of a program and also express properties that cannot be expressed by traditional testing techniques.

Two popular temporal logics are linear temporal logic LTL [Pnu77] and branching temporal logic CTL [CE81]. Branching temporal logic CTL [CE81] has been widely used as a specification language in model checking. The complexity of model checking using CTL is linear in the size of the structure and the formula, whereas the complexity is exponential in the formula size for LTL [LP85].

Our partial order trace model leads to a state space (the set of possible global states of a system) which is a *finite distributive lattice.* A partial order set forms a *lattice* if the greatest lower bound (meet) and the least upper bound (join) exist and are contained in the set for every pair of elements. A lattice is *distributive* if its meet operator distributes over its join operator.

CTL semantics has so far been defined for infinite execution traces. We define a finite trace semantics for CTL and interpret it on a distributive lattice structure. CTL is built up from atomic propositions and temporal modalities. Every global state is annotated with atomic propositions that hold in that state. Some of the widely used temporal modalities in CTL are EF, AG, EG, AF, EX, AX, EU, and AU. The modalities are interpreted over sequences of global states or paths. The operators E and A are used to denote the branching nature of time starting from the current state. Specifically, we use E and A to denote that a formula is true "for some path" and "for all paths" starting from the current state, respectively. We use F and G to denote that a formula is true "eventually" and "globally" on a path, respectively. We use X to denote that a formula is true "at the next state" on a path. Finally, we use U to denote that a combination of two formulas is possible where the first must be true on the path states, beginning at the current state, until the second formula becomes true. For example, $\mathsf{EF}(p)$ means that "*for some path* starting from the current state, p *eventually* holds on the path" or $\mathsf{AG}(p)$ means that "*for all paths* starting from the current state, p *globally* holds on the path". Similarly, $\mathsf{EG}(p)$ means that "*for some path* starting from the current state, p *globally* holds on the path" or $\mathsf{AF}(p)$ means that "*for all paths* starting from the current state, p *eventually* holds on the path". Now, we can specify the starvation freedom property mentioned above in CTL as $\mathsf{AG}(request \Rightarrow \mathsf{AF}(acquire))$, where *request* and *acquire* are atomic propositions to denote that the process is in the corresponding state. The detailed semantics of CTL operators is given in Chapter 3.

Although CTL model checking has time complexity linear in the size of the structure, the size of the structure (i.e., the number of global states) may itself be exponential in the size of its description as a program. Given a program with n boolean variables, the reachable state space may be of size 2^n—state explosion. Surprisingly, even when CTL model checking is used for partial order execution traces rather than programs, the state explosion exists. Given a partial order trace generated by n processes (or threads) with k events on each process, the possible number of global states can be as high as $O(k^n)$, where a global state is comprised of n components. This is the state explosion problem in partial order trace model.

The problem of CTL model checking on finite traces has been denoted by *predicate detection* in the context of distributed systems [CM91, CG98]. We also chose to use predicate detection to differentiate between program and trace checking. Hereafter, we will use predicate detection instead of model checking.

Our goal in this dissertation is to develop efficient predicate detection algorithms (polynomial-time in the number of processes). For this purpose we develop abstraction techniques and exploit the lattice structure of the state space.

1.3 Abstraction Technique: Slicing of Traces

Many different methods have been devised for automatically checking temporal logic specifications on execution traces by examining the state space models of system behavior. These methods all depend on decision procedures that explicitly represent the state space using a list or a table that grows exponentially in the number of processes, components, or state elements (*state explosion problem*). We developed a general method that represents the state space that satisfies a temporal logic specification using a "slice" instead of using an explicit representation.

Computation slicing was introduced in [GM01, MG01a] as an abstraction technique for analyzing traces of distributed programs, that is, distributed computations. Intuitively, a *slice* of a trace with respect to a temporal logic specification (predicate) p is a sub-trace that contains all the states of the trace that satisfy p. Note that the set of states that satisfy p may be large, so one could not simply enumerate all the states efficiently either in space or time. A slice contains all the states that satisfy p such that it is computed efficiently (without traversing the state space) and represented concisely (without explicit representation of individual states). The slice has much fewer states than the trace itself—exponentially smaller in many cases—resulting in substantial savings. Therefore, in order to determine whether a predicate is satisfied, rather than searching the state space of the trace, it is much more efficient to compute the slice and search the state space of the slice. Other state space reduction techniques for reducing the time and/or space complexity, such as partial order reduction, BDDs and SAT [McM93, GW91, Val91, Pel93, Esp94, SL98, SUL00, BCCZ99], are orthogonal to slicing, and as such can be used in conjunction with slicing. While techniques such as partial order reduction, BDDs and SAT rely on state space traversal, with our slicing technique, the state space is never traversed (or built), rather our algorithms work on the partial order trace model itself.

Computing the slice for an arbitrary predicate is known to be intractable in general [MG01a]. However, by exploiting the structure of the predicate, polynomial-time algorithms have been developed for non-temporal regular and linear predicates [GM01, MG01a, MG03] . A non-temporal predicate is such that it does not contain any temporal operator. A predicate is regular if the set of global states that satisfy it forms a sublattice, that is, it is closed under intersection and union. The slice with respect to a regular predicate contains *precisely* those global states for which the predicate evaluates to true. Regular predicates widely occur in practice during verification. Some examples of regular predicates are conjunction of local predicates such as "all processes are in *red* state" and channel predicates such as "at most k messages are in transit from process P_i to P_j". A predicate is linear if the set of global states that satisfy it forms an inf-semilattice, that is, it is closed under intersection. Linear predicates contain the class of regular predicates.

Using our results in this dissertation, it is now possible to compute the slice efficiently for many more classes of *non-temporal* and *temporal* predicates. The non-temporal predicates include stable, co-stable, observer-independent, relational, and co-linear predicates and the temporal predicates include a subset of temporal logic CTL. In this dissertation, we also present on-line slicing algorithms in which the slice is computed as and when a new event arrives. Next, we summarize these contributions.

Slicing with respect to predicate classes with efficient $\mathsf{EF}(p)$ detection: The predicate detection problem under EF operator is only concerned with answering the question whether there exists *at least one* global state of the trace that satisfies the given predicate. Slicing, on the other hand, is concerned with computing a succinct representation of *all* global states of the trace for which the given predicate evaluates to true. Clearly, detecting a predicate under EF operator is no harder than computing its slice in the sense that the predicate detection problem under EF operator can be easily solved given the slice for the predicate (it suffices to test for the emptiness of the slice). In this dissertation, we prove a somewhat surprising result that detecting a predicate under EF operator is no easier than computing its slice. In other words, given an algorithm A for detecting a predicate $\mathsf{EF}(p)$, there exists an algorithm B for computing the slice for p such that the time-complexity of B is at most a small multiple of the time-complexity of A. In particular, the multiple is $n|E|$, where n is the number of processes and E is the set of events. As a corollary, it can be derived that there exists a polynomial-time algorithm for detecting a predicate if and only if there exists a polynomial-time algorithm for computing its slice.

Using this result, it is now possible to compute the slice efficiently for many more classes of non-temporal predicates including stable and co-stable predicates, observer-independent predicates, co-linear predicates, and relational predicates. This is because there are efficient $\mathsf{EF}(p)$ detection algorithms in the literature when p belongs to one of these classes. A stable predicate is such that once the predicate becomes true it stays true. For example, deadlock or termination are stable predicates. The complement of a stable predicate is denoted by a co-stable predicate. An observer-independent predicate is such that if the predicate is eventually true for some path, it is eventually true for all paths. For example, the disjunction of local predicates such as "at least one server is not busy" is an observer-independent predicate. An example of a relational predicate is $x_1 + x_2 + \ldots + x_n \leqslant k$ for constant k, where x_i is an integer variable on process P_i.

Slicing with respect to Temporal Logic Predicates: We identify a regular subset of temporal logic CTL denoted by RCTL with predicates for which the slices contain precisely those global states that satisfy the predicate. To that end, we prove that temporal predicates $\mathsf{EF}(p)$, $\mathsf{EG}(p)$, and $\mathsf{AG}(p)$ are regular, whereas $\mathsf{AF}(p)$, $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, $\mathsf{E}(p \mathbin{\mathsf{U}} q)$, and $\mathsf{A}(p \mathbin{\mathsf{U}} q)$, in general, are not regular when p and q are regular. We present polynomial-time algorithms to compute slices for $\mathsf{EF}(p)$, $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, and a local version of EX denoted by $\mathsf{EX}(p)[j]$, to specify that the predicate holds at the next action of process j.

On-line Slicing: The algorithms described in earlier papers [GM01, MG01a, MG03] for computing a slice are all *off-line* in nature; they assume that the entire set of events is available *a priori*. While this is quite adequate for applications such as testing and debugging; for other applications such as software fault tolerance, it is desirable that the slice be computed incrementally in an *on-line* manner. In other words, the current slice is updated, as and when a new event is generated, to reflect its arrival. The reason is that for software fault tolerance, it is important to detect the fault as early as possible before it can cause any severe damage. If we compute the slice only after a certain number of events have been collected and then analyze it for the presence of a faulty global state, it may be too late for any meaningful recovery. At the same time, whenever an event arrives, we want the cost of incrementally updating the slice to be less than the cost of recomputing the slice from scratch using an off-line algorithm. In this dissertation, we give an efficient incremental algorithm for computing the slice for predicates with efficient $\mathsf{EF}(p)$ detection algorithm and for the class of regular predicates.

1.4 Analysis Technique: Predicate Detection

Our approach to ameliorate state explosion in partial order trace model uses two techniques: (1) slicing and (2) exploiting the structure of the predicate itself—by imposing restrictions—to evaluate its value efficiently for a given execution trace. Some examples of the predicates for which the predicate detection can be solved efficiently are: conjunctive [GW94, HMRS96], stable [CL85], observer-independent [CBDGF95], linear [CG98], an relational [TG97] predicates. So far, these predicate classes have been detected under some or all of the temporal operators EF, EG, AG, AF of CTL, but not under any nesting of these operators. For example, a nested temporal predicate $\mathsf{EF}(p \wedge \mathsf{EG}(q))$, where p and q are conjunctive predicates, cannot be efficiently detected using only the efficient algorithms for conjunctive predicates.

We show how to use the slicing algorithms developed in this dissertation for efficient detection of predicates in $\mathsf{RCTL+}$, which contains nested temporal predicates. In $\mathsf{RCTL+}$, the temporal operators are EF, EG, AG, $\mathsf{EX}[j]$, EX and the atomic propositions are regular, co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates. We also develop efficient detection algorithms for temporal operators that do not allow efficient slicing, hence do not belong to $\mathsf{RCTL+}$, such as AF, AU, and EU. Specifically, we provide algorithms for predicates without nested temporal operators (unnested) such as $\mathsf{AF}(p)$, $\mathsf{A}(p\,\mathsf{U}\,q)$, and $\mathsf{E}(p\,\mathsf{U}\,q)$. For unnested temporal predicates of the form $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$ we improve the complexity results in [MG03], when p is a non-temporal regular predicate.

We consider another problem related to predicate detection, namely the problem of locating a global state of a trace that satisfies the given predicate (counterexample), if it exists. While it is sufficient to determine whether there exists a faulty state in a trace for testing purposes; for debugging purposes, it is desirable to actually *locate* the faulty state. An examination of such a state may provide valuable insight into the bug that caused the fault. This problem is closely related to the problem of setting a *global breakpoint* when debugging a distributed program.

1.5 Tools for Predicate Detection

We validate the effectiveness of our algorithms with experimental studies. For this purpose, we implemented a prototype system called Partial Order Trace Analyzer (POTA). The tool consists of 3 main modules: analyzer, translator, and instrumentor. We have implemented predicate detection algorithms in the analyzer module. The translator module translates traces into input language of SPIN model checker and enables us to compare POTA with partial order reduction techniques of SPIN [Hol97]. Since we are working with concurrent and distributed programs which exhibit a lot of parallelism and independency, partial order reduction techniques can take advantage of these properties of programs. The instrumentation module generates an instrumented version of input program such that when run, every process outputs "relevant" events.

Our experimental results demonstrate that slicing is a very effective abstraction technique. We generated traces of several scalable protocols including CORBA General Inter-ORB Protocol (GIOP), Asynchronous Transfer Mode Ring (ATMR) and MESI cache coherence protocol with up to 200 processes and verified them using several safety and liveness properties. In almost all cases, we obtained more than three orders of magnitude speed-up compared to SPIN. Furthermore, SPIN could not find bugs, which were found by POTA, due to running out of memory.

1.6 Overview of the Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2 , we define our partial order trace model. Chapter 3 discusses the temporal logic and the classes of predicates that we use in this dissertation. In Chapter 4, we present a background on slicing, our results on slicing of traces with respect to non-temporal predicates and our on-line slicing algorithm. Chapter 5 investigates our temporal logic slicing approach. Our predicate detection algorithms for temporal and non-temporal predicates are presented in Chapter 6. We present POTA and our predicate detection experiments in Chapter 7. In Chapter 8, we give a summary of the related work. Finally, we draw conclusions and describe future work in Chapter 9.

Chapter 2

System Model

In this chapter we formally describe the system model and notation used in this dissertation.

2.1 Overview

We can model an execution trace of a program, which consists of events, in two ways. The first model imposes a total order (interleaving) of events. The second model imposes a partial order of events. Traditionally, the former model has been used in testing and runtime verification [KKL$^+$01, HR01]. We use the latter approach which has several advantages over the former. First, it is a more faithful representation of concurrency [Lam78], that is, only the events that have a causal dependency are ordered. Second, a partial order encodes possibly exponential number of interleavings. This may translate into finding bugs that are not found using a single interleaving. Hence, we obtain better coverage in terms of testing. Third, our partial order approach is applicable to both message passing and shared memory programs. We use partial order relations in [Lam78] and [SRA03] for representing traces of message passing and shared memory programs, respectively. All of these aspects aid in modeling execution traces in a very compact and efficient manner.

We first define lattice theoretic concepts used in developing our trace model and slicing theory. Then we extend our trace model and related notions. The extended model allows us to handle a trace and its slice in a uniform fashion.

2.2 Partially Ordered Sets and Lattices

A pair (X, P) is called a partially ordered set or *poset* if X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. We call X the *ground set* while P is a *partial order* on X. The $\leqslant$ and *divides* relations on the set of natural numbers are some examples of partial orders.

We write $x \leqslant y$ and $y \geqslant x$ in P when $(x, y) \in P$. Also, $x < y$ and $y > x$ in P means $x \leqslant y$ in P and $x \neq y$. Let $x, y \in X$ with $x \neq y$. If either $x < y$ or $y < x$, we say x and y are *comparable.* On the other hand, if neither $x < y$ nor $x > y$, then we say x and y are incomparable.

A poset (X, P) is called a *total order* or a *linear order* if every distinct pair of points from X is comparable in P. It is possible to extend any partial order to a linear order by adding order between pairs of unordered elements. Each such linear order is called a *linearization* of the partial order. Similarly, we call a poset an *antichain* if every distinct pair of points from X is incomparable in P. The *height* of a poset is defined to be the largest chain in the poset and is denoted by $height(P)$. Similarly, the *width* of a poset is defined to be the largest antichain in the poset and is denoted by $width(P)$.

Finite posets are often depicted graphically using a *Hasse diagram.* To define Hasse diagrams, we first define a relation *covers* as follows. For any two elements x, y, we say y covers x if $x < y$ and $\forall z \in X : x \leqslant z < y$ implies $z = x$. In other words, there should not be any element z with $x < z < y$. A Hasse diagram of a poset is a graph with the property that there is an edge from x to y if and only if y covers x. Furthermore, when drawing the figure in an Euclidean plane, x is drawn lower than y when y covers x. For example, consider the poset $(X, \leqslant)$.

$$X \triangleq \{a,b,c,d,e\} \quad \leqslant \triangleq \{(a,a),(b,b),(c,c),(d,d),(a,b),(a,c),(a,d),$$
$$(a,e),(b,d),(b,e),(c,e),(d,e)\}.$$

The first Hasse diagram in Figure 2.1 corresponds to this poset.

An element $y \in X$ is called an *upper bound* for $S \subseteq X$ if $s \leqslant y$ in P, for every $s \in S$. An upper bound y for S is the *least upper bound* for S, provided $y \leqslant y'$ in P for every upper bound y' of S. *Lower bounds* and *greatest lower bounds* are defined similarly. The greatest lower bound is also referred to as *infimum* or *meet*. Similarly, the least upper bound is also referred to as *supremum* or *join*. We denote the *meet* of $\{a,b\}$ by $a \sqcap b$, and the *join* of $\{a,b\}$ by $a \sqcup b$.

In the set of natural numbers ordered by the *divides* relation, the *join* corresponds to finding the greatest common divisor and the *meet* corresponds to finding the least common multiple of two natural numbers.

The greatest lower bound or the least upper bound may not always exist. In the third poset in Figure 2.1, the set $\{b,c\}$ does not have any least upper bound (although both d and e are upper bounds).

Definition 2.1 (lattice) *A set of partially ordered elements (or poset) forms a lattice L if the* greatest lower bound *and the* least upper bound *exist and are contained in the set for every pair of elements.*

Thus, the first two posets in Figure 2.1 are lattices, whereas the third one is not. As another example, the power set of a given set forms a lattice under $\subseteq$ relation.

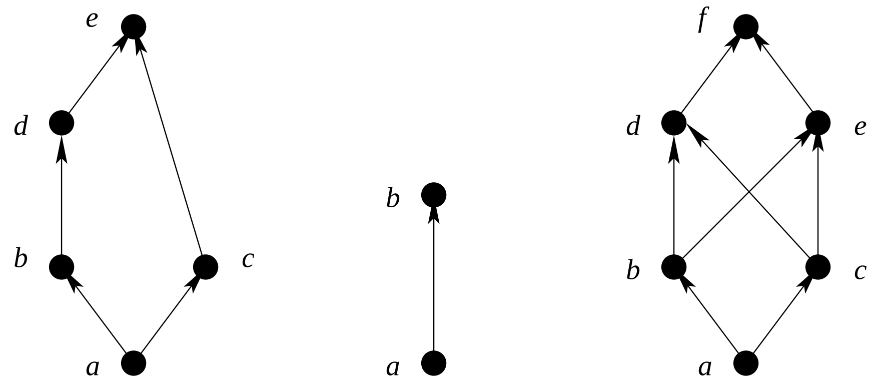


Figure 2.1: Only the first two posets are lattices.

Example 2.2 *For the set $\{x, y, z\}$, the power set is given by $\{\emptyset, \{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$. The meet of the two elements of a power set is given by their intersection. For example, the meet of $\{x, y\}$ and $\{y, z\}$ is $\{y\}$. Dually, the join is given by their union. For example, the join of $\{x, y\}$ and $\{y, z\}$ is $\{x, y, z\}$. In other words, the meet and join operators of the lattice correspond to intersection ($\cap$) and union ($\cup$), respectively.* □

The lattice in Example 2.2 is called a *Boolean* lattice.

Definition 2.3 (sublattice) *A subset of a lattice is a sublattice if it is closed under the meet and join operations.*

For example, in the Boolean lattice the set of all subsets of $\{x, y, z\}$ that contain x forms a sublattice. However, the set of all subsets with at most two elements does not form a sublattice.

Definition 2.4 (distributive lattice) *A lattice is distributive if its meet operator distributes over its join operator.*

For example, since intersection distributes over union, a Boolean lattice is distributive. The lattice of natural numbers with $\leqslant$ defined as the relation *divides* is also distributive. Two important nondistributive lattices, called diamond and pentagon, are shown in Figure 2.2.

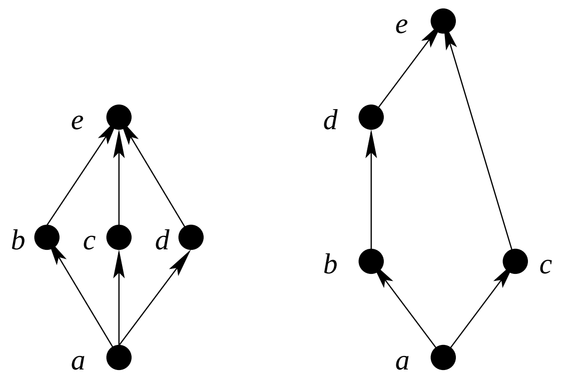


Figure 2.2: Examples of nondistributive lattices

Now, consider a (finite) set of partially ordered elements (not necessarily a lattice). A subset of elements forms an *order ideal* (or simply an *ideal*) if whenever an element is contained in the subset then all its preceding elements are also contained in the subset.

Definition 2.5 (order ideal) *Given a poset (X, P), a subset S of X is an order ideal if it satisfies $\langle \forall\, x, y : x, y \in X : (x \in S) \wedge (y \leqslant x) \Rightarrow (y \in S)\rangle$*

Example 2.6 *For the poset in Figure 2.3(b), some examples of ideals are $\{a, b, c\}$ and $\{a, b\}$. However, $\{a, d\}$ is not an ideal because it contains d but not b, which precedes d.* $\square$

We denote the set of ideals of a poset (X, P) by $\mathcal{C}(P)$.

Theorem 2.7 ([DP90]) *The set of ideals of a poset forms a distributive lattice under $\subseteq$ relation.*

By using the notion of ideals, we went from a poset to a distributive lattice. Is it possible to go in the reverse direction? The answer is provided by Birkhoff's Representation Theorem [DP90]. Intuitively, the result says that a finite distributive lattice can be uniquely characterized by only a small subset of its elements known as *join-irreducible elements*.

Definition 2.8 (join-irreducible element) *An element of a lattice L is join-irreducible if (1) it is not the least element, and (2) it cannot be expressed as join of two elements, both different from itself. Formally, $a \in L$ is join-irreducible if,*

$$\langle \exists\, x :: x < a \rangle \bigwedge \langle \forall\, x, y : x, y \in L : a = x \sqcup y \Rightarrow (a = x) \vee (a = y) \rangle$$

Clearly, the join-irreducible elements of a Boolean lattice are the singleton sets.

Example 2.9 *The Boolean lattice in Example 2.2 has three join-irreducible elements, namely $\{x\}$, $\{y\}$ and $\{z\}$. As expected, every other element that is different from $\emptyset$ can be expressed as the union of some or all of these three elements.* □

Pictorially, in a finite lattice, an element is join-irreducible if and only if it has exactly one lower cover, that is, there is exactly one edge coming into the element in the Hasse diagram. Intuitively, the join-irreducible elements of a distributive lattice act as *basis elements* for the lattice. Every element of the lattice, except for the least one (*e.g.*, $\emptyset$ in a Boolean lattice), can be written as the join of some or all of these join-irreducible elements. For a distributive lattice L, $\mathcal{JI}(L)$ refers to the set of its join-irreducible elements. The notion of *meet-irreducible elements* can be defined dually.

Definition 2.10 (meet-irreducible element) *An element of a lattice L is meet-irreducible if (1) it is not the greatest element, and (2) it cannot be expressed as* *meet of two elements, both different from itself. Formally, $a \in L$ is meet-irreducible if,*

$$\langle \exists\, x :: x > a\rangle \bigwedge \langle \forall\, x, y : x, y \in L : a = x \sqcap y \Rightarrow (a = x) \vee (a = y)\rangle$$

For a distributive lattice L, $\mathcal{MI}(L)$ refers to the set of its meet-irreducible elements. The meet-irreducible elements of a Boolean lattice are given by those subsets of the ground set that have exactly one element missing. Thus, the meet-irreducible elements of the Boolean lattice in Example 2.2 are $\{x, y\}$, $\{y, z\}$ and $\{x, z\}$. Clearly, every other element that is different from $\{x, y, z\}$ can be expressed as the intersection of some or all of these three elements.

Theorem 2.11 (Birkhoff's Theorem [DP90]) *Let L be a finite distributive lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible-elements. Then the map $f : L \rightarrow \mathcal{C}(\mathcal{JI}(L))$ defined by,*

$$f(a) = \{x \in \mathcal{JI}(L) \mid x \leqslant a\}$$

is an isomorphism of L onto $\mathcal{C}(\mathcal{JI}(L))$. Dually, let P be a finite poset. Then the map $g : P \rightarrow \mathcal{JI}(\mathcal{C}(P))$ defined by

$$g(a) = \{x \in P \mid x \leqslant a\}$$

is an isomorphism of P onto $\mathcal{JI}(\mathcal{C}(P))$.

Thus, Birkhoff's Theorem establishes the duality between finite posets and finite distributive lattices. We can go from a finite poset to its dual finite distributive lattice by constructing the set of its order ideals and from the finite distributive lattice to the poset by restricting it to join-irreducible elements. For example, Figure 2.3(b) gives the poset corresponding to the lattice in Figure 2.3(a). Note that the theorem can be defined dually using *meet-irreducible elements.*

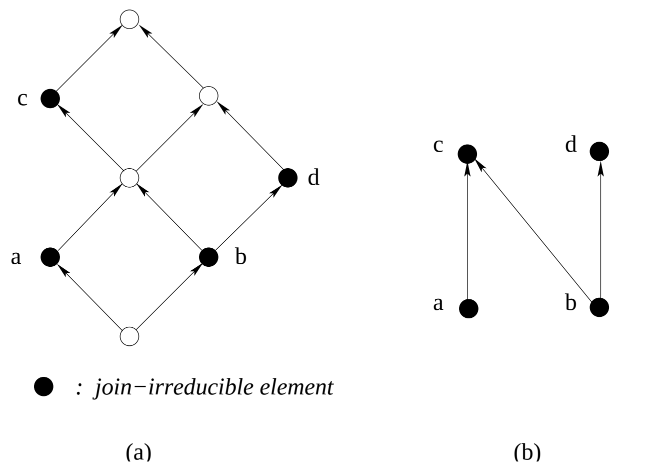


Figure 2.3: (a) An example of a distributive lattice, (b) its partial order representation

2.3 Trace Model

A *program* consists of n sequential processes (or threads) denoted by $P_1, P_2, \ldots, P_n$. The *local computation* of a process is given by the sequence of events that transforms the *initial* state of the process into the *final* state. At each step, the *local* state is captured by the initial state together with the sequence of events that have been executed up to that step. Each event is an *internal event* or an *external event.* Examples of external events are: a *send event*, *a receive event*, *a read event*, and a *write event.* An event causes the local state of a process to be updated. The send and receive events cause a set of messages to be sent or received, respectively. We assume the presence of fictitious initial and final events on each process. The initial event on process P_i, denoted by $\bot_i$, occurs before any other event on P_i. Likewise, the final event on process P_i, denoted by $\top_i$, occurs after all other events on P_i. For convenience, let $\bot$ and $\top$ denote the set of all initial events and final events, respectively.

Let *proc*(e) denote the process on which event e occurs. The predecessor and

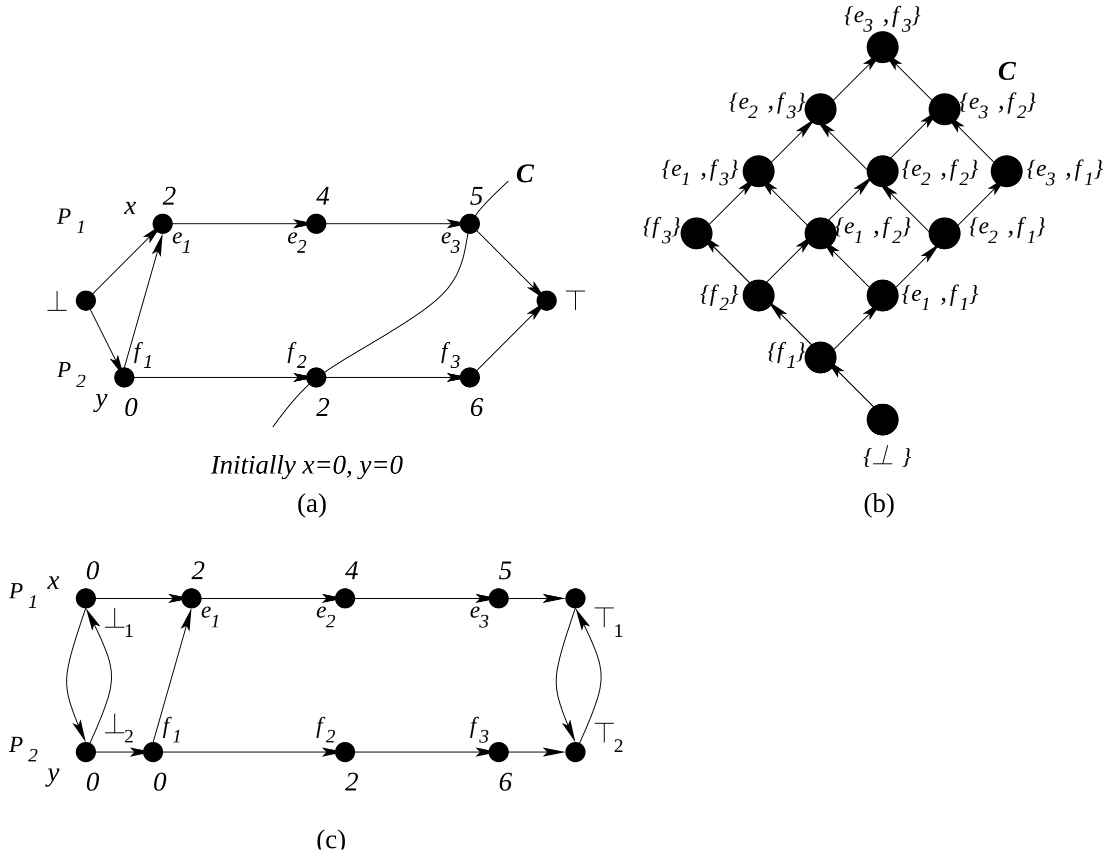


Figure 2.4: (a) An execution trace, (b) its lattice corresponding to nontrivial consistent cuts, and (c) another representation of the computation in (a)

successor events of e on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist. We denote the order of events on processes by $\xrightarrow{P}$ which is referred to as *process order*. The projection of $\xrightarrow{P}$ on process i is denoted by $\xrightarrow{P_i}$. The reflexive closure of $\xrightarrow{P}$ is represented by $\underline{\xrightarrow{P}}$ and its transitive closure is denoted by $\xrightarrow{P}^{+}$.

Example 2.12 *Consider an execution of a message passing program. The resulting execution trace is shown in Figure 2.4(a). Such figures are usually called* space-time diagrams, *or* happened-before diagrams. *In the trace, there are two processes* P_1 *and* P_2 *with integer variables* x *and* y*, respectively. The local computations of* *each process advances from left to right as shown in the figure. The events are represented by circles and the order relation is represented by arrows. (Observe that the execution traces are drawn from left to right, whereas Hasse diagrams are drawn from bottom to up). The local computation of P_1 is given by the sequence $\bot_1 e_1 e_2 e_3 \top_1$. The event f_1 is a send event, the event e_1 is a receive event and the event e_2 is an internal event. Here, $\bot$ and $\top$ are the set of fictitious initial and final events. Also, $proc(e_2) = P_1$, $pred(e_2) = e_1$ and $succ(e_2) = e_3$. The process order $\xrightarrow{P}$ is given by $\{(\bot, e_1), (e_1, e_2), (e_2, e_3), (e_3, \top),\ (\bot, f_1),\ (f_1, f_2), (f_2, f_3),\ (f_3, \top)\}^+$. Process P_2 sends a message to process P_1 by executing event f_1 and process P_1 receives that message by executing event e_1. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of x immediately after executing e_1 is 2. The first event $\bot$ initializes the state of each process.*

2.3.1 Traces as Graphs

We model an execution trace as a partial order on the set of events in the trace. For example, in the case of message passing programs, the partial order is Lamport's happened-before relation [Lam78]. In this section, we relax the restriction that the order on events must be a partial order so that we can use directed graphs to model execution traces as well as slices. However, several notions developed for partial order sets such as ideals and distributive lattices have equivalent notions for directed graphs as well.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote the set of vertices and edges, respectively. We define a *consistent cut* (consistent global state) on directed graphs as a subset of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. Formally,

Definition 2.13 (consistent cut) *Given a directed graph G, a subset C of $\mathsf{V}(G)$*

is a consistent cut if,

$$\langle \forall\, e, f : e, f \in \mathsf{V}(G) : (f \in C) \wedge (e, f) \in \mathsf{E}(G) \Rightarrow (e \in C)\rangle$$

Remark 2.14 *Observe that the notion of consistent cuts is similar to the notion of order ideals in Definition 2.5.*

We denote the set of consistent cuts of a directed graph G by $\mathcal{C}(G)$, which forms a distributive lattice under $\subseteq$ relation.

Theorem 2.15 ([MG01a]) *Given a directed graph* G*,* $(\mathcal{C}(G), \subseteq)$ *forms a distributive lattice.*

Remark 2.16 *The above theorem is a generalization of the result in lattice theory described in Theorem 2.7.*

The meet ($\sqcap$) and join ($\sqcup$) of the two consistent cuts is given by their set intersection ($\cap$) and set union ($\cup$), respectively. Observe that the empty set $\emptyset$ and the set of vertices $\mathsf{V}(G)$ trivially belong to $\mathcal{C}(G)$. We call them *trivial* consistent cuts. We use $\mathcal{P}(G)$ to denote the set of pairs of vertices (u, v) such that there is a path from u to v in G. We assume that each vertex has a path to itself.

We model an *execution trace* (or a *computation*), denoted by $\langle E, \rightarrow\rangle$, as a directed graph with vertices as the set of events E and edges as $\rightarrow$. We use event and vertex interchangeably. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that $\mathcal{P}(\langle E, \rightarrow\rangle)$ contains at least the partial order relation. We assume that all initial events belong to the same strongly connected component. Similarly, all final events belong to the same strongly connected component. This ensures that any nontrivial consistent cut will contain all initial events and none of the the final events. Therefore, every consistent cut of a computation in the model without $\perp, \top$ (traditional model) is a nontrivial consistent cut of the computation in our model and vice versa. Note that the *initial* *consistent cut* $\emptyset$ and the *final consistent cut* E in the traditional model correspond to $\{\bot\}$ and $\mathcal{E} = E - \{\top\}$, respectively, in our model.

Observe that a computation in our model can contain cycles. This is because whereas a computation in the partial order model captures the *observable* order of execution of events, a computation in our model captures the set of possible consistent cuts.

A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally, $\mathit{frontier}(C) \triangleq \{\, e \in C \mid \mathit{succ}(e) \text{ exists } \Rightarrow \mathit{succ}(e) \notin C\}$. A consistent cut is uniquely characterized by its frontier and vice versa. Thus in figures, we specify a consistent cut by simply listing the events in its frontier instead of enumerating all its events.

We say that an event e is *enabled* at a consistent cut C if there exists $g \in \mathit{frontier}(C)$ such that $e = \mathit{succ}(g)$ and $C \cup \{e\}$ is a consistent cut.

We say that a cut D is *reachable* from a cut C if $C \subseteq D$. We define *successor* of a cut by a relation $\rhd \subseteq \mathcal{C}(G) \times \mathcal{C}(G)$ such that $C \rhd D$ if and only if $D = C \cup \{e\}$, where e is the set of vertices in a strongly connected component in $G = \langle E, \rightarrow \rangle$ and $\{e\} \cap C = \emptyset$. We denote the reflexive closure of this relation by $\unrhd$. A *fullpath* $C_0, C_1, \ldots, C_k = \mathcal{E}$ of the distributive lattice $(\mathcal{C}(G), \subseteq)$ satisfies that for each $0 \leqslant i < k$, $C_i \rhd C_{i+1}$. We will use the notation $|\pi|$ to denote the length of the fullpath π and the notation π^i to denote the consistent cut C_i of fullpath π, provided $i < |\pi|$.

Definition 2.17 (observation, interleaving) *An observation (interleaving) of a computation is a fullpath that starts from the initial consistent cut,* $\pi^0 = \{\bot\}$.

Remark 2.18 *Note that an observation is similar to a linearization of a partial order described in the previous section and there are possibly exponential number of observations of a computation.*

Example 2.19 *In Figure 2.4(a), the set of events* $E = \{\bot, e_1, e_2, e_3, f_1, f_2, f_3, \top\}$ *and the edges* $\rightarrow = \{(\bot, e_1), (e_1, e_2), (e_2, e_3), (e_3, \top),\ (\bot, f_1), (f_1, f_2), (f_2, f_3), (f_3, \top),$

$(f_1, e_1)\}^+$. Figure 2.4(b) contains the lattice of all (nontrivial) consistent cuts of the computation in Figure 2.4(a). We represent a cut by a line drawn from top to bottom passing through exactly one event on each process; an event belongs to the cut if it either lies on the line or lies to the left of the line. The initial consistent cut is $\{\perp\}$ and the final consistent cut $\mathcal{E} = \{\perp, e_1, e_2, e_3, f_1, f_2, f_3\}$. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $C = \{e_3, e_2, e_1, f_2, f_1, \perp\}$ is represented by $frontier(C) = \{e_3, f_2\}$. Observe that $\{e_1, \perp\}$ is not a consistent cut because it depicts a situation where a message from P_2 has been received by P_1, but P_2 has not yet sent the message. Event e_1 is not enabled at $\{\perp\}$, since $\{e_1, \perp\}$ is not a consistent cut. Similarly, f_3 is not enabled at $\{e_1, f_1, \perp\}$, whereas f_2 is enabled at $\{f_1, \perp\}$. Consistent cut $\{e_3, f_3\}$ is reachable from $\{e_1, f_3\}$. Also, $\{e_1, f_3\} \triangleright \{e_2, f_3\}$ and the sequence $\{e_1, f_3\}, \{e_2, f_3\}, \{e_3, f_3\}$ is a fullpath. Finally, the fullpath $\{\perp\}$, $\{f_1\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_2\}$, $\{e_3, f_3\}$ is an interleaving of the computation. In Figure 2.4(c) we depict the computation in Figure 2.4(a) displaying its initial and final events explicitly. In the figure, all initial and final events belong to the same strongly connected component.

Given a computation with the events as in Figure 2.4(a), by using a directed graph model based on a partial order representation, we are able to capture all possible interleavings of events, namely ten in total. One such interleaving is $\{\perp\}$, $\{f_1\}$, $\{e_1, f_1\}$, $\{e_2, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_2\}$, $\{e_3, f_3\}$. If this were the only interleaving observed for the given computation, and if there were an error at the consistent cut $\{e_2, f_3\}$ then this error would not be found from the mentioned interleaving. However, we can find this potential error by using a model based on the partial order representation and capturing all possible interleavings for a given interleaving.

2.4 Representing Partial Order Traces

A mechanism known as *vector clocks* has been used to represent partial order relations on traces. A vector clock assigns timestamps to events such that the partial order relation between events can be determined by using the timestamps.

Definition 2.20 (vector clock) *Given a poset (X, P), a vector clock v is a map from X to $\mathcal{N}^k$ (vectors of natural numbers) such that*

$$\langle \forall\, x, y : x, y \in X : (x, y) \in P \iff x.v < y.v \rangle$$

where $x.v$ is the vector assigned to the element x and $k = width(P)$.

Given two vectors $x.v$ and $y.v$ of dimension N, we compare them as follows.

$$\begin{aligned} x.v \leqslant y.v &\triangleq \langle \forall\, k : 1 \leqslant k \leqslant N : x.v[k] \leqslant y.v[k] \rangle \\ x.v < y.v &\triangleq (x.v \leqslant y.v) \wedge (x.v \neq y.v) \end{aligned}$$

It is clear that this order is only partial for $N \geqslant 2$. A vector clock timestamps each event with a vector of natural numbers. For modeling message passing program traces, a partial order relation known as Lamport's happened-before relation [Lam78] has been used. Lamport's *happened-before relation* is defined as the smallest transitive relation satisfying the following properties:

- if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and
- if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

A well-known vector clock algorithm for message passing programs is Fidge and Mattern algorithm [Fid91, Mat89], which implements Lamport's happened-before relation. For a computation on n processes the dimension of the vector clock

k has been shown to be at least n [GS01]. We will present details of vector clock algorithms for both message passing and shared memory programs in Chapter 7.

Chapter 3

Temporal Logic Predicate Classes

In this chapter, we describe the temporal logic and the predicate classes that we use to specify properties of programs.

3.1 Overview

Many specifications of concurrent and distributed systems are temporal in nature because we are interested in properties related to the sequence of states during an execution rather than just the initial and final states. We can specify both *safety* and *liveness* properties of a system using temporal logic. A safety property specifies that something bad will never happen, whereas a liveness property specifies that something good will eventually happen. For example, the safety property in a mutual exclusion algorithm, “no two processes are in the critical state at the same time”, is a temporal property, as is the liveness property in dining philosophers protocol, “a philosopher, whenever gets hungry, eventually gets to eat”. With the use of temporal logic, we can unambiguously describe the properties of a program and also express the properties that cannot be expressed in traditional testing techniques.

In Section 3.2, we present our temporal logic, which is a subset of branching temporal logic CTL [CE81]. CTL has widely been used as a specification language in model checking. CTL semantics has so far been given for infinite execution traces. However, our trace model leads to a state space which is a finite distributive lattice. In Section 3.2 we give a finite trace semantics for CTL and interpret it on a distributive lattice structure.

Our approach to ameliorate state explosion problem exploits the structure of the predicate itself—by imposing restrictions—to evaluate its value efficiently for a given computation. Some of these predicate classes are conjunctive [GW94, HMRS96], stable [CL85], observer-independent [CBDGF95], linear [CG98], relational [TG97], and non-temporal regular [GM01, MG01a] predicates. In Section 3.3, we formally define these predicate classes.

In Section 3.4, we study the regularity of a predicate p when temporal operators are applied to it. In particular, we prove that several temporal predicates are regular. This result enables us to obtain efficient slicing algorithms for a regular subset of CTL, which we denote by RCTL, and for an extended version of temporal logic, denoted by RCTL+.

3.2 CTL For Finite Execution Traces

We will define the syntax and semantics of the well-known temporal logic CTL for finite execution traces. The other logics considered, RCTL+ and RCTL, are sublogics of CTL, and will be defined as such.

CTL ("Computational Tree Logic") has mostly been used for reasoning about programs rather than execution traces. Furthermore, it has been interpreted with respect to Kripke structures, which have infinite length fullpaths, whereas we interpret it with respect to finite distributive lattices, which have finite length fullpaths.

Propositional temporal logics use a finite set of atomic propositions AP, each one of which represents some property of the consistent cut. A labeling function $\lambda : \mathcal{C}(G) \rightarrow 2^{AP}$ assigns to each consistent cut the set of propositions from AP that hold in it. Given a consistent cut, an atomic proposition is evaluated with respect to the values of variables resulting after executing all events in the cut. The formal syntax of CTL is given below.

- Every proposition $ap \in AP$ is a CTL formula.
- If p and q are CTL formulas, then so are $\neg p$, $p \vee q$, $p \wedge q$, $\mathsf{EF}(p)$, $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, $\mathsf{AF}(p)$, $\mathsf{E}(p \,\mathsf{U}\, q)$, $\mathsf{A}(p \,\mathsf{U}\, q)$, $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, $\mathsf{EX}(p)[j]$.

The symbols $\vee$, $\wedge$ and $\neg$ have their usual meanings. There are two path quantifiers: A denotes *for all fullpaths* and E denotes *for some fullpath.* There are four linear temporal operators: G is the *always* operator, F is the *eventually* operator, U is the *until* operator, and X is the *next-time* operator.

The formula $\mathsf{AG}(p)$ (resp. $\mathsf{EG}(p)$) intuitively means that *for all fullpaths* (resp. *for some fullpath*), p *always* holds on the path. The formula $\mathsf{AF}(p)$ (resp. $\mathsf{EF}(p)$) intuitively means that *for all fullpaths* (resp. *for some fullpath*), p *eventually* holds on the path. The formula $\mathsf{A}(p \,\mathsf{U}\, q)$ (resp. $\mathsf{E}(p \,\mathsf{U}\, q)$) intuitively means that *for all fullpaths* (resp. *for some fullpath*), p holds *until* q holds on the path. The formula $\mathsf{AX}(p)$ (resp. $\mathsf{EX}(p)$) intuitively means that *for all fullpaths* (resp. *for some fullpath*), p holds *next-time* on the path. The formula $\mathsf{EX}(p)[j]$ intuitively means that when process j executes an event, *for some fullpath*, p holds *next-time* on the path.

Given a finite distributive lattice $L = (\mathcal{C}(G), \subseteq)$ of a graph G, the formulas of CTL are interpreted over the consistent cuts in $\mathcal{C}(G)$. We leave the formulas undefined for the trivial consistent cuts. Let p be a CTL formula and C be a consistent cut in $\mathcal{C}(G)$. Then, the satisfaction relation $L, C \models p$ means that predicate

p holds at consistent cut C in lattice $L = (\mathcal{C}(G), \subseteq)$ and is defined inductively below. We denote $C \models p$ as a short form for $L, C \models p$, when L is clear from the context.

- $C \models ap$ iff $ap \in \lambda(C)$ for an atomic proposition ap.
- $C \models \neg p$ iff $C \not\models p$.
- $C \models p \wedge q$ iff $C \models p$ and $C \models q$.
- $C \models p \vee q$ iff either $C \models p$ or $C \models q$.
- $C \models \mathsf{EG}(p)$ iff for some fullpath π starting from C, $\langle \forall\, i : 0 \leqslant i < |\pi| : \pi^i \models p \rangle$.
- $C \models \mathsf{AG}(p)$ iff for all fullpaths π starting from C, $\langle \forall\, i : 0 \leqslant i < |\pi| : \pi^i \models p \rangle$.
- $C \models \mathsf{EF}(p)$ iff for some fullpath π starting from C, $\langle \exists\, i : 0 \leqslant i < |\pi| : \pi^i \models p \rangle$.
- $C \models \mathsf{AF}(p)$ iff for all fullpaths π starting from C, $\langle \exists\, i : 0 \leqslant i < |\pi| : \pi^i \models p \rangle$.
- $C \models \mathsf{E}(p \,\mathsf{U}\, q)$ iff for some fullpath π starting from C, $\langle \exists\, i : 0 \leqslant i < |\pi| : \pi^i \models q \wedge \langle \forall\, j : 0 \leqslant j < i : \pi^j \models p \rangle \rangle$.
- $C \models \mathsf{A}(p \,\mathsf{U}\, q)$ iff for all fullpaths π starting from C, $\langle \exists\, i : 0 \leqslant i < |\pi| : \pi^i \models q \wedge \langle \forall\, j : 0 \leqslant j < i : \pi^j \models p \rangle \rangle$.
- $C \models \mathsf{EX}(p)$ iff for some fullpath π starting from C, $\pi^1 \models p$.
- $C \models \mathsf{AX}(p)$ iff for all fullpaths π starting from C, $\pi^1 \models p$.
- $C \models \mathsf{EX}(p)[j]$ iff for some fullpath π starting from C, $\pi^1 \models p$ such that $\pi^1 = C \cup \{e\}$, where e is an event on process j that is enabled at C.

We do not consider empty fullpaths. Also, next-time operator is defined for fullpaths with length > 1 only.

We define $L \models p$ if and only if $L, \{\bot\} \models p$. By an abuse of notation, we also write $G \models p$ for $L \models p$ when $L = (\mathcal{C}(G), \subseteq)$. We also use the following equivalences in writing CTL formulas. $\mathsf{EG}(p)$ is equivalent to $\neg\mathsf{AF}(\neg p)$, $\mathsf{AG}(p)$ is equivalent to $\neg\mathsf{EF}(\neg p)$, $\mathsf{EF}(p)$ is equivalent to $\neg\mathsf{AG}(\neg p)$, and $\mathsf{AF}(p)$ is equivalent to $\neg\mathsf{EG}(\neg p)$.

For convenience, we define a *temporal predicate* as a predicate that contains temporal operators such as EG, AG, EF, AF, EU, AU, EX, AX, or $\mathsf{EX}(p)[j]$. We define a *non-temporal predicate* as a predicate that does not contain any temporal operator. For example, an atomic proposition is a non-temporal predicate. We say that a temporal predicate is *nested* if it contains > 1 level of nesting of temporal operators such as $\mathsf{AG}(\mathsf{EF}(p))$. We say that a predicate is *unnested* if it contains a single temporal operator and is of the form $\mathsf{EF}(p)$, $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, $\mathsf{AF}(p)$, $\mathsf{E}(p \,\mathsf{U}\, q)$, $\mathsf{A}(p \,\mathsf{U}\, q)$, $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, $\mathsf{EX}(p)[j]$ where p is non-temporal. The temporal operators EG, AG, EF, and AF have also been referred to as *controllable*, *invariant*, *possibly* and *definitely*, respectively [Gar02].

Example 3.1 *Figure 3.1 illustrates* CTL *operators on a distributive lattice. Some typical CTL formulas that may arise for specifying properties of concurrent and distributed systems are given below:*

- $\mathsf{EF}(critical_i \wedge critical_j)$ *intuitively means that "it is possible to get to a global state where processes i and j are in their critical state".*

- $\mathsf{AG}(request \Rightarrow \mathsf{AF}(acknowledge))$ *intuitively means that "whenever request occurs, it will be eventually acknowledged".*

- *The complement of the previous formula is* $\mathsf{EF}(request \wedge \mathsf{EG}(\neg acknowledge))$.

- $\mathsf{AG}(\mathsf{EF}(reset))$ *intuitively means that "reset is possible from every global state".*

- $\mathsf{AG}(request \Rightarrow \mathsf{A}(request \,\mathsf{U}\, acknowledge))$ *intuitively means that "if a request occurs then it continues to hold until it is eventually acknowledged".*

Sublogics of CTL are defined by restrictions on the operators allowed, and by restrictions on the classes of atomic propositions.

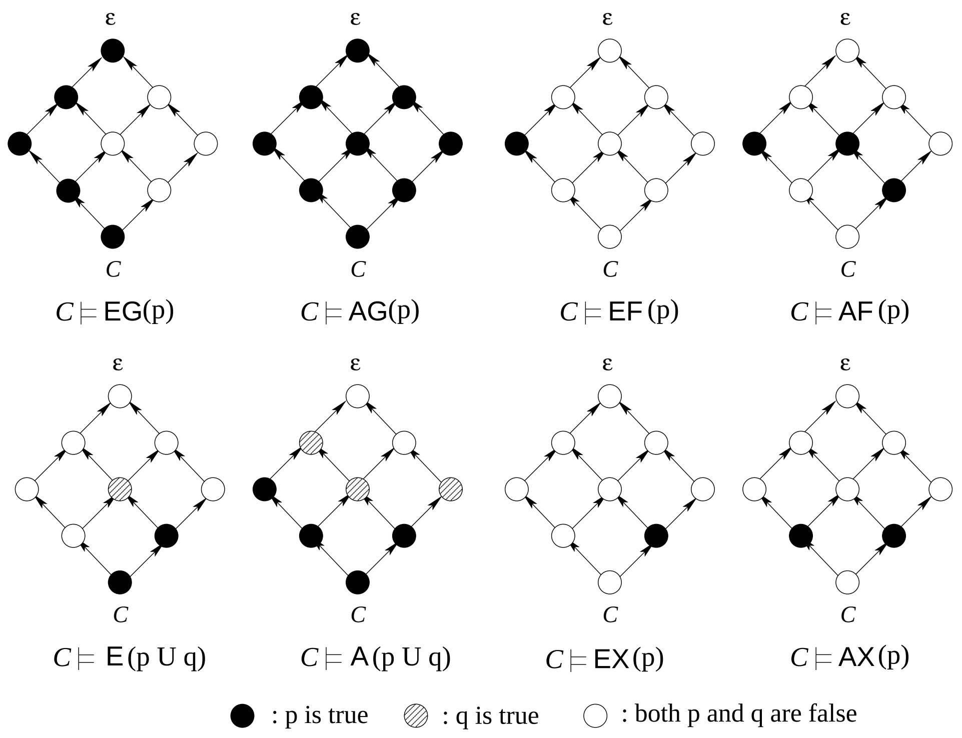


Figure 3.1: Basic CTL operators

- RCTL+ is the subset of CTL where atomic propositions are regular, co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates, which we will define in the next section, boolean operators are $\vee$ and $\wedge$, and temporal operators are EF, AG, EG, EX$[j]$, and EX.

- RCTL is the subset of RCTL+ where atomic propositions are regular predicates and both disjunction and EX operators are left out.

The *predicate detection* problem is to decide whether the initial consistent cut of a computation satisfies a predicate. More formally,

Definition 3.2 (predicate detection) *Given a distributive lattice $L = (\mathcal{C}(G), \subseteq)$ that represents a computation $G = \langle E, \rightarrow \rangle$ and a temporal logic predicate p expressing some desired specification, decide whether $L, \{\bot\} \models p$ holds or not.*

Remark 3.3 *Observe that our definition of predicate detection is similar to that of model checking [CE81, QS82, CGP00]. However, we consider execution traces rather than programs and we interpret specifications on a finite distributive lattice.*

3.3 Predicate Classes

Our approach to predicate detection is based on exploiting the structure of the predicate. We first give definitions of several widely used predicate classes and then present relationship among these classes.

Definition 3.4 (local predicate) *A predicate is* local *predicate if its truth value depends only on the variables of a single process.*

For example, "the value of x on process i is 2" is a local predicate.

Definition 3.5 (conjunctive predicate) *A predicate p is conjunctive if it can be written as a conjunction of local predicates.*

For example, in a mutual-exclusion algorithm between two processes, where cs_i represents the local predicate that process P_i is in the critical section, $cs_1 \wedge cs_2$ represents whether both processes are in the critical section.

Definition 3.6 (disjunctive predicate) *A predicate p is disjunctive if it can be written as a disjunction of local predicates.*

For example, in a dining philosophers protocol, "at least one philosopher is thinking" could be represented by $think_1 \vee think_2 \vee, \ldots, \vee think_n$ where $think_i$ represents the local predicate that philosopher P_i is thinking.

Definition 3.7 (stable predicate [CL85]) *A predicate p is stable if the predicate remains true, once it becomes true. Formally, for all consistent cuts C, D of a computation G:*

$$C \models p \ \wedge \ (C \subseteq D) \ \Rightarrow \ D \models p$$

For example, termination property is a stable predicate because once a system has terminated it will stay terminated. Deadlock is another example of a stable predicate.

Definition 3.8 (observer-independent predicate [CBDGF95]) *A predicate p is observer-independent, if p holds for some observation then p holds for all observations of a computation.*

Note that if p holds in the initial consistent cut then it is an observer-independent predicate because the initial consistent cut belongs to all observations.

Next we will define the class of meet-closed predicates. These predicates are useful because they allow us to compute the least consistent cut that satisfies a given predicate.

Definition 3.9 (meet-closed predicates) *A predicate p is meet-closed if for all consistent cuts C, D of a computation G:*

$$C \models p \ \wedge \ D \models p \ \Rightarrow \ (C \sqcap D) \models p$$

For example, the predicate "does not contain x" in the Boolean lattice generated by all subsets of $\{x, y, z\}$ is meet-closed whereas the predicate "has size k" is not. Any conjunctive predicate is meet-closed.

It follows from the definition that if there exists any consistent cut that satisfies a meet-closed predicate p, then there exists the least one. Note that the predicate *false* which corresponds to the empty subset and the predicate *true* which corresponds to the entire set of consistent cuts are meet-closed predicates. We now give another characterization of meet-closed predicates that will be useful for computing the least consistent cut that satisfies the predicate. To this end, we first define the notion of a crucial event for a consistent cut.

Definition 3.10 (crucial element [CG98, GMS03]) *For a consistent cut $C \subsetneqq \mathcal{E}$ and a predicate p, we define $e \in \mathcal{E} - C$ to be crucial for C as:*

$$crucial(C, e, p) \;\triangleq\; \langle \forall\, D : D \supseteq C : (e \in D) \;\vee\; D \not\models p \rangle$$

Definition 3.11 (linear predicates [CG98]) *A predicate p is linear if for all consistent cuts $C \subsetneqq \mathcal{E}$,*

$$C \not\models p \;\Rightarrow\; \langle \exists\, e : e \in \mathcal{E} - C : crucial(C, e, p) \rangle$$

Intuitively, this means that any consistent cut D, that is at least C, cannot satisfy the predicate unless it contains e. Now, we have

Theorem 3.12 ([CG98]) *A predicate p is linear if and only if it is meet-closed.*

Example 3.13 *Consider the Boolean lattice generated by all subsets of $\{1, ..., n\}$. Let the predicate p defined to be true on a consistent cut C as "If C contains any odd $i < n$, then it also contains $i + 1$." It is easy to verify that p is meet-closed. Given any C for which p does not hold, the crucial elements consist of*

$$\{i | i \textit{ is even},\; 2 \leqslant i \leqslant n, i - 1 \in C, i \notin C\}$$

Example 3.14 *Consider a computation on two processes P_1 and P_2 and the predicate p to be true on a consistent cut if both the processes are in the critical section. Given any consistent cut C for which p does not hold, there are two cases; either P_1 is not in the critical section, or P_2 is not in the critical section. Let e be the projection of events from frontier(C) onto the events on P_1. Let g be the event on P_1 that occurs after e, that is, $e \xrightarrow{P_1} g$, and where P_1 enters the critical section. In the first case, event g is crucial. Similarly, we can find the crucial event in the second case. This example can be easily generalized to any conjunctive predicate.*

Our interest is in detecting whether there exists a consistent cut that satisfies a given predicate p. We make two assumptions for this purpose:

Property 3.15 (Efficient Predicate Evaluation) *Given a consistent cut C and a predicate p there exists an efficient (polynomial-time) function to determine whether p is true for C or not.*

Remark 3.16 *We assume that the time-complexity for the above property is polynomial in the input size. However, for convenience, throughout this dissertation, for non-temporal predicates, we specify the time-complexity of our algorithms assuming that the time-complexity of the above property is linear in the number of processes whose variables the predicate depends on. In case the time-complexity is higher, the time-complexity of the algorithms will increase correspondingly. However, we will not assume the above property for temporal predicates.*

On account of linearity of p, if p is evaluated to be false in some consistent cut C, then we know that there exists a crucial event in $\mathcal{E} - C$. We make an additional assumption:

Property 3.17 (Efficient Advancement) *There exists an efficient (polynomial-time) function to determine the crucial event.*

Remark 3.18 *We will not assume the above property for temporal predicates.*

We now have

Theorem 3.19 ([CG98, GMS03]) *If p is a linear predicate with the efficient advancement property, then there exists an efficient algorithm to determine the least consistent cut that satisfies p (if any).*

Proof: An efficient algorithm to find the *least* cut in which p is true is given in Figure 3.2. We search for the least consistent cut starting from the *initial* consistent cut. If the predicate is false in the consistent cut, which we determine using the efficient predicate evaluation property, then we find the crucial element using the

```
Algorithm Algo$_{3.1}$:

    Input: (1) a computation $G = \langle E, \rightarrow\rangle$, and (2) a linear predicate $p$

    Output: whether $p$ is satisfied or not

1   $C$: consistent cut initially $C := \{\perp\}$;
2   while $((C \not\models p) \wedge (C \neq \mathcal{E}))$ do
3       Let $e$ be such that $crucial(C, e, p)$ in $\langle E, \rightarrow\rangle$;
4       $C := C \cup \{e\}$;
    endwhile;
5   if $C \models p$ then
6       return true;
    else
7       return false;
    endif;
```

Figure 3.2: An efficient algorithm to detect a linear predicate

efficient advancement property and then repeat the procedure. If this is the last event on the process, then we return false else we advance along the process that has the crucial event. □

Complexity Analysis 3.20 *Each iteration of the while loop at line* 2 *has* $O(n)$ *time-complexity assuming the efficient property evaluation for determining* $C \not\models p$ *and efficient advancement property for determining* $crucial(C, e, p)$*. Thus the time-complexity of the algorithm* Algo$_{3.1}$ *is* $O(n|E|)$*.*

Assuming that both $C \models p$ and $crucial(C, e, p)$ can be evaluated efficiently for a given computation, we can determine the least consistent cut that satisfies p efficiently even though the number of consistent cuts may be exponentially larger than the size of the computation.

Remark 3.21 *In practice, most meet-closed predicates p satisfy the efficient advancement property. All the non-temporal meet-closed predicates in this dissertation* *satisfy the efficient advancement property.*

So far we have focused on meet-closed predicates. All the definitions and ideas carry over to join-closed predicates. If the predicate p is join-closed, then one can search for the largest consistent cut that satisfies p in a fashion analogous to finding the least consistent cut when it is meet-closed.

Predicates that are both meet-closed and join-closed are called regular predicates.

Definition 3.22 (regular predicates [GM01]) *A predicate is regular if the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts. Equivalently, a predicate p is* regular *with respect to P if it is closed under $\sqcup$ and $\sqcap$, i.e., for all consistent cuts C, D of a computation G:*

$$C \models p \ \wedge \ D \models p \ \Rightarrow \ (C \sqcup D) \models p \ \wedge \ (C \sqcap D) \models p$$

Some examples of regular predicates are:

- Consider the predicate p as "there is no outstanding message in the channel." We show that this predicate is regular. Observe that p holds on a consistent cut C if only if for all send events in C the corresponding receive events are also in C. It is easy to see that if $C \models p$ and $D \models p$, then $(C \cup D) \models p$. To see that it holds for $C \cap D$, let e be any send event in $C \cap D$. Let f be the receive event corresponding to e. From $C \models p$, we get that $f \in C$ and from $D \models p$, we get that $f \in D$. Thus $f \in C \cap D$. Hence, $(C \cap D) \models p$. Similarly, the following predicates are also regular.

  - There is no token message in transit.
  - No token message is in transit between processes P_1 and P_5.
  - Every "request" message has been "acknowledged" in the system.

- Any local predicate is regular. Thus the following predicates are regular.
  - The leader has sent all "prepare to commit" messages.
  - Process P_i is in a "red" state.
- Channel predicates such as "there are at most k messages in transit from P_i to P_j" and "there are at least k messages in transit from P_i to P_j" are also regular.

It is easy to verify that the class of regular predicates is closed under conjunction. The closure under conjunction implies that the following predicates are also regular:

- No process has the token, and no channel has the token.
- Any conjunction of local predicates.

The closure is not true for disjunction as can be readily verified by taking p_1 and p_2 to be local predicates. Similarly, the closure is not true for negation. For convenience, we denote the complement of a predicate from a class A by *co-A*. For example, we denote the complement of a regular predicate by a *co-regular* predicate. Similarly, we denote the complement of a linear predicate by a *co-linear* predicate and the complement of a stable predicate by a *co-stable* predicate.

Figure 3.3 displays a classification of the above predicate classes. We also give proofs of these relations in [SG01]. Note that linear predicates include regular predicates and regular predicates include conjunctive predicates. Similarly, observer-independent predicates include stable and disjunctive predicates. Also note that, linear predicates are not necessarily disjoint from observer-independent predicates, for example, termination or a local predicate are both linear and observer-independent.

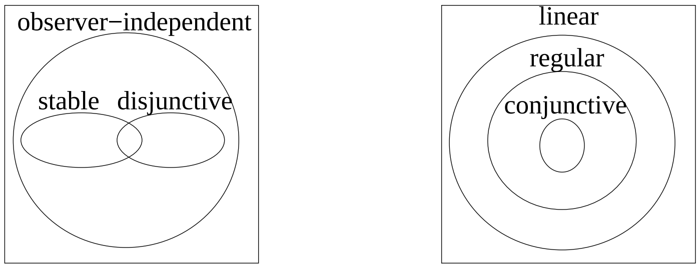


Figure 3.3: Predicate classes

3.4 Temporal Regular Predicates

In this section, we study the regularity of a predicate p when temporal operators are applied to it. For example, given a predicate p, applying the temporal operator EF to p gives the predicate $\mathsf{EF}(p)$. We prove that temporal predicates $\mathsf{EF}(p)$, $\mathsf{AG}(p)$, $\mathsf{EG}(p)$, and $\mathsf{EX}(p)[j]$ are regular when p is regular. This result shows that regular predicates are closed under EF, AG, EG, and $\mathsf{EX}[j]$ temporal operators in addition to the boolean conjunction operator. Our temporal logic RCTL is generated by exactly these five operators. However, the regularity does not follow in the case of AF, EU, AU, EX, and AX. The results of this section enable us to compute efficient slicing algorithms in the following chapters.

Given a temporal predicate p, to prove that $\mathsf{EF}(p)$ is regular, we show that for all consistent cuts C, D if both C and D satisfy $\mathsf{EF}(p)$ then both $(C \cap D)$ and $(C \cup D)$ satisfy $\mathsf{EF}(p)$.

3.4.1 Proof of $\mathsf{EF}(p)$ is regular

Lemma 3.23 *If p is a regular predicate then $\mathsf{EF}(p)$ is a regular predicate.*
Proof:

$$\begin{array}{ll} & C \models \mathsf{EF}(p) \wedge D \models \mathsf{EF}(p) \\ \equiv & \{ \text{ definition of } \mathsf{EF}(p) \ \} \end{array}$$

$$\begin{array}{ll}
 & \langle \exists\, C' : C \subseteq C' : C' \models p\rangle \wedge \langle \exists\, D' : D \subseteq D' : D' \models p\rangle \\
\equiv & \{ \text{ rewriting } \} \\
 & \langle \exists\, C', D' : C \subseteq C' \wedge D \subseteq D' : C' \models p \wedge D' \models p\rangle \\
\Rightarrow & \{ \text{ Let } W = C' \cup D' \text{ and use definition of regular predicates } \} \\
 & \langle \exists\, W : C \subseteq W \wedge D \subseteq W : W \models p\rangle \\
\equiv & \{ \text{ set theory } \} \\
 & \langle \exists\, W : (C \cap D) \subseteq W \wedge (C \cup D) \subseteq W : W \models p\rangle \\
\equiv & \{ \text{ definition of } \mathsf{EF}(p) \ \} \\
 & (C \cap D) \models \mathsf{EF}(p) \wedge (C \cup D) \models \mathsf{EF}(p)
\end{array}$$

This establishes the lemma. Note that we only use the join-closedness of predicate p. □

3.4.2 Proof of $\mathsf{AG}(p)$ is regular

Lemma 3.24 *If p is a regular predicate then $\mathsf{AG}(p)$ is a regular predicate.*

Proof:

$$\begin{array}{ll}
 & C \models \mathsf{AG}(p) \\
\equiv & \{ \text{ definition of } \mathsf{AG}(p) \ \} \\
 & \langle \forall\, C' : C \subseteq C' : C' \models p\rangle \\
\Rightarrow & \{ \ (C \cup F) \subseteq C' \ \Rightarrow \ C \subseteq C' \ \} \\
 & \langle \forall\, C', F : (C \cup F) \subseteq C' : C' \models p)\rangle \\
\equiv & \{ \text{ definition of } \mathsf{AG}(p) \ \} \\
 & \langle \forall\, F :: (C \cup F) \models \mathsf{AG}(p)\rangle \\
\Rightarrow & \{ \text{ instantiating for } F \ \} \\
 & (C \cup D) \models \mathsf{AG}(p)
\end{array}$$

This shows that AG is a monotonic (stable) operator, i.e.,

$$C \models \mathsf{AG}(p) \wedge C \subseteq C' \ \Rightarrow \ C' \models \mathsf{AG}(p)$$

We have not used regularity of p in this part of the proof.

$C \models \mathsf{AG}(p) \wedge D \models \mathsf{AG}(p)$

$\Rightarrow$ { monotonicity of AG }

$\langle \forall\, F :: (C \cup F) \models \mathsf{AG}(p) \wedge (D \cup F) \models \mathsf{AG}(p) \rangle$

$\Rightarrow$ { $\mathsf{AG}(p) \Rightarrow p$ }

$\langle \forall\, F :: (C \cup F) \models p \wedge (D \cup F) \models p \rangle$

$\Rightarrow$ { p is closed under meet }

$\langle \forall\, F :: (C \cup F) \cap (D \cup F) \models p \rangle$

$\Rightarrow$ { distributivity }

$\langle \forall\, F :: (C \cap D) \cup F \models p \rangle$

$\equiv$ { definition of AG(p) }

$(C \cap D) \models \mathsf{AG}(p)$

Note that we only use the meet-closedness of predicate p in this part.

□

3.4.3 Proof of $\mathsf{EG}(p)$ is regular

Lemma 3.25 *If p is a regular predicate then $\mathsf{EG}(p)$ is a regular predicate.*

Proof:

$C \models \mathsf{EG}(p) \wedge D \models \mathsf{EG}(p)$

$\equiv$ { definition of $\mathsf{EG}(p)$ }

$\langle \exists\, \pi_0 : (\pi_0^0 = C) : \langle \forall\, i : 0 \leqslant i < |\pi_0| : \pi_0^i \models p \rangle \rangle$

$\langle \exists\, \pi_1 : (\pi_1^0 = D) : \langle \forall\, j : 0 \leqslant j < |\pi_1| : \pi_1^j \models p \rangle \rangle$

$\Rightarrow$ { definition of regular predicates }

There exists a sequence

$(\pi_0^0 \cup \pi_1^0) = (C \cup D) \trianglerighteq (\pi_0^0 \cup \pi_1^1) \trianglerighteq \ldots \trianglerighteq (\pi_0^0 \cup \pi_1^{|\pi_1|-1}) = \mathcal{E}$

such that $(\pi_0^0 \cup \pi_1^0) \models p \wedge (\pi_0^0 \cup \pi_1^1) \models p \wedge \ldots \wedge (\pi_0^0 \cup \pi_1^{|\pi_1|-1}) \models p$

$\equiv$ { eliminating stuttering consistent cuts, we obtain fullpath π_2 }

$\langle \exists\, \pi_2 : (\pi_2^0 = \pi_0^0 \cup \pi_1^0) \ldots (\pi_2^{|\pi_2|-1} = \pi_0^0 \cup \pi_1^{|\pi_1|-1}) :$
$\langle \forall\, j : 0 \leqslant j < |\pi_2| : \pi_2^j \models p \rangle\rangle$

$\equiv$ { rewriting }
$\langle \exists\, \pi_2 : (\pi_2^0 = (C \cup D)) : \langle \forall\, j : 0 \leqslant j < |\pi_2| : \pi_2^j \models p \rangle\rangle$

$\equiv$ { definition of $\mathsf{EG}(p)$ }
$(C \cup D) \models \mathsf{EG}(p)$

$C \models \mathsf{EG}(p) \wedge D \models \mathsf{EG}(p)$

$\equiv$ { definition of $\mathsf{EG}(p)$ }
$\langle \exists\, \pi_0 : (\pi_0^0 = C) : \langle \forall\, i : 0 \leqslant i < |\pi_0| : \pi_0^i \models p \rangle\rangle\ \wedge$
$\langle \exists\, \pi_1 : (\pi_1^0 = D) : \langle \forall\, j : 0 \leqslant j < |\pi_1| : \pi_1^j \models p \rangle\rangle$

$\Rightarrow$ { definition of regular predicates }
There exists a sequence
$(\pi_0^0 \cap \pi_1^0) = (C \cap D) \trianglerighteq (\pi_0^0 \cap \pi_1^1) \trianglerighteq \ldots \trianglerighteq (\pi_0^0 \cap \pi_1^k) \trianglerighteq \pi_0^0 \triangleright \pi_0^1 \triangleright \ldots \triangleright \pi_0^{|\pi_0|-1} = \mathcal{E}$
such that $(\pi_0^0 \cap \pi_1^0) \models p \wedge (\pi_0^0 \cap \pi_1^1) \models p \wedge \ldots \wedge (\pi_0^0 \cap \pi_1^k) \models p$

$\equiv$ { eliminating stuttering consistent cuts, we obtain fullpath π_3 }
$\langle \exists\, \pi_3 : (\pi_3^0 = \pi_0^0 \cap \pi_1^0) \ldots (\pi_2^{|\pi_3|-1} = \pi_0^{|\pi_0|-1}) :$
$\langle \forall\, j : 0 \leqslant j < |\pi_3| : \pi_3^j \models p \rangle\rangle$

$\equiv$ { rewriting }
$\langle \exists\, \pi_3 : (\pi_3^0 = (C \cap D)) : \langle \forall\, j : 0 \leqslant j < |\pi_3| : \pi_3^j \models p \rangle\rangle$

$\equiv$ { definition of $\mathsf{EG}(p)$ }
$(C \cap D) \models \mathsf{EG}(p)$

$\square$

3.4.4 Proof of $\mathsf{EX}(p)[j]$ is regular

We prove in the next section that $\mathsf{EX}(p)$ is not regular, in general. However, for a special case of $\mathsf{EX}(p)$, that is $\mathsf{EX}(p)[j]$, we obtain regularity.

The following lemma states that $\mathsf{EX}(p)$ is equivalent to the disjunction of

$\mathsf{EX}(p)[j]$ for all processes j. This result follows directly from the definitions of $\mathsf{EX}(p)$ and our computation model.

Lemma 3.26 $\mathsf{EX}(p)$ *iff there exists a process* j *such that* $\mathsf{EX}(p)[j]$.

Lemma 3.27 *Given a process* j*, if* p *is a regular predicate then* $\mathsf{EX}(p)[j]$ *is a regular predicate.*

Proof:

$$
\begin{aligned}
& C \models \mathsf{EX}(p)[j] \ \wedge \ D \models \mathsf{EX}(p)[j] \\
\equiv \ & \{\text{Let } e \text{ and } f \text{ be events on process } j \text{ and be enabled at cuts } C \text{ and } D, \\
& \text{respectively, and use definition of } \mathsf{EX}(p)[j] \ \} \\
& \langle \exists\, C' : C' = (C \cup \{e\}) : C' \models p \rangle \wedge \langle \exists\, D' : D' = (D \cup \{f\}) : D' \models p \rangle \\
\equiv \ & \{ \text{ rewriting } \} \\
& \langle \exists\, C', D' : C' = C \cup \{e\} \wedge D' = D \cup \{f\} : C' \models p \wedge D' \models p \rangle \\
\Rightarrow \ & \{ \text{ Let } W = C' \cup D',\ V = C' \cap D' \text{ and use definition of regular predicates } \} \\
& \langle \exists\, W : W = (C \cup \{e\}) \cup (D \cup \{f\}) : W \models p \rangle \wedge \\
& \langle \exists\, V : V = (C \cup \{e\}) \cap (D \cup \{f\}) : V \models p \rangle \\
\equiv \ & \{ \text{ Assume } e \overset{P}{\rightarrow} f \text{ on process } j \ \} \\
& \langle \exists\, W : W = (C \cup D) \cup \{f\} : W \models p \rangle \wedge \\
& \langle \exists\, V : V = (C \cap D) \cup \{e\} : V \models p \rangle \\
\equiv \ & \{ \text{ definition of } \mathsf{EX}(p) \ \} \\
& (C \cap D) \models \mathsf{EX}(p) \ \wedge \ (C \cup D) \models \mathsf{EX}(p)
\end{aligned}
$$

This establishes the lemma. □

3.4.5 Proof of $\mathsf{AF}(p)$ is closed under union

We prove in the next section that $\mathsf{AF}(p)$ is not regular, in general. However, it is closed under union, from which we infer that there exists a greatest consistent cut that satisfies $\mathsf{AF}(p)$.

Lemma 3.28 *If the set of consistent cuts that satisfy p is closed under union then the set of consistent cuts that satisfy* $\mathsf{AF}(p)$ *is closed under union (join-closed).*

Proof:

$$
\begin{array}{ll}
 & C \models \mathsf{AF}(p) \wedge D \models \mathsf{AF}(p) \\
\equiv & \{ \text{ definition of } \mathsf{AF}(p) \text{ } \} \\
 & \langle \forall\, \pi_0 : \pi_0^0 = C : \langle \exists\, i : 0 \leqslant i < |\pi_0| : \pi_0^i \models p \rangle\rangle \wedge \\
 & \langle \forall\, \pi_1 : \pi_1^0 = D : \langle \exists\, j : 0 \leqslant j < |\pi_1| : \pi_1^j \models p \rangle\rangle \\
\Rightarrow & \{ \text{ predicate calculus, range weakening } \} \\
 & \langle \forall\, \pi_0 : \pi_0^0 = C \wedge \langle \exists\, k : k \geqslant 0 : \pi_0^k = (C \cup D)\rangle : \langle \exists\, i : 0 \leqslant i < |\pi_0| : \pi_0^i \models p \rangle\rangle \wedge \\
 & \langle \forall\, \pi_1 : \pi_1^0 = D \wedge \langle \exists\, m : m \geqslant 0 : \pi_1^m = (C \cup D)\rangle : \langle \exists\, j : 0 \leqslant j < |\pi_1| : \pi_1^j \models p \rangle\rangle
\end{array}
$$

Our goal is to show that for each full path $(C \cup D), \ldots, \mathcal{E}$, there exists a consistent cut in the path that satisfies p. We have,

Case 1: $i > k$

$$
\begin{array}{ll}
 & \langle \forall\, \pi_0 : \pi_0^0 = C \wedge \langle \exists\, k : k \geqslant 0 : \pi_0^k = (C \cup D)\rangle : \langle \exists\, i : 0 \leqslant i < |\pi_0| : \pi_0^i \models p \rangle\rangle \\
\Rightarrow & \{ \text{ Let } g = i - k \text{ and start } \pi_0 \text{ from } C \cup D \text{ } \} \\
 & \langle \forall\, \pi_0 : \pi_0^0 = (C \cup D) : \langle \exists\, g : 0 \leqslant g < |\pi_0| : \pi_0^g \models p \rangle\rangle
\end{array}
$$

Case 2: $j > m$. Similar to Case 1 by symmetry between C and D.

Case 3: $i \leqslant k \wedge j \leqslant m$

$$
\begin{array}{ll}
 & \{ \text{ from the definition of } \mathsf{AF}(p) \text{ above } \} \\
 & (C \subseteq \pi_0^i) \wedge (D \subseteq \pi_1^j) \\
\Rightarrow & \{ \text{ set theory } \} \\
 & (C \cup D) \subseteq (\pi_0^i \cup \pi_1^j) \\
\Rightarrow & \{ \text{ Case 3 } \} \\
 & ((C \cup D) \subseteq (\pi_0^i \cup \pi_1^j)) \wedge ((\pi_0^i \cup \pi_1^j) \subseteq (\pi_0^k \cup \pi_1^m)) \\
\equiv & \{ \ \pi_0^k = \pi_1^m = C \cup D \ \} \\
 & ((C \cup D) \subseteq (\pi_0^i \cup \pi_1^j)) \wedge ((\pi_0^i \cup \pi_1^j) \subseteq (C \cup D)) \\
\equiv & \{ \text{ set theory } \}
\end{array}
$$

$((C \cup D) = (\pi_0^i \cup \pi_1^j))$

$\Rightarrow$ { p is closed under join }

$(C \cup D) \models p$

$\Rightarrow$ { $p \Rightarrow \mathsf{AF}(p)$ }

$(C \cup D) \models \mathsf{AF}(p)$

□

3.5 Temporal Non-Regular Predicates

These are temporal predicates which are not regular. We will present examples of distributive lattices of consistent cuts to show that the regularity does not follow in the case of $\mathsf{AF}(p)$, $\mathsf{A}(p \,\mathsf{U}\, q)$, $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, and $\mathsf{E}(p \,\mathsf{U}\, q)$ when p and q are regular.

In the case of $\mathsf{AF}(p)$, the set of consistent cuts that satisfy the temporal predicate $\mathsf{AF}(p)$ is closed under union but not under intersection. In Figure 3.4(a) consistent cuts C and D satisfy $\mathsf{AF}(p)$, but their intersection $(C \cap D)$ does not. This is because there exists a path starting from $(C \cap D)$ and ending at the final cut $\mathcal{E}$ where p never holds on the path.

In the case of $\mathsf{A}(p \,\mathsf{U}\, q)$, the set of consistent cuts that satisfy the temporal predicate $\mathsf{A}(p \,\mathsf{U}\, q)$ is also closed under union. The proof is similar to that of $\mathsf{AF}(p)$ proof above. The set of consistent cuts that satisfy the temporal predicate $\mathsf{A}(p \,\mathsf{U}\, q)$ is not closed under intersection. In Figure 3.4(b) consistent cuts V and W satisfy $\mathsf{A}(p \,\mathsf{U}\, q)$, but their intersection $(V \cap W)$ does not.

Figure 3.5 displays a distributive lattice of consistent cuts where p and q are closed under union and intersection. However, none of $\mathsf{EX}(p), \mathsf{AX}(p)$, and $\mathsf{E}(p \,\mathsf{U}\, q)$ is closed under union or intersection. Specifically, consistent cuts V and W satisfy $\mathsf{EX}(p)$ and $\mathsf{AX}(p)$, but their intersection $(V \cap W)$ and union $(V \cup W)$ do not. Similarly, consistent cuts C and D satisfy $\mathsf{E}(p \,\mathsf{U}\, q)$, but $(C \cap D)$ and $(C \cup D)$ do not.

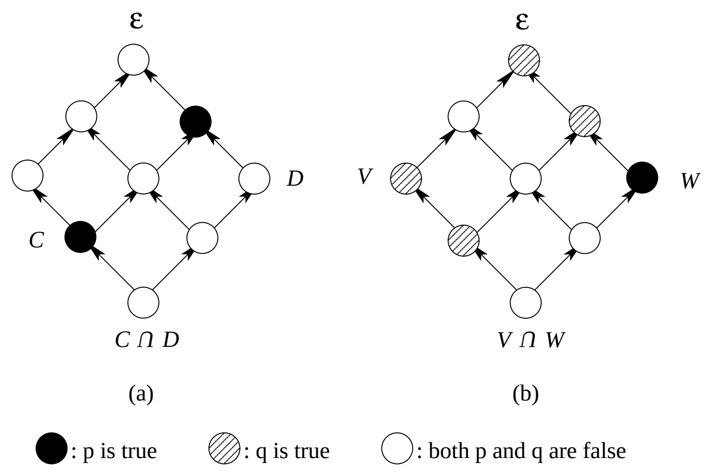


Figure 3.4: (a) $\mathsf{AF}(p)$ may not be regular, (b) $\mathsf{A}(p \,\mathsf{U}\, q)$ may not be regular, when p is regular

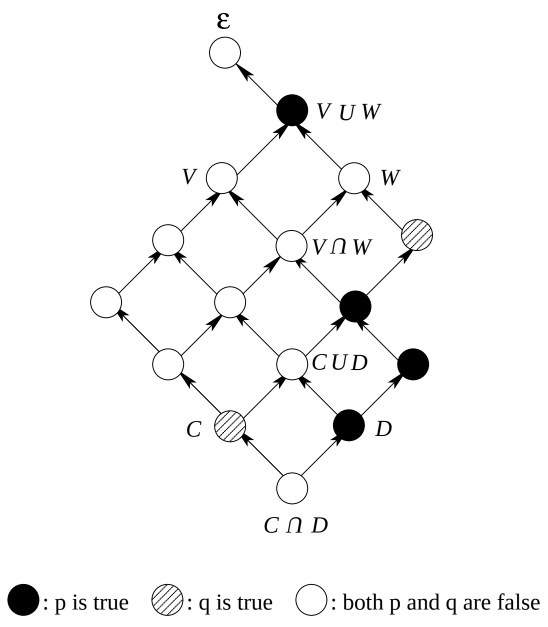


Figure 3.5: $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, and $\mathsf{E}(p \,\mathsf{U}\, q)$ may not be regular when p and q are regular

3.6 Summary

Here we summarize the list of predicate classes that we commonly refer to in this dissertation.

Table 3.1: Predicate Classes

Predicate Class	Description	Example p, q regular
Regular	set of consistent cuts are closed under set union and intersection	$p \wedge q$
Non-Regular	not regular	$p \vee q$, $\neg p$
Temporal	contains temporal operators	$\mathsf{EX}(p)$
Non-Temporal	does not contain temporal operators	p
Nested Temporal	contains nested temporal operators	$\mathsf{EF}(p \wedge \mathsf{EG}(q))$
Unnested Temporal	contains a single temporal operator	$\mathsf{EG}(p), \mathsf{AF}(p)$
Temporal Regular	temporal operators are EF, AG, EG, $\mathsf{EX}[j]$ and atomic propositions are regular	$\mathsf{EF}(p)$, $\mathsf{AG}(p)$, $\mathsf{EG}(p)$, $\mathsf{EX}(p)[j]$
Temporal Non-Regular	contains a temporal operator and is not regular	$\mathsf{AF}(p)$, $\mathsf{EX}(p)$
RCTL+	atomic propositions are regular, co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates, boolean operators are $\vee$ and $\wedge$, and temporal operators are EF, AG, EG, $\mathsf{EX}[j]$, and EX	all of the above
RCTL	subset of RCTL+ where atomic propositions are regular predicates and both $\vee$ and EX operators are left out	

Chapter 4

Slicing for Non-Temporal Predicates

In this chapter, we discuss our results in computation slicing which we will use for developing predicate detection algorithms in Chapter 6.

4.1 Overview

We first give a background on "computation slicing" in Section 4.2. *Computation slicing* was introduced in [GM01, MG01a] as an abstraction technique for analyzing traces of distributed programs, that is, computations. A *computation slice*, defined with respect to a global predicate, is the computation with the least number of consistent cuts that contains all consistent cuts of the original computation for which the predicate evaluates to true. Note that the set of consistent cuts that satisfy a predicate may be large, so one could not simply enumerate all the states efficiently either in space or time.

In Section 4.3, we present efficient computation slicing algorithms with respect to non-temporal predicates for which there are efficient detection algorithms under EF operator of $\mathsf{RCTL+}$. The predicate detection problem under EF operator is only concerned with answering the question whether there exists *at least one* consistent cut of the computation that satisfies the given predicate. Computation slicing, on the other hand, is concerned with computing a succinct representation of *all* consistent cuts of the computation for which the given predicate evaluates to true. Clearly, detecting a predicate under EF operator is no harder than computing its slice in the sense that the predicate detection problem under EF operator can be easily solved given the slice for the predicate (it suffices to test for the emptiness of the slice). In this chapter, we prove a somewhat surprising result that detecting a predicate under EF operator is no easier than computing its slice. In other words, given an algorithm A for detecting a predicate $\mathsf{EF}(p)$, there exists an algorithm B for computing the slice for p such that the time-complexity of B is at most a small multiple of the time-complexity of A. In particular, the multiple is $n|E|$, where n is the number of processes and E is the set of events. As a corollary, it can be derived that there exists a polynomial-time algorithm for detecting a predicate under EF operator if and only if there exists a polynomial-time algorithm for computing its slice.

Mittal and Garg [MG01a] has shown that it is intractable in general to compute the slice for an arbitrary predicate and present efficient slicing algorithms for non-temporal regular and linear predicates [GM01, MG01a, MG03]. Using the above equivalence, it is now possible to compute the slice efficiently for many more classes of non-temporal predicates including stable, co-stable, observer-independent, relational, and co-linear predicates.

If predicate detection under EF operator is "equivalent" to computation slicing, then how can slicing be used to improve the complexity of predicate detection in general? Slicing can indeed be used to facilitate predicate detection as illustrated by the following example. Consider a predicate p that is a conjunction of two clauses p_1 and p_2. Now, assume that p_1 is such that it can be detected efficiently under EF operator but p_2 has no structural property that can be exploited for efficient detection. To detect $\mathsf{EF}(p)$, without computation slicing, we are forced to use techniques such as *breadth first search* [CM91], *depth first search* [AV01], and *partial-order methods* [SUL00], which do not take advantage of the fact that $\mathsf{EF}(p_1)$ can be detected efficiently. With computation slicing, however, we can first compute the slice for p_1. If only a small fraction of consistent cuts satisfy p_1, then instead of detecting $\mathsf{EF}(p)$ in the computation, it is much more efficient to detect $\mathsf{EF}(p)$ in the slice. Therefore, by spending only polynomial amount of time in computing the slice, we can throw away exponential number of consistent cuts, thereby obtaining an exponential speedup overall.

Note that other techniques for reducing the time-complexity [SUL00] and/or the space-complexity [AV01] of predicate detection are orthogonal to slicing, and as such can actually be used in conjunction with slicing.

Finally, the algorithms described in earlier papers [GM01, MG01a, MG03] for computing a slice are all *off-line* in nature; they assume that the entire set of events is available *a priori*. While this is quite adequate for applications such as testing and debugging, for other applications such as software fault tolerance, it is desirable that the slice be computed incrementally in an *on-line* manner. In other words, the current slice is updated, as and when a new event is generated, to reflect its arrival. The reason is that for software fault tolerance, it is important to detect the fault as early as possible before it can cause any severe damage. If we compute the slice only after a certain number of events have been collected and then analyze it for the presence of a faulty consistent cut, it may be too late for any meaningful recovery. At the same time, whenever an event arrives, we want the cost of incrementally updating the slice to be less than the cost of recomputing the slice from scratch using an off-line algorithm. In Section 4.4, we give an efficient incremental algorithm for computing the slice for predicates that can be efficiently detected under EF operator.

4.2 Slicing Background

The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [DP90] which we described in Section 2.2. We now illustrate Birkhoff's Theorem on a computation and its distributive lattice of consistent cuts.

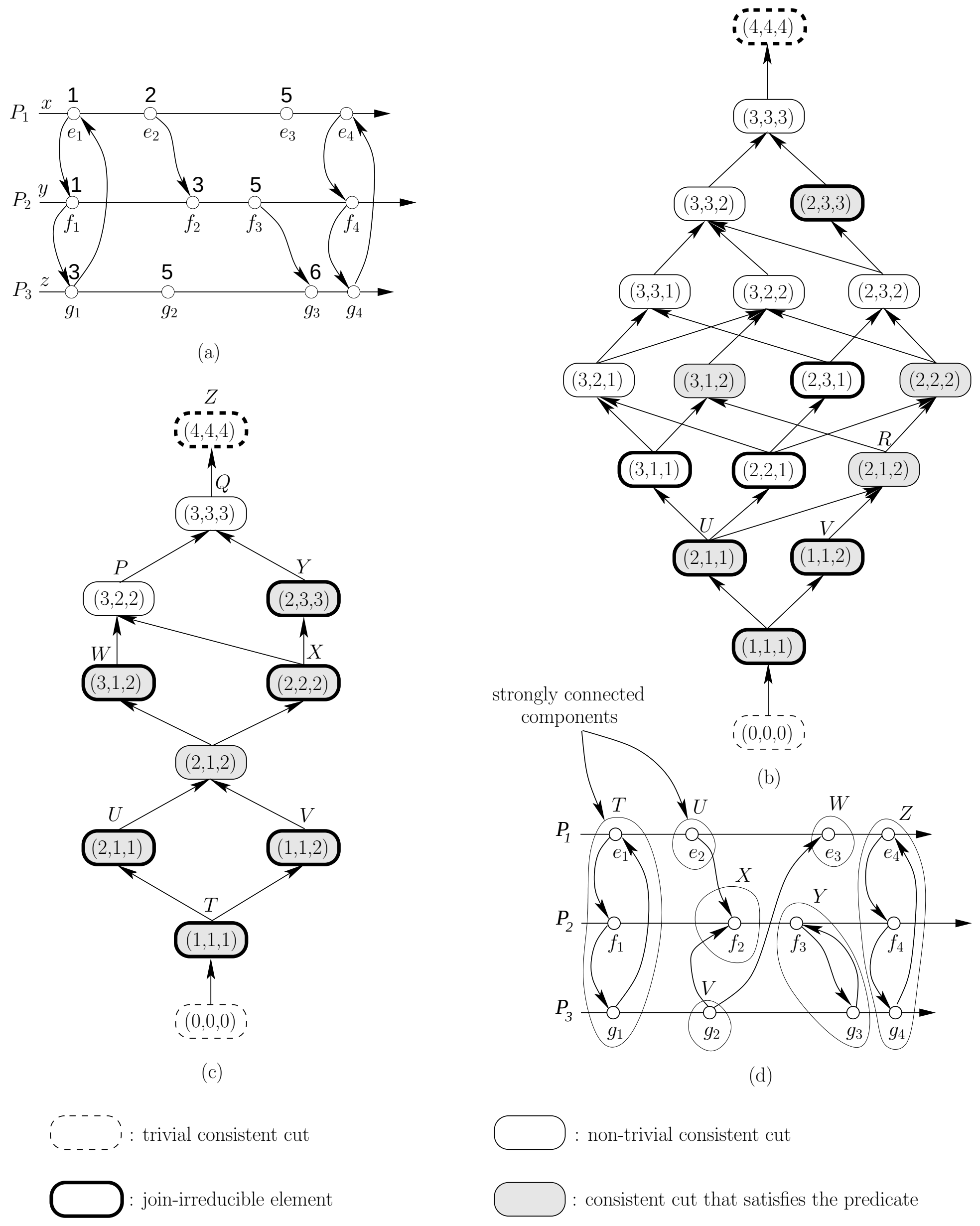
4.2.1 Birkhoff's Theorem: Example

Let L be a distributive lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible elements. Form Birkhoff's Theorem, we have that every element in L can be expressed as join of some subset of elements in $\mathcal{JI}(L)$ and vice versa. In other words, the partial order defined on $\mathcal{JI}(L)$ represents L. This is significant because $|\mathcal{JI}(L)|$ is generally much smaller—exponentially in many cases—than $|L|$. Hence if some computation on L can instead be performed on $\mathcal{JI}(L)$, we obtain a significant computational advantage.

Consider a computation $G = \langle E, \rightarrow \rangle$ and its set of its consistent cuts $\mathcal{C}(G)$. The set of join-irreducible elements of $\mathcal{C}(G)$ is isomorphic to the set of strongly connected components of G [MG01a].

Now, consider a *sublattice* $\mathcal{D}$ of $\mathcal{C}(G)$. It can be proved that any sublattice of a distributive lattice is also a distributive lattice [DP90]. Thus if $\mathcal{D}$ is a sublattice of $\mathcal{C}(G)$, then, using Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$ completely characterizes $\mathcal{D}$. This forms the basis for the notion of computation slice.

Example 4.1 *Consider the computation shown in Figure 4.1(a). The (distributive)*



P_1 x 1 2 5 e_1 e_2 e_3 e_4
P_2 y 1 3 5 f_1 f_2 f_3 f_4
P_3 z 3 5 6 g_1 g_2 g_3 g_4
(a)
(4,4,4)
(3,3,3)
(3,3,2) (2,3,3)
(3,3,1) (3,2,2) (2,3,2)
(3,2,1) (3,1,2) (2,3,1) (2,2,2)
(3,1,1) (2,2,1) (2,1,2) R
(2,1,1) U (1,1,2) V
(1,1,1)
(0,0,0)
(b)
Z (4,4,4)
Q (3,3,3)
P (3,2,2) Y (2,3,3)
W (3,1,2) X (2,2,2)
(2,1,2)
U (2,1,1) V (1,1,2)
T (1,1,1)
(0,0,0)
(c)
strongly connected components
T U W Z X Y V
P_1 P_2 P_3
e_1 e_2 e_3 e_4 f_1 f_2 f_3 f_4 g_1 g_2 g_3 g_4
(d)
: trivial consistent cut
: non-trivial consistent cut
: join-irreducible element
: consistent cut that satisfies the predicate

Figure 4.1: (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leqslant 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

lattice spanned by its set of consistent cuts is shown in Figure 4.1(b). In the figure, each consistent cut is labeled with the number of events that have to be executed on each process to reach the cut. The join-irreducible elements of the lattice have been drawn with thick boundaries. They have exactly one incoming edge. The lattice has eight join-irreducible elements which is same as the number of strongly connected components of the computation. It can be verified that every consistent cut of the computation can be obtained as the join of some subset of these eight join-irreducible elements and vice versa. For instance, the consistent cut R (in Figure 4.1(b)) can be expressed as the join of the consistent cuts U and V.

4.2.2 Slice

Informally, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. Formally,

Definition 4.2 (slice [GM01, MG01a]) *A* slice *of a graph G with respect to a predicate is the directed graph obtained from G by adding edges such that:*
(1) it contains all consistent cuts of the computation that satisfy the predicate and
(2) of all the graphs that satisfy (1), it has the least number of consistent cuts.

We denote the slice of a computation $G = \langle E, \rightarrow \rangle$ with respect to a predicate p by $\mathsf{slice}(G, p)$. Note that $G = \mathsf{slice}(G, \mathsf{true})$. In the rest of the dissertation, we use the terms "computation", "slice", "graph", and "directed graph" interchangeably. Every slice derived from the computation G has the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts. In the rest of the dissertation, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called *lean*.

Definition 4.3 (lean slice) *The slice of a computation with respect to a predicate is lean if every consistent cut of the slice satisfies the predicate.*

Theorem 4.4 ([GM01]) *The slice of a computation with respect to a predicate is lean if and only if the predicate is regular.*

We next present an algorithm for generating the slice. To that end, we first define some concepts. Given a sublattice L' and a vertex e, let $J_{L'}(e)$ denote the *least* consistent cut in L that is part of L' and contains e. If there is no consistent cut in L' that includes e, then $J_{L'}(e)$ is set to E—the trivial consistent cut. In case $e = \top$ then $J_{L'}(e)$ is set to E—the trivial consistent cut and in case $e = \bot$ then $J_{L'}(\bot)$ corresponds to the least element of L'.

Given a distributive lattice L generated by a graph G, every sublattice of L can be generated by a graph obtained by adding edges to G.

Theorem 4.5 *Let L' be any sublattice of a finite distributive lattice L generated by the directed graph G. Then, there exists a graph G' that can be obtained by adding edges to G that generates L'.*

Proof: We show an algorithm to compute G'. Since L' is a sublattice, it is clear that if the set of consistent cuts include e and are part of L', then their intersection (meet) also includes e and is part of L'. Thus, $J_{L'}(e)$ is well-defined.

Now we add the following edges to the graph G. For every pair of vertices e, f such that $J_{L'}(e) \subseteq J_{L'}(f)$, we add an edge from e to f. We now claim the resulting graph G' generates L'.

Pick any nontrivial consistent cut C of G'. We show that $C = \cup_{e \in C} J_{L'}(e)$. This will be sufficient to show that $C \in L'$ because C is a union of consistent cuts in L' and L' is a lattice. Since $e \in J_{L'}(e)$ it is clear that $C \subseteq \cup_{e \in C} J_{L'}(e)$. We show that $C \supseteq \cup_{e \in C} J_{L'}(e)$. Let $f \in J_{L'}(e)$ for some e. This implies that $J_{L'}(f) \subseteq J_{L'}(e)$ because $J_{L'}(f)$ is the least consistent cut containing f in L'. By our algorithm, there is an edge from f to e in G', and since C is a consistent cut of G' that includes e, it also includes f.

Conversely, pick any element C of L'. We show that C is a nontrivial consistent cut of G'. Since $L' \subseteq L$ and L corresponds to nontrivial consistent cuts of G, it is clear that C is a nontrivial consistent cut of G. Our obligation is to show that it is a nontrivial consistent cut of G' as well. Assume, if possible, C is not a nontrivial consistent cut of G'. This implies that there exists vertices e, f in G' such that $f \in C$, $e \notin C$ and (e, f) is an edge in G'. The presence of this edge in G', but not in G implies that $J_{L'}(e) \subseteq J_{L'}(f)$. Since $f \in C$ and $C \in L'$, from the definition of $J_{L'}(f)$, we get that $J_{L'}(f) \subseteq C$. But this implies that $J_{L'}(e) \subseteq C$, that is, $e \in C$, a contradiction. $\square$

Observe that $J_{L'}(e)$ corresponds to the join-irreducible elements of the sublattice L'. Hence, the algorithm in Theorem 4.5 (as we know from Birkhoff's Theorem) generates G' by computing the poset induced on $J_{L'}(e)$.

Theorem 4.6 *For any directed graph G and any predicate p, $\mathsf{slice}(G, p)$ exists and is unique.*

Proof: Note that the intersection of sublattices is also a sublattice. Given any predicate p consider all sublattices that contain all the consistent cuts that satisfy p. The intersection of all these sublattices gives us the smallest sublattice that contains all the consistent cuts that satisfy p. From Theorem 4.5, we get that there exists a graph that generates this sublattice. $\square$

The procedure outlined in the proof of Theorem 4.6 is not efficient because it requires us to take intersection of all bigger sublattices. However, for non-temporal regular predicates, since the set of consistent cuts that satisfy the predicate generates a sublattice L', we can use the algorithm in Theorem 4.5 to compute slices efficiently.

In particular, $J_{L'}(e)$ corresponds to $J_p(e)$, where $J_p(e)$ denotes the *least* consistent cut of a computation G that satisfies p and contains e. $J_p(e)$ can also be interpreted as the least consistent cut that satisfies the predicate p_e. We say that a consistent cut satisfies predicate p_e if the cut contains event e and satisfies predicate p.

Example 4.7 *Consider the lattice of consistent cuts depicted in Figure 4.1(b). The consistent cuts that satisfy the predicate $x + y - z \leqslant 1$ have been shaded in the figure. Figure 4.1(c) depicts the smallest sublattice that contains these consistent cuts. The consistent cuts P and Q do not satisfy the predicate but have been included to complete the sublattice. The join-irreducible elements of the sublattice have been drawn with thick boundaries. There are, in total, seven join-irreducible elements, namely T, U, V, W, X, Y and Z. Figure 4.1(d) portrays the partial order induced on the set $\mathcal{J} = \{T, U, V, W, X, Y, Z\}$. There is a one-to-one correspondence between the set of join-irreducible elements and the set of strongly connected components of the graph shown in Figure 4.1(d). It can be verified that every consistent cut in the sublattice can be expressed as join of some subset of $\mathcal{J}$ and, furthermore, the join of every subset of $\mathcal{J}$ is a consistent cut of the sublattice.*

However, the graph obtained using Theorem 4.5 can have as many as $\Omega(|E|^2)$ edges. Next we will show a representation with less number of edges.

Skeletal Representation of a Slice

In general, there can be multiple directed graphs with the same set of consistent cuts. Therefore more than one graph may constitute a valid representation of the given slice. The following lemma states that all such graphs are in fact related.

Lemma 4.8 ([MG01a]) *Consider directed graphs G and H on the same set of vertices. Then,*

$$\mathcal{P}(G) \subseteq \mathcal{P}(H) \equiv \mathcal{C}(G) \supseteq \mathcal{C}(H)$$

which in turn implies that:

Lemma 4.9 ([MG01a]) *Consider directed graphs G and H on the same set of vertices. Then,*

$$\mathcal{P}(G) = \mathcal{P}(H) \ \equiv \ \mathcal{C}(G) = \mathcal{C}(H)$$

In other words, two directed graphs G and H, on identical sets of vertices, are *cut-equivalent* (that is, $\mathcal{C}(G) = \mathcal{C}(H)$) if and only if they are *path-equivalent* (that is, $\mathcal{P}(G) = \mathcal{P}(H)$). We consider a special directed graph to capture a slice, called the *skeletal representation* of a slice [MG01a]. Let $F_p(e)$ be a vector of events where the i^{th} entry in the vector denotes the earliest event f on process P_i such that $J_p(e) \subseteq J_p(f)$. Informally, $F_p(e)[i]$ is the earliest event on P_i that is reachable from e in the $\mathsf{slice}(G, p)$. The skeletal representation of a slice has E as the set of vertices and the following edges:

1. for each event $e \notin \top$, there is an edge from e to $succ(e)$, and
2. for each event e and process P_i, there is an edge from e to $F_p(e)[i]$.

To see that the skeletal representation represents the slice, the following lemma is useful.

Lemma 4.10 ($J_p(e)$ is order-preserving [GM01]) *Given events e and f,*

$$e \rightarrow f \ \Rightarrow \ J_p(e) \subseteq J_p(f)$$

Proof: Consider $J_p(f)$. Since $e \rightarrow f$ and $f \in J_p(f)$, $e \in J_p(f)$. Thus $J_p(f)$ is a consistent cut that contains e and satisfies p. Since $J_p(e)$ is the least such cut, $J_p(e) \subseteq J_p(f)$. □

Using this lemma we can prove that if $J_p(e) \subseteq J_p(f)$ then there is a path from event e to event f in the skeletal representation and vice versa. Finally, we can show that the slice is cut-equivalent to the skeletal representation.

4.2.3 Slicing for Non-Temporal Regular Predicates

Garg and Mittal present an efficient algorithm to compute the slice for a non-temporal regular predicate [GM01]. In particular, they construct the graph corresponding to the skeletal representation of $\mathsf{slice}(G, p)$. To that end, they give an algorithm to compute $F_p(e)$.

They first compute $J_p(e)$ for each event e. Since p_e is a non-temporal regular predicate, it is also a linear predicate. The algorithm in Figure 3.2 in Chapter 3 can be used to find the least consistent cut that satisfies p_e. Their algorithm computes $J_p(e)$ for each event e in a single scan of the computation from left to right with $O(n^2|E|)$ time-complexity. This algorithm assumes the efficient predicate evaluation and the efficient advancement properties (Property 3.15 and Property 3.17 in Section 3.3). Similarly, they compute $F_p(e)$ for each event e, in a single scan of the computation from left to right with $O(n|E|)$ time-complexity. The overall time-complexity of computing the slice for a non-temporal regular predicate is therefore $O(n^2|E|)$. The order-preserving property of $F_p(e)$ is crucial for obtaining this complexity bound.

Lemma 4.11 ($F_p(e)$ is order-preserving [GM01]) *Given events e and f and a process P_j,*

$$e \rightarrow f \;\Rightarrow\; F_p(e)[j] \xrightarrow{P} F_p(f)[j]$$

Proof: Assume that $e \rightarrow f$. Let $g = F_p(e)[j]$ and $h = F_p(f)[j]$. Note that $proc(g) = proc(h) = P_j$. By definition of $F_p(e)$, $J_p(f) \subseteq J_p(h)$. Since, from Lemma 4.10, $J_p(e) \subseteq J_p(f)$, $J_p(e) \subseteq J_p(h)$. Again, by definition of $F_p(e)$, g is the earliest event on P_j such that $J_p(e) \subseteq J_p(g)$. Therefore $g \xrightarrow{P} h$. $\square$

Figure 4.2 displays the algorithm to compute $F_p(e)$ using the algorithm to compute $J_p(e)$. Let E_j denote the set of events on process P_j. The outer for loop at

```
Algorithm Algo$_{4.1}$:

    Input: (1) a directed graph $G$, and   (2) $J_p(e)$ for each event $e$
    Output: $F_p(e)$ for all events $e$

1   for each process $P_i$ do
2       for each process $P_j$ do
3           $f := \bot_j$;
4           for each event $e$ on $P_i$ do               // visited in the order given by $\xrightarrow{P}$
5               while $J_p(e) \not\subseteq J_p(f)$ do
6                   $f := succ(f)$;                        // advance to the next event on $P_j$
                endwhile;
7               $F_p(e)[j] := f$;
            endfor;
        endfor;
    endfor;
```

Figure 4.2: The algorithm to compute $F_p(e)$ for all events e.

line 2 is executed exactly n times. For j^{th} iteration of the outer for loop, the while loop at line 5 is executed at most $O(|E_i| + |E_j|)$ times. Each iteration of the while loop has $O(1)$ time-complexity because whether $J_p(e) \not\subseteq J_p(f)$ can be ascertained by performing only a single comparison. Thus the overall time-complexity of the algorithm is $O(n|E_i| + |E|)$. Summing up over all processes, $F_p(e)$ for each event e can be determined in $O(n|E|)$ time.

An advantage of the skeletal representation is that it has $O(|E|)$ vertices and only $O(n|E|)$ edges, where n is the number of processes and E is the set of events, and hence generally leads to more efficient algorithms involving slices.

4.3 Slicing for Non-Temporal Predicates

In this section, we present efficient computation slicing algorithms with respect to non-temporal predicates for which there are efficient detection algorithms under EF operator of RCTL+. Specifically, we study the relationship between the following two problems, which allows us to extend efficient computation slicing algorithms.

Containing Cut (CONTC) Given a directed graph G and a predicate p, does G contain a consistent cut that satisfies p? Equivalently, check $G \models \mathsf{EF}(p)$.

Computing Slice (COMPS) Given a directed graph G and a predicate p, compute the slice of G with respect to p. Equivalently, compute $\mathsf{slice}(G, p)$.

4.3.1 Computing Slice is equivalent to Containing Cut

In this section, we prove that the problem of computing a succinct representation of *all* consistent cuts satisfying a predicate is equivalent the problem of determining whether there exists *at least one* consistent cut satisfying the predicate. From the definition of slice, clearly, it follows that the slice for a directed graph with respect to a predicate (not necessarily regular) is nonempty if and only if the graph contains a consistent cut that satisfies the predicate. Formally,

$$\mathsf{CONTC}(G, p) \;\equiv\; \mathsf{COMPS}(G, p) \text{ is nonempty}$$

Therefore COMPS is at least as hard as CONTC. We now prove the converse. Consider a directed graph G and a predicate p. Now, G and $\mathsf{slice}(G, p)$ are directed graphs on identical sets of vertices. However, more pairs of vertices are "connected" in $\mathsf{slice}(G, p)$ than in G, that is, the slice contains "additional edges". In the next lemma, we give a complete characterization of the additional edges in $\mathsf{slice}(G, p)$. Let $G[e, f]$ denote the directed graph obtained by adding an edge from e to f in G.

Lemma 4.12 *There is a path from an event e to an event f in $\mathsf{slice}(G, p)$ if and only if no consistent cut in $\mathcal{C}(G) - \mathcal{C}(G[e, f])$ satisfies p.*

Proof: We have,

$$\begin{array}{ll} & \text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G, p) \\ \equiv & \{ \text{ definition of } \mathsf{slice}(G, p) \text{ } \} \end{array}$$

$$
\begin{array}{ll}
 & (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G,p)) \wedge \Big(\mathcal{C}(\mathsf{slice}(G,p)) \subseteq \mathcal{C}(G)\Big) \\
\equiv & \{ \text{ from Lemma 4.8, } \mathcal{C}(\mathsf{slice}(G,p)) \subseteq \mathcal{C}(G) \equiv \mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G,p)) \ \} \\
 & (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G,p)) \wedge \Big(\mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G,p))\Big) \\
\equiv & \{ \text{ definition of } G[e,f] \ \} \\
 & \mathcal{P}(G[e,f]) \subseteq \mathcal{P}(\mathsf{slice}(G,p)) \\
\equiv & \{ \text{ from Lemma 4.8 } \} \\
 & \mathcal{C}(\mathsf{slice}(G,p)) \subseteq \mathcal{C}(G[e,f]) \\
\equiv & \{ \ \mathcal{C}(\mathsf{slice}(G,p)) \text{ contains all consistent cuts of } \mathcal{C}(G) \text{ satisfying } p \ \} \\
 & \text{no consistent cut in } \ \mathcal{C}(G) - \mathcal{C}(G[e,f]) \ \text{ satisfies } p
\end{array}
$$

This establishes the lemma. □

Lemma 4.12 is useful provided it is possible to ascertain efficiently whether some consistent cut in $\mathcal{C}(G) - \mathcal{C}(G[e,f])$ satisfies p. To that end, we show that the set $\mathcal{C}(G) - \mathcal{C}(G[e,f])$ actually forms a sublattice and therefore can be captured faithfully using a directed graph. Let $\widehat{G}[e,f]$ denote the directed graph obtained by adding an edge from f to $\bot_1$ and an edge from $\top_1$ to e. It suffices to show the following:

Lemma 4.13 $\mathcal{C}(\widehat{G}[e,f]) - \{\emptyset, E\} \ = \ \mathcal{C}(G) - \mathcal{C}(G[e,f])$

Proof: Consider a *nontrivial* consistent cut C in $\mathcal{C}(G)$. It suffices to show that $C \in \mathcal{C}(\widehat{G}[e,f]) \equiv C \notin \mathcal{C}(G[e,f])$.

(⇒) We need to prove that $C \in \mathcal{C}(\widehat{G}[e,f]) \ \Rightarrow \ C \notin \mathcal{C}(G[e,f])$. Intuitively, it means that $G[e,f]$ and $\widehat{G}[e,f]$ do not have any common nontrivial consistent cut. Equivalently, $\mathcal{C}(G[e,f]) \cap \mathcal{C}(\widehat{G}[e,f]) = \{\emptyset, E\}$. We have,

$$
\begin{array}{ll}
 & C \in \mathcal{C}(\widehat{G}[e,f]) \\
\Rightarrow & \{ \text{ follows from the definition of } \widehat{G}[e,f] \ \} \\
 & (f \in C) \wedge (e \notin C) \\
\Rightarrow & \{ \text{ follows from the definition of } G[e,f] \ \}
\end{array}
$$

$C \notin \mathcal{C}(G[e,f])$

($\Leftarrow$) We need to prove that $C \notin \mathcal{C}(G[e,f]) \Rightarrow C \in \mathcal{C}(\widehat{G}[e,f])$. Intuitively, it means that every nontrivial consistent cut of G is either a consistent cut of $G[e,f]$ or a consistent cut of $\widehat{G}[e,f]$. Equivalently, $\mathcal{C}(G[e,f]) \cup \mathcal{C}(\widehat{G}[e,f]) = \mathcal{C}(G)$. The proof consists of two steps. In the first step, we show that if C is not a consistent cut of $G[e,f]$, then it is the case that $e \notin C$ and $f \in C$. Assume that $C \notin \mathcal{C}(G[e,f])$. Therefore there exist events u and v such that there is a path from u to v in $G[e,f]$, $u \notin C$ and $v \in C$. Since C is a consistent cut of G, there is no path from u to v in G. That is, $(u,v) \in \mathcal{P}(G[e,f])$ but $(u,v) \notin \mathcal{P}(G)$. Thus every path from u to v in $G[e,f]$ involves the edge (e,f). This implies that $(u,e) \in \mathcal{P}(G)$ and $(f,v) \in \mathcal{P}(G)$. Since $(u,e) \in \mathcal{P}(G)$ and $u \notin C$, $e \notin C$. Similarly, since $(f,v) \in \mathcal{P}(G)$ and $v \in C$, $f \in C$. Therefore $e \notin C$ and $f \in C$.

In the second step, we show that if C is not a consistent cut of $\widehat{G}[e,f]$, then it is the case that $(e \in C) \vee (f \notin C)$. Assume that $C \notin \mathcal{C}(\widehat{G}[e,f])$. Hence there exist events x and y such that there is a path from x to y in $\widehat{G}[e,f]$, $x \notin C$ and $y \in C$. Since C is a consistent cut of G, there is no path from x to y in G. That is, $(x,y) \in \mathcal{P}(\widehat{G}[e,f])$ but $(x,y) \notin \mathcal{P}(G)$. Thus every path from x to y in $\widehat{G}[e,f]$ either involves the edge $(f,\bot_1)$ or the edge $(\top_1,e)$. In the first case, there is a path from x to f in G which implies that $f \notin C$. In the second case, there is path from e to y in G which implies that $e \in C$. Consequently, either $e \in C$ or $f \notin C$—a logical negation of what we obtained in the first step.

Combining the two steps, it can be inferred that it cannot be the case that C is neither a consistent cut of $G[e,f]$ nor a consistent cut of $\widehat{G}[e,f]$. This establishes the lemma. $\square$

We now present an example illustrating the Lemma 4.12 and Lemma 4.13.

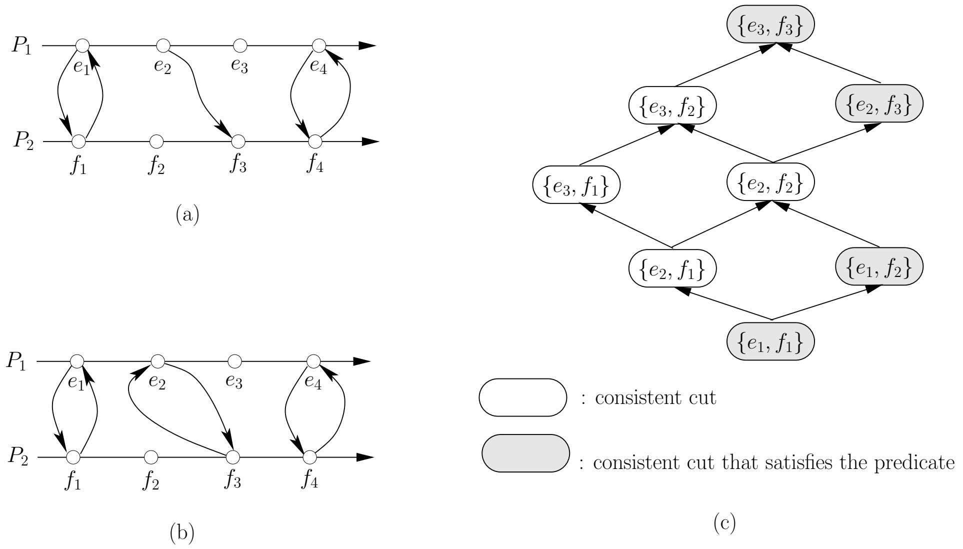


Figure 4.3: (a) A directed graph G, (b) $\mathsf{slice}(G, p)$, where p is the predicate "all channels are empty", and (c) the set of nontrivial consistent cuts of H.

Example 4.14 *Consider the directed graph G shown in Figure 4.3(a). In the graph, e_1 and f_1 are the initial events, whereas e_4 and f_4 are the final events. Figure 4.3(b) depicts $\mathsf{slice}(G, p)$, where p is the predicate "all channels are empty". The slice is obtained by adding the edge from f_3 to e_2 because, for all channels to be empty, send and receive events of the same message have to be executed atomically. The set of nontrivial consistent cuts of G are shown in Figure 4.3(c). The cuts for which all channels are empty have been shaded. Now, consider directed graphs $G[e_3, e_2]$ and $\widehat{G}[e_3, e_2]$ shown in Figure 4.4(a) and Figure 4.4(c), respectively. Their sets of consistent cuts (excluding trivial consistent cuts) are shown in Figure 4.4(b) and Figure 4.4(d), respectively. As expected, the two sets satisfy Lemma 4.13. Also, since there is no path from e_3 to e_2 in $\mathsf{slice}(G, p)$, $\widehat{G}[e_3, e_2]$ contains a consistent cut that satisfies p. Figure 4.5 illustrates the case when the edge (f_3, e_3) is such that $\mathsf{slice}(G, p)$ contains a path from f_3 to e_3.*

Combining the two lemmas, we obtain the following:

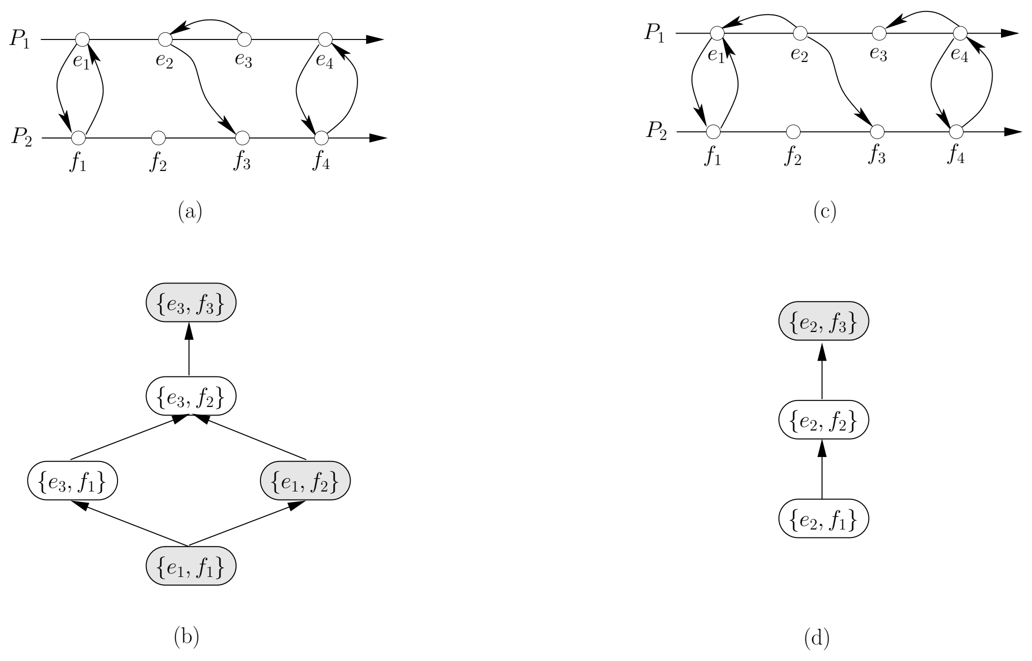


Figure 4.4: (a) The directed graph $G[e_3, e_2]$, where there is no path from e_3 to e_2 in $\mathsf{slice}(G, p)$, (b) the set of nontrivial consistent cuts of $G[e_3, e_2]$, (c) the directed graph $\widehat{G}[e_3, e_2]$, and (d) the set of nontrivial consistent cuts of $\widehat{G}[e_3, e_2]$.

Theorem 4.15 *There is a path from an event e to an event f in $\mathsf{slice}(G, p)$ if and only if no consistent cut in $\widehat{G}[e, f]$ satisfies p, that is, $\mathsf{CONTC}(\widehat{G}[e, f], p)$ evaluates to false.*

Figure 4.6 depicts the algorithm for solving COMPS using an algorithm that solves CONTC. The algorithm constructs a directed graph that is transitively closed.

Theorem 4.16 *The time-complexity of the algorithm for solving* COMPS *in Figure 4.6 is $O(|E|^2 T)$, where E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC*.*

Proof: The initialization at line 1 requires $O(|E|^2)$ time, where E is the set of events, because G has $|E|$ vertices and therefore $O(|E|^2)$ edges. The for loop at

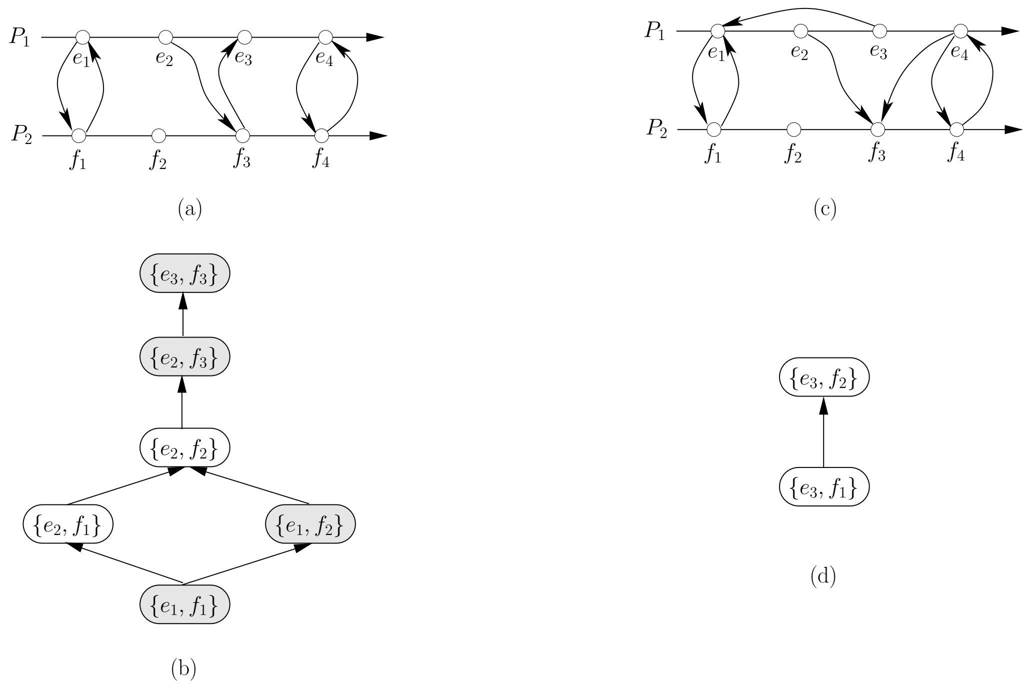


Figure 4.5: (a) The directed graph $G[f_3, e_3]$, where there is a path from f_3 to e_3 in $\mathsf{slice}(G, p)$, (b) the set of nontrivial consistent cuts of $G[f_3, e_3]$, (c) the directed graph $\widehat{G}[f_3, e_3]$, and (d) the set of nontrivial consistent cuts of $\widehat{G}[f_3, e_3]$.

```
Algorithm Algo$_{4.2}$:

    Input: (1) a directed graph $G$,   (2) a predicate $p$, and
           (3) an algorithm to evaluate CONTC$(H, p)$ for an arbitrary directed graph $H$

    Output: the slice of $G$ with respect to $p$

1   $K := G$;
2   for every pair of events $(e, f)$ do
3       if $not$(CONTC($\widehat{G}[e, f], p$)) then          // $\widehat{G}[e, f] \not\models$ EF($p$)
4           add an edge from $e$ to $f$ in $K$;              // $K := K[e, f]$
        endif;
    endfor;
5   output $K$;
```

Figure 4.6: An algorithm to solve COMPS using an algorithm to solve CONTC.

```
Algorithm Algo$_{4.3}$:

    Input: (1) a directed graph $G$, and   (2) a predicate $p$, and
           (3) an algorithm to evaluate CONTC$(H, p)$ for an arbitrary directed graph $H$

    Output: $F_p(e)$ for all events $e$

1   for each process $P_i$ do
2       for each process $P_j$ do
3           $f := \perp_j$;
4           for each event $e$ on $P_i$ do                  // visited in the order given by $\xrightarrow{P}$
5               while CONTC$(\widehat{G}[e, f], p)$ do
6                   $f := succ(f)$;                         // advance to the next event on $P_j$
                endwhile;
7               $F_p(e)[j] := f$;
            endfor;
        endfor;
    endfor;
```

Figure 4.7: The algorithm to compute $F_p(e)$ for all events e.

line 2 executes $|E|^2$ times. Each iteration of the for loop requires solving an instance of CONTC. The construction of the particular instance of CONTC involves adding two edges to G, and therefore can be done in $O(1)$ time. Depending on the result of the if statement at line 3, an edge may be required to be added to K at line 4, which can be done in $O(1)$ time. At the end of the iteration, the two edges that were added to G have to be deleted. The deletion can be accomplished in $O(1)$ time by maintaining pointers to the two edges if using adjacency list representation. The overall time-complexity of the for loop is $O(|E|^2T)$, which is also the time-complexity of the algorithm. $\square$

The time-complexity of the algorithm can be reduced to $O(n|E|T)$, where n is the number of processes, by constructing the skeletal representation of the slice discussed in Section 4.2.2. To that end, it suffices to compute the vector $F_p(e)$ for each event e; the i^{th} entry of $F_p(e)$ denotes the earliest event on process P_i reachable from e in the slice. From Lemma 4.11, we have that F_p is order-preserving which means that if $e \rightarrow f$ then $F_p(e)[j] \overset{P}{\rightarrow} F_p(f)[j]$ for each j. Consequently, it is possible to compute $F_p(e)[j]$ for each event e on process P_i by scanning the computation once from left to right. The algorithm is presented in Figure 4.7.

Theorem 4.17 *The time-complexity of the algorithm for computing $F_p(e)$ for each event e in Figure 4.7 is $O(n|E|T)$, where n is the number of processes, E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC*.*

Proof: Note that the while loop at line 5 terminates in at most $|E_i|+|E_j|$ iterations, where E_i and E_j denote the set of events on processes P_i and P_j, respectively. This is because the directed graph $\widehat{G}[e, f]$ when $f = \top_j$ has an edge from the final event $\top_j$ to the initial event $\bot_1$ implying that $\widehat{G}[e, \top_j]$ has no nontrivial consistent cut. Therefore $\mathsf{CONTC}(\widehat{G}[e, f], p)$ when $f = \top_j$ will, trivially, evaluate to false. This gives the time-complexity of $O((|E_i| + |E_j|)\ T)$ for the inner for loop at line 4. Hence the time-complexity of the outer for loop at line 2 is $O((n|E_i| + |E|)\ T)$. This implies that the overall time-complexity of computing $F_p(e)$ for each event e is $O(n|E|T)$. □

4.3.2 Applications of the Result

In earlier results, Mittal and Garg gave efficient algorithms for computing the slice for non-temporal predicate classes such as regular predicates [GM01], co-regular predicates [MG01a], and linear predicates [MG03]. Using the result of this chapter, it is now possible to compute the slice efficiently for many more classes of non-temporal predicates including stable and co-stable predicates, observer-independent predicates, co-linear predicates, and relational predicates. For instance, an observer-independent predicate can be detected under EF operator in $O(n|E|)$ time using the algorithm presented in [CBDGF95]. This implies that its slice can be computed in

$O(n^2|E|^2)$ time using the algorithm given in Section 4.3.1. It is possible that a faster and more efficient slicing algorithm exists for an observer-independent predicate, which perhaps exploits the specific properties of the class of observer-independent predicates. Similarly, relational predicates can be detected under EF operator efficiently using max-flow techniques [Gar96] and we can use this algorithm to compute the slice with respect to a relational predicate. Our result is still useful because it gives a ready-made algorithm for computing the slice.

Note that the input to our slicing algorithm is a directed graph G, a predicate p, and an algorithm to evaluate $\mathsf{CONTC}(H, p)$ for an arbitrary directed graph H. In fact, in our algorithms of this section, H is obtained from G by addition of edges. Equivalently, the lattice of consistent cuts of H is a sublattice of the lattice of consistent cuts of G.

In Chapter 3, we defined predicate classes such as linear, stable, observer-independent with respect to a graph G. We now show that given a predicate p from a class A and defined with respect to a graph G, p belongs to class A for an arbitrary graph H that is obtained from G. In particular, we are concerned with classes of non-temporal predicates stable and co-stable, observer-independent, co-linear, and relational predicates.

Lemma 4.18 (universal predicates) *If p belongs to one of linear, co-linear, stable, co-stable, local, disjunctive, conjunctive and relational predicate classes with respect to a graph G, then then it belongs to the same class with respect to all H obtained from G by addition of edges.*

Proof: We know that $L' = (\mathcal{C}(H), \subseteq)$ forms a sublattice of the lattice $L = (\mathcal{C}(G), \subseteq)$.

Given a linear predicate p and two arbitrary consistent cuts C and D in $\mathcal{C}(H)$, $C \cap D$ belongs to $\mathcal{C}(H)$ and $\mathcal{C}(G)$ since $\mathcal{C}(H)$ is a sublattice. Furthermore, if both C and D satisfy p, then $C \cap D$ satisfies p, since $C \cap D$ belongs to $\mathcal{C}(G)$ and satisfies p. Hence, p is linear with respect to H.

Given a predicate p which is co-linear with respect to G, $\neg p$ is linear with respect to G. Consequently, p is co-linear with respect to H, since $\neg p$ is linear with respect to H, which we showed above.

A stable predicate is such that once the predicate is true then it stays true. In other words, all cuts reachable from the cut where p holds, satisfy p. This implies that if there exists a consistent cut C that satisfies p in L', and if L' also contains cuts reachable from C in L, then all such cuts satisfy p, hence p is stable with respect to H. Using an argument similar to co-linear predicates, we can show the universality of co-stable predicates.

A local predicate is such that it holds true at a local event of a process. If H contains a consistent cut that has an event that satisfies the local predicate in its frontier then it is clear that p is local with respect to H as well. Using an argument similar to local predicates, we can prove that disjunctive, conjunctive and relational predicates are also universal. $\square$

Note that a co-linear predicate can be detected by using the complement of the algorithm to detect a linear predicate under AG operator. We assume observer-independent predicates are universal too.

The above results enable us to use the efficient algorithms for $\mathsf{CONTC}(H, p)$ as an input to our slicing algorithm.

Since the predicate detection problem under EF operator is NP-complete in general [CG95], the problem of computing the slice is also NP-complete. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently. An approximate slice may be bigger than the actual slice but may be much smaller than the computation itself. Therefore, in order to detect a predicate, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice. In Chapter 6 we show how to use the results of this section for predicate detection.

4.4 On-Line Slicing Algorithm

In this section, we present an on-line algorithm for computing the slice for a predicate for which the slice can indeed be computed efficiently in an off-line manner. Our on-line slicing algorithm is basically derived from the off-line algorithm for computing the slice described in Figure 4.7. On generation of a new event in the system, our on-line algorithm updates the current slice to reflect the arrival of the new event.

Before discussing the algorithm, we state our assumptions and describe some notation. We assume that a newly arrived event is "enabled" in the sense that all events that happened-before it have already arrived and been incorporated into the slice. This can be achieved by buffering the new event—in case it is not "enabled"—and processing it later when it becomes "enabled". Whether an event is "enabled" can be determined efficiently by examining its vector clock timestamp. A well-known vector clock algorithm for message passing programs is Fidge/Mattern algorithm [Fid91, Mat89].

Initially, the computation consists of only the fictitious—initial and final—events. Let the k^{th} arriving event, $k \geqslant 1$, be denoted by $e^{(k)}$, and let $G^{(k)}$ denote the resulting computation. Sometimes we represent the computation more explicitly using $\langle E^{(k)}, \rightarrow \rangle$ whenever necessary, where $E^{(k)}$ denotes the set of events and $\rightarrow$ denotes the set of edges in $G^{(k)}$. Without loss of generality, assume that $G^{(k)}$ is a transitively closed graph and thus $\rightarrow$ is a transitive relation.

Clearly, every nontrivial consistent cut of $G^{(k-1)}$ is a consistent cut of $G^{(k)}$ as well.

Observation 4.19 $\mathcal{C}(G^{(k-1)}) - E^{(k-1)} \subseteq \mathcal{C}(G^{(k)})$

Furthermore, every consistent cut of $G^{(k)}$ that is not a consistent cut of

$G^{(k-1)}$ contains $e^{(k)}$.

Observation 4.20 $C \in \mathcal{C}(G^{(k)}) - \mathcal{C}(G^{(k-1)}) \;\Rightarrow\; e^{(k)} \in C$

The on-line algorithm, whenever a new event arrives, computes the new slice by updating $F_p(e)$ for each event e. We use ${F_p}^{(k)}$ to refer to the value of F_p for the computation $G^{(k)}$. Now, in order to incorporate an event into the slice, we may have to recompute the entry $F_p(e)[i]$ for each event e and every process P_i.

Definition 4.21 (critical event) *Let P_{j_k} denote the process on which the event $e^{(k)}$ occurred. An event $e \in E^{(k-1)}$ is said to be a* critical event *with respect to $e^{(k)}$ if $F_p^{k-1}(e)[j_k] = \top_{j_k}$.*

Intuitively, no nonfinal event on P_{j_k} is reachable from e in $\mathsf{slice}(G^{(k-1)}, p)$. This may change, however, on arrival of $e^{(k)}$ because $e^{(k)}$ is an event on P_{j_k}. Let *critical*(k) denote the set of all events in $E^{(k-1)}$ that are critical with respect to $e^{(k)}$.

First, we show that the new value for events on the same process cannot move backward in the space-time diagram. It is easy to verify that $F_p^k(e)[j]$ is order-preserving using the result that $F_p(e)$ is order-preserving from Lemma 4.11.

Lemma 4.22 ($F_p^k(e)[j]$ is order-preserving) *Given events e and f and a process P_j,*

$$e \overset{P}{\rightarrow} f \;\Rightarrow\; F_p^k(e)[j] \overset{P}{\underline{\rightarrow}} F_p^k(f)[j]$$

Section 4.4.2 contains a formal proof a similar lemma as above. Lemma 4.22 may greatly restrict the amount of work that needs to be done in order to recompute F_p. In particular, to determine the new value of $F_p(f)[j]$ for an event f and a process P_j, rather than starting the scan from $\bot_j$, we can instead start the scan from the value of $F_p(e)[j]$, where $e \overset{P}{\rightarrow} f$. The next lemma specifies the conditions under which either $F_p(e)[j]$ will not change or can be determined cheaply.

Lemma 4.23 *Given an event $e \in E^{(k-1)}$ and a process P_j,*

$$(e \to e^{(k)}) \wedge \Big((j \neq j_k) \vee (e \notin critical(k))\Big) \quad \Rightarrow \quad F_p^{k-1}(e)[j] = F_p^k(e)[j] \tag{4.3}$$

$$(e \to e^{(k)}) \wedge \Big((j = j_k) \wedge (e \in critical(k))\Big) \quad \Rightarrow \quad F_p^k(e)[j] = e^{(k)} \tag{4.4}$$

Proof: Let $f = F_p^{k-1}(e)[j]$ and $g = F_p^k(e)[j]$.

Equation (4.3) Given f and g are on process P_j. It suffices to show that $f \overset{P}{\not\to} g$ and $g \overset{P}{\not\to} f$.

(a) $f \overset{P}{\not\to} g$: Assume, on the contrary, that $f \xrightarrow{P} g$. In this case, edge (e, f) does not belong to $\mathsf{slice}(G^{(k)}, p)$. This means that, there exists a consistent cut C in $G^{(k)}$ such that $f \in C$, $e \notin C$, and $C \models p$. Since $e \to e^{(k)}$, $e^{(k)} \notin C$, as well. From Observation 4.20, a consistent cut of $G^{(k)}$ that does not include $e^{(k)}$ is a consistent cut of $G^{(k-1)}$, that is, $C \in G^{(k-1)}$. Using Theorem 4.15, there is no edge (e, f) in $\mathsf{slice}(G^{(k-1)}, p)$. Equivalently, $f \neq F_p^{k-1}(e)[j]$, contradiction.

This implies that the value for a new entry cannot move forward.

(b) $g \overset{P}{\not\to} f$: Assume, on the contrary, that $g \xrightarrow{P} f$. In this case, edge (e, g) does not belong to $\mathsf{slice}(G^{(k-1)}, p)$. This means that, there exists a consistent cut C in $G^{(k-1)}$ such that $g \in C$, $e \notin C$, and $C \models p$. From Observation 4.19, every nontrivial consistent cut of $G^{(k-1)}$ (in this case C) is a consistent cut of $G^{(k)}$. Using Theorem 4.15, there is no edge (e, g) in $\mathsf{slice}(G^{(k)}, p)$. Equivalently, $g \neq F_p^k(e)[j]$, contradiction. Note that, we did not use the fact that $e \to e^{(k)}$ in this case.

This implies that the value for a new entry cannot move backward.

Equation (4.4) Note that $\mathcal{C}(\mathsf{slice}(G^{(k)}, p)) \subseteq \mathcal{C}(G^{(k)})$. In case $e \to e^{(k)}$, there is a path from e to $e^{(k)}$ in $G^{(k)}$. Thus, from Lemma 4.8, there is a path from e to $e^{(k)}$ in $\mathsf{slice}(G^{(k)}, p)$ as well. Consequently, $F_p^k(e)[j] \overset{P}{\underline{\to}} e^{(k)}$. Since e is a critical event, $F_p^{k-1}(e)[j] = \top_j$. From part (b) we know that, the value of $F_p^k(e)[j]$ cannot move backward, that is, it is either $e^{(k)}$ or $\top_j$. This, in turn, implies that $F_p^k(e)[j]$ is $e^{(k)}$.

$\square$

Lemma 4.23 implies that F_p needs to be (re)computed only for the following events in $E^{(k)}$: (1) the newly arrived event $e^{(k)}$, (2) those events in $E^{(k-1)}$ that did not happen-before $e^{(k)}$. It turns out that F_p for the newly arrived event can be determined rather easily. Specifically,

Lemma 4.24 *Given a process P_j,*

$$\begin{aligned} j \neq j_k \quad &\Rightarrow \quad F_p^k(e^{(k)})[j] = F_p^{k-1}(\top_{j_k})[j] \\ j = j_k \quad &\Rightarrow \quad F_p^k(e^{(k)})[j] = \min\{e^{(k)},\, F_p^{k-1}(\top_{j_k})[j]\} \end{aligned}$$

Section 4.4.2 also contains the proofs of the lemmas in this section.

4.4.1 Algorithm

Figure 4.8 shows the algorithm to update the slice on arrival of a new event. Our on-line slicing algorithm is basically derived from the off-line algorithm for computing the slice described in Figure 4.7. The algorithm is the same as the off-line algorithm except for the lines 1–3, 7 and 8. Intuitively, at line 8 (due to Lemma 4.23), instead of starting the search on process P_i from $e = \bot_i$ we start from the earliest event $e \not\rightarrow e^{(k)}$. Also, at line 9 (due to Lemma 4.22), instead of starting the search on process P_j from $f = \bot_j$, we start from $f := F_p(pred(e))[j]$. Finally, at lines 1–3, the value of $F_p(e^{(k)})$ is computed using Lemma 4.24.

We now analyze the time-complexity of the algorithm. For a set of events X, let X_j denote the subset of those events that occurred on process P_j. Note that for an event e in $E^{(k-1)}$, if $e^{(k)} \rightarrow e$ then $e \in \top$; otherwise, when e was incorporated into the slice, it was not "enabled"—a contradiction. As a result, events in $E^{(k-1)}$ that did not happen-before $e^{(k)}$ consists of either those events that are concurrent with $e^{(k)}$ or the final events. Now, let $C^{(k)}$ contain those events from $E^{(k)}$ that are concurrent with $e^{(k)}$. It can be verified that, given processes P_j and P_i, the number of times an instance of CONTC is invoked at line 10 is given by

```
Algorithm Algo$_{4.4}$:

    Input: (1) a directed graph $G^{(k)} = \langle E^{(k)}, \rightarrow \rangle$,   (2) a predicate $p$,
           (3) for each event $e \in E^{(k-1)}$, $F_p(e)$ currently set to $F_p^{k-1}(e)$, and
           (4) an algorithm to evaluate CONTC$(H, p)$ for an arbitrary directed graph $H$

    Output: for each event $e \in E^{(k)}$, $F_p(e)$ now set to $F_p^k(e)$

1   $F_p(e^{(k)}) := F_p(\top_{j_k})$;                          // compute $F_p$ for the new event
2   for each event $e$ in $E^{(k)}$ do
3       if $F_p(e)[j_k] = \top_{j_k}$ then $F_p(e)[j_k] := e^{(k)}$; endif;        // is $e$ a critical event?
    endfor;

4   for each process $P_i$ do
5       for each process $P_j$ do
6           let $e$ be the earliest event on $P_i$ such that $e \not\rightarrow e^{(k)}$;
7           $f := F_p(pred(e))[j]$;
8           for each event $e$ on $P_i$ starting from the event at line 6 do
9               $f := \max\{f, F_p(e)[j]\}$;     // $F_p$ is order-preserving and Lemma 4.22
10              while $(f \neq \top_j)$ and CONTC$(\widehat{G}^{(k)}[e, f], p)$ do
11                  $f := succ(f)$;                    // advance to the next event on $P_j$
                endwhile;
12              $F_p(e)[j] := f$;
13          endfor;
        endfor;
    endfor;
```

Figure 4.8: An on-line algorithm to update $F_p(e)$ for all events e on arrival of a new event.

$O(|E_j^{(k)}| + |C_i^{(k)}|)$. This is because between two consecutive invocations of CONTC, either e or f advances to its next event. Further, whereas e, if different from $\top_i$, is constrained to be concurrent with $e^{(k)}$, there is no such constraint on f. Summing over all possible values for j and i, CONTC is invoked $O(n|E^{(k)}|)$ times. This gives us a time-complexity of $O(n|E|T)$ for updating the slice, which is same as that of computing the slice from scratch. (Note that the earliest event on a process that did not happen-before $e^{(k)}$—at line 6—can be determined in $O(1)$ time using the Fidge/Mattern's vector timestamp.)

In order to reduce the time complexity further, we proceed as follows. Sup-

```
11a        if $f \rightarrow e^{(k)}$ then
11b            set $f$ to the earliest event on $P_i$ such that $f \not\rightarrow e^{(k)}$;
11c        else $f := succ(f)$;
           endif;
```

Figure 4.9: Improving the time-complexity of the algorithm in Figure 4.8.

pose, at line 10, $\mathsf{CONTC}(\widehat{G}^{(k)}[e,f],p)$ evaluates to true and $f \rightarrow e^{(k)}$. It can be shown that $\mathsf{CONTC}(\widehat{G}^{(k)}[e,g],p)$ will also evaluate to true for all events g such that $g \rightarrow e^{(k)}$. Formally,

Lemma 4.25 *Consider an event $e \in E^{(k-1)}$ and a process P_j. Further, let f be an event on P_j with $f \rightarrow e^{(k)}$ such that $F_p^{k-1}(e)[j] \overset{P}{\underline{\rightarrow}} f$. Then,*

$$(\mathsf{CONTC}(\widehat{G}^{(k)}[e,f],p) \textit{ evaluates to true}) \wedge (g \rightarrow e^{(k)}) \;\Rightarrow$$

$$\mathsf{CONTC}(\widehat{G}^{(k)}[e,g],p) \textit{ evaluates to true}$$

Proof: Since $f \overset{P}{\rightarrow} e^{(k)}$, $f \in E^{(k-1)}$. Further, $F_p^{k-1}(e)[j] \overset{P}{\underline{\rightarrow}} f$. From Theorem 4.15, $\mathsf{CONTC}(\widehat{G}^{(k-1)}[e,f],p)$ evaluates to false. Equivalently, there is no consistent cut of $\widehat{G}^{(k-1)}[e,f]$ that satisfies p. However, $\mathsf{CONTC}(\widehat{G}^{(k)}[e,f],p)$ evaluates to true. This implies that there exists a consistent cut of $\widehat{G}^{(k)}[e,f]$, say C, that satisfies p. In other words, C is a consistent cut of $G^{(k)}$, $f \in C$, $e \notin C$ and C satisfies p. Clearly, $e^{(k)} \in C$; otherwise, C is a consistent cut of $\widehat{G}^{(k-1)}[e,f]$ that satisfies p—a contradiction. Since $g \rightarrow e^{(k)}$, C also contains g. Therefore $\mathsf{CONTC}(\widehat{G}^{(k)}[e,g],p)$ also evaluates to true. $\square$

Therefore, when the condition of the while loop at line 10 evaluates to true and $f \rightarrow e^{(k)}$, rather than advancing f to $succ(f)$, we can advance f directly to the earliest event on P_j that did not happen-before $e^{(k)}$. This reduces the number of times an instance of CONTC is evaluated to $O(|C_j^{(k)}|+|C_i^{(k)}|+1)$. The modification is described in Figure 4.9. Now, summing over all possible values for j and i, when $e^{(k)}$ arrives, CONTC needs to be invoked $O(n|C^{(k)}|+n^2)$ times to update the slice. Next, summing over the arrival of $|E|$ events, the total number of times CONTC is invoked is given by $O(n|C| + n^2|E|)$, where C is the set of concurrent pairs of events in the computation. Assuming that the time-complexity of solving CONTC increases with the number of events, the overall time-complexity is given by $O(n|C|T + n^2|E|T)$, where $O(T)$ is the worst-case time-complexity of solving CONTC for a computation consisting of $|E|$ events. Note that the time-complexity of executing lines 1-3, over $|E|$ events, is given by $O(|E|^2)$, which can be ignored assuming that $T = \Omega(|E|)$. Finally, the amortized time-complexity for updating the slice *once*—on arrival of an event—is given by $O(n(c+n)T)$, where $c = |C|/|E|$ denotes the average concurrency in the computation. Formally,

Theorem 4.26 *The time-complexity of the algorithm to update the slice on arrival of a new event, described in Figure 4.8 and Figure 4.9, amortized over $|E|$ events, is $O(n(c + n)T)$, where n is the number of processes, c is the average concurrency in the computation and $O(T)$ is the worst-case time-complexity of solving* CONTC *for a computation consisting of $|E|$ events.*

In case c is low, say $O(n)$, the on-line algorithm has an amortized time-complexity of $O(n^2T)$. In this case, therefore, rather than computing the slice from scratch whenever an event arrives, it is much faster to *update* it using the incremental algorithm. The (on-line) algorithm in this section only assumes that the predicate can be detected efficiently; no other assumption is made about the structure of the predicate. For a special class of predicates, however, namely non-temporal regular predicates, we have developed a much faster $O(n^2)$ amortized time-complexity algorithm to compute the slice in an on-line manner [MSGA03].

4.4.2 Technical Details

Lemma 4.22 *Given an event $e \in E^{(k-1)}$ and a process P_j,*

$$(j \neq j_k) \vee (e \notin critical(k)) \quad \Rightarrow \quad F_p^{k-1}(e)[j] \overset{P}{\underline{\rightarrow}} F_p^k(e)[j] \tag{4.1}$$

$$(j = j_k) \wedge (e \in critical(k)) \quad \Rightarrow \quad F_p^k(e)[j] \in \{e^{(k)}, \top_j\} \tag{4.2}$$

Proof: First, consider an event $f \in E^{(k-1)}$ on process P_j. We have,

$$\begin{array}{ll}
 & f \overset{P}{\rightarrow} F_p^{k-1}(e)[j] \\
\equiv & \{ \; F_p^{k-1}(e)[j] \text{ is the earliest event on } P_j \text{ reachable from } e \text{ in } \mathsf{slice}(G^{(k-1)}, p) \; \} \\
 & (e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k-1)}, p)) \\
\equiv & \{ \; f \in E^{(k-1)} \text{ and using Theorem 4.15 } \} \\
 & \langle \exists \, C : C \text{ is a consistent cut of } \widehat{G}^{(k-1)}[e, f] : C \text{ satisfies } p \rangle \\
\equiv & \{ \text{ definition of } \widehat{G}^{(k-1)}[e, f] \; \} \\
 & \langle \exists \, C : C \text{ is a consistent cut of } G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } p) \rangle \\
\Rightarrow & \{ \text{ Observation 4.19 } \} \\
 & \langle \exists \, C : C \text{ is a consistent cut of } G^{(k)} : (f \in C) \wedge (e \notin C) \wedge (C \text{ satisfies } p) \rangle \\
\equiv & \{ \text{ definition of } \widehat{G}^{(k)}[e, f] \; \} \\
 & \langle \exists \, C : C \text{ is a consistent cut of } \widehat{G}^{(k)}[e, f] : C \text{ satisfies } p \rangle \\
\equiv & \{ \text{ using Theorem 4.15 } \} \\
 & (e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k)}, p)) \\
\equiv & \{ \; F_p^{k}(e)[j] \text{ is the earliest event on } P_j \text{ reachable from } e \text{ in } \mathsf{slice}(G^{(k)}, p) \; \} \\
 & f \overset{P}{\rightarrow} F_p^{k}(e)[j]
\end{array}$$

In other words, whenever $f \in E^{(k-1)}$, $f \overset{P}{\rightarrow} F_p^{k-1}(e)[j]$ implies that $f \overset{P}{\rightarrow} F_p^k(e)[j]$. Now, we prove the two implications.

Equation (4.1) Consider an event $f \in E^{(k)}$ on process P_j. Suppose $f \overset{P}{\rightarrow} F_p^{k-1}(e)[j]$. In case j is different from j_k, f is different from $e^{(k)}$ and therefore $f \in E^{(k-1)}$. On the other hand, if $e \notin critical(k)$, then $F_p^{k-1}(e)[j] \overset{P}{\rightarrow} e^{(k)}$ and therefore

$f \in E^{(k-1)}$. In either case, $f \in E^{(k-1)}$. Using the above result, $f \xrightarrow{P} F_p^k(e)[j]$. In other words, whenever $f \in E^{(k)}$, $f \xrightarrow{P} F_p^{k-1}(e)[j]$ implies that $f \xrightarrow{P} F_p^k(e)[j]$. This, in turn, means that $F_p^{k-1}(e)[j] \overset{P}{\underline{\rightarrow}} F_p^k(e)[j]$.

Equation (4.2) Consider an event $f \in E^{(k)}$ on process P_j, where $j = j_k$. Suppose $f \xrightarrow{P} e^{(k)}$. Clearly, $f \in E^{(k-1)}$. Further, in case e is a critical event, $f \xrightarrow{P} F_p^{k-1}(e)[j]$. Using the above result, $f \xrightarrow{P} F_p^k(e)[j]$. In other words, whenever $f \in E^{(k)}$, $f \xrightarrow{P} e^{(k)}$ implies that $f \xrightarrow{P} F_p^k(e)[j]$. This, in turn, means that $e^{(k)} \overset{P}{\underline{\rightarrow}} F_p^k(e)[j]$. $\square$

Lemma 4.23 *Given an event $e \in E^{(k-1)}$ and a process P_j,*

$$(e \rightarrow e^{(k)}) \wedge \Big((j \neq j_k) \vee (e \notin critical(k))\Big) \;\Rightarrow\; F_p^{k-1}(e)[j] = F_p^k(e)[j] \quad (4.3)$$

$$(e \rightarrow e^{(k)}) \wedge \Big((j = j_k) \wedge (e \in critical(k))\Big) \;\Rightarrow\; F_p^k(e)[j] = e^{(k)} \quad (4.4)$$

Proof: *Equation (4.3)* From Lemma 4.22, $F_p^{k-1}(e)[j] \overset{P}{\underline{\rightarrow}} F_p^k(e)[j]$. Assume, on the contrary, that $F_p^{k-1}(e)[j] \xrightarrow{P} F_p^k(e)[j]$. For convenience, let $f = F_p^{k-1}(e)[j]$. We have,

$f \xrightarrow{P} F_p^k(e)[j]$

$\equiv$ { $F_p^k(e)[j]$ is the earliest event on P_j reachable from e in $\mathsf{slice}(G^{(k)}, p)$ }

$(e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k)}, p))$

$\equiv$ { using Theorem 4.15 }

$\langle \exists\, C : C$ is a consistent cut of $\widehat{G}^{(k)}[e, f] : C$ satisfies $p\rangle$

$\equiv$ { definition of $\widehat{G}^{(k)}[e, f]$ }

$\langle \exists\, C : C$ is a consistent cut of $G^{(k)} : (f \in C) \wedge (e \notin C) \wedge (C$ satisfies $p)\rangle$

$\Rightarrow$ { $e \rightarrow e^{(k)}$ }

$\langle \exists C : C$ is a consistent cut of $G^{(k)}$: $(f \in C) \wedge (e \notin C) \wedge (C$ satisfies $p) \wedge$ $(e^{(k)} \notin C)\rangle$

$\equiv$ { a consistent cut of $G^{(k)}$ that does not contain $e^{(k)}$ is a consistent cut of $G^{(k-1)}$ }

$\langle \exists C : C$ is a consistent cut of $G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C$ satisfies $p) \wedge$

$(e^{(k)} \notin C)\rangle$

$\equiv$ { $e^{(k)}$ is not an event in $G^{(k-1)}$ }

$\langle \exists\, C : C$ is a consistent cut of $G^{(k-1)} : (f \in C) \wedge (e \notin C) \wedge (C$ satisfies $p)\rangle$

$\equiv$ { definition of $\widehat{G}^{(k-1)}[e, f]$ }

$\langle \exists\, C : C$ is a consistent cut of $\widehat{G}^{(k-1)}[e, f] : C$ satisfies $p\rangle$

$\equiv$ { using Theorem 4.15 }

$(e, f) \notin \mathcal{P}(\mathsf{slice}(G^{(k-1)}, p))$

$\Rightarrow$ { f is $F_p^{k-1}(e)[j]$ }

a contradiction

Equation (4.4) Note that $\mathcal{C}(\mathsf{slice}(G^{(k)}, p)) \subseteq \mathcal{C}(G^{(k)})$. In case $e \rightarrow e^{(k)}$, there is a path from e to $e^{(k)}$ in $G^{(k)}$. Thus, from Lemma 4.8, there is a path from e to $e^{(k)}$ in $\mathsf{slice}(G^{(k)}, p)$ as well. Consequently, $F_p^k(e)[j] \xrightarrow{P} e^{(k)}$. From Lemma 4.22, $F_p^k(e)[j]$ is either $e^{(k)}$ or $\top_j$. This, in turn, implies that $F_p^k(e)[j]$ is $e^{(k)}$. □

Lemma 4.24 *Given a process P_j,*

$$j \neq j_k \quad \Rightarrow \quad F_p^k(e^{(k)})[j] = F_p^{k-1}(\top_{j_k})[j] \tag{4.5}$$

$$j = j_k \quad \Rightarrow \quad F_p^k(e^{(k)})[j] = \min\{e^{(k)},\ F_p^{k-1}(\top_{j_k})[j]\} \tag{4.6}$$

Proof: Consider an event $f \in E^{(k)}$ with $f \neq e^{(k)}$. We have,

$\langle \exists\, C : C$ is a consistent cut of $\widehat{G}^{(k)}[e^{(k)}, f] : C$ satisfies $p\rangle$

$\equiv$ { definition of $\widehat{G}^{(k)}[e^{(k)}, f]$ }

$\langle \exists\, C : C$ is a consistent cut of $G^{(k)} : (f \in C) \wedge (e^{(k)} \notin C) \wedge (C$ satisfies $p)\rangle$

$\equiv$ { a consistent cut of $G^{(k)}$ that does not contain $e^{(k)}$ is a consistent cut of $G^{(k-1)}$ }

$\langle \exists\, C : C$ is a consistent cut of $G^{(k-1)} : (f \in C) \wedge (e^{(k)} \notin C) \wedge (C$ satisfies $p)\rangle$

$\equiv$ { $e^{(k)}$ is not an event in $G^{(k-1)}$ }

$\langle \exists\, C : C$ is a consistent cut of $G^{(k-1)} : (f \in C) \wedge (C$ satisfies $p)\rangle$

$\equiv$ { definition of $\widehat{G}^{(k-1)}[\top_{j_k}, f]$ }
$\langle \exists\, C : C$ is a consistent cut of $\widehat{G}^{(k-1)}[\top_{j_k}, f] : C$ satisfies $p\rangle$

Equation (4.5) Clearly, using the above result and Theorem 4.15, $F_p^k(e^{(k)})[j] = F_p^{k-1}(\top_{j_k})[j]$.

Equation (4.6) In case $F_p^{k-1}(\top_{j_k})[j]$ is different from $\top_{j_k}$, using the above result and Theorem 4.15, $F_p^k(e^{(k)}) = F_p^{k-1}(\top_{j_k})[j]$. On the other hand, if $F_p^{k-1}(\top_{j_k})[j] = \top_{j_k}$, then $\mathsf{CONTC}(\widehat{G}^{(k)}[e^{(k)}, e^{(k)}], p)$ evaluates to false and therefore $F_p^k(e^{(k)})[j] = e^{(k)}$. $\square$

Chapter 5

Slicing for Temporal Predicates

In this chapter, we discuss our results in slicing for temporal predicates.

5.1 Overview

Many different methods have been devised for automatically checking temporal logic predicates on execution traces by examining the state space models of system behavior. These methods all depend on decision procedures that explicitly represent the state space using a list or a table that grows exponentially in the number of processes, components, or state elements (*state explosion problem*). In this chapter we develop algorithms that represent the state space that satisfies a temporal logic predicate using a "slice" instead of using an explicit representation.

We first define the problem formally in Section 5.2. Informally, the slice of a computation with respect to a temporal predicate contains all consistent cuts of the slice that satisfies the temporal predicate. In [MG01a], it was shown that computing slices with respect to an arbitrary predicate is an intractable problem and efficient slicing algorithms with respect to non-temporal regular predicates were presented. Regularity of a predicate plays an important role in developing efficient slicing algorithms for temporal predicates as well. In Chapter 3, we defined a regular subset of CTL called RCTL, which contains four temporal operators EF, AG, EG, and $\mathsf{EX}[j]$ and where the atomic propositions are non-temporal regular predicates. We obtained this subset by proving in Chapter 3 that temporal predicates $\mathsf{EF}(p)$, $\mathsf{AG}(p)$, $\mathsf{EG}(p)$, and $\mathsf{EX}(p)[j]$ are regular when p is regular.

Our slicing algorithms for temporal regular predicates have an overall time-complexity of $O(n|E|)$, where n is the number of processes and E is the set of events in a given computation. The complexity of computing the slice with respect to a predicate from RCTL is dominated by the complexity of computing the slice for an atomic proposition, which is a non-temporal regular predicate. The slicing algorithm complexity is $O(n^2|E|)$ for a non-temporal regular predicate. Therefore, the complexity of computing the slice for a predicate from RCTL is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p. We present our slicing algorithms for temporal regular predicates in RCTL in Section 5.3.

In Section 5.4 we show that we can use our slicing algorithms for temporal regular predicates to obtain slices for temporal predicates in $\mathsf{RCTL}+$, which extends RCTL with temporal operator EX and boolean operator $\vee$ and where the atomic propositions are non-temporal predicates. The slicing algorithm complexity is $O(n|E|T)$ for a non-temporal predicate as we showed in the previous chapter, where T is the complexity of detecting the predicate under EF operator. Therefore, the complexity of computing the slice for a predicate from $\mathsf{RCTL}+$ is $O(|p| \cdot n|E|T)$r.

Note that in this chapter we do not assume the efficient predicate evaluation and the efficient advancement properties (Property 3.15 and Property 3.17 of Section 3.3) in developing our slicing algorithms for temporal predicates. This is because, an efficient way of computing the crucial element (efficient advancement property) for a temporal predicate is not known. Also, our goal in this dissertation is to develop efficient predicate detection algorithms to determine the satisfiability of temporal predicates, so assuming the efficient predicate evaluation property defeats this purpose. Therefore, in particular, we cannot use Garg and Mittal slicing algorithm for non-temporal regular predicates to compute slices for temporal regular predicates. Similarly, we cannot use our slicing algorithm from previous chapter to obtain an efficient temporal predicate slicing algorithm. That algorithm assumes the presence of an efficient $\mathsf{EF}(p)$ algorithm. This implies that when p is temporal such as $\mathsf{EG}(p)$, we need an efficient algorithm to detect $\mathsf{EF}(\mathsf{EG}(p))$. Although there are efficient algorithms to detect $\mathsf{EG}(p)$ (which we will show in next chapter), those algorithms can not be used when there is a nesting of temporal operators. Intuitively, $\mathsf{EF}(\mathsf{EG}(p))$ checks whether there exists *any* consistent cut that satisfies $\mathsf{EG}(p)$, whereas the algorithms in the literature (and which we will present in the next chapter for p linear) checks whether the *initial* consistent cut satisfies $\mathsf{EG}(p)$.

5.2 Problem Statement

In this chapter we are concerned with computing those consistent cuts of a computation that satisfy a temporal predicate. In other words, given a computation G and a temporal predicate $\mathsf{AG}(p)$, we are interested in computing consistent cuts $C \in \mathcal{C}(G)$ such that $G, C \models \mathsf{AG}(p)$. The semantics of a temporal predicate is given with respect to a graph. More formally, a cut C that belongs to a graph G satisfies $\mathsf{AG}(p)$ if for all fullpaths π starting from C, $\langle \forall\, i : 0 \leqslant i < |\pi| : \pi^i \models p \rangle$. A *full-path* $C_0, C_1, \ldots, C_k = \mathcal{E}$ of the distributive lattice $(\mathcal{C}(G), \subseteq)$ satisfies that for each $0 \leqslant i < k$, $C_i \rhd C_{i+1}$. This implies that the successor relation $\rhd$ is also defined with respect to G. Formally,

Definition 5.1 (slice for temporal predicates) *A* slice *of a graph G with respect to a temporal predicate is a directed graph obtained from G by adding edges such that:*

(1) it contains all consistent cuts of the computation that satisfy the temporal predicate and
(2) of all the graphs that satisfy (1), it has the least number of consistent cuts.

We denote the slice of a computation $G = \langle E, \rightarrow \rangle$ with respect to a temporal predicate $\mathsf{AG}(p)$ by $\mathsf{slice}(G, \mathsf{AG}(p))$.

5.3 Slicing for RCTL Predicates

From the definition of a slice, we know that the slice of a computation is obtained by adding edges to the computation. In the previous chapter, we showed an efficient way of adding edges to compute the slice of a computation with respect to non-temporal predicates. In this chapter, we show an efficient way of adding edges to compute the slice of a computation with respect to temporal predicates.

Clearly, every nontrivial consistent cut of $\mathsf{slice}(G, p)$ is a consistent cut of G. From the definition of $\mathsf{AG}(p)$ we know that every consistent cut that satisfies $\mathsf{AG}(p)$ also satisfies p. This implies that every nontrivial consistent cut of $\mathsf{slice}(G, \mathsf{AG}(p))$ is a consistent cut of $\mathsf{slice}(G, p)$. Similarly for $\mathsf{EG}(p)$.

Observation 5.2

$$\mathcal{C}(\mathsf{slice}(G, \mathsf{AG}(p))) \subseteq \mathcal{C}(\mathsf{slice}(G, p)) \subseteq \mathcal{C}(G)$$

$$\mathcal{C}(\mathsf{slice}(G, \mathsf{EG}(p))) \subseteq \mathcal{C}(\mathsf{slice}(G, p)) \subseteq \mathcal{C}(G)$$

Furthermore, using Lemma 4.8, we obtain

Observation 5.3

$$\mathcal{P}(\mathsf{slice}(G, \mathsf{AG}(p))) \supseteq \mathcal{P}(\mathsf{slice}(G, p)) \supseteq \mathcal{P}(G)$$

$$\mathcal{P}(\mathsf{slice}(G, \mathsf{EG}(p))) \supseteq \mathcal{P}(\mathsf{slice}(G, p)) \supseteq \mathcal{P}(G)$$

which in turn implies that the slice with respect to $\mathsf{AG}(p)$ (or $\mathsf{EG}(p)$) can be obtained by adding edges to the slice for p.

Since the set of consistent cuts that satisfy $\mathsf{EF}(p)$ and $\mathsf{EX}(p)[j]$ may not be a subset of the set of consistent cuts of the slice for p, the slice with respect to $\mathsf{EF}(p)$ and $\mathsf{EX}(p)[j]$ cannot always be obtained by adding edges to the slice for p. Rather, we obtain the slice with respect to these temporal predicates by adding edges to the computation G.

Every slicing algorithm in this section takes as input a graph G, and its slice with respect to a regular predicate p. The output of each algorithm is the slice of G with respect to a temporal predicate such as $\mathsf{EF}(p)$.

In Figure 5.2(a) and (d), we show two slices of the computation given in Figure 5.1. The lattices of consistent cuts of the slices are displayed in Figure 5.2(b) and (e). In this chapter, we chose to display the set of consistent cuts of the slices as in Figure 5.2(c) and (f), that is, explicitly showing those consistent cuts of the computation that satisfy the predicate and that do not satisfy the predicate. This makes it easier to see which consistent cuts of the computation belong to the slice.

5.3.1 Computing Slices with respect to $\mathsf{EF}(p)$

In this section, we explain algorithm $\mathsf{Algo}_{5.1}$ in Figure 5.3 for computing the slice of a graph G with respect to $\mathsf{EF}(p)$.

Consider a graph G and $\mathsf{slice}(G, p)$. Let W denote the greatest cut that satisfies p. Notice that W is the final consistent cut of $\mathsf{slice}(G, p)$. In the algorithm, we construct a graph H with vertices as the vertices in G and the following edges:

1. all the edges in G, and
2. from $\top$ to the successors of events in $\mathit{frontier}(W)$.

The first type of edges ensure that the consistent cuts of H are a subset of

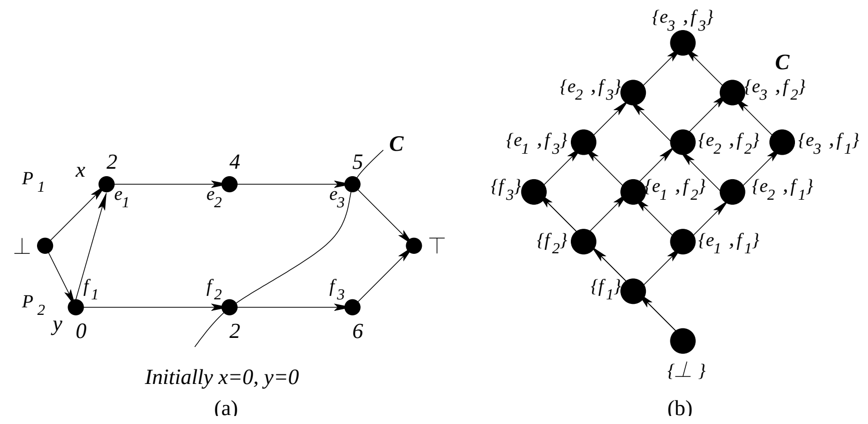


Figure 5.1: (a) A computation, and (b) the set of consistent cuts

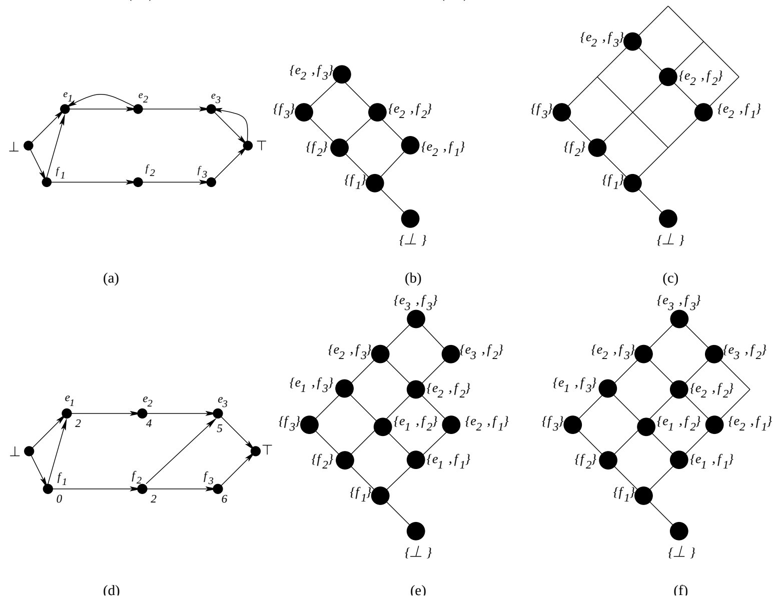


Figure 5.2: (a) A slice of the computation above, (b) the set of consistent cuts, (c) the set of consistent cuts displayed as a subset of the set of consistent cuts of the computation, (d) A slice of the computation above, (e) the set of consistent cuts, (f) the set of consistent cuts displayed as a subset of the set of consistent cuts of the computation

```
Algorithm Algo$_{5.1}$:

    Input: (1) a directed graph $G$, and (2) slice($G, p$)
    Output: slice($G$, EF($p$))

1   $H := G$;
2   if slice($G, p$) is nonempty then
3       $W :=$ the final consistent cut of slice($G, p$);
4       $\forall\ e \in frontier(W)$: add an edge from the vertex $\top$ to $succ(e)$ in $H$;
    else
5       add an edge from the vertex $\top$ to $\bot$ in $H$;        // $H$ becomes an empty slice
    endif;
6   return $H$;
```

Figure 5.3: The algorithm to compute slice(G, EF(p)).

the consistent cuts of G. The second type of edges ensure that the final consistent cut of H is W, therefore all consistent cuts of G that can reach W is a consistent cut of H. From the definition of EF(p), all consistent cuts of the computation that can reach the greatest consistent cut that satisfies p, that is W, will also satisfy EF(p) and furthermore these are the only cuts that satisfy EF(p). We can find the cut W using slice(G, p) when it is nonempty. We construct the slice for EF(p) from the computation so that the slice contains all consistent cuts of the computation that can reach W. To ensure that all cuts that cannot reach W do not belong to the slice, we add edges from $\top$ to the successors of events in the frontier of W in the computation. Note that adding an edge from $\top$ to an event makes any cut that contains the event trivial.

Consider the computation depicted in Figure 5.1(a) and its slice depicted in Figure 5.4(a). The application of algorithm Algo$_{5.1}$ is shown in Figure 5.4(c). The frontier of the final cut of the slice in Figure 5.4(a) is $frontier(W) = \{e_2, f_3\}$. After adding edges from $\top$ to the successor of e_2, that is e_3, and to the successor of f_3, that is $\top$, we obtain the slice with respect to EF(p) depicted in Figure 5.4(c).

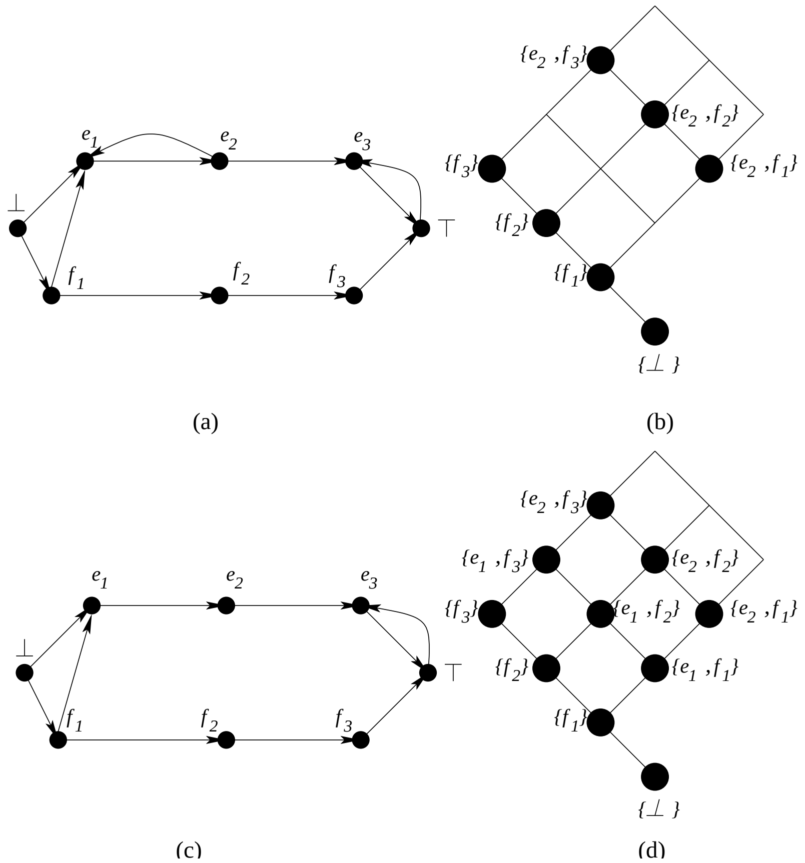


Figure 5.4: (a) A slice of the computation in Figure 5.1(a), (b) the corresponding sublattice, (c) The application of the temporal operator EF on the slice in (a), (d) the corresponding sublattice

We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy $\mathsf{EF}(p)$.

Lemma 5.4 *Every consistent cut of H satisfies* $\mathsf{EF}(p)$.

Proof: Consider a consistent cut C of H. $\mathsf{slice}(G, p)$ must be nonempty in order for C to exist, otherwise H is an empty slice. Since the final consistent cut of H is W, $C \subseteq W$. Therefore, C satisfies $\mathsf{EF}(p)$. $\Box$

Next, we show that the above constructed graph retains all consistent cuts of the computation that satisfy $\mathsf{EF}(p)$.

Lemma 5.5 *Every consistent cut of G that satisfies $\mathsf{EF}(p)$ is a consistent cut of H.*

Proof: Consider a consistent cut C of G that satisfies $\mathsf{EF}(p)$. In this case, $\mathsf{slice}(G, p)$ is nonempty. Assume, on the contrary, that C is not a consistent cut of H. Thus, there exist events e and f such that there is an edge from e to f in H, f belongs to C but e does not. Since C is a consistent cut of G, the edge from e to f could be only of type (2). This implies that C contains an event from the successors of events in W. Since C satisfies $\mathsf{EF}(p)$, there exist a cut $D \supseteq C$ such that D satisfies p. Since W is the greatest cut that satisfies p, $D \subseteq W$. However, D contains an event from the successors of events in W so $D \not\subseteq W$—a contradiction. $\square$

From the previous two lemmas, it follows that:

Theorem 5.6 *H is cut-equivalent to $\mathsf{slice}(G, \mathsf{EF}(p))$.*

Complexity Analysis 5.7 *The graph H has $O(|E|)$ vertices, $O(|E|)$ edges (in fact $O(n+|E|)$ edges, but we assume that n is much smaller than $|E|$) and can be built in $O(n|E|)$ time as explained next. The slice with respect to a regular predicate contains $O(n|E|)$ edges using the skeletal representation. The nonemptiness check at line 2 can be done by checking whether the number of strongly connected components of the input slice is greater than one, which takes $O(n|E|)$ time. We can compute the final consistent cut of this slice, that is W, by proceeding backwards from vertex $\top$ as follows: First, we compute the strongly connected component of the slice that contains $\top$, in $O(n|E|)$ time. Second, for each process P_i, starting from the final event on P_i, we find the predecessors of events until we reach events on P_i that do not belong to the strongly connected component. This step takes $O(|E_i|)$ time.*

```
Algorithm Algo$_{5.2}$:

    Input: (1) a directed graph $G$, and (2) slice$(G, p)$
    Output: slice$(G, \mathsf{AG}(p))$

1   $H :=$ slice$(G, p)$;                  // $F_{\mathsf{AG}(p)}(e) = F_p(e)$
2   for each process $P_i$ do
3       for each event $e$ on $P_i$ do              // visited in the order given by $\xrightarrow{P}$
4           for each process $P_j$ do
5               $f := F_p(e)[j]$;
6               if $f \xrightarrow{P} F(e)[j]$ then        // $(e, f) \in \mathcal{P}(\text{slice}(G, p)) \wedge (e, f) \notin \mathcal{P}(G)$
7                   add an edge from $e$ to $\perp$ in $H$;    // $F_{\mathsf{AG}(p)}(e)[j] = \perp$
8                   continue;              // no need to add an edge from $e$ to $\perp$ again
                endif;
            endfor;
        endfor;
    endfor;
9   return $H$;
```

Figure 5.5: The algorithm to compute slice$(G, \mathsf{AG}(p))$ using $F_p(e)$ for each event e.

Summing up for all processes, we can compute the frontier of W in $O(|E|)$ time. Finally, there are n successors of the events in the frontier of W, therefore adding edges from $\top$ to these successor events take $O(n)$ time. Thus the algorithm has $O(n|E|)$ overall time-complexity.

5.3.2 Computing Slices with respect to $\mathsf{AG}(p)$

In this section, we explain algorithm Algo$_{5.2}$ in Figure 5.5 for computing the slice of a graph G with respect to $\mathsf{AG}(p)$.

Consider a graph G and its slice(G, p). In the algorithm, we construct a graph H with vertices as the vertices in G and the following edges:

1. all the edges in slice(G, p), and
2. from e to $\perp$ if $F_p(e)[j] \xrightarrow{P} F(e)[j]$.

The first type of edges ensure that the consistent cuts of H are a subset of the consistent cuts of $\mathsf{slice}(G, p)$. This follows from Observation 5.2 and Observation 5.3. The second type of edges ensure that consistent cuts of $\mathsf{slice}(G, p)$ that do not include vertex e of an additional edge (e, f) do not belong to $\mathsf{slice}(G, \mathsf{AG}(p))$, whereas the rest of the consistent cuts belong to $\mathsf{slice}(G, \mathsf{AG}(p))$. The semantics of $\mathsf{AG}(p)$ is given with respect to a graph and a consistent cut. Notice that a consistent cut C of $\mathsf{slice}(G, p)$ (which is also a consistent cut of G) satisfies $\mathsf{AG}(p)$ *with respect to* G, if all fullpaths starting from C in G also exist in $\mathsf{slice}(G, p)$. This is because all cuts in $\mathsf{slice}(G, p)$ satisfies p. When $\mathsf{slice}(G, p)$ contains an additional edge (e, f) that does not exist in G, some fullpaths that start from C in G, namely the ones in which f is included (executed) before e, do not exist in $\mathsf{slice}(G, p)$. Hence, C does not satisfy $\mathsf{AG}(p)$. Since every nontrivial consistent cut includes $\perp$, by adding an edge from e to $\perp$ (by the definition of a consistent cut), e is included in every nontrivial consistent cut of H.

Consider the computation depicted in Figure 5.1(a) and its slice depicted in Figure 5.6(a), which contains an additional edge (f_2, e_3). The application of algorithm $\mathsf{Algo}_{5.2}$ is shown in Figure 5.6(c). The consistent cuts $\{\perp\}$, $\{f_1\}$, $\{e_1, f_1\}$, and $\{e_2, f_1\}$ of the slice in Figure 5.6(a) do not include vertex f_2 of the additional edge (f_2, e_3). Hence, it is easy to see that these four consistent cuts do not satisfy $\mathsf{AG}(p)$. Therefore we add an edge from vertex f_2 to $\perp$ and obtain the slice with respect to $\mathsf{AG}(p)$ depicted in Figure 5.6(c). Note that we exploit the transitivity of the edge relation and not display other added edges such as the edge from vertex f_1 to $\perp$.

We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy $\mathsf{AG}(p)$.

In order to check for the existence of an edge that does not exist in the graph but in the slice, it suffices to check the $F(e)$. The next lemma follows from the definition of $F(e)$.

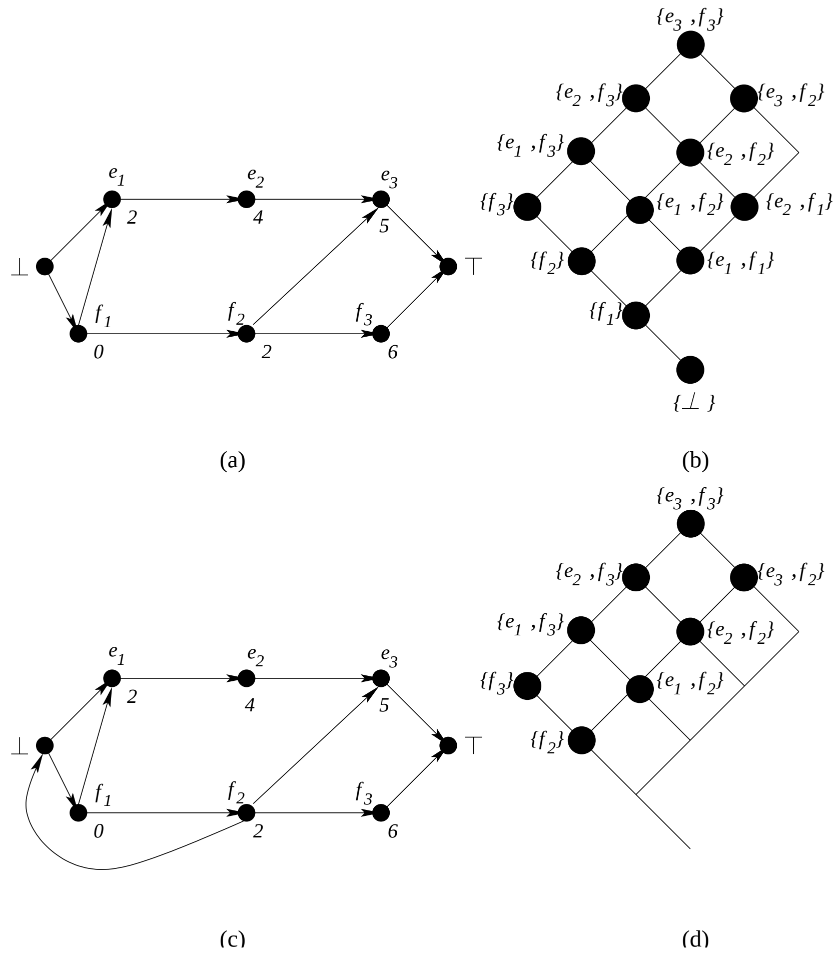


Figure 5.6: (a) A slice of the computation in Figure 5.1(a), (b) the corresponding sublattice, (c) The application of the temporal operator AG on the slice in (a), (d) the corresponding sublattice

Lemma 5.8 *Given a graph G, there exist vertices e and f in $\mathsf{V}(G)$ such that $(e, f) \in \mathcal{P}(\mathsf{slice}(G, q)) \wedge (e, f) \notin \mathcal{P}(\mathsf{slice}(G, r)) \iff F_q(e)[j] \xrightarrow{P} F_r(e)[j]$, where $f = F_q(e)[j]$ and $j = proc(f)$.*

Proof:

($\Rightarrow$) If $(e, f) \in \mathcal{P}(\mathsf{slice}(G, q))$ and $(e, f) \notin \mathcal{P}(\mathsf{slice}(G, r))$ then there is no path from

e to any vertices before f (in process order) on P_j in $\mathsf{slice}(G, r)$. We also know that there is an edge from e to $F_r(e)[j]$ in $\mathsf{slice}(G, r)$. Then $F_r(e)[j] \overset{P}{\not\to} f$.

($\Leftarrow$) If $F_q(e)[j] \overset{P}{\to} F_r(e)[j]$ then there is an edge from e to f in $\mathsf{slice}(G, q)$ but no such edge exists in $\mathsf{slice}(G, r)$. $\Box$

Lemma 5.9 *Every consistent cut of H satisfies* $\mathsf{AG}(p)$.

Proof: Consider a consistent cut C of H. Assume, on the contrary, that C does not satisfy $\mathsf{AG}(p)$. Thus, there exists a consistent cut $D \supseteq C$ in G such that D does not satisfy p. Therefore, D does not belong to $\mathsf{slice}(G, p)$. This implies that there exist events e and f such that there is an edge from $(e, f) \in \mathsf{slice}(G, p)$, f belongs to D but e does not. Setting $q = p$, $r = \mathit{true}$, and $j = \mathit{proc}(f)$ in Lemma 5.8, $F_p(e)[j] \overset{P}{\to} F(e)[j]$. This means that there exists an edge from e to $\perp$ of type (2) in H. We know that every nontrivial consistent cut of every directed graph contains $\perp$. Since there is an edge $(e, \perp)$ in H, every nontrivial consistent cut also contains e. This implies that C contains e. However, since $e \notin D$ and $C \subseteq D$, we have $e \notin C$—contradiction. $\Box$

Lemma 5.10 *Every consistent cut of G that satisfies* $\mathsf{AG}(p)$ *is a consistent cut of H.*

Proof: Consider a consistent cut C of G that satisfies $\mathsf{AG}(p)$. Assume, on the contrary, that C is not a consistent cut of H. Thus, there exist events e and f such that there is an edge from e to f in H, f belongs to C but e does not. Since C is a consistent cut of G, the edge from e to f could be only of type (1) or of type (2). If the edge is of type (1) then C does not belong to $\mathsf{slice}(G, p)$ since e does not belong to C. Thus C does not satisfy p. However, C satisfies $\mathsf{AG}(p)$—contradiction. If the edge is of type (2) then $f = \perp$ and there exists a process j such that $F_p(e)[j] \overset{P}{\to} F(e)[j]$. Equivalently, there is an edge from e to $F_p(e)[j]$ in

$\mathsf{slice}(G,p)$ that does not exist in G. For convenience, let $g = F_p(e)[j]$. If $g \in C$ then C does not belong to $\mathsf{slice}(G,p)$ since C does not contain e. Thus C does not satisfy p. However, C satisfies $\mathsf{AG}(p)$—contradiction. If $g \notin C$ then consider a path in G from C to the final consistent cut in which g is executed before e (it is always possible to find such a path because there is no path from e to g in G). Consider a consistent cut immediately after the execution of g, say D. Again D does not belong to $\mathsf{slice}(G,p)$ because D contains g but not e. Thus D does not satisfy p. It follows that there exists a consistent cut D on a path from C to the final consistent cut such that D does not satisfy p. Hence, C does not satisfy $\mathsf{AG}(p)$—contradiction.

□

From the previous two lemmas, it follows that:

Theorem 5.11 *H is cut-equivalent to* $\mathsf{slice}(G, \mathsf{AG}(p))$.

Remark 5.12 *The semantics of* $\mathsf{AG}(p)$ *is given with respect to a graph and a consistent cut. Notice that a consistent cut C of* $\mathsf{slice}(G,p)$ *(which is also a consistent cut of G) satisfies* $\mathsf{AG}(p)$ with respect to G, *if all fullpaths starting from C in G also exist in* $\mathsf{slice}(G,p)$. *A fullpath $C_0, C_1, \ldots, C_k = \mathcal{E}$ of the distributive lattice $(\mathcal{C}(G), \subseteq)$ satisfies that for each $0 \leqslant i < k$, $C_i \triangleright C_{i+1}$. Observe that the successor relation is also defined with respect to a graph.*

Let $H =$ $\mathsf{slice}(G,p)$. Since p holds for all consistent cuts of $C \in \mathcal{C}(H)$, we have that $H, C \models$ $\mathsf{AG}(p)$. The consistent cuts of the slice belong to the the set of consistent cuts of the computation as well, and one might falsely infer that for all $C \in \mathcal{C}(H)$, we have that $G, C \models$ $\mathsf{AG}(p)$. However, this may not be true since some fullpaths starting from C in G may not exist in $\mathsf{slice}(G,p)$.

For example, consider the slice in Figure 5.2(a). The consistent cut $\{e_2, f_1\}$ satisfies $\mathsf{AG}(p)$ *with respect to the slice. However, the same consistent cut does not satisfy* $\mathsf{AG}(p)$ *with respect to the computation. This is because there exists a* *consistent cut $\{e_3, f_1\}$ that does not satisfy p on a fullpath starting from $\{e_2, f_1\}$ in the computation, which can be easily seen in Figure 5.2(c). However, such a fullpath does not exist in the slice. In other words, $\{e_2, f_1\}$, $\{e_3, f_1\}$, $\{e_3, f_2\}$, $\{e_3, f_3\}$ is a fullpath of the computation, yet it is not a fullpath of the slice in Figure 5.2(c). Similarly, $\{f_1\}$, $\{f_2\}$, $\{e_2, f_2\}$, $\{e_2, f_3\}$ is a fullpath of the slice in Figure 5.2(a), however, it is not a fullpath of the computation G.*

We assume the computation is given to us as n queues of events—one for each process. Further, the vector clock for each event e is available to us, that is $J(e)$, from which $F(e) = F_{true}(e)$ can be easily computed using algorithm $\mathsf{Algo}_{4.1}$ in $O(n|E|)$ time. Furthermore, we assume that the skeletal representation of $\mathsf{slice}(G, p)$ and thus $F_p(e)$ is given.

Complexity Analysis 5.13 *The graph H has $O(|E|)$ vertices, $O(n|E|)$ edges (one edge is added for each event e in the graph at line 7 and $\mathsf{slice}(G, p)$ has at most $O(n|E|)$ edges) and can be built in $O(n|E|)$ time as explained next. Let E_i denote the set of events on process P_i. The for loop at line 4 is executed exactly n times. Each iteration of the for loop has $O(1)$ time-complexity because the condition at line 6 can be ascertained by a single comparison and adding an edge takes constant time. Thus the complexity of for loop at line 3 is $O(n|E_i|)$. Summing up over all processes, we have $O(n|E|)$ time-complexity.*

5.3.3 Computing Slices with respect to $\mathsf{EG}(p)$

In this section, we explain algorithm $\mathsf{Algo}_{5.2}$ in Figure 5.7 for computing the slice of a graph G with respect to $\mathsf{EG}(p)$.

The algorithm for $\mathsf{EG}(p)$ slicing displayed in Figure 5.7 is similar to the $\mathsf{AG}(p)$ slicing algorithm. However in this case, for each additional edge (e, f) *that generates a nontrivial strongly connected component* in $\mathsf{slice}(G, p)$, we add an edge from the vertex e to the vertex $\bot$. This is different from the algorithm for $\mathsf{AG}(p)$ where for

```
Algorithm Algo$_{5.3}$:

    Input: (1) a directed graph $G$, and (2) slice$(G, p)$
    Output: slice$(G, \mathsf{EG}(p))$

1   $H :=$ slice$(G, p)$;                      // $F_{\mathsf{EG}(p)}(e) = F_p(e)$
2   for each process $P_i$ do
3       for each event $e$ on $P_i$ do                // visited in the order given by $\xrightarrow{P}$
4           for each process $P_j$ do
5               $f := F_p(e)[j]$;
6               if $f \xrightarrow{P} F(e)[j]$ and $F_p(f)[i] \overset{P}{\underline{\rightarrow}} e$ then
                                   // $(e, f), (f, e) \in \mathcal{P}$(slice$(G, p)$) and $(e, f) \notin \mathcal{P}(G)$
7                   add an edge from $e$ to $\perp$ in $H$;     // $F_{\mathsf{EG}(p)}(e)[j] = \perp$
8                   continue;              // no need to add an edge from $e$ to $\perp$ again
                endif;
            endfor;
        endfor;
    endfor;
9   return $H$;
```

Figure 5.7: The algorithm to compute slice$(G, \mathsf{EG}(p))$ using $F_p(e)$ for each event e.

each additional edge (e, f) in slice(G, p), we add an edge from the vertex e to the vertex $\perp$.

Consider a graph G and its slice(G, p). In the algorithm, we construct a graph H with vertices as the vertices in G and the following edges:

1. all the edges in slice(G, p), and

2. from e to $\perp$ if $F_p(e)[j] \xrightarrow{P} F(e)[j]$ and $F_p(f)[i] \overset{P}{\underline{\rightarrow}} e$.

The first type of edges ensure that the consistent cuts of H are a subset of the consistent cuts of slice(G, p). This follows from Observation 5.2 and Observation 5.3. The second type of edges ensure that consistent cuts of slice(G, p) that do not include strongly connected components generated by an additional edge (e, f) are disallowed from slice$(G, \mathsf{EG}(p))$, whereas the rest of the consistent cuts belong to slice$(G, \mathsf{EG}(p))$. The semantics of $\mathsf{EG}(p)$ is given with respect to a graph and a consistent cut. Notice that a consistent cut C of $\mathsf{slice}(G, p)$ (which is also a consistent cut of G) satisfies $\mathsf{EG}(p)$ *with respect to* G, if some fullpaths starting from C in G also exist in $\mathsf{slice}(G, p)$. When there is no such strongly connected component in G, the fullpaths in G can be classified as the ones that execute e before f and those that execute f before e. When $\mathsf{slice}(G, p)$ contains a strongly connected component that does not exist in G, then a consistent cut upon including a single vertex from the component includes all vertices from the component. This implies that no fullpaths that start from C in G exist in $\mathsf{slice}(G, p)$. Hence, C does not satisfy $\mathsf{EG}(p)$. Since every nontrivial consistent cut includes $\perp$, by adding an edge from e to $\perp$ (by the definition of a consistent cut), the strongly connected component generated by an additional edge (e, f) is included in every nontrivial consistent cut of H.

Consider the computation depicted in Figure 5.1(a) and its slice depicted in Figure 5.8(a), which contains two additional edges, (e_2, e_1) and (f_3, e_3). Only one of these edges, (e_2, e_1), generates a nontrivial strongly connected component in the slice. Therefore an edge from e_2 to $\perp$ is added to obtain the slice with respect to $\mathsf{EG}(p)$ depicted in Figure 5.8(c).

As another example, consider the computation in Figure 5.1(a) and the slice in Figure 5.6(a) as input to the $\mathsf{EG}(p)$ algorithm. The resulting slice is going to be the same as Figure 5.6(a) because the additional edge (f_2, e_3) does not generate a nontrivial strongly connected component.

We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy $\mathsf{EG}(p)$.

Lemma 5.14 *Every consistent cut of H satisfies* $\mathsf{EG}(p)$.

Proof: It suffices to show that every fullpath in H is a fullpath in G. Consider a fullpath π starting from a consistent cut C in H. Assume, on the contrary, that π is not a fullpath starting from C in G. This implies that there exists a first consistent cut C_j of π such that $C_j \;\triangleright\; C_{j+1}$ in H, but $C_j \;\not\triangleright\; C_{j+1}$ in G. Equivalently,

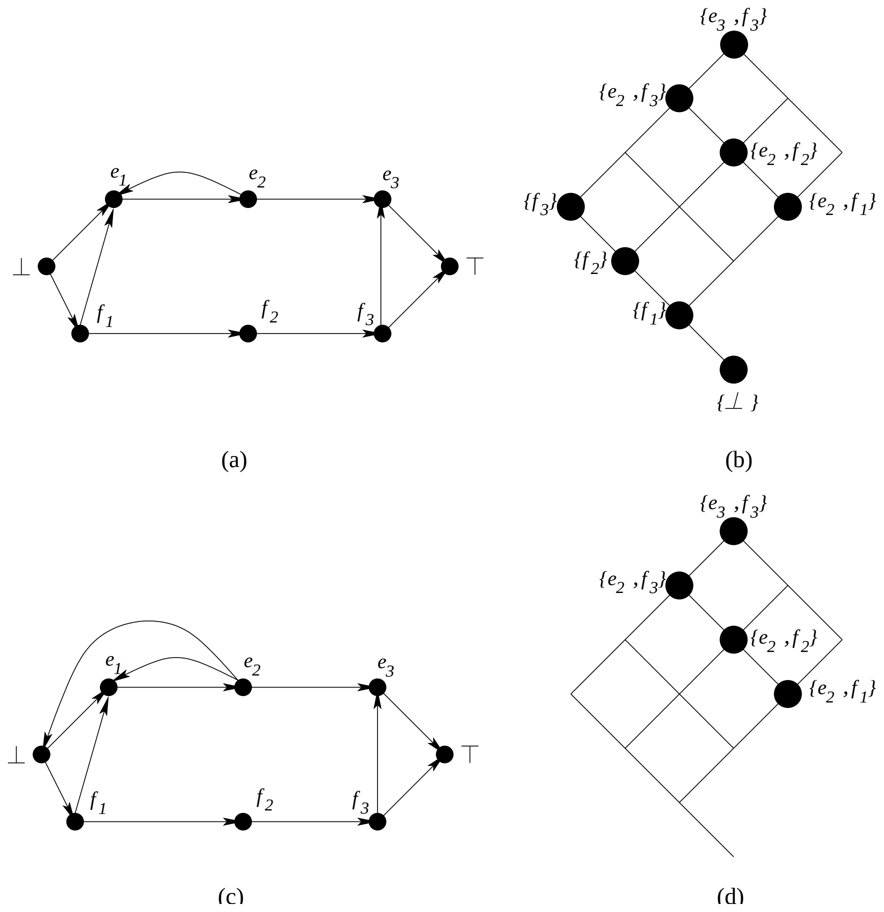


Figure 5.8: (a) A slice of the computation in Figure 5.1(a), (b) the corresponding sublattice, (c) The application of the temporal operator EG on the slice in (a), (d) the corresponding sublattice

C_{j+1} is not a successor of C_j in G. We know that $C_j \ \triangleright\ C_{j+1}$ if and only if $C_{j+1} = C_j \cup \{e\}$, where e is the set of vertices in a strongly connected component in a graph and $\{e\} \cap C_j = \emptyset$. From the definition of $\triangleright$, it follows that there exists a strongly connected component in H that does not exist in G. Let e be a vertex of this component. Then there exists an edge from e to $\perp$ of type (2) in H. We know that every consistent cut of every directed graph contains $\perp$. Since there is an edge

$(e, \bot)$ in H, every consistent cut that contains $\bot$ also contains e. This implies that C_j contains e. However, since $e \cap \in C_j$, we have $e \notin C_j$—contradiction. $\Box$

Lemma 5.15 *Every consistent cut of G that satisfies* $\mathsf{EG}(p)$ *is a consistent cut of H.*

Proof: Consider a consistent cut C of G that satisfies $\mathsf{EG}(p)$. Assume, on the contrary, that C is not a consistent cut of H. Thus, there exist events e and f such that there is an edge from e to f in H, f belongs to C but e does not. Since C is a consistent cut of G, the edge from e to f could be only of type (1) or of type (2). If the edge is of type (1) then C does not belong to $\mathsf{slice}(G, p)$ since e does not belong to C. Thus C does not satisfy p. However, C satisfies $\mathsf{EG}(p)$—contradiction. If the edge is of type (2) then $f = \bot$ and there exists a process j such that $F_p(e)[j] \xrightarrow{P} F(e)[j]$ and $F_p(g)[i] \overset{P}{\underline{\to}} e$, where $g = F_p(e)[j]$. Equivalently, there is an edge from e to g in $\mathsf{slice}(G, p)$ that does not exist in G and there is an edge from g to e in $\mathsf{slice}(G, p)$. If $g \in C$ then C does not belong to $\mathsf{slice}(G, p)$ since C does not contain e. Thus C does not satisfy p. However, C satisfies $\mathsf{EG}(p)$—contradiction. If $g \notin C$ then there are two types of paths in G from C to the final consistent cut. First type of paths is in which g is executed before e (it is always possible to find such a path because there is no path from e to g in G). Consider a consistent cut immediately after the execution of g, say D. Again D does not belong to $\mathsf{slice}(G, p)$ because D contains g but not e. Thus D does not satisfy p. Second type of paths is in which e is executed before g (it is possible to find such a path if there is no path from g to e in G, otherwise such a path does not exist and we are done). Consider a consistent cut immediately after the execution of e, say D. Again D does not belong to $\mathsf{slice}(G, p)$ because D contains e but not g. Thus D does not satisfy p. It follows that there exists a consistent cut D on all paths from C to the final consistent cut such that D does not satisfy p. Hence, C does not satisfy $\mathsf{EG}(p)$—contradiction. $\Box$

From the previous two lemmas, it follows that:

Theorem 5.16 *H is cut-equivalent to* $\mathsf{slice}(G, \mathsf{EG}(p))$.

We assume the computation is given to us as n queues of events—one for each process. Further, the vector clock for each event e is available to us, that is $J(e)$, from which $F(e) = F_{true}(e)$ can be easily computed using algorithm $\mathsf{Algo}_{4.1}$ in $O(n|E|)$ time. Furthermore, we assume that the skeletal representation of $\mathsf{slice}(G, p)$ and thus $F_p(e)$ is given.

Complexity Analysis 5.17 *The graph H has $O(|E|)$ vertices, $O(n|E|)$ edges (one edge is added for each event e in the graph at line 7 and $\mathsf{slice}(G, p)$ has $O(n|E|)$ edges) and can be built in $O(n|E|)$ time as explained next. Let E_i denote the set of events on process P_i. The for loop at line 4 is executed exactly n times. Each iteration of the for loop has $O(1)$ time-complexity because the condition at line 6 can be ascertained by a single comparison and adding an edge takes constant time. Thus the complexity of for loop at line 3 is $O(n|E_i|)$. Summing up over all processes, we have $O(n|E|)$ time.*

5.3.4 Computing Slices with respect to $\mathsf{EX}(p)[j]$

In this section, we explain algorithm $\mathsf{Algo}_{5.4}$ in Figure 5.9 for computing the slice of a graph G with respect to $\mathsf{EX}(p)[j]$.

Consider a graph G and its $\mathsf{slice}(G, p)$. For each event e in G, let $f = F_p(e)[proc(f)]$. In the algorithm, we construct a graph H with vertices as the vertices in G and the following edges:

1. all the edges in G,

2. from $pred(e)$ to $pred(f)$ if $proc(e) = proc(f) = j$, and

3. from $pred(e)$ to f if $proc(e) = j$, $proc(f) \neq j$, and

```
Algorithm Algo$_{5.4}$:

    Input: (1) a directed graph $G$, (2) slice$(G, p)$, and (3) a process $j$
    Output: slice$(G, \mathsf{EX}(p)[j])$

1   $H := G$;                                  // $F_{\mathsf{EX}(p)}(e) = F(e)$
2   for each process $P_i$ do
3       for each event $e$ on $P_i$ do                    // visited in the order given by $\xrightarrow{P}$
4           for each process $P_k$ do
5               $f := F_p(e)[k]$;
6               if ($P_k = P_j$ and $P_i = P_j$) then
7                   add an edge from $pred(e)$ to $pred(f)$ in $H$;
                endif;
8               if ($P_i = P_j$ and $P_k \neq P_j$) then
9                   add an edge from $pred(e)$ to $f$ in $H$;
                endif;
10              if ($P_k = P_j$ and $P_i \neq P_j$) then
11                  add an edge from $e$ to $pred(f)$ in $H$;
                endif;
12              if ($P_k \neq P_j$ and $P_i \neq P_j$) then
13                  add an edge from $e$ to $f$ in $H$;
                endif;
            endfor;
        endfor;
    endfor;
14  return $H$;
```

Figure 5.9: The algorithm to compute slice$(G, \mathsf{EX}(p))$ using $F_p(e)$ for each event e.

4. from e to $pred(f)$ if $proc(f) = j$, $proc(e) \neq j$, and

5. from e to f if $proc(e) \neq j$, $proc(f) \neq j$.

The first type of edges ensure that the consistent cuts of H are a subset of the consistent cuts of G. The second, third, fourth and fifth type of edges ensure that only the consistent cuts that satisfy p at the next action of process j belong to the slice. In other words, we would like the slice to be the same as the slice with respect to p, except the edges to and from the process j, hence the next action of process j. Specifically, when the condition for the third type of edge is satisfied, then we know that there may exist a consistent cut that includes e and f and that satisfies p. We capture the predecessor of this cut by adding an edge from $\mathit{pred}(e)$ to f, thereby, when process j executes e, p holds. When the condition for the fourth type of edge is satisfied, then we know that e must be executed before f. Hence, by adding an edge from e to $\mathit{pred}(f)$, we make sure that e has been executed, thereby, the consistent reached when next time process j executes satisfies p.

Consider the computation depicted in Figure 5.1(a) and its slice depicted in Figure 5.10(a), which contains two additional edges, (f_1, e_1) and (f_2, e_3). We are interested in computing the slice of the computation with respect to $\mathsf{EX}(p)[j]$, where j is process 1. We add edges from f_1 and f_2 to the predecessors of e_1 and e_3, respectively. We obtain the slice with respect to $\mathsf{EX}(p)[1]$ depicted in Figure 5.10(c).

We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy $\mathsf{EX}(p)[j]$.

Lemma 5.18 *Given a process j, every consistent cut of H satisfies* $\mathsf{EX}(p)[j]$.

Proof: Consider a consistent cut C of H. Let g be the frontier of C on process j. Assume, on the contrary, that C does not satisfy $\mathsf{EX}(p)[j]$. Thus, $D = C \cup \{\mathit{succ}(g)\}$ does not satisfy p. Therefore, D does not belong to $\mathsf{slice}(G, p)$. This implies that there exist events e and f such that there is an edge from $(e, f) \in \mathsf{slice}(G, p)$, f belongs to D but e does not. We have the following cases:

- Both e and f are on process j: Since f belongs to D and f is on process j, $\mathit{pred}(f) \overset{P}{\underline{\rightarrow}} g$. Therefore, $\mathit{pred}(f)$ belongs to C. In this case, since there is an edge from e to f in $\mathsf{slice}(G, p)$, there exists an edge of type (2), from $\mathit{pred}(e)$ to $\mathit{pred}(f)$ in H. Hence, C contains $\mathit{pred}(e)$. Since D does not contain e and e and f are on the same process, C does not contain $\mathit{pred}(e)$—contradiction.

- $\mathit{proc}(f) \neq j$, $\mathit{proc}(e) = j$: Since e does not belong to D and e is on process j, $\mathit{succ}(g) \overset{P}{\underline{\rightarrow}} \mathit{pred}(e)$. Therefore, $\mathit{pred}(e)$ does not belong to C. In this case, since there is an edge from e to f in $\mathsf{slice}(G, p)$, there exists an edge of type

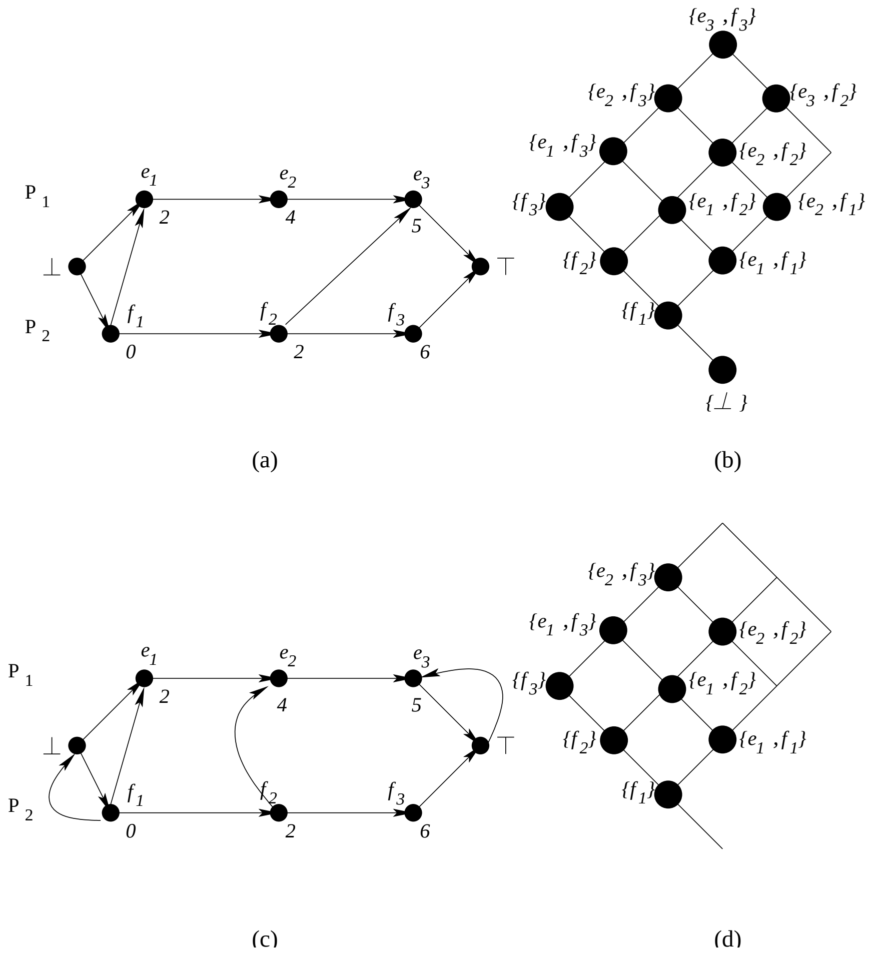


Figure 5.10: (a) A slice of the computation in Figure 5.1(a), (b) the corresponding sublattice, (c) The application of the temporal operator $\mathsf{EX}(p)[1]$ on the slice in (a), (d) the corresponding sublattice

(3), from $pred(e)$ to f in H. Since, D contains f and f is not on process j, C contains f. Hence, C contains $pred(e)$—contradiction.

- $proc(e) \neq j$, $proc(f) = j$: Since f belongs to D and f is on process j, $pred(f) \overset{P}{\rightarrow} g$. Therefore, $pred(f)$ belongs to C. In this case, since there is an edge from e to f in $\mathsf{slice}(G, p)$, there exists an edge of type (4), from e to *pred*(f) in H. Hence, C contains e. Since D does not contain e, C does not contain e—contradiction.

- *proc*$(e) \neq j$ and *proc*$(f) \neq j$: In this case, since there is an edge from e to f in $\mathsf{slice}(G, p)$, there exists an edge of type (5), from e to f in H. Since both e and f are not on process j, C contains f but not e. Therefore, C is not a consistent cut of H—contradiction.

□

Lemma 5.19 *Given a process j, every consistent cut of G that satisfies* $\mathsf{EX}(p)[j]$ *is a consistent cut of H.*

Proof: Consider a consistent cut C of G that satisfies $\mathsf{EX}(p)[j]$. Let g be the frontier of C on process j. Assume, on the contrary, that C is not a consistent cut of H. Thus, there exist events e and f such that there is an edge from e to f in H, f belongs to C but e does not. Since C is a consistent cut of G, the edge from e to f could be only of type (2), (3), (4) or of type (5).

- If (e, f) is of type (2), then both e and f are on process j. In this case, there is an edge from *succ*(e) to *succ*(f) in $\mathsf{slice}(G, p)$. Since C does not include e, $D = C \cup \{succ(g)\}$ does not include *succ*(e). Furthermore, since C includes f, $D = C \cup \{succ(g)\}$ includes *succ*(f). Therefore, D is not a consistent cut of $\mathsf{slice}(G, p)$ and does not satisfy p–contradiction.

- If (e, f) is of type (3), then e is on process j. In this case, there is an edge from *succ*(e) to f in $\mathsf{slice}(G, p)$. Since C does not include e, $D = C \cup \{succ(g)\}$ does not include *succ*(e). However, D includes f. Therefore, D is not a consistent cut of $\mathsf{slice}(G, p)$ and does not satisfy p–contradiction.

- If (e, f) is of type (4), then f is on process j. In this case, there is an edge from e to *succ*(f) in $\mathsf{slice}(G, p)$. Since C does not include e, $D = C \cup \{succ(g)\}$

does not include e, but D includes $succ(f)$. Therefore, D is not a consistent cut of $\mathsf{slice}(G,p)$ and does not satisfy p–contradiction.

- If (e,f) is of type (5), then both e and f are not on process j. In this case, there is an edge from e to f in $\mathsf{slice}(G,p)$. Since C does not include e, $D = C \cup \{succ(g)\}$ does not include e. However, D includes f. Therefore, D is not a consistent cut of $\mathsf{slice}(G,p)$ and does not satisfy p–contradiction.

□

From the previous two lemmas, it follows that:

Theorem 5.20 *Given a process j, H is cut-equivalent to* $\mathsf{slice}(G,\mathsf{EX}(p)[j])$.

We assume the computation is given to us as n queues of events—one for each process. Further, the vector clock for each event e is available to us, that is $J(e)$, from which $F(e) = F_{true}(e)$ can be easily computed using algorithm $\mathsf{Algo}_{4.1}$ in $O(n|E|)$ time. Furthermore, we assume that the skeletal representation of $\mathsf{slice}(G,p)$ and thus $F_p(e)$ is given.

Complexity Analysis 5.21 *The graph H has $O(|E|)$ vertices, $O(n|E|)$ edges (at most 4 edges are added for each event e in the graph at line* 7*, line* 11*, line* 9*, and line* 13 *and since* $\mathsf{slice}(G,p)$ *has $O(n|E|)$ edges) and can be built in $O(n|E|)$ time as explained next. Let E_i denote the set of events on process P_i. The for loop at line* 4 *is executed exactly n times. Each iteration of the for loop has $O(1)$ time-complexity because each of the conditions at line* 6*, line* 10*, line* 8*, and line* 12 *can be ascertained by a single comparison and adding an edge takes constant time.*

5.4 Slicing for RCTL+ Temporal Predicates

Using our slicing algorithms for temporal regular predicates, for non-temporal predicates, and algorithms for composing slices under boolean operators given in [MG01a], it is possible to compute slices with respect to temporal predicates in RCTL+.

We explain algorithm $\mathsf{Algo}_{5.5}$ in Figure 5.11 for computing the slice of a graph G with respect to RCTL+ predicates. The slices are computed recursively starting from the deepest nesting of predicates and applying the appropriate slicing algorithm, either temporal or boolean, while proceeding outwards. In the figure, the input to each algorithm has been specified next to the algorithm name in parentheses.

Following algorithm $\mathsf{Algo}_{5.5}$, now we present an example in Figure 5.12(a) for computing the slice of the computation in Figure 5.1(a) with respect to $\mathsf{AG}(p \wedge \mathsf{EF}(q))$, when $p = (y \geqslant 6)$ and $q = (2 \leqslant x \leqslant 4) \wedge (y \geqslant 2)$. First, we compute the slices with respect to non-temporal predicates p and q using algorithm $\mathsf{Algo}_{4.3}$. Next, the slices of the computation with respect to predicates $\mathsf{EF}(q)$, p, $p \wedge \mathsf{EF}(q)$ and $\mathsf{AG}(p \wedge \mathsf{EF}(q))$ are obtained, which are displayed in Figure 5.12(a), (c), (e), (g), respectively.

Complexity Analysis 5.22 *Our slicing algorithms for temporal regular predicates have an overall time-complexity of $O(n|E|)$, where n is the number of processes and E is the set of events in a given computation. The complexity of computing the slice with respect to a predicate from* RCTL *is dominated by the complexity of computing the slice for an atomic proposition, that is, a non-temporal predicate. The slicing algorithm complexity is $O(n^2|E|)$ for a non-temporal regular predicate. Therefore, the complexity of computing the slice for a predicate from* RCTL *using algorithm* $\mathsf{Algo}_{5.5}$ *is $O(|p| \cdot n^2|E|)$, where $|p|$ is the number of boolean and temporal operators in p. When the predicate is from* RCTL+, *the slicing algorithm complexity is $O(n|E|T)$ for a non-temporal predicate as we showed in the previous chapter, where T is the complexity of detecting the predicate under* EF *operator. Therefore,*

```
Algorithm Algo$_{5.5}$:

    Input: (1) a directed graph $G$, and (2) a predicate $p$
    Output: slice$(G, p)$

1   if ($p$ is an atomic proposition) then
        return Algo$_{4.3}(G, p,$ CONTC);                  // returns skeletal representation

    // adding edges to $G$ in every algorithm below
2   else if ( $p$ is boolean combination of RCTL predicates) then
3       if ( $p = (q \wedge r)$ ) then
4           $H := AND$(slice$(G, q)$, slice$(G, r)$);        // the union of edges in both slices
5       else if ( $p = (q \vee r)$ ) then
6           $H := OR$(slice$(G, q)$, slice$(G, r)$);// the intersection of edges in both slices
        endif;
7   else if ( $p =$ EF$(q)$ ) then
8       $H :=$ Algo$_{5.1}(G,$ slice$(G, q))$;
9   else if ( $p =$ AG$(q)$ ) then
10      $H :=$ Algo$_{5.2}(G,$ slice$(G, q))$;
11  else if ( $p =$ EG$(q)$ ) then
12      $H :=$ Algo$_{5.3}(G,$ slice$(G, q))$;
13  else if ( $p =$ EX$(q)[j]$ ) then
14      $H :=$ Algo$_{5.4}(G,$ slice$(G, q), j)$;
15  else if ( $p =$ EX$(q)$ ) then
16      $H := OR_j$(Algo$_{5.4}(G,$ slice$(G, q), j)$);// for each process $j$ using Lemma 3.26
    endif;
17  compute skeletal representation of $H$;                 // using Algo$_{4.1}$ and Algo$_{5.6}$
18  return $H$;
```

Figure 5.11: The algorithm to compute slice(G, p) for a predicate in RCTL+.

the complexity of computing the slice for a predicate from RCTL+ *using algorithm* Algo$_{5.5}$ *is* $O(|p| \cdot n|E|T)$*, where* $|p|$ *is the number of boolean and temporal operators in* p*.*

5.4.1 Discussion

We know that slicing for an arbitrary predicate is intractable. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently. An approximate slice may be bigger than the actual slice but may be much smaller

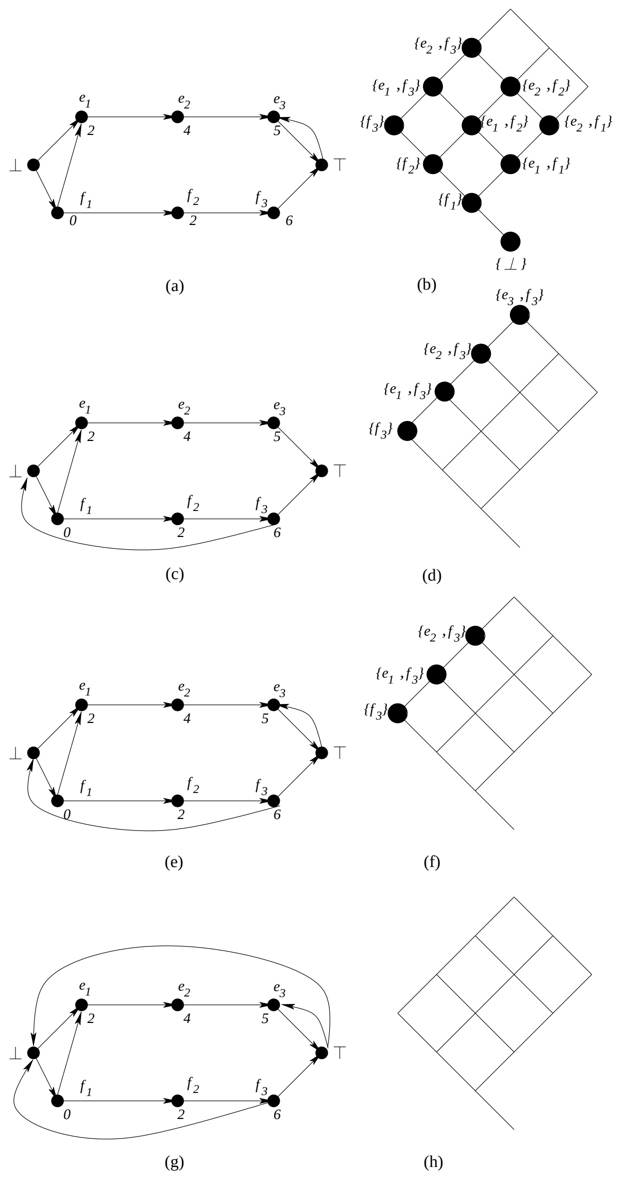


Figure 5.12: (a) The slice of the computation in Figure 5.1(a) wrt. $\mathsf{EF}(q)$, (b) the corresponding sublattice, (c) wrt. p, (d) the corresponding sublattice, (e) wrt. $p \wedge \mathsf{EF}(q)$, (f) the corresponding sublattice, (g) wrt. $\mathsf{AG}(p \wedge \mathsf{EF}(q))$, (h) the corresponding sublattice

than the computation itself. Therefore, in order to detect a predicate, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice.

The slices for RCTL predicates are lean, however this is not the case for $\mathsf{RCTL}+$ predicates. This is because regular predicates are not closed under $\vee$, $\neg$, or EX operators. Also, slices for atomic propositions such as co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates may not be lean.

Depending on the predicate p, a slice may be *lean* or *approximate*. Figure 5.13 shows the relationship of slice categories on the set of all consistent cuts of the computation,

- the shaded region (denoted by $\models p$) corresponds to the set of all consistent cuts in $\mathcal{C}(G)$ that satisfy predicate p.
- *slice* corresponds to the *smallest* sublattice of $\mathcal{C}(G)$ that contains the region $\models p$. If the region X in *slice* is empty (or equivalently, if the region $\models p$ is a sublattice of $\mathcal{C}(G)$) then the slice is a *lean slice*.
- *approximate slice* corresponds to *any* sublattice of $\mathcal{C}(G)$ that contains the region $\models p$.

For example, given non-temporal regular predicates p_1, p_2, p_3, and p_4, and a non-temporal predicate p_5:

- We generate lean slices for the following predicates: p_1, p_2, $p_1 \wedge p_2$, $\mathsf{EF}(p_1)$, $\mathsf{AG}(\mathsf{EF}(p_1 \wedge p_2))$, $\mathsf{EG}(p_2)$.
- We generate slices for the following predicates: $p_1 \vee p_2$, $(p_1 \wedge p_2) \vee (p_3 \wedge p_4)$, p_5, $\mathsf{EX}(p_1)$.

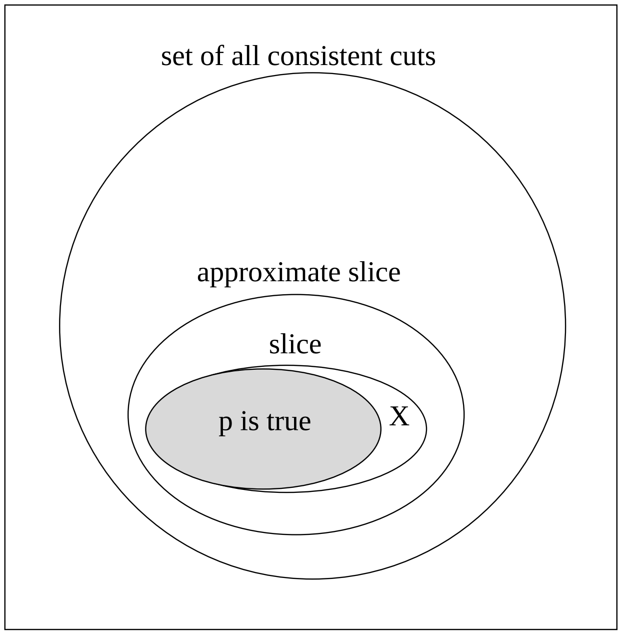


Figure 5.13: Relationship of slice categories

- We may generate approximate slices for the following predicates: $(p_1 \vee p_2) \wedge (p_3 \vee p_4)$, $\mathsf{EG}(p_1 \vee p_2)$, $\mathsf{EF}(p_5)$.

5.4.2 Implementation Details

In the complexity analysis of the slicing algorithms in this dissertation, we assume the computation is given to us as n queues of events—one for each process. Further, the vector clock for each event e in the computation is available to us, that is $J(e)$. In Chapter 7, we will show how to obtain vector clock for the events in a computation G.

Furthermore, in the analysis of slicing algorithms, we assume that the skeletal representation of the computation G and $\mathsf{slice}(G, p)$ are given and thus $F(e) = F_{true}(e)$ and $F_p(e)$, respectively. Observe that we may add edges to the computation at line 2 through line 16 of algorithm $\mathsf{Algo}_{5.5}$. Therefore, the skeletal representation

```
Algorithm Algo$_{5.6}$:

    Input: a directed graph $G$
    Output: $J(e)$ for each event $e$ in $G$

1   topologically sort $G$;
2   for each SCC in $G$ do                    // visited in topological order
        Let JVector be the jvector of this SCC;
3       for each event $e$ in SCC do
4           add index of $e$ to JVector
        endfor;
5       compute the component-wise maximum of the JVector for each SCC that has an edge
        to this SCC and the JVector for this SCC.
6       for each event $e$ in SCC do
7           $J(e)$ = JVector;
        endfor;
    endfor;
```

Figure 5.14: The details of the algorithm to compute $J(e)$ for each event e.

needs to be recomputed. This is shown in line 17 of algorithm Algo$_{5.5}$.

Once we have $J(e)$ values, it is easy to check whether $J(e) \not\subseteq J(f)$ by performing a single comparison of the vector clocks. Hence, the skeletal representation can be easily computed, using algorithm Algo$_{4.1}$ in $O(n|E|)$ time.

In Figure 5.14, we present algorithm Algo$_{5.6}$ to compute $J(e)$ for each event e in a given graph. We first topologically sort the graph in $O(n|E|)$ time at line 1. Then, at line 4 and line 5, while traversing the strongly connected components (denoted by scc) in topological order, we compute the vector clock of an scc (denoted by JVector) by first adding the index (in terms of process order) of all the events in the scc and then taking the component-wise maximum of the JVector of all scc's that have an edge to the current scc. Finally, at line 7, we set $J(e)$ to the JVector of the scc that contains it. This algorithm has a computational complexity of $O(n|E|)$ as well.

Chapter 6

Predicate Detection

In this chapter, we present our results pertaining to predicate detection.

6.1 Overview

Informally, the predicate detection problem is to decide whether a computation satisfies a given predicate. We give a formal definition of the problem in Section 6.2. Predicate detection aids in increasing the reliability of programs by checking their specifications (given as predicates) and generating counter examples when the programs do not satisfy their specifications, that is, when there are bugs.

The main problem in predicate detection in the partial order trace model is the *state explosion problem*—the set of possible consistent cuts of a program with n individual processes can be of size exponential in n.

Our approach to ameliorate state explosion uses two techniques: (1) exploiting the structure of the predicate itself—by imposing restrictions—to evaluate its value efficiently for a given execution trace, and (2) slicing.

Some examples of the predicates for which the structure can be exploited are:

conjunctive [GW94, HMRS96], stable [CL85], observer-independent [CBDGF95], linear [CG98], relational [TG97], and non-temporal regular [GM01, MG01a] predicates. These predicate classes have so far been detected under some or all of the temporal operators EF, EG, AG, AF, but not under any nesting of these operators. For example, a nesting of these operators exists in the predicate $\mathsf{EF}(p \wedge \mathsf{EG}(q))$, where p and q are conjunctive predicates. This nested temporal predicate cannot be efficiently detected using only the efficient algorithms for conjunctive predicates.

Slicing allows us to concisely and precisely represent the consistent cuts of a computation that satisfies a given predicate. In order to detect a predicate, it is exponentially more efficient to work on the slice than on the computation. In Section 6.3, we show how to use the slicing algorithms developed in this dissertation for efficient detection of predicates in $\mathsf{RCTL}+$, which contains nested temporal predicates. In $\mathsf{RCTL}+$, temporal operators are EF, EG, AG, $\mathsf{EX}[j]$, EX and atomic propositions are regular, co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates.

We also develop efficient detection algorithms for unnested temporal predicates of the form $\mathsf{AF}(p)$, $\mathsf{A}(p \;\mathsf{U}\; q)$, and $\mathsf{E}(p \;\mathsf{U}\; q)$. Note that temporal operators AF, AU, and EU do not belong to $\mathsf{RCTL}+$.

Cooper and Marzullo [CM91] present an algorithm for detecting $\mathsf{AF}(p)$ for arbitrary predicate p. The worst-case space and time complexity of the their algorithm is exponential in the number of processes. Tarafdar and Garg [TG98b] proved that it is, in general, NP-complete to detect a predicate under EG operator. Since the problem of detecting a predicate under AF operator is the dual of the problem of detecting a predicate under EG operator, it is, in general, coNP-complete to detect a predicate under AF operator. Using Tarafdar and Garg's [TG99] NP-completeness result for detecting $\mathsf{EG}(p)$ when p is a special case of 2-CNF predicates, called *independent mutual exclusion* predicates, we can easily deduce that detecting a special case of 2-DNF predicates, which is the dual of independent mutual exclusion predicates, under AF operator is coNP-complete in general.

In Section 6.4 we present efficient algorithms for detecting $\mathsf{AF}(p)$. We first present a polynomial-time state-space reduction algorithm that enables us to work on a computation that is in general much smaller than the original computation. We prove that the original computation satisfies $\mathsf{AF}(p)$ if and only if the smaller computation satisfies it. Then, we present a simple algorithm that uses polynomial-space in Section 6.4.1. We determine necessary conditions and sufficient conditions under which detecting $\mathsf{AF}(p)$ may be efficiently solved in Section 6.4.2 and Section 6.4.3.

In Section 6.5, we develop algorithms for unnested temporal predicates $\mathsf{E}(p \mathsf{U} q)$ and $\mathsf{A}(p \,\mathsf{U}\, q)$. These predicates help in detecting properties where a condition has to hold until another condition eventually holds. Efficient algorithms for this operator did not exist before. A mutual exclusion predicate such as "a process enters the trying state before entering the critical state" can be specified as $\mathsf{A}(try \,\mathsf{U}\, critical)$.

In Section 6.6, we give efficient detection algorithms for unnested temporal predicates $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$, where p is a non-temporal linear predicate. Although EG and AG operators belong to $\mathsf{RCTL}+$, and as such $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$ can be detected using slicing, the algorithms in this section are more efficient.

It is useful to know what classes of predicates do not have an efficient predicate detection algorithm. To that end, Chase and Garg [CG98] show that detecting $\mathsf{EF}(p)$ when atomic proposition p belongs to 3-CNF is an NP-complete problem. In Section 6.7, we present intractability results for observer-independent predicates. Charron-Bost et al. [CBDGF95] introduced observer-independent predicates to capture the class of predicates for which the detection under EF and AF operators are equivalent. Observer-independent predicates include predicate classes such as stable predicates and disjunctive predicates. We prove that detecting an observer-independent predicate under EG and AG operators is, in general, NP-complete and co-NP-complete, respectively.

In Section 6.8 we consider the problem of finding a counter example, namely the problem of locating a consistent cut of a computation that satisfies the given predicate (or its complement), if it exists. While it is sufficient to determine whether there exists a faulty consistent cut in a computation for testing purposes, for debugging purposes, it is desirable to actually *locate* the faulty consistent cut. An examination of such a cut may provide valuable insight into the bug that caused the fault. This problem is closely related to the problem of setting a *global breakpoint* when debugging a program.

Finally, in Section 6.9 we discuss how to interpret the output of our predicate detection algorithms in the context of program correctness. Table 6.1 displays the classes of predicates for which we develop an algorithm or prove complexity results in this chapter.

Table 6.1: Predicate detection algorithms and complexity results

Operator and Predicate Class	**Complexity**
RCTL	$O(\|p\|.n^2\|E\|)$
RCTL+	exponential-time, may be $O(\|p\|.n\|E\|T)$
AX(p), p conjunctive, disjunctive	$O(n^2\|E\|)$
AF(p), p regular	reduced complexity
E(p U q), p conjunctive, q linear	$O(n\|E\|)$
A(p U q), p disjunctive, q disjunctive	$O(n\|E\|)$
EG(p), p linear	$O(n\|E\|)$
AG(p), p linear	$O(n\|E\|)$
EG(p), p observer-independent	NP-complete
AG(p), p observer-independent	co-NP-complete

6.2 Problem Statement

For completeness, we now give the formal definition of predicate detection, which was previously given in Chapter 3.

The *predicate detection* problem is to decide whether *the* initial consistent cut of a computation satisfies a given predicate. More formally,

```
Algorithm Algo$_{6.1}$:

    Input: (1) a directed graph $G$, and (2) an RCTL+ predicate $p$
    Output: $G$ satisfies $p$ or not

1   Let slice$(G, p)$ = Algo$_{5.5}(G, p)$;
                                  // initial denotes the initial consistent cut of a graph
2   if initial$(G) \neq$ initial(slice$(G, p)$) then
3       return false and a counter example;
    else
4       if $p$ belongs to RCTL then
5           return true;
        else                                      // the slice may not be lean
6           use a model checker;
        endif;
    endif;
```

Figure 6.1: Predicate detection algorithm for RCTL+.

Definition 6.1 (predicate detection) *Given a distributive lattice $L = (\mathcal{C}(G), \subseteq)$ that represents a computation $G = \langle E, \rightarrow \rangle$ and a temporal logic predicate p expressing some desired specification, decide whether $L, \{\bot\} \models p$ holds or not.*

Remark 6.2 *Observe that our definition of predicate detection is similar to that of model checking [CE81, QS82, CGP00]. However, we consider execution traces rather than programs and we interpret specifications on a finite distributive lattice.*

We define $L \models p$ if and only if $L, \{\bot\} \models p$. By an abuse of notation, we also write $G \models p$ for $L \models p$ when $L = (\mathcal{C}(G), \subseteq)$.

We also define a *path* as a sequence of consistent cuts such that the consistent cuts in the sequence are ordered with the successor relation $\triangleright$. Note that a path is different from a fullpath in that it does not have to end at the final consistent cut.

6.3 Predicate Detection for RCTL+

Many different methods have been devised for automatically checking temporal logic predicates on execution traces by examining the state space models of system behavior. These methods all depend on decision procedures that explicitly represent the state space using a list or a table that grows exponentially in the number of processes, components, or state elements (*state explosion problem*). In the previous chapter we developed algorithms that represent the state space that satisfies a temporal logic predicate using a "slice" instead of using an explicit representation. In this section we show how to use slices for predicate detection.

We explain algorithm $\mathsf{Algo}_{6.1}$ in Figure 6.1 for detection of RCTL+ predicates. We first compute the slice with respect to the predicate at line 1 using algorithm $\mathsf{Algo}_{5.5}$. The slice contains all consistent cuts that satisfy the predicate. Since the predicate detection problem is concerned with checking whether the initial consistent cut of a computation satisfies the predicate, we check this at line 2. If the initial consistent cut of the computation does not belong to the slice then the predicate is false and we return a counter example at line 3. We describe details of counter example generation in Section 6.8.

If the initial consistent cut of the computation belongs to the slice then depending on the class of the predicate either we return true or we have to take extra steps. When the predicate belongs to RCTL (at line 4), we know that the slice with respect to the predicate is lean, therefore the slice does not contain cuts that do not satisfy the predicate. Hence, the initial consistent cut of the computation satisfies the predicate and we return true at line 5.

However, when the predicate does not belong to RCTL, the slice may not be lean, therefore it may contain cuts that do not satisfy the predicate. In fact, the slice may even be approximate as described in Section 5.4.1. Hence, the initial consistent cut of the computation may not satisfy the predicate although it belongs to the slice. In this case, we can use a model checker to check whether the initial consistent cut of the computation satisfies the predicate or not.

Consider the example in Figure 5.12. We observe that the initial cut of the original computation, which is $\{\bot\}$, is different from the initial cut of the slice with respect to $\mathsf{AG}(\neg p \vee \mathsf{EF}(q))$, since the slice is empty. Therefore the predicate is not satisfiable.

Complexity Analysis 6.3 *The complexity of predicate detection for* RCTL *is dominated by the complexity of computing the slice which has* $O(|p| \cdot n^2|E|)$ *time-complexity as shown in Section 5.4. Therefore, the overall time-complexity of predicate detection for* RCTL *is* $O(|p| \cdot n^2|E|)$*, where* $|p|$ *is the number of boolean and temporal operators in* p*.*

When the initial consistent cut of the computation is different from the initial consistent cut of the slice, the complexity of predicate detection for $\mathsf{RCTL+}$ *is dominated by the complexity of computing the slice which has* $O(|p| \cdot n|E|T)$ *time-complexity as shown in Section 5.4. Therefore, the time-complexity of predicate detection for* $\mathsf{RCTL+}$ *is* $O(|p| \cdot n|E|T)$*, where* T *is the complexity of detecting an atomic proposition under* EF *operator.*

When the initial consistent cut of the computation is the same as the initial consistent cut of the slice, due to use of a model checker, we may have exponential-time complexity. However, the slice is, in general, much smaller than the computation and therefore we still have efficient predicate detection results. We validate this conclusion with experiments in the next chapter.

Remark 6.4 *The condition at line 4 of the algorithm* $\mathsf{Algo}_{6.1}$ *is used for checking whether the slice for p is lean or not. In fact, a condition weaker than a "lean" slice would suffice. We say that a slice with respect to a predicate p is "join-irreducible exact" if (1) all join-irreducible elements of the slice satisfy p and (2) the initial consistent cut* $\{\bot\}$ *satisfies p.*

Note that when the predicate in the algorithm is non-temporal, we can use the slice as input to the model checker rather than the computation.

We can use the duality $\mathsf{AX}(p) \equiv \neg\mathsf{EX}(\neg p)$ and give $\mathsf{EX}(\neg p)$ as an input predicate to algorithm $\mathsf{Algo}_{6.1}$ to obtain a necessary and sufficient algorithm to detect $\mathsf{AX}(p)$. However, such a procedure may not have polynomial-time complexity since $\mathsf{EX}(\neg p)$ is not a regular predicate, in general. For special cases of p, such as disjunctive and conjunctive predicates, we can obtain efficient algorithms by using the duality and the fact that the negation of a local predicate is also a local predicate, hence regular. Therefore, we can detect $\mathsf{AX}(p)$ for conjunctive and disjunctive p in $O(n^2|E|)$ time-complexity.

Next we will present predicate detection algorithms for detecting specific classes of unnested temporal predicates.

6.4 Predicate Detection for $\mathsf{AF}(p)$

The $\mathsf{AF}(p)$ detection problem has efficient solutions when the predicate p is conjunctive or disjunctive [GW96, GW92]. However, the complexity problem is open when p is regular. In this section, we present space and time efficient conditions to solve the problem when p is arbitrary or regular.

6.4.1 Polynomial-Space Algorithm

The performance of algorithms for detecting $\mathsf{AF}(p)$ in a computation can be improved by considering a smaller state-space, that is, a smaller computation (with less number of consistent cuts) than the original computation. In this section, we present a polynomial-time algorithm for reducing the size of the computation. We show that detecting $\mathsf{AF}(p)$ on the original computation is the same as detecting $\mathsf{AF}(p)$ on the smaller computation.

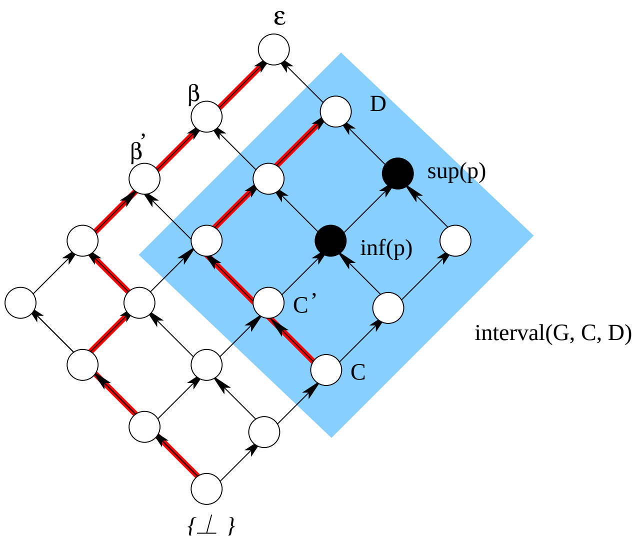


Figure 6.2: The lattice of consistent cuts of a computation G and an $interval(G, C, D)$

Let $interval(G, C, D)$ denote the slice of a computation G with respect to the predicate $G_{[C,D]}$. We say that a consistent cut satisfies $G_{[C,D]}$ if the cut belongs to the interval lattice $[C, D]$. Formally,

Definition 6.5 (interval predicate) *Given a computation G and consistent cuts C and D such that $C \subseteq D$, $F \models G_{[C,D]}$ iff $F \in [C, D]$.*

Figure 6.2 depicts the lattice of consistent consistent cuts of a computation G. The shaded region in the figure depicts the set of consistent cuts of $interval(G, C, D)$. Note that an interval lattice is also a sublattice. It can be proved that $G_{[C,D]}$ is a regular predicate. We can obtain $interval(G, C, D)$ as follows: We add edges from $\top$ to the successor of every vertex in $frontier(D)$. Thus, all cuts that include one of these successors becomes a trivial cut, hence such a cut does not belong to the slice (similar to line 4 in algorithm $\mathsf{Algo}_{5.1}$). We also add edges from every vertex in $frontier(C)$ to $\perp$. Since $\perp$ is the initial consistent cut, by adding these edges, $frontier(C)$ becomes the initial consistent cut of the slice.

For a computation G and a predicate p, we use $\mathit{inf}(p)$ and $\mathit{sup}(p)$ to denote the least and the greatest consistent cuts of the computation that satisfy the predicate, respectively. Observe that both $\mathit{inf}(p)$ and $\mathit{sup}(p)$ exist for a regular predicate.

We will prove below that $\mathit{interval}(G, C, D)$, which is smaller than the original computation G, can be used for detecting $\mathsf{AF}(p)$, but first we prove the following lemma which presents an observation on the lattice structure of the set of consistent cuts of a computation.

Lemma 6.6 *Given a computation $G = \langle E, \rightarrow \rangle$ and consistent cuts $C, D, F \in \mathcal{C}(G)$, if $C \triangleright F$ and $D \subseteq F$ then $(C \cap D) = D$ or $(C \cap D) \triangleright D$.*

Proof: From the definition of $\triangleright$, we have $C \triangleright F \Rightarrow \langle \exists e \in E : e \notin C : C \cup \{e\} = F \rangle$. Hence, we have that for any D, $D \subseteq F \Rightarrow D \subseteq (C \cup \{e\})$. We have the following two cases:

Case 1: $e \notin D$: Hence, $D \subseteq C$, and so $(C \cap D) = D$, as required.

Case 2: $e \in D$: Hence, $(C \cap D) = (D - \{e\})$ and it is a consistent cut. Further, using the definition of $\triangleright$ and $(e \notin C)$, we get $(C \cap D) \triangleright D$, as required. □

Theorem 6.7 (NSC) *Given a computation G, $G \models \mathsf{AF}(p)$ iff $\mathit{interval}(G, C, D) \models \mathsf{AF}(p)$, where C is the meet of predecessors of $\mathit{inf}(p)$, if the predecessors exist, otherwise $C = \mathit{inf}(p)$, and D is the join of successors of $\mathit{sup}(p)$, if the successors exist, otherwise $D = \mathit{sup}(p)$.*

Proof: Without loss of generality, assume that both C and D exist and are different from the initial and final consistent cuts of G. We prove the contrapositives.

$\Rightarrow$:

We obtain a path starting from the initial consistent cut and ending at the final consistent cut in G along which p is false as follows. Pick an arbitrary path starting from the initial consistent cut and ending at a predecessor of C in G. We know that none of the cuts along this path satisfy p since all cuts that satisfy p belong to $interval(G, C, D)$. Next, extend this arbitrary path with a path in $interval(G, C, D)$ where none of the cuts on the path satisfy p. We can find such a path in $interval(G, C, D)$ due to the assumption in this case. Finally, pick an arbitrary path starting from a successor of D and ending at the final consistent cut of G.

$\Leftarrow$:

We prove the claim in two steps.

Step 1: We first show that if there exists a path $\mathcal{P}$, from the initial to the final consistent cut in G such that all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final consistent cut in $interval(G, C, \mathcal{E})$ such that all cuts on the path satisfy $\neg p$. For convenience, we depict an example path $\mathcal{P}$ in Figure 6.2 with thick lines.

Let β be the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. Let β' be the predecessor of β on the path $\mathcal{P}$. From Lemma 6.6, the meet of β' and $inf(p)$ is either $inf(p)$ or a predecessor of $inf(p)$, denoted by C'. However, in the former case, $\mathrm{inf}(p) \subseteq \beta'$. Since β' is also on the path $\mathcal{P}$, this leads to a contradiction since β is the first cut such that $inf(p) \subseteq \beta$. Therefore the meet of β' and $inf(p)$ is C' and $\mathrm{inf}(p) \not\subseteq \beta'$.

There exists a path from C' to β' because $C' \subseteq \beta'$. We now prove that every cut on this path satisfies $\neg p$, that is, for all F such that $C' \subseteq F$, $F \subseteq \beta'$, F satisfies $\neg p$. Since $inf(p) \not\subseteq \beta'$ and $F \subseteq \beta'$, we have $inf(p) \not\subseteq F$. The consistent cuts that satisfy p belong to $interval(G, inf(p), sup(p))$. Therefore, β' and all such F do not satisfy p. Since C is the meet of all predecessors of $inf(p)$ and C' is a predecessor of $inf(p)$, $C \subseteq C'$. Therefore $C \subseteq F$. Similarly, all cuts from C to C' do not satisfy p since they do not belong to $interval(G, inf(p), sup(p))$.

We obtain the required path as follows. Start the path from C and extend it

with a path from C' to β'. Such a path exists as we showed in the above paragraph. Then, extend the resulting path with a path from β to the final cut. Such an extension exists in $\mathcal{P}$ because there is a path from β to the final cut in path $\mathcal{P}$.
Step 2: Now we show that if there exists a path $\mathcal{P}$, from the initial to the final consistent cut in $interval(G, C, \mathcal{E})$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final consistent cut in $interval(G, C, D)$ where all cuts on the path satisfy $\neg p$.

The proof is similar to Step 1 with the paths reversed. In this case we choose β as the last cut on the path $\mathcal{P}$ such that $\beta \subseteq sup(p)$ and β' as the successor of β on the path $\mathcal{P}$. Furthermore, we choose D' as a successor of $inf(p)$. We can show in a similar fashion as in Step 1 that there exists a path from β' to D' where all cuts on the path satisfy $\neg p$. Finally, we can construct a path from C to D as the concatenation of the paths from C to β, β' to D', and D. $\square$

Complexity Analysis 6.8 *When p is regular, we can compute $inf(p)$ in $O(n|E|)$ time using algorithm $\mathsf{Algo}_{3.1}$. The same algorithm can also be used to compute $sup(p)$ by starting the algorithm from the final consistent cut and moving backwards. We can compute the predecessors and successors of a cut in $O(n)$ time. Finally we can obtain $interval(G, C, D)$ by adding edges from $\top$ to the successor of every vertex in $frontier(sup(p))$ and an edge from $\bot$ to the every vertex in $frontier(inf(p))$ in $O(n)$ time. Therefore, the overall complexity is $O(n|E|)$.*

Note that the above theorem is not restricted to predicates with a single least and greatest cut only. For example, if the predicate has several least cuts then first we take the intersection of all those cuts. Second, we find the predecessors of the intersection. Finally, we compute the intersection of the predecessors to obtain C in $interval(G, C, D)$. Also, we can use the above theorem to reduce the state space for other temporal operators such as EF, EG, and AG. In other words, in order to detect a predicate of the form $\mathsf{EF}(p)$ we can use the $interval(G, C, D)$.

Although the time complexity of computing $interval(G, C, D)$ is polynomial, the time and space complexity of detecting $\mathsf{AF}(p)$ on this reduced state-space may still be exponential since $interval(G, C, D)$ may contain exponential number of consistent cuts. However, it is always better to work on $interval(G, C, D)$ rather than G since $interval(G, C, D)$ is a subset of G. In fact, we believe that $interval(G, C, D)$ is generally much smaller than the original computation G which we validate with experimental work in the next chapter. Furthermore, Theorem 6.7 is orthogonal to other techniques we will present in the next sections for detecting $\mathsf{AF}(p)$, that is, we can always first compute $interval(G, C, D)$ and then apply those conditions.

Next, we present a polynomial-space algorithm for $\mathsf{AF}(p)$. Cooper and Marzullo [CM91] presented a worst-case exponential-space and exponential-time algorithm for $\mathsf{AF}(p)$. Their algorithm detects $\mathsf{AF}(p)$ using level sets where a *level set* is the set of successors of a consistent cut. Their algorithm starts from the initial consistent cut. If p is true in the initial consistent cut $\mathsf{AF}(p)$ is satisfied. Otherwise, it constructs the next level set including only those consistent cuts in which $\neg p$ is true. Continuing in this manner, if the algorithm can reach the final consistent cut, then $\mathsf{AF}(p)$ is false; otherwise, it is true. This algorithm requires space proportional to the size of the largest level set, which is exponential in the number of processes. This is because there are possibly exponential number of consistent cuts of a computation but polynomial number of level sets. In fact, the number of level sets is exactly $|E|$ (the length of an observation).

We obtain a simple space efficient algorithm for detecting $\mathsf{AF}(p)$ by generating all observations for the given computation. This algorithm is based on generating linearizations of a partial order [PR94]. For each such observation, we check whether $\neg p$ holds on every cut on the observation. If such an observation exists then $\mathsf{AF}(p)$ is not satisfied, otherwise it is satisfied.

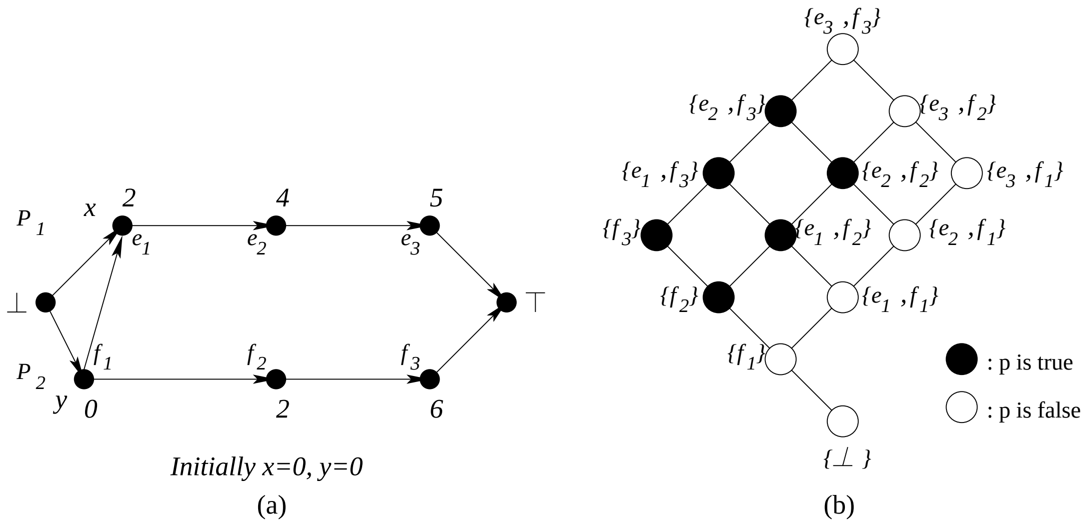


Figure 6.3: (a) A computation, and (b) its corresponding lattice

Complexity Analysis 6.9 *The length of every observation is $|E|$, the total number of events in the computation. The frontier of a consistent cut can be represented by an n-dimensional vector for a computation with n processes. Therefore, for each consistent cut $O(n)$ space is required giving us the space complexity of $O(n|E|)$. The time complexity is bounded by the number of observations, which may be exponential in the number of processes. Note that if we use the slice of the computation with respect to $\neg p$ instead of the computation, then we can reduce the number of observations.*

6.4.2 Polynomial-Time Necessary Conditions

Now we present a polynomial-time necessary condition to detect $\mathsf{AF}(p)$ that uses meet-irreducible elements [DP90]. For example, the predecessors of the final consistent cut of a computation (e.g., predecessors of the final consistent cut $\{e_3, f_3\}$ in Figure 6.3(b)) are all meet-irreducible elements.

Theorem 6.10 (NC) *Given a computation $G = \langle E, \rightarrow\rangle$ and a regular predicate p,* *if $\neg p$ holds at the initial consistent cut and at the successor of every meet-irreducible element then $G \not\models \mathsf{AF}(p)$.*

Proof: We show that there exists a path from the initial to the final consistent cut in the computation G where all cuts on the path satisfy $\neg p$. Given an arbitrary consistent cut C that satisfies $\neg p$ and that is different from the final consistent cut, we first show that there exists a successor of C that satisfies $\neg p$. There are two cases.

Case 1: C has a single successor. In this case C is a meet-irreducible element and from the assumption, $\neg p$ holds at the successor of C.

Case 2: C has at least two successors. Observe that if more than one successor of C satisfies p then from the regularity of p, the intersection of those successor cuts, which is C, satisfies p. This leads to a contradiction. Therefore, there exists at least one successor of C where $\neg p$ holds.

We construct the required path as follows: Start from the initial consistent cut for which we know that $\neg p$ holds. From above 2 cases, we have that for every consistent cut that satisfies $\neg p$ we can find a successor consistent cut that satisfies $\neg p$. Finally, we reach the final consistent cut which is the successor of a cut that satisfies $\neg p$.

Note that we only use the linearity of predicate p only. $\Box$

The converse of Theorem 6.10 is false. Figure 6.3(b) displays the lattice of consistent cuts of the computation in Figure 6.3(a). From the lattice we observe that this computation satisfies the right side of Theorem 6.10, that is $G \not\models \mathsf{AF}(p)$. However, the left side of the theorem does not hold because the successor of the meet-irreducible element $\{f_3\}$ satisfies p.

A similar necessary condition can be given for join-irreducible elements.

Theorem 6.11 *Given a computation G and a regular predicate p, if $\neg p$ holds at*

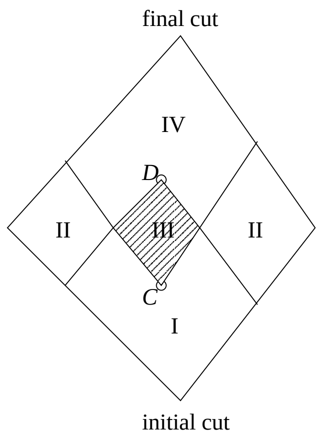


Figure 6.4: *interval*(G, C, D) partitions the lattice of consistent cuts of G

the final consistent cut and at the predecessor of every join-irreducible element then $G \not\models \mathsf{AF}(p)$.

Complexity Analysis 6.12 *We can check the condition in Theorem 6.10 (resp. in Theorem 6.11) by finding the meet-irreducible (resp. join-irreducible) cuts of the computation in* $O(n|E|)$ *time as will be explained in the complexity analysis of algorithm* $\mathsf{Algo}_{6.5}$.

Observe that the consistent cuts of *interval*(G, C, D) may partition the lattice of consistent cuts of a computation G as in Figure 6.4. The patterned region in the figure denotes the cuts that belong to *interval*(G, C, D), i.e., the set of cuts that satisfy the interval predicate $G_{[C,D]}$. A cut F belongs to partition I if $(C \not\subseteq F) \wedge (F \subseteq D)$, partition II if $(C \not\subseteq F) \wedge (F \not\subseteq D)$, partition III if $(C \subseteq F) \wedge (F \subseteq D)$, and partition IV if $(C \subseteq F) \wedge (F \not\subseteq D)$. Given that *interval*$(G, C, D)$ exists, that is, partition III exists, other partitions may not exist. For example, if C is the initial consistent cut of G and D is the final consistent cut of G then only partition III exists.

Next we present another polynomial-time necessary condition for detecting $\mathsf{AF}(p)$ based on the above characterization of partitions generated by an interval predicate.

Theorem 6.13 *Given a computation G and an interval predicate $p = G_{[C,D]}$, there exists a consistent cut $F \in \mathcal{C}(G)$ that belongs to partition II iff $G \not\models \mathsf{AF}(p)$.*

Proof:

$\Rightarrow$:

We know that F is reachable from the initial consistent cut. For the purpose of contradiction, assume that there exists a cut H on a path from $\{\perp\}$ to F such that H satisfies p. For F to be reachable from H, we must have that $H \subseteq F$. However since H satisfies p, $C \subseteq H$ and therefore $C \subseteq F$. Since F is in partition II, $C \not\subseteq F$— a contradiction. Similarly, we can show that there does not exist a cut H' on a path from F to $\mathcal{E}$ such that H' satisfies p. C cannot be $\{\perp\}$ and D cannot be $\mathcal{E}$ because we assume that partition II exists. Therefore, partitions I and IV also exist. Now we obtain a path where all cuts satisfy $\neg p$ by starting from $\{\perp\}$ and following an arbitrary path in partition I such that the path reaches F in partition II. Then we follow an arbitrary path from F to the final consistent cut.

$\Leftarrow$:

We prove by contradiction. Suppose that partition II does not exist and there exists a path in G from initial to the final consistent cut where all cuts on the path satisfy $\neg p$. Since there exists such a path, we have that partitions I and IV exist. Otherwise, $C = \{\perp\}$ and $D = \mathcal{E}$ and we do not have a path from $\{\perp\}$ to $\mathcal{E}$ where $\neg p$ holds on the path. Since partition II does not exist and a path of cuts satisfying $\neg p$ exists, there is a path from partition I to partition IV without passing through partition III (since p is an interval predicate). We will show that this is impossible.

Consider two cuts F and H on a path from $\{\perp\}$ to $\mathcal{E}$ where $\neg p$ holds on the path, F belongs to partition I, H belongs to partition IV, and H is a successor of F. From the definition of partitions, we have that $(C \not\subseteq F) \wedge (F \subseteq D)$ and $(C \subseteq H) \wedge (H \not\subseteq D)$. Furthermore, from the definition of successor of a cut, we know that $H = F \cup \{e\}$, where e is an event in G and $e \notin F$. To obtain H from F, there are two cases: In the first case, we should add $e \notin D$ to F (therefore $e \notin C$) so that $H \not\subseteq D$. In the second case, we should add $e \in C$ to F (therefore $e \in D$) so that $C \subseteq H$. However, $e \in D$ and $e \notin D$ leads to a contradiction. $\square$

We present a weaker result for regular predicates. The necessary conditions of Theorem 6.10 and Theorem 6.11 are not comparable with the condition of Theorem 6.14 below. Furthermore, observe that the converse of the next condition is false.

Theorem 6.14 *Given a computation G and a regular predicate p, if there exists a consistent cut F that belongs to partition II in G then $G \not\models \mathsf{AF}(p)$.*

We can use slicing to detect whether there exists a consistent cut F in partition II as follows. We compute the slice of the computation with respect to the predicate $(C \not\subseteq F)$ $\wedge(F \not\subseteq D)$, where $C = \mathit{inf}(p)$ and $D = \mathit{sup}(p)$. If the slice is nonempty then $\mathsf{AF}(p)$ does not hold. In the following, C_i denotes the maximal event (in terms of process order) of the consistent cut C from process i.

$$\begin{aligned}
& (C \not\subseteq F) \wedge (F \not\subseteq D) \\
\equiv\ & \{ \text{ from the definition of a consistent cut } \} \\
& (\exists i : F_i \overset{P_i}{\rightarrow} C_i) \wedge (\exists j : D_j \overset{P_j}{\rightarrow} F_j) \\
\equiv\ & \{ \text{ rewriting } \} \\
& \Big((F_0 \overset{P_0}{\rightarrow} C_0) \vee (F_1 \overset{P_1}{\rightarrow} C_1) \vee \ldots \vee (F_{n-1} \overset{P_{n-1}}{\rightarrow} C_{n-1})\Big) \bigwedge \\
& \quad \Big((D_0 \overset{P_0}{\rightarrow} F_0) \vee (D_1 \overset{P_1}{\rightarrow} F_1) \vee \ldots \vee (D_{n-1} \overset{P_{n-1}}{\rightarrow} F_{n-1})\Big)
\end{aligned}$$

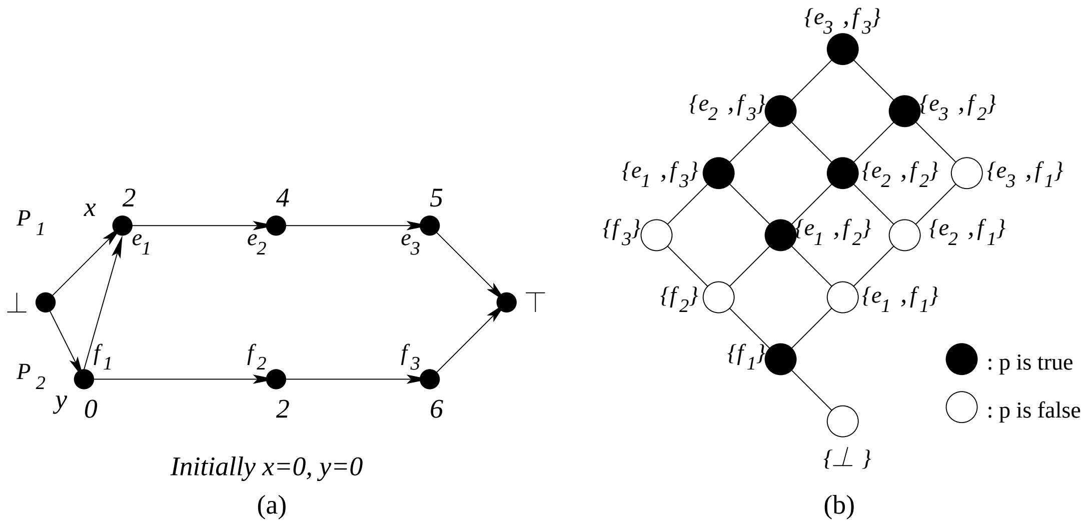


Figure 6.5: (a) A computation, and (b) its corresponding lattice

Complexity Analysis 6.15 *The above predicate is a disjunction of n^2 terms where each term is a conjunction of local predicates $((F_i \overset{P_i}{\to} C_i) \wedge (D_j \overset{P_j}{\to} F_j))$. Using the slicing algorithm $\mathsf{Algo}_{5.5}$, we can compute a slice for this predicate in $O(n^4|E|)$ time. If the slice is not empty then F exists otherwise it does not exist.*

6.4.3 Polynomial-Time Sufficient Condition

We can use the duality between $\mathsf{AF}(p) \equiv \neg\mathsf{EG}(\neg p)$ and give $\mathsf{EG}(\neg p)$ as an input predicate to $\mathsf{Algo}_{6.1}$ to obtain a necessary and sufficient algorithm to detect $\mathsf{AF}(p)$. However, such a procedure may not have polynomial-time complexity since $\mathsf{EG}(\neg p)$ is not a regular predicate. Instead we use $\mathsf{Algo}_{6.1}$ to obtain a simple sufficient condition.

Theorem 6.16 (SC) *Given a computation G and a predicate $\mathsf{EG}(\neg p)$, if algorithm $\mathsf{Algo}_{6.1}$ returns false at line 3, then $G \models \mathsf{AF}(p)$.*

The converse of Theorem 6.16 is false. Figure 6.5(a) displays a computation that satisfies $\mathsf{AF}(p)$. When we compute the union and intersection closure of the cuts that satisfy the predicate (the closure of white filled circles), we obtain all consistent cuts that belong to the computation, that is, $\mathsf{slice}(G, \neg p,)$ has the same set of cuts as G. Therefore, algorithm $\mathsf{Algo}_{6.1}$ does not return false at line 3.

6.4.4 A General $\mathsf{AF}(p)$ Algorithm

Algorithm $\mathsf{Algo}_{6.2}$ in Figure 6.6 displays a general procedure that combines the conditions developed in this section for detecting whether a computation satisfies a predicate of the form $\mathsf{AF}(p)$, when p is a regular predicate.

6.5 Predicate Detection for $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ and $\mathsf{A}(p \mathbin{\mathsf{U}} q)$

In this section, we explain algorithm $\mathsf{Algo}_{6.3}$ in Figure 6.7 for detecting whether a computation satisfies a predicate of the form $\mathsf{E}(p \mathbin{\mathsf{U}} q)$, when p is a conjunctive predicate and q is a non-temporal linear predicate.

EU and AU operators aid in detecting conditions where a condition has to hold until another condition eventually holds. A computation satisfies $\mathsf{E}(p \mathbin{\mathsf{U}} q)$, if there exists a fullpath starting from the initial consistent cut such that p holds along the path until q holds on the path. In detecting $\mathsf{E}(p \mathbin{\mathsf{U}} q)$, if we constructed the lattice of nontrivial consistent cuts then, in the worst-case, we would have to check all fullpaths in the lattice. It is clear that this is very inefficient due to both state explosion and exponential number of fullpaths in the number of events.

Algorithm $\mathsf{Algo}_{6.3}$ shows a procedure to detect $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ by checking for fullpaths in a smaller state-space. At line 1, the least consistent cut that satisfies a non-temporal linear predicate is computed using algorithm $\mathsf{Algo}_{3.1}$. Then in order to check $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ we check whether there exists a path starting from the initial consistent cut and ending at a predecessor of Jq such that along the path p holds. The algorithm adds an edge from vertex $\top$ to the successor of every element in the frontier of Jq at line 4. At the end of the for loop at line 3, a new graph that has the

```
Algorithm Algo$_{6.2}$:

    Input: (1) a directed graph $G$, and (2) a non-temporal regular predicate $p$
    Output: $G$ satisfies AF($p$) or not

    // reduce the state-space using Theorem 6.7
1   compute $inf(p)$ and $sup(p)$;
2   let $C$ be the intersection of predecessors of $inf(p)$;
3   let $D$ be the union of successors of $sup(p)$;
4   use $C$ and $D$ to obtain $K := interval(G, C, D)$;      // reduce the number of cuts
    // apply polynomial-time necessary condition in Theorem 6.10
5   let $SMI :=$ the set of Successors of all Meet-Irreducible elements in $K$;
6   if initial consistent cut of $K$ and all cuts in $SMI$ satisfy $\neg p$ then
7       return false;
    endif;
    // apply polynomial-time necessary condition in Theorem 6.11
8   let $PJI :=$ the set of Predecessors of all Join-Irreducible elements in $K$;
9   if final consistent cut of $K$ and all cuts in $PJI$ satisfy $\neg p$ then
10      return false;
    endif;
    // apply polynomial-time necessary condition in Theorem 6.14
11  let $q :=$ the be DNF predicate in Complexity Analysis 6.15;
12  if slice($K, q$) is nonempty then
13      return false;
    endif;
    // apply polynomial-time sufficient condition in Theorem 6.16
14  compute slice($K$, EG($\neg p$));
15  if initial($G$) $\neq$ initial(slice($K, \neg p$)) then
16      return true;
    endif;
    // apply polynomial-space algorithm in Section 6.4.1
17  for each path in $interval(G, C, D)$ do     // obtain paths using [PR94]
18      let $G$ be the first cut on the path;
19      while $G$ satisfies $\neg p$ do
20          $G :=$ successor of $G$ on the path;
        endwhile;
21      if $G = D$ then     // final consistent cut is reached
22          return false;
        endif;
    endfor;
23  return true;
```

Figure 6.6: The algorithm to decide whether AF(p) is satisfied or not.

```
Algorithm Algo$_{6.3}$:

     Input: (1) a directed graph $G$,   (2) a conjunctive predicate $p$, and
            (3) a non-temporal linear predicate $q$

     Output: $G$ satisfies $\mathsf{E}(p\,\mathsf{U}\,q)$ or not

1    $Jq$ := Algo$_{3.1}(G, q)$;   // the least consistent cut that satisfies $q$
2    $Q := frontier(Jq)$;
3    for each $e \in Q$ do
4        add an edge from $\top$ to $succ(e)$ in $G$; // the final consistent cut of $G$ is $Jq$
     endfor;
5    for each $e \in Q$ do
6        $H := G$;            // $H$ is (re)set to $G$
7        add an edge from $\top$ to $e$ in $H$;       // the final consistent cut of $H$ is $Jq - \{e\}$
8        if $H \models \mathsf{EG}(p)$ then
9            return true;
         endif;
     endfor;
10   return false;
```

Figure 6.7: The algorithm to decide whether $\mathsf{E}(p\,\mathsf{U}\,q)$ is satisfied or not.

same set of consistent cuts as the input graph up to Jq is obtained. This is because adding an edge from $\top$ to a vertex makes every consistent cut that includes that vertex trivial (similar to line 4 in algorithm Algo$_{5.1}$). Then during each iteration of the for loop at line 5, we compute yet another graph H that has the same set of consistent cuts as the input graph up to a predecessor of Jq. We check whether there exists a graph H that satisfies $\mathsf{EG}(p)$ or not at line 8. If the answer is positive then the algorithm returns true at line 9, otherwise it returns false at line 10.

Consider the computation depicted in Figure 6.8(a) and the predicate $\mathsf{E}(p\mathsf{U}q)$ where p denotes that "variable z of process P_3 is less than 6 and variable x of process P_1 is less than 4" and q denotes that "channels are empty and variable x of process P_1 is greater than 1". Note that p is a conjunctive predicate and q is a non-temporal linear predicate. It can be observed from Figure 6.8(b) that the predicate is true since there exists a path of consistent cuts denoted by patterned circles leading to consistent cuts denoted by filled circles. The least consistent cut that satisfies q is $Jq = \{e_1, f_2, f_1, g_1\}$ and $Q = \mathit{frontier}(Jq) = \{e_1, f_2, g_1\}$. Algorithm $\mathsf{Algo}_{6.3}$, at line 4, adds an edge from $\top$ to $\mathit{succ}(e_1) = e_2$. Similarly, edges are added from $\top$ to the successors of f_2 and g_1, which is $\top$ itself. Then in the for loop at line 5, the algorithm adds an edge from $\top$ to $g_1 \in Q$ and we obtain the graph depicted in Figure 6.8(c). It is clear that $\mathsf{EG}(p)$ holds for this new graph since there exists a path of patterned circles, that is, consistent cuts $\{\perp\}$, $\{f_1\}$, $\{e_1, f_1\}$, $\{e_1, f_2, f_1\}$. Furthermore, this path can be extended with a filled circle, that is, $Jq = \{e_1, f_2, f_1, g_1\}$. Hence, $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ holds for the computation depicted in Figure 6.8(a). Out of a possible 7 paths starting from the initial consistent cut and satisfying $\mathsf{E}(p \mathbin{\mathsf{U}} q)$, it is enough to consider only the ones that lead to Jq, of which there are only 2 in this case.

The correctness of the algorithm follows from the theorem below.

Theorem 6.17 *Given a finite distributive lattice $L = (\mathcal{C}(G), \subseteq)$ for a computation $G = \langle E, \rightarrow \rangle$, a conjunctive predicate p, and a linear predicate q; $L \models \mathsf{E}(p \mathbin{\mathsf{U}} q)$ if and only if there exists a finite sequence $D_0, \ldots, D_j$ of consistent cuts such that*

$$\text{(a) } D_0 = \{\perp\}, \text{(b) } D_i \triangleright D_{i+1} \textit{ for all } 0 \leqslant i < j, \text{(c) } D_j = Jq, \textit{ and}$$

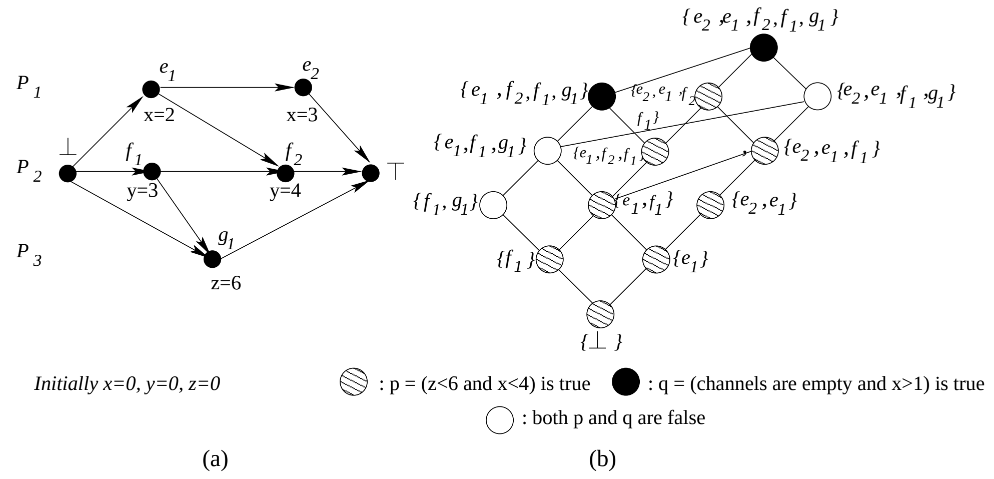
$$\text{(d) } D_i \models p, \textit{ for all } 0 \leqslant i < j$$

Proof:

$\Leftarrow$: $D_0, D_1, \ldots, D_j$ is a sequence that satisfies the predicate $\mathsf{E}(p \mathbin{\mathsf{U}} q)$.

$\Rightarrow$: Assume $L \models \mathsf{E}(p \mathsf{U} q)$. This means that there exists a finite sequence $C_0, C_1, \ldots, C_k$ of consistent cuts such that (a) $C_0 = \{\perp\}$, (b) $C_i \triangleright C_{i+1}$ for all $0 \leqslant i < k$, (c) $C_k \models q$, and (d) $C_i \models p$ for all $0 \leqslant i < k$. Interesting case is when $k \geqslant 1$; otherwise the theorem is trivially true.

We know that Jq exists because q is a linear predicate. Therefore $Jq \subseteq C_k$. Consider the sublattice, $L' = (\mathcal{C}(H), \subseteq)$ corresponding to the computation H



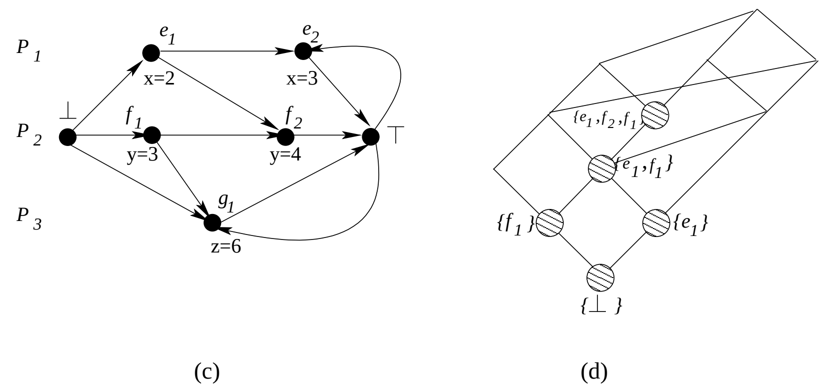


Figure 6.8: (a) A computation, (b) the corresponding sublattice, (c) The application of the algorithm on the computation in (a), (d) the corresponding sublattice

obtained from G such that the final consistent cut of H is $C_{(k-1)}$ and all cuts that can reach this final cut in G belong to H.

$$\begin{array}{ll}
 & \{\ C_0, ..., C_{(k-1)} \text{ is a path in } L' \text{ along which } p \text{ is true }\} \\
 & L' \models \mathsf{EG}(p) \\
\equiv & \{\ \mathsf{EG}(p) \equiv \neg\mathsf{AF}(\neg p)\ \} \\
 & L' \models \neg\mathsf{AF}(\neg p) \\
\equiv & \{\ \text{disjunctive predicate } \neg p \text{ is observer-independent and universal }\}
\end{array}$$

$L' \models \neg\mathsf{EF}(\neg p)$

$\equiv$ $\{ \neg\mathsf{EF}(\neg p) \equiv \mathsf{AG}(p) \}$

$L' \models \mathsf{AG}(p)$

$\Rightarrow$ $\{$ since $C_{(k-1)} \cap Jq \subseteq C_{k-1}$ $\}$

There exists a path $K_0, \ldots, K_m$ in L' starting from the initial consistent cut and ending at $C_{(k-1)} \cap Jq$ such that p holds along the sequence.

Setting $C = C_{(k-1)}$, $D = Jq$, $F = C_k$ in Lemma 6.6, there are two cases for $(C_{(k-1)} \cap Jq)$:

Case 1: $(C_{(k-1)} \cap Jq) = Jq$, then $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ is true since there exists a sequence $K_0, \ldots, K_m$.

Case 2: $(C_{(k-1)} \cap Jq) \triangleright Jq$, then $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ is true since there exists a sequence $K_0, \ldots, K_m, Jq$. □

Using Theorem 6.17, in algorithm $\mathsf{Algo}_{6.3}$ we check only for the existence of a path starting from the initial consistent cut and ending at Jq. This greatly simplifies the task of detecting $\mathsf{E}(p \mathbin{\mathsf{U}} q)$ because otherwise we would have to check for the existence of a path starting from the initial consistent cut and ending at an *arbitrary* consistent cut that satisfies q.

Remark 6.18 *Observe that predicate q in Theorem 6.17 could be "weaker" than a linear predicate in that only the existence of a least consistent cut that satisfies the predicate q is required. Also, predicate p should be such that its complement is an observer-independent predicate. Conjunctive predicates is one such class of predicates.*

Complexity Analysis 6.19 *When q is a linear predicate, algorithm $\mathsf{Algo}_{3.1}$ in Section 3.3 is applicable at line 1. We can compute Jq in $O(n|E|)$ time assuming the efficient predicate evaluation and efficient advancement properties (Property 3.15* *and Property 3.17 in Section 3.3) in $O(n|E|)$ time. The for loops at line* 3 *and at line* 5 *are executed n times. At line* 8*, the optimal algorithm presented in [MG01a] for computing the slice of a conjunctive predicate p is applicable for detecting* $\mathsf{EG}(p)$*. The complexity of checking* $\mathsf{EG}(p)$ *using this algorithm is* $O(n|E|)$*. (Note that we can also use the* $\mathsf{EG}(p)$ *algorithm which we will describe next at this step since a conjunctive predicate is also linear.) Adding an edge takes constant time, therefore the complexity of the for loop at line* 5 *is* $O(n|E|)$*. The overall time-complexity of the algorithm is* $O(n|E|)$*.*

Remark 6.20 *We can use the equivalence* $\mathsf{A}(p\mathsf{U}q) \equiv \neg(\mathsf{EG}(\neg q) \vee \mathsf{E}(\neg q \mathsf{U}(\neg p \wedge \neg q)))$ *to detect predicates under* AU *operator. This equality gives a representation of* AU *operator in terms of* EU *and* EG *operators. When predicates p and q are disjunctive, we can use the algorithm developed for* EU*, since* $(\neg q)$ *is a conjunctive predicate and* $(\neg p \wedge \neg q)$ *is a linear predicate. Similarly we can use the algorithm presented in [MG01a] for detecting conjunctive predicates under* EG *operator to detect* $\mathsf{EG}(\neg q)$*. The overall time-complexity of* $\mathsf{A}(p \ \mathsf{U} \ q)$ *algorithm is therefore* $O(n|E|) + O(|E|)$*, which is* $O(n|E|)$*.*

6.6 Predicate Detection for $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$

In this section we present efficient algorithms for detecting non-temporal linear predicates under EG and AG operators. Linear predicates include several useful predicate classes like conjunctive predicates, regular predicates, monotonic channel predicates and some relational predicates. Garg et al. [CG98] presented an efficient algorithm for detecting non-temporal linear predicates under EF operator. We presented this in algorithm $\mathsf{Algo}_{3.1}$ of Section 3.3. Our results improve the algorithm in Section 6.3 when detecting predicates $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$, for non-temporal regular or non-temporal linear predicate p.

```
Algorithm Algo$_{6.4}$:

    Input: (1) a directed graph $G$,   (2) a non-temporal linear predicate $p$
    Output: $G$ satisfies EG($p$) or not

1   $W$ := the final consistent cut of $G$;
2   if $W \not\models p$ then
3       return false;
    endif;
4   while $W \neq$ initial($G$) do
5       $Q := \{C \mid C \models p \ \wedge \ C \triangleright W\}$;     // $Q$ is the set of predecessors of $W$ that satisfy $p$
6       if $Q = \emptyset$ then
7           return false;                        // no predecessor that satisfies $p$ exists
        else
8           let $W$ be an arbitrary element from $Q$;
        endif;
    endwhile;
9   return true;
```

Figure 6.9: The algorithm to decide whether EG(p) is satisfied or not

6.6.1 EG(p) Detection

In this section, we explain algorithm Algo$_{6.4}$ in Figure 6.9 for detecting whether a computation satisfies a predicate of the form EG(p), when p is a non-temporal linear predicate.

A computation satisfies EG(p) if there exists a fullpath starting from the initial consistent cut such that every cut on the path satisfies the predicate. Our algorithm generates such a path backwards, that is, starting from the final consistent cut moving toward the initial consistent cut. If the final consistent cut of the computation (computed at line 1) does not satisfy p, then such a path does not exist and the algorithm returns false at line 3. Otherwise, in each iteration of the while loop at line 4, we move backward by one predecessor cut starting from the final consistent cut until we reach the initial consistent cut. In the while loop, we compute the set of consistent cuts that precede the current consistent cut and that satisfy p at line 5. If this set is nonempty then we choose an *arbitrary* element from

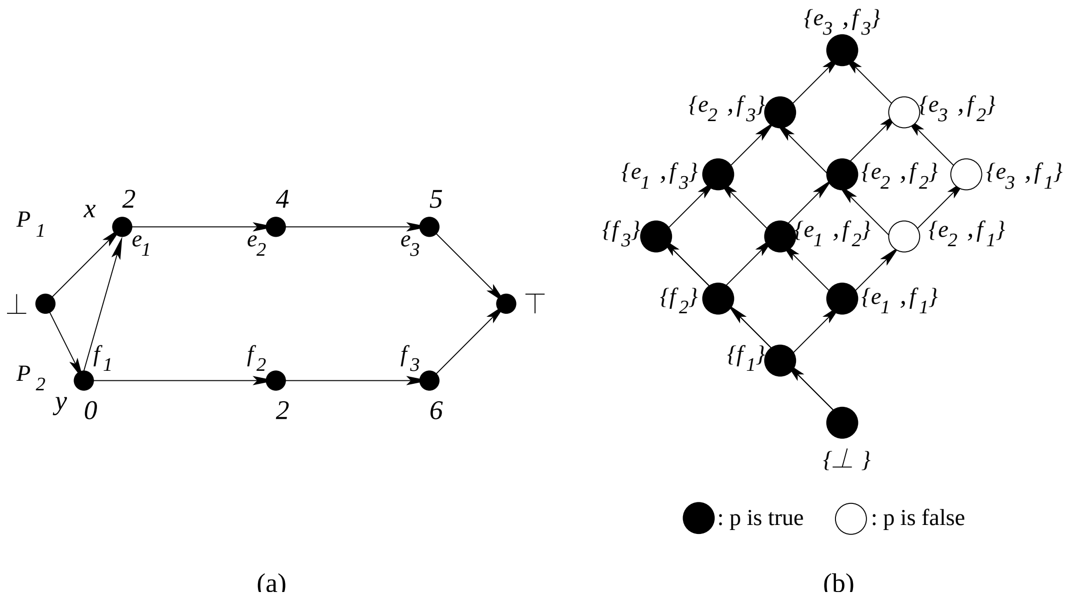


Figure 6.10: (a) A computation, and (b) its lattice corresponding to nontrivial consistent cuts

this set at line 8. If the set is empty then there is no path that starts from the final consistent cut and reaches the initial consistent cut along which p holds. Therefore, the algorithm returns false at line 7. Finally, if the while loop terminates when W is the initial consistent cut then $\mathsf{EG}(p)$ is satisfied.

Consider the computation depicted in Figure 6.10(a). It is clear that the computation satisfies $\mathsf{EG}(p)$. The application of algorithm $\mathsf{Algo}_{6.4}$ returns an arbitrary fullpath along which p holds. One such fullpath, given in reverse order, is $\{e_3, f_3\}$, $\{e_2, f_3\}$, $\{e_2, f_2\}$, $\{e_1, f_2\}$, $\{f_2\}$, $\{f_1\}$, $\{\perp\}$.

Next we will prove the correctness of algorithm $\mathsf{Algo}_{6.4}$.

Theorem 6.21 *Given a computation G and a linear predicate p, algorithm* $\mathsf{Algo}_{6.4}$ *returns true iff $G \models \mathsf{EG}(p)$.*

Proof: If the algorithm returns true then it is clear that $\mathsf{EG}(p)$ holds. Now we prove that if $\mathsf{EG}(p)$ holds then the algorithm returns true. Since $\mathsf{EG}(p)$ holds, there exists a fullpath π starting from the initial consistent cut along which p holds.

The invariant for the while loop is "there exists a fullpath starting from W along which p holds". We prove by induction that after every iteration j the invariant still holds. It suffices to prove that Q is nonempty. In other words, there exists a predecessor of W that satisfies p.

Base case ($j = 1$): Since $\mathsf{EG}(p)$ holds in the computation and W is not the initial consistent cut, Q is nonempty. At line 8, the new value of W is updated such that it satisfies p and it is a predecessor of the old value of W. Therefore, the invariant holds after the first iteration.

Induction step ($j = k+1$): Assuming that the invariant is true up to $j = k$, we now prove that it holds for $j = k+1$. When the condition of the while loop holds, we know that W is not the initial consistent cut. Let π^m be the first cut on π such that $W \subseteq \pi^m$. Setting $D = W$, $C = \pi^{m-1}$, and $F = \pi^m$ in Lemma 6.6, we have $(\pi^{m-1} \cap W) = W$ or $(\pi^{m-1} \cap W) \triangleright W$. The former case is not possible because then $W \subseteq \pi^{m-1}$, which implies that π^{m-1} is the first cut on π such that $W \subseteq \pi^{m-1}$. Furthermore, since p is linear and it holds at π^{m-1} and W, p holds at $(\pi^{m-1} \cap W)$. Therefore, there exists a predecessor of W, which is $(\pi^{m-1} \cap W)$, that satisfies p.

At the end of the $(|E|-1)$'th iteration of the while loop, W becomes the initial consistent cut. Since it satisfies p, the algorithm returns true at line 9. $\square$

Complexity Analysis 6.22 *We can compute the final consistent cut of G in $O(n)$ time. Assuming a property similar to efficient advancement property (Property 3.17), we can compute a predecessor consistent cut that satisfies p in $O(n)$ time. (Consider a conjunctive predicate predicate. The crucial element in this case is the event from a process on which the value of the local predicate stays true). Any fullpath in the lattice of consistent cuts has at most $|E|$ length since there are $|E|$ events in the computation. Therefore the complexity of the while loop is $O(n|E|)$ time. The overall time-complexity is $O(n|E|)$. Since regular predicates are contained in linear*

```
Algorithm Algo$_{6.5}$:

    Input: (1) a directed graph $G$, and (2) a non-temporal linear predicate $p$
    Output: $G$ satisfies AG($p$) or not

1   $V := \mathcal{M}(L) \cup \{\mathcal{E}\}$    // the set of all meet-irreducible elements plus the final consistent cu
            // check whether all consistent cuts in $V$ satisfy $p$
2   for each consistent cut $C$ in $V$ do
3       if $C \not\models p$ then
4           return false;
        endif;
    endfor;
5   return true
```

Figure 6.11: The algorithm to decide whether $\mathsf{AG}(p)$ is satisfied or not.

predicates, this result applies to non-temporal regular predicates as well and improves the $O(n^2|E|)$ complexity in [GM01].

6.6.2 $\mathsf{AG}(p)$ Detection

In this section, we explain algorithm $\mathsf{Algo}_{6.5}$ in Figure 6.11 for detecting whether a computation satisfies a predicate of the form $\mathsf{AG}(p)$, when p is a linear predicate. For this purpose, we use Birkhoff's Representation Theorem for Finite Distributive Lattices (Theorem 2.11) explained in Chapter 2.

Birkhoff's Theorem implies that there is a one-to-one correspondence between a finite poset and a finite distributive lattice. Every element of a finite distributive lattice (except for the final element) can be defined as the meet of a subset of meet-irreducible elements of the lattice. The following is a corollary of Theorem 2.11.

Corollary 6.23 ([DP90]) *Given an element $x \in L$ for a finite distributive lattice L, $x = \bigsqcap \{y \in \mathcal{MI}(L) \mid x \subseteq y\}$.*

A computation satisfies $\mathsf{AG}(p)$ if all consistent cuts of the computation satisfy p. Using the above corollary, in order to detect a linear predicate under AG operator, it is necessary and sufficient to check whether the predicate is satisfied at the meet-irreducible elements and the final consistent cut of the computation. We compute these elements and the final consistent cut at line 1 of algorithm $\mathsf{Algo}_{6.5}$. If all cuts in this set satisfy p then the predicate is satisfied and the algorithm returns true at line 5, otherwise the algorithm returns false at line 4. The correctness of the algorithm follows from the corollary above.

Consider the computation depicted in Figure 6.12(a) and its lattice of non-trivial consistent cuts in Figure 6.12(b). Pictorially, in a finite distributive lattice an element is meet-irreducible if and only if it has exactly one upper cover, that is, it has exactly one outgoing edge. The meet-irreducible elements of the computation are shown as patterned circles in Figure 6.12(b). The consistent cut $\{f_2\}$ can be obtained by the intersection of meet-irreducible elements $\{f_3\}, \{e_1, f_3\}, \{e_2, f_3\}$, and $\{e_3, f_2\}$. Similarly, we can obtain other consistent cuts by the intersection of a subset of meet-irreducible elements.

The number of meet-irreducible elements of a distributive lattice is generally exponentially smaller than the number of all cuts in the lattice. In fact, for a finite distributive lattice, the number of meet-irreducible elements is exactly equal to the size of the longest chain in the lattice [DP90]. In our case, the length of the longest chain is equal to the number of events $|E|$. Hence, if some computation can be done on meet-irreducible elements, we get a significant computational advantage.

Complexity Analysis 6.24 *A vector clock algorithm such as Fidge/Mattern's is used to compute the join-irreducible elements of a computation. We can use the same algorithm with slight modifications to compute the meet-irreducible elements as follows. We apply the vector clock algorithm backwards on the computation, that is, with all the edges in the computation reversed. Whenever the local component*

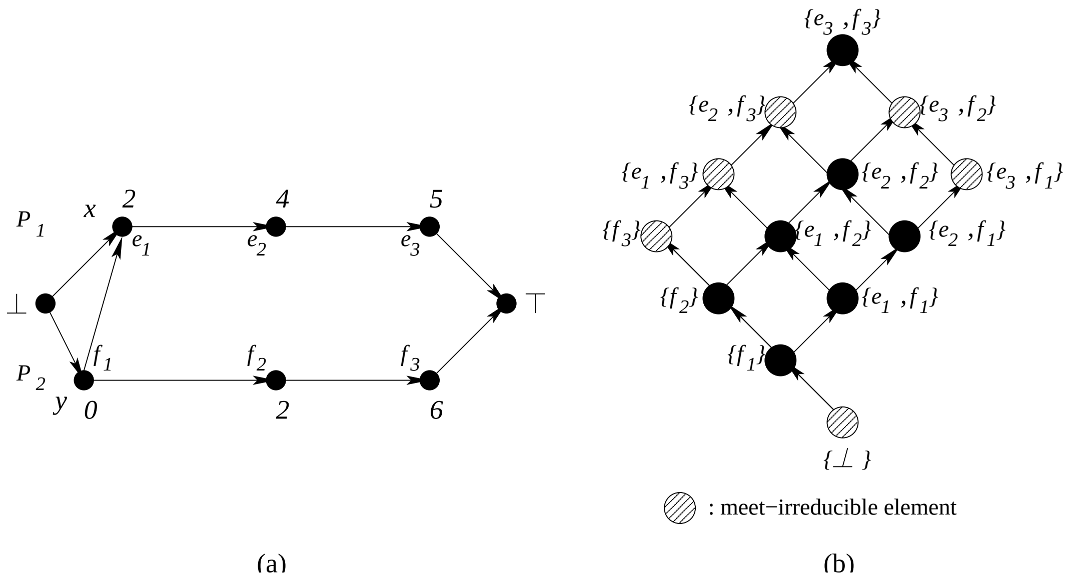


Figure 6.12: (a) A computation, and (b) its lattice corresponding to nontrivial consistent cuts

of a vector clock is incremented in the original algorithm, instead we decrement it. Whenever the vector clock of a process is updated by taking a component-wise maximum with that of the received vector clock, instead we take the component-wise minimum with that of the received vector clock. Using this modified vector clock algorithm the complexity of computing the set of meet-irreducible elements at line 1 *is* $O(|E|)$ *time. The for loop at line* 2 *is executed at most* $|E| + 1$ *times since the number of meet-irreducible elements is* $|E|$*. Further, assuming the efficient predicate evaluation property, the check at line* 3 *takes linear time. Therefore, the overall time-complexity of the algorithm is* $O(n|E|)$*.*

Remark 6.25 *We can use the above* $\mathsf{EG}(p)$ *and* $\mathsf{AG}(p)$ *detection algorithms when* p *is a* post-linear *predicate [CG98] as well. Post-linear predicates are such that the set of consistent cuts that satisfy a post-linear predicate is closed under join, whereas the set of consistent cuts that satisfy a linear predicate is closed under meet.*

Algorithm $\mathsf{Algo}_{6.4}$ *can be modified so that it starts from the initial consistent cut and moves toward the final consistent cut. The next consistent cut to move is determined based on picking one of the consistent cuts that is a successor of the current consistent cut and that satisfies the predicate. Again, it does not matter which one of these consistent cuts is chosen as long as there exists one.*

Algorithm $\mathsf{Algo}_{6.5}$ *can be modified such that at line* 1 *instead of computing meet-irreducible elements and the final consistent cut, the new algorithm computes the join-irreducible elements and the initial consistent cut. This is because every consistent cut except the initial one can be obtained by the join of a subset of join-irreducible elements.*

6.7 Detecting Observer-Independent Predicates

In this section we present intractability results for detecting observer-independent predicates under EG and AG operators. A predicate is called *observer-independent* if there exists a consistent cut that satisfies the predicate for some observation, then there exists a consistent cut that satisfies the for all observations, and vice versa. Since the initial consistent cut belongs to all observations, if a predicate is satisfied at the initial consistent cut, then the predicate is observer-independent. Some examples of observer-independent predicates are stable predicates and local predicates. Unlike linear predicates, observer-independent predicates are closed under disjunction, but not under conjunction.

It was shown in [CG98] and [TG98b] that detecting an arbitrary predicate is NP-complete under EF and EG operators, respectively. Next, we prove that detecting an observer-independent predicate is NP-complete under EG operator and co-NP-complete under AG operator.

Theorem 6.26 *Given a computation, detecting an observer-independent predicate* p *under* EG *operator is NP-complete.*

Proof: The proof of the theorem is very similar to the proof of NP-completeness for an arbitrary predicate under EG operator [TG98b]. The problem is in NP because it takes polynomial-time to check that a candidate fullpath is valid and that it satisfies the predicate p. To show that it is NP-hard, we reduce SAT [GJ91] to an instance this problem. If p is the boolean expression in SAT, then for each variable $x_1, \ldots, x_m$ in p, we assign a separate process with two events, true and false (as shown in Figure 6.13(a), for convenience, we do not display the trivial final event $\top$ in the figure). We define a process for an extra boolean variable x_{m+1} which starts true, goes through a false event, and ends true again. We define $B = p \vee x_{m+1}$. It is clear that B is observer-independent since it is satisfied at the initial consistent cut. Then we apply EG algorithm to detect predicate B. If there exists a fullpath along which B is true (that is $\mathsf{EG}(B)$ is satisfied), then the consistent cut with $x_{m+1} = false$ will have a satisfying assignment for the variables of p. Conversely, if p is satisfiable, then there exists a satisfying fullpath. $\square$

Theorem 6.27 *Given a computation, detecting an observer-independent predicate p under* AG *operator is co-NP-complete.*

Proof: The problem is in co-NP because it takes polynomial-time to check that a candidate consistent cut satisfies the negation of the predicate p. To show that it is co-NP-hard, we reduce TAUTOLOGY [GJ91], a co-NP-complete problem, to an instance of this problem. If p is the boolean expression in TAUTOLOGY, then for each variable $x_1, \ldots, x_m$ in p, we assign a separate process with true and false events (Figure 6.13(b)). We define a process for an extra boolean variable x_{m+1} which starts true, and ends at a false event. We define $B = p \vee x_{m+1}$. Similar to the proof of above theorem, it is clear that B is observer-independent since it is satisfied at the initial consistent cut. We then apply AG algorithm to detect invariance of B. If the algorithm returns true, then all fullpaths satisfy the predicate and all consistent cuts with $x_{m+1} = false$ will have satisfying assignments for the variables of p. Conversely, if p is a tautology, then all consistent cuts in all sequences will satisfy p. $\square$

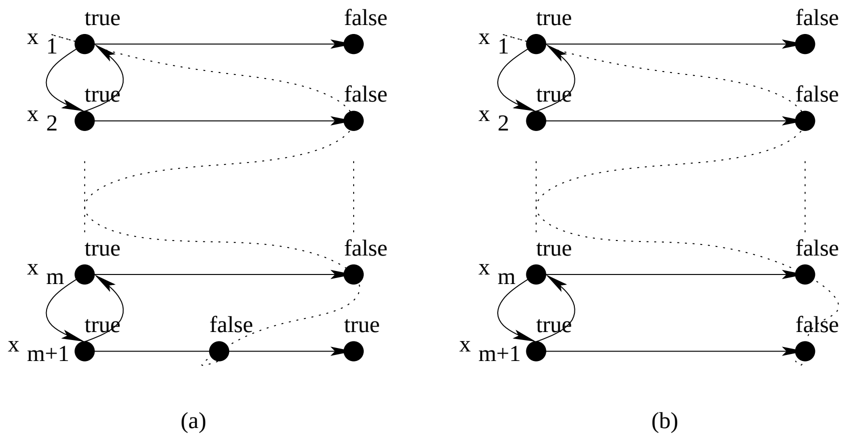


Figure 6.13: Detecting an observer-independent predicate p under (a) EG operator is NP-complete, (b) AG operator is co-NP-complete

Although predicate detection of observer-independent predicates under EG and AG operators is intractable, there are efficient algorithms under EF and AF operators [CBDGF95]. Also, subclasses of observer-independent predicates such as disjunctive predicates have polynomial-time detection algorithms under EG and AG operators. These algorithms can be obtained using the duality of temporal operators such as $\mathsf{AG}(p) \equiv \mathsf{EF}(\neg p)$, where $\neg p$ is a conjunctive predicate [GW92] and similarly for EG. Finally, we can check whether a stable predicate is satisfied under EG and AG operators by simply checking whether the predicate is true at the initial consistent cut.

6.8 Generating Counter Examples

In this section we show how to obtain counter examples in case there is a consistent cut that does not satisfy the given predicate. Equivalently, we show how to obtain a witness when there is a consistent cut that satisfies the given predicate. For this we define the following problem.

Computing Cut (COMPC) Given a directed graph G and a predicate p, compute a consistent cut of G that satisfies p, if any.

We establish that the problem COMPC is equivalent to the problem CONTC defined in Section 4.3, that is, $G \models \mathsf{EF}(p)$. While for testing purposes, it is sufficient to determine whether there exists a faulty consistent cut in a computation; for debugging purposes, it is desirable to actually *locate* the faulty consistent cut. This is because an examination of such a cut may provide valuable insight into the bug that caused the fault. Evidently, COMPC is at least as hard as CONTC. We prove the converse. The main idea is follows. Starting with the lattice of consistent cuts of the given directed graph, successively shrink the lattice—by adding edges to the graph—until it consists of only a single nontrivial consistent cut. Of course, at each step, an edge is added in such a way that the resultant (shrunken) lattice contains at least one consistent cut that satisfies the predicate. The algorithm is presented in Figure 6.14. The next theorem establishes the correctness of the algorithm.

Theorem 6.28 *The algorithm in Figure 6.14 solves* COMPC.

Proof: The algorithm first checks whether some consistent cut of G satisfies p. In case no such consistent cut exists, it simply outputs the appropriate message at line 1 and terminates. Now, assume that G does contain a consistent cut that satisfies p. In that case, the for loop at line 3 executes exactly $|E|$ times, where E is the set of events. Let K_i, where $0 \leqslant i \leqslant |E|$ and $K_0 = G$, be the value of K when the i^{th} iteration of the for loop ends. The algorithm maintains the invariant that each K_i contains at least one consistent cut that satisfies p. The proof is by induction on

```
Algorithm Algo$_{6.6}$:

    Input: (1) a directed graph $G$,   (2) a predicate $p$, and
           (3) an algorithm to evaluate CONTC$(H, p)$ for an arbitrary directed graph $H$

    Output: a consistent cut of $G$ that satisfies $p$, if any

1   if $not$(CONTC$(G, p)$) then output "no consistent cut of $G$ satisfies $p$" endif;
2   $K := G$;
3   for each event $e$ do
4       if CONTC$(K[e, \bot_1], p)$ then              // $K[e, f]$ is the graph obtained
                                                      // by adding an edge from $e$ to $f$ in K
5           add an edge from $e$ to $\bot_1$ in $K$;  // $K := K[e, \bot_1]$
6       else add an edge from $\top_1$ to $e$ in $K$; // $K := K[\top_1, e]$
        endif;
    endfor;

    // at this point, $K$ has exactly one nontrivial consistent cut
7   output the nontrivial consistent cut of $K$;
```

Figure 6.14: An algorithm to solve COMPC using an algorithm to solve CONTC.

i. When $i = 0$, by definition, $K_0 = G$ and the proposition holds. Assume that the proposition holds for K_i. We need to show that it also holds for K_{i+1}. Let e be the event in consideration at the $(i+1)^{st}$ iteration of the for loop. At line 4 the algorithm tests whether $K_i[e, \bot_1]$ contains a consistent cut that satisfies p. If the answer is yes, then K_{i+1} is set to $K_i[e, \bot_1]$ at line 5. Clearly, in this case, K_{i+1} contains a consistent cut that satisfies p. On the other hand, if the test fails, then K_{i+1} is set to $K_i[\top_1, e]$. From Lemma 4.13, $\mathcal{C}(K_i) = \mathcal{C}(K_i[e, \bot_1]) \cup \mathcal{C}(\widehat{K}_i[e, \bot_1])$. From induction hypothesis, K_i contains a consistent cut that satisfies p. However, $K_i[e, \bot_1]$ does not contain any such consistent cut. Therefore $\widehat{K}_i[e, \bot_1]$ should contain a consistent cut that satisfies p. Note that $\widehat{K}_i[e, \bot_1]$ is obtained from K_i by adding edges $(\bot_1, \bot_1)$ and $(\top_1, e)$. Clearly, the presence or absence of the edge $(\bot_1, \bot_1)$ does not affect the set of consistent cuts. Therefore K_{i+1} contains a consistent cut that satisfies p.

We next prove that $K_{|E|}$ contains exactly one nontrivial consistent cut. Assume the contrary. Therefore $K_{|E|}$ contains two *nontrivial* consistent cuts C and D

with $C \neq D$. Evidently, either $C - D \neq \emptyset$ or $D - C \neq \emptyset$. Without loss of generality assume that it is the former, and let e be an event in $C - D$. The for loop at line 3 considers events in E one by one. Let the iteration corresponding to e be i. After the i^{th} iteration ends, K_i contains either $(e, \bot_1)$ or $(\top_1, e)$. Since no edge is ever deleted, $K_{|E|}$ also contains either $(e, \bot_1)$ or $(\top_1, e)$. In the first case, since $e \notin D$, $\bot_1 \notin D$—a contradiction. In the second case, since $e \in C$, $\top_1 \in C$—a contradiction.

$\square$

An illustration of the algorithm is given below.

Example 6.29 *Consider the computation shown in Figure 6.15(a). In the figure,* e_1 *and* f_1 *are the initial events and* e_4 *and* f_4 *are the final events. Suppose we want to find a consistent cut for which both* x_1 *and* x_2 *evaluate to true. In the figure, we represent an event for which the corresponding local predicate evaluates to true—referred to as a* true event*—using a solid circle. Clearly, there are two consistent cuts which satisfy* $x_1 \wedge x_2$*, namely* $\{e_1, e_2, f_1\}$ *and* $\{e_1, e_2, f_1, f_2\}$*. Suppose the events are considered in the order* $e_3\, f_2\, f_3\, e_2$*. (There is no need to consider the initial and final events.) Figure 6.15(b) to Figure 6.15(e) depict the sequence of iterations of the for loop. In the first iteration, an edge is added from* $\top_1$ *to* e_3*. In the second iteration, an edge is added from* f_2 *to* $\bot_2$*, which is equivalent to adding an edge from* f_2 *to* $\bot_1$*. In the third iteration, an edge is added from* $\top_2$ *to* f_3*, which is same as adding an edge from* $\top_1$ *to* f_3*. Finally, in the fourth iteration, an edge is added from* e_2 *to* $\bot_1$*. Clearly, the slice has exactly one nontrivial consistent cut, namely* $\{e_1, e_2, f_1, f_2\}$*.*

In general, a graph may contain more than one consistent cut satisfying the given predicate. The specific consistent cut that is output at line 7 depends on the order in which the events are considered at line 3.

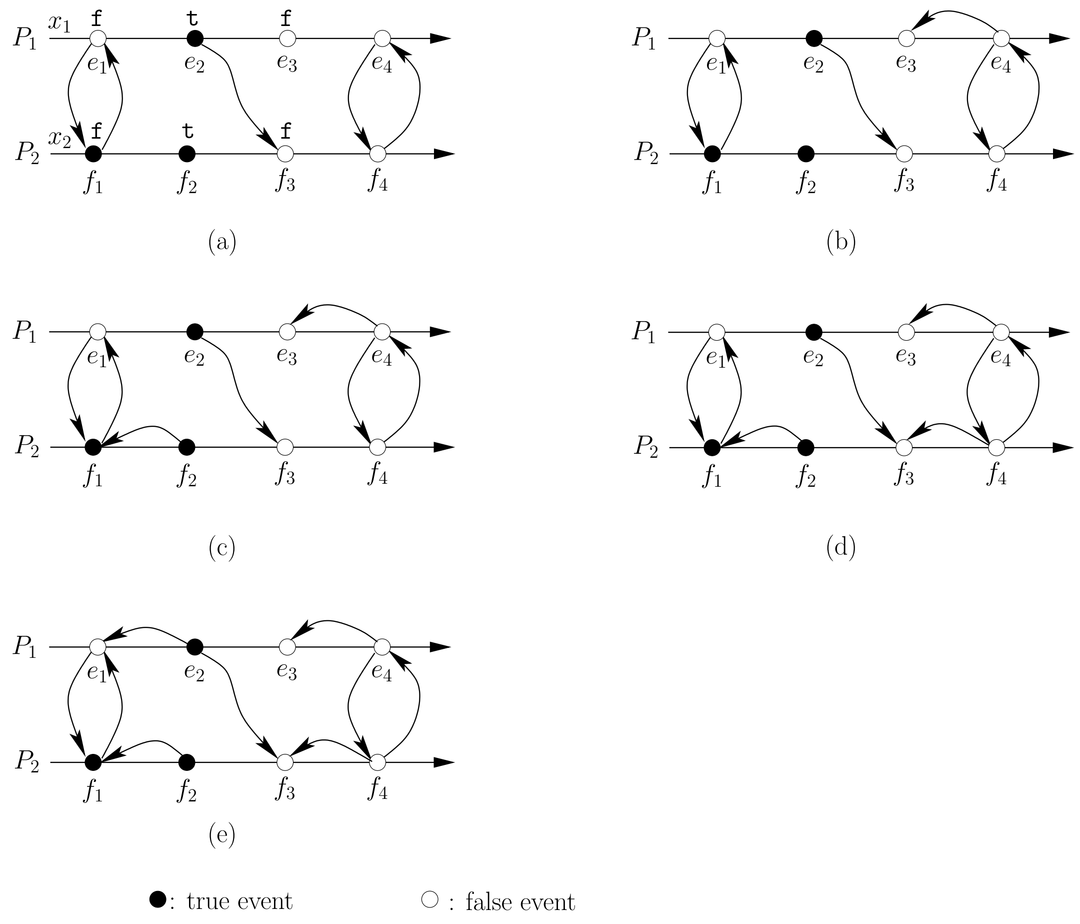


Figure 6.15: All illustration of the algorithm in Figure 6.14.

Theorem 6.30 *The time-complexity of the algorithm for solving* COMPC *in Figure 6.14 is $O(|E|T)$, where E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC.

Proof: Clearly, the initialization at line 2 has $O(n|E|)$ time-complexity, where n is the number of processes and E is the set of events, assuming that the skeletal representation is used. Also, each iteration of the for loop at line 3 has $O(T)$ time-complexity. Therefore the overall time-complexity of the algorithm is $O(n|E| + |E|\,T)$. It is reasonable to assume that $T = \Omega(n)$. This gives the time-complexity of $O(|E|\,T)$. $\square$

It is possible to further reduce the time-complexity to $O(n \log(|E|)T)$ using a technique similar to binary search. The main idea is as follows. Consider an event e on process P_i. Suppose the if statement at line 4 evaluates to true for e. In that case, we will add an edge from e to $\bot_1$. The edge will also create a path from every event that occurred before e on P_i to $\bot_1$. These events do not need to be considered at all by the for loop and can be simply ignored. On the other hand, in case the if statement evaluates to false, we will add an edge from $\top_1$ to e. Similar to before, the edge will also create a path from $\top_1$ to every event that occurred after e on P_i. Again, these events do not need to be considered by the for loop. Therefore, by selecting e appropriately, it is possible to eliminate close to half of the remaining events on P_i from consideration in a similar fashion as binary search.

Remark 6.31 *Using our results in this section and Section 4.3, we have that* $\mathsf{CONTC} \cong \mathsf{COMPC} \cong \mathsf{COMPS}$.

We use the algorithm of this section to generate a witness when the predicate is of the form $\mathsf{EF}(p)$*. Note that our slicing algorithms can also be used to generate a witness. (A counter example is simply a witness to the complement of the predicate.) In that, when the predicate* $\mathsf{EF}(p)$ *is satisfied, any consistent cut of the* $\mathsf{slice}(G, \mathsf{EF}(p))$ *serves as a witness. When the predicate* $\mathsf{EG}(p)$ *is satisfied, any observation in* $\mathsf{slice}(G, \mathsf{EG}(p))$ *serves as a witness. When the predicate* $\mathsf{AG}(p)$ *is satisfied, any observation in G serves as a witness. However, the witness generation based on slicing has a higher complexity than witness generation algorithm of this section.*

6.9 Discussion

Predicate detection algorithms of this chapter are useful in increasing the understanding of program behavior when a property fails or holds. Also, due to the nature of working on a single trace, these algorithms are most appropriate for finding bugs rather than proving programs correct. For example, when a safety property such as $\mathsf{AG}(p)$ does not hold on a trace, we can conclude that the program does not satisfy the property. However, when the safety property holds on a trace, it might not hold for other traces or when the same trace is extended.

In order to detect liveness property violations, we need to reason about infinite execution traces. For finite state systems, such infinite traces are generated when a consistent cut is revisited (resulting in a loop). However, our current algorithms do not find such loops, hence when we report a violation of a liveness property, the property may indeed hold for the program. For example, when a liveness property such as $\mathsf{EF}(p \wedge \mathsf{EG}(q))$ fails on a trace, it might hold for other traces or when the same trace is extended by new events.

Chapter 7

POTA System and Experiments

In this chapter, we describe an experimental evaluation of our partial order trace analysis method which is based on the predicate detection algorithms in Chapter 6.

7.1 Overview

In order to quantify the effectiveness of our predicate detection algorithms, we developed a prototype system named Partial Order Trace Analyzer (POTA). POTA contains implementation of our novel computation slicing technique for predicate detection.

Section 7.2 gives an overview of the tool architecture with the implementation details of representing the partial order relation between the events in a trace.

We present predicate detection experiments in Section 7.3. The first set of experiments demonstrate the effectiveness of our slicing based predicate detection algorithms. The second set of experiments demonstrate the effectiveness of our $\mathsf{AF}(p)$ detection algorithms. Our experimental work covers a wide range of protocols.

Some of these protocols are a message passing dining philosopher protocol, a shared variable mutual exclusion protocol, a CORBA General Inter-ORB Protocol, and a directory based cache coherence protocol.

Finally, we present a discussion of our experimental work in Section 7.4.

7.2 POTA Architecture

The overall structure of POTA architecture is shown in Figure 7.1. We implemented the tool using the Java programming language [GJSB00]. The input to POTA is a program and a specification. The specification is given in a text file and then parsed. The program is instrumented by the instrumentor module and executed, generating a partial order trace. The trace is then fed into the analyzer module, which implements our slicing and predicate detection algorithms. The tool also contains a translator module that enables the usage of a model checker on a program trace rather than on a program. Next we will describe the details of each module.

7.2.1 Instrumentor

The instrumentor module inserts code at the appropriate places in the program to be monitored. The instrumented program is such that it outputs the relevant events and a vector clock that is updated for each such event. The events can be classified as internal and external events. An external event is a send, receive, read, or write event. A relevant event is such that it assigns values to the atomic propositions in the predicate, hence, the value of the predicate might change or a relevant event is a send or a receive event. An example of a relevant event in a message passing program is an internal event that updates the value of a variable that appears in the predicate. An example of a relevant event in a shared variable program is a write event of a shared variable that appears in the predicate.

In order to represent the partial order relationship between the events, we

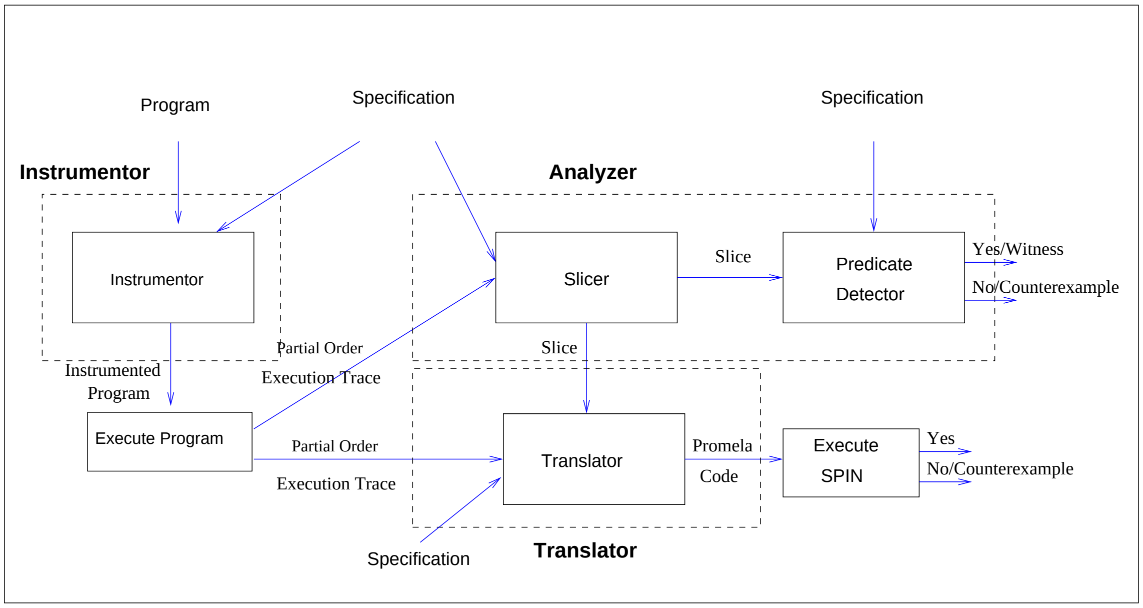


Figure 7.1: Overview of POTA Architecture

use a vector clock mechanism which we described in Chapter 2. We refer the reader to [Fid91, Mat89] for a detailed discussion on vector clock mechanisms. Now we will present details of vector clock algorithms implemented in POTA for message passing and shared memory programs, which are taken from [Gar02] and [SRA03] for message passing and shared variable programs, respectively. It was proven in [Gar02] and [SRA03] that the mentioned vector clock algorithms correctly implement causality.

Our implementation of vector clocks uses vectors of size n, the number of processes in the system. Each process (thread) is associated with a vector clock v. The algorithms presented in Figure 7.2 and Figure 7.4 are described by the initial conditions and the actions taken for each event type.

In algorithm $\mathsf{Algo}_{7.1}$, a process increments its own component of the vector clock only after a relevant event. A process includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by

```
Algorithm Algo$_{7.1}$:

    Input: vector clock $v$: array$[1..n]$ of integers for process $P_j$
    Output: vector clock $v$ updated for process $P_j$

1   initially $(\forall i : i \neq j : v[i] := 0) \wedge (v[j] := 1)$;

    // update the vector clocks for each event $e$ generated by a process $P_j$ as follows:

2   if $e$ is a relevant event then
3       $v[j] := v[j] + 1$;
    endif;

4   if $e$ is the send of an event to process $P_k$ then
5       send $v$ to $P_k$;
    endif;

6   if $e$ is the receive of an event $g$ then
7       for $i := 1$ to $n$ do
8           $v[i] := \max(v[i], g.v[i])$;
        endfor;
    endif;

9   if $e$ is a relevant event then
10      output the pair $(e, v)$;
    endif;
```

Figure 7.2: A vector clock algorithm for message passing programs.

taking a component-wise maximum with the vector clock included in the message. Finally, if the event is relevant, then the event and its vector clock is output. For message passing programs, all of internal events that assign values to the atomic propositions in the predicate, send events and receive events are relevant. A sample execution of the algorithm is given in Figure 7.3.

In algorithm Algo$_{7.2}$, for each shared variable x there are two vector clocks v_x^a and v_x^w, denoted by access and write vector clocks, respectively. A process increments its own component of the vector clock only after a relevant event. A process updates its vector clock on reading a shared variable x by taking a component-wise maximum with the write vector clock of x. Then the access vector clock of x is

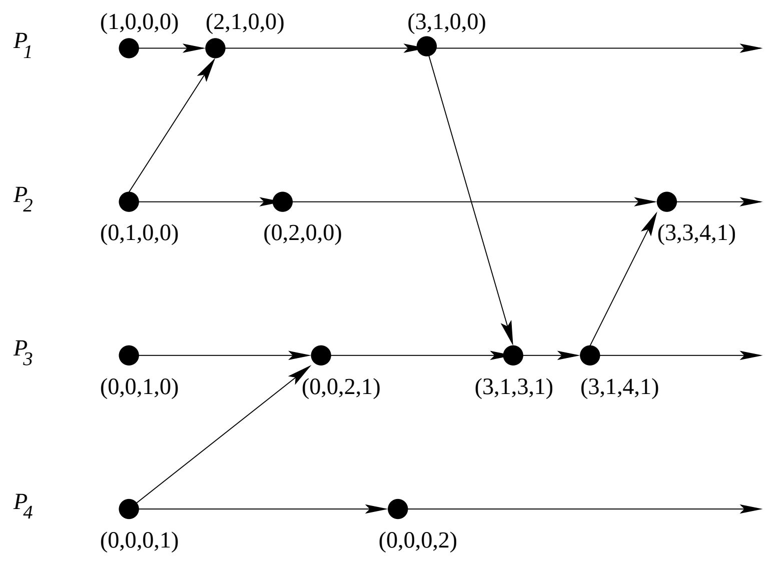


Figure 7.3: A sample execution of the vector clock algorithm

updated by taking a component-wise maximum with that of the vector clock of the process. On writing a shared variable, a process updates its vector clock by taking a component-wise maximum with the access vector clock of x. Then, the write and access vector clocks of x are set to the vector clock of the process. Finally, if the event is relevant, then the event and its vector clock pair is output.

Upon running the instrumented program, a separate log file is created for each process. Each log file consists of a sequence of (event, vector clock) pairs that a process generates. Furthermore, each such pair is also appended by the values of the variables that the event in the pair manipulates. The log files from all processes are then combined to obtain a partial order representation of the execution trace. Instead of using a log file, if every process sends its trace to a dedicated process which combines them during runtime, we can obtain a simple *on-line* verification environment. This environment can further be used to implement our on-line slicing algorithm in Section 4.4.

```
Algorithm Algo$_{7.2}$:

    Input: (1) a vector clock $v$: array[$1..n$] of integers for process $P_j$, and
           (3) read vector clock $v_x^a$ and write vector clock $v_x^w$ for each shared variable $x$

    Output: vector clocks $v$, $v_x^a$, and $v_x^w$ updated for process $P_j$ and for each shared variable $x$

1   initially $(\forall i : i \neq j : v[i] := 0) \ \wedge \ (v[j] := 1)$;
2   initially $(\forall i : v_x^a[i] := v_x^w[i] := 0)$;

    // update the vector clocks for each event $e$ generated by a process $P_j$ as follows:

3   if $e$ is a relevant event then
4       $v[j] := v[j] + 1$;
    endif;

5   if $e$ is the read of a shared variable $x$ then
6       for $i := 1$ to $n$ do
7           $v[i] := \max(v[i], v_x^w[i])$;
        endfor;
8       for $i := 1$ to $n$ do
9           $v_x^a[i] := \max(v_x^a[i], v[i])$;
        endfor;
    endif;

10  if $e$ is the write of a shared variable $x$ then
11      for $i := 1$ to $n$ do
12          $v[i] := \max(v_x^a[i], v[i])$;
        endfor;
13      $v_x^w := v_x^a := v$;
    endif;

14  if $e$ is a relevant event then
15      output the pair $(e, v)$;
    endif;
```

Figure 7.4: A vector clock algorithm for shared variable programs.

Currently, the instrumentation code is manually added to the programs.

7.2.2 Translator

The translator module takes a partial order representation of a trace and translates it into specific languages. Since we are working with concurrent and distributed programs which exhibit a lot of parallelism and independency, partial order reduction techniques can take advantage of these properties of programs. The SPIN model checker [Hol97] contains implementation of partial order reduction techniques and is one of the most effective model checking tools.

SPIN is an on-the-fly linear time logic (LTL) model checker that uses explicit state enumeration and the partial order reduction. The partial order reduction generates a subset of the lattice of consistent cuts. This is due to executing only a subset of events enabled from the current consistent cut rather than executing all enabled events. The selection of the subset of events exploits the commutativity of concurrent events, which results in the same consistent cut when the events are executed in different orders. For example, read events by different processes on different variables are commutative. The details of the partial order reduction can be found elsewhere [GW91, Val91, Pel93, Esp94, SUL00].

Currently, translation from traces to Promela (input language of SPIN) is supported. The translation mechanism is similar to the technique explained in [LL96] for translations from Message Sequence Charts (MSC) to Promela.

Translator module serves two purposes: (1) It enables comparison of our slicing technique with other techniques such as partial order reduction, (2) It enables detection of predicates that do not belong to RCTL. The second usage is performed when the predicate detection algorithm $\mathsf{Algo}_{6.1}$ reaches line 6. Note that we can use SPIN when there are equivalent specifications in linear temporal logic LTL for the corresponding RCTL+ predicate.

7.2.3 Analyzer

The analyzer module contains our computation slicing and predicate detection algorithms. In particular, it contains an implementation of our slicing algorithm $\mathsf{Algo}_{5.5}$, and predicate detection algorithms $\mathsf{Algo}_{6.1}$ and $\mathsf{Algo}_{6.2}$, respectively. In order to obtain the slice with respect to a predicate, our algorithms add edges to the input graph. Upon adding edges, we traverse the graph in order to compute the new values of $J(e)$ and $F(e)$ for each event e as explained in Section 5.4.2. This procedure aids in obtaining a skeletal representation of the slice.

7.3 Experiments

In order to evaluate the effectiveness of POTA, we perform experiments with scalable protocols; protocols which are composed of many processes. We compare our computation slicing based approach with partial order reduction based approach of SPIN.

7.3.1 Setup

All experiments were performed on a 1.4 Ghz Pentium 4 machine running Linux. We restricted the memory usage to 512MB, but did not set a time limit. The two performance metrics we measured are running time and memory usage. In the case of slicing both metrics also include the overhead of computing the slice. We measure the time with Unix "time" command for SPIN runs and with Java's "System.Milliseconds" function for POTA runs. We measure the memory usage by "runtime.totalMemory() - runtime.freeMemory()" for POTA runs and by the usage displayed by SPIN for SPIN runs. We run all the programs for 20 seconds and our measurements are averaged over 20 traces for each program.

We consider programs such as *distributed dining philosophers*, *primary-secondary*, *distributed mutual exclusion*, *GIOP*, *ATMR*, and *leader election* protocols. Further experimental results can be obtained from POTA website [POT03].

We are primarily interested in checking the violations of the safety and liveness properties of the programs. For this purpose, we check for the complements of these properties.

7.3.2 Distributed Dining Philosophers

We use the Java protocol from [Har98] for this exercise. The protocol consists of multiple philosophers who sit around a table and spend their time thinking and eating. However, a philosopher requires shared resources, such as forks, to eat. The protocol coordinates access to the shared resources. Each philosopher has 3 local states namely *think*, *hungry*, and *eat*. The philosophers do not have a central server that they can query for fork availability. Instead each philosopher has a servant who communicates with the two neighboring servants to negotiate the use of the forks. The servants send "need left fork", "need right fork", "pass left fork", and "pass right fork" messages back and forth. Each fork is always in the possession of some philosopher, one of the two on either side of the fork. When a philosopher finishes eating, it labels its two forks as dirty. A hungry philosopher's servant is required to give up a dirty fork in its possession, if asked for by its hungry neighbor's servant. This prevents starvation. We check the following properties.

1. We require mutually exclusive use of forks, that is, a shared resource should not be used by more than one philosopher at a time. This can be ascertained by checking whether two neighbor philosophers are eating at the same time. This safety property can be stated as $\bigwedge_{i,j \in 0 \ldots (n-1)} (\mathsf{AG}(\neg eat_i \vee \neg eat_j))$, where eat_i denotes that philosopher i is in eating state and j denotes the neighbor of philosopher i. We can check whether this property is violated by checking the complement of the safety property, which is $\bigvee_{i,j \in 0 \ldots (n-1)} (\mathsf{EF}(eat_i \wedge eat_j))$. Figure 7.5 displays our results for this property. SPIN took 16.06 seconds and 74 MB to complete for 6 processes. SPIN ran out of memory for more than 90 % of the runs for 7 processes and took 189.4 seconds and 493.2 MB to complete for the remaining runs. Whereas, POTA took 383 seconds and 52 MB to complete for 200 processes. Due to the overhead associated in generating traces, we stopped generating traces for more than 200 processes.

2. We check starvation freedom of the philosophers, that is, every hungry philosopher should be able to eat eventually. Symbolically, $\bigwedge_{i \in 0 \ldots (n-1)}(\mathsf{AG}(hungry_i \Rightarrow \mathsf{AF}(eat_i)))$, for each philosopher i. We check the complement of the property, which is $\bigvee_{i \in 0 \ldots (n-1)}(\mathsf{EF}(hungry_i \wedge \mathsf{EG}(\neg eat_i)))$, for each philosopher i. Figure 7.6 displays our results for this property. Observe that the negation of a local predicate $\neg eat_i$ is also a local predicate and furthermore it is a regular predicate. SPIN took 246.7 seconds and 301.7 MB to complete for 6 processes and it ran out of memory for more than 6 processes. Whereas, POTA took 835 seconds and 106 MB to complete for 200 processes.

3. We check the property $\mathsf{AG}(\mathsf{EF}(eat_i))$ which denotes that eating is possible from every state. Note that, this property does not need to be satisfied unless the philosopher is in hungry state. Figure 7.7 displays our results for this property. POTA took 1118 seconds and 80 MB to complete for 200 processes. There is no equivalent specification in LTL in this case, hence we do not present SPIN results.

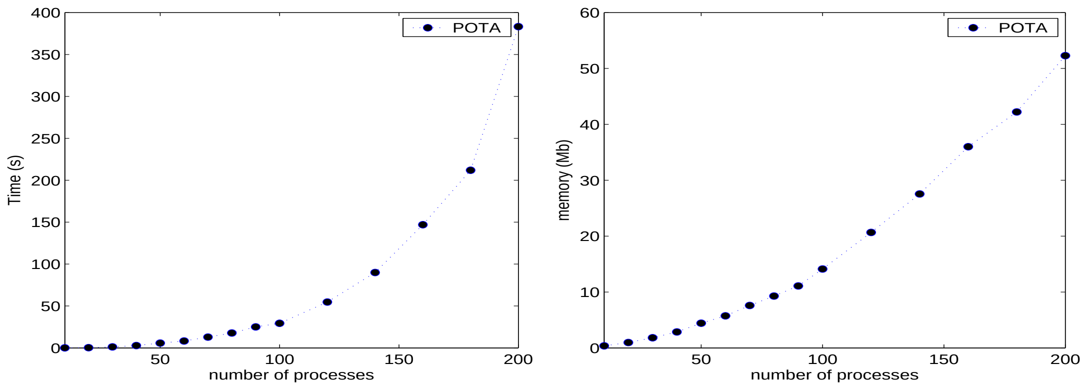


Figure 7.5: Dining philosophers verification results for Property (1)

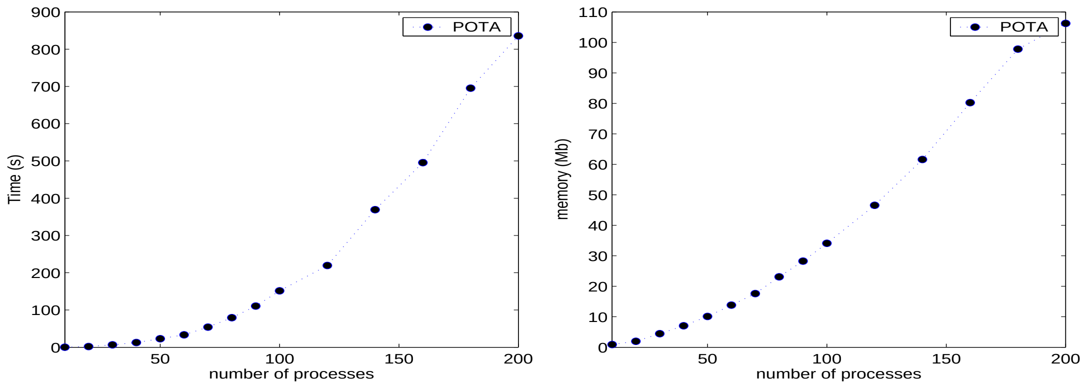


Figure 7.6: Dining philosophers verification results for Property (2)

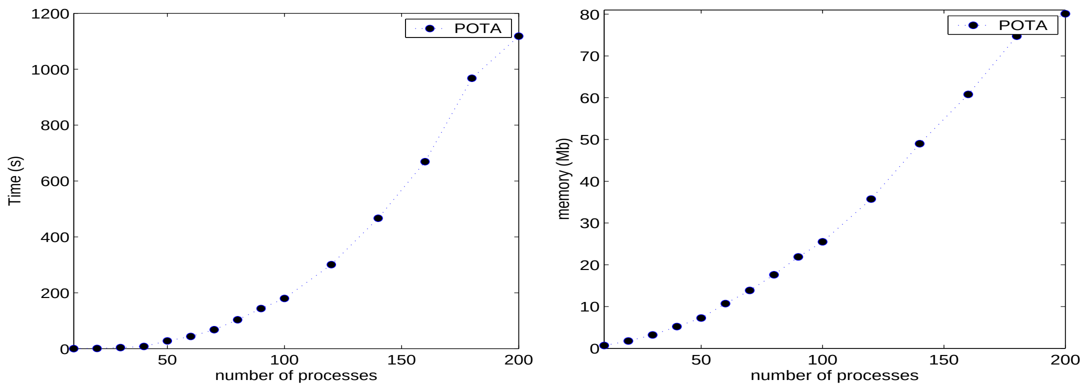
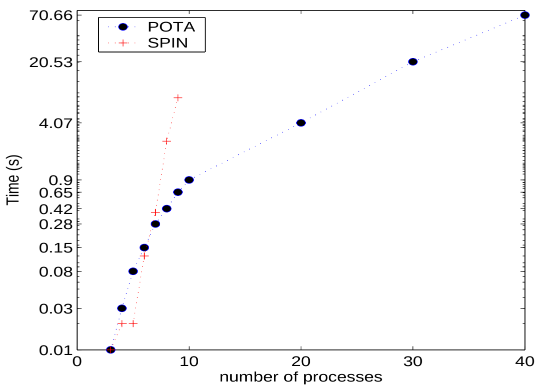


Figure 7.7: Dining philosophers verification results for Property (3)

7.3.3 Primary Secondary

The primary secondary program [SUL00] is an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The details of the protocol can be found at [SUL00]. The invariant of the protocol requires that there is a pair of processes P_i and P_j such that (a) P_i is acting as a primary and correctly thinks that P_j is its secondary, and (b) P_j is acting as a secondary and correctly thinks that P_i is its primary. Both the primary and secondary may choose new processes as their successor at any time.



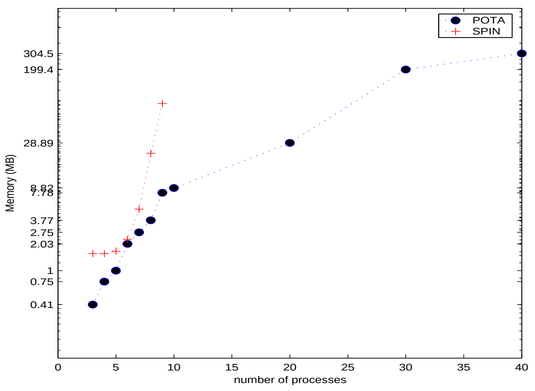


Figure 7.8: Primary Secondary verification results for Property (1)

1. We check the complement of the safety property, which is $\mathsf{EF} \bigwedge_{i,j \in 0\ldots(n-1), i \neq j} (\neg isPrimary_i \vee \neg isSecondary_j \vee (secondary_i \neq P_j) \vee (primary_j \neq P_i))$. Note that this predicate contains disjunction operators and the slice may not be lean. We use logscale for time and memory in Figure 7.8. The figure shows that even for predicates with disjunction operator, slicing can reduce the state space substantially. SPIN ran out of memory for more than 10 processes, whereas POTA ran out of memory for more than 40 processes. Since the slice for the predicate may be approximate, the number of processes verified with POTA is not as high as it was in the previous experiment.

7.3.4 Distributed Mutual Exclusion (Message Passing)

Mutual exclusion is one of the most studied topics in distributed systems. It reveals many important issues such as safety and liveness properties of such systems. We use the Java protocol from [Har98] for this exercise. The protocol is an implementation of Ricart and Agrawal's distributed mutual exclusion algorithm. The algorithm uses timestamps assigned to requests of shared resources to resolve conflict in use of resources. Each process has two local states, namely $tryCS$, which denotes that the process has made a request to access the resource (or the critical section), and $inCS$, which denotes that the process has acquired the resource. Each process maintains the logical time of its request. On receiving any request with a lower timestamp than its own, it replies immediately. Otherwise, it adds that process to the list of processes that will be replied after this process releases the resource. The details of the algorithm can be found in [RA81]. We check the following properties.

1. Two processes cannot access the critical section simultaneously. We can check whether this property is violated by checking its complement, which is $\bigvee_{i,j \in 0\ldots(n-1)} (\mathsf{EF}(inCS_i \wedge inCS_j))$, where i and j denote processes. Figure 7.9 displays our results for this property. In this case the traces did not satisfy the predicate. Hence, the safety property was not violated.

2. Every request for the critical section is eventually granted. We check the complement of the liveness property, which is $\bigvee_{i \in 0\ldots(n-1)} (\mathsf{EF}\,(tryCS_i \wedge \mathsf{EG}(\neg inCS_i)))$, for each process i. Observe that the negation of a local predicate $\neg inCS_i$ is also a local predicate, hence, it is a regular predicate. Figure 7.10 displays our results for this property. SPIN took 55.82 seconds and 393 MB to complete for 5 processes and it ran out of memory for more than 5 processes. Whereas, POTA took 2335 seconds and 272 MB to complete for 100 processes and it ran out of memory for more than 100 processes.

7.3.5 Distributed Mutual Exclusion (Shared Variable)

We use the Java protocol from [Gar04] for this exercise. The protocol is an implementation of Bakery mutual exclusion algorithm. The algorithm uses shared variables. The basic idea is that of a bakery; customers take numbers, and whoever has the lowest number gets service next. Here, of course, “service” means entry to the critical section. We check the same safety property as in the message passing

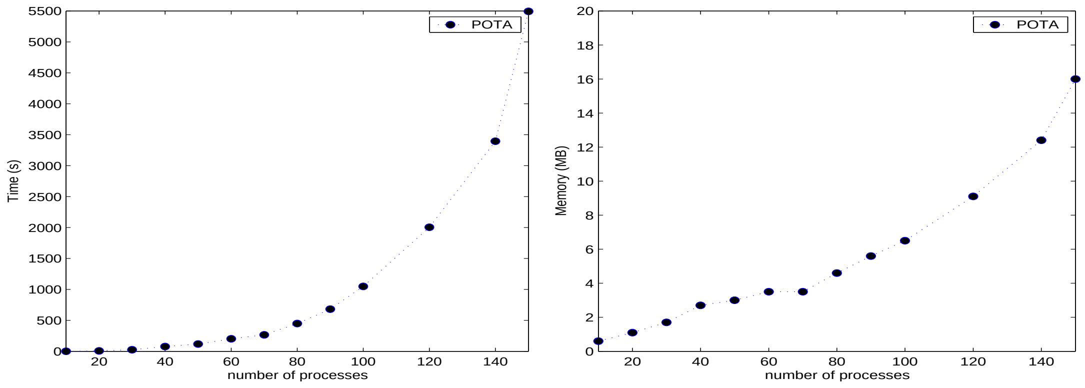


Figure 7.9: Mutual Exclusion (Message Passing) verification results for Property (1)

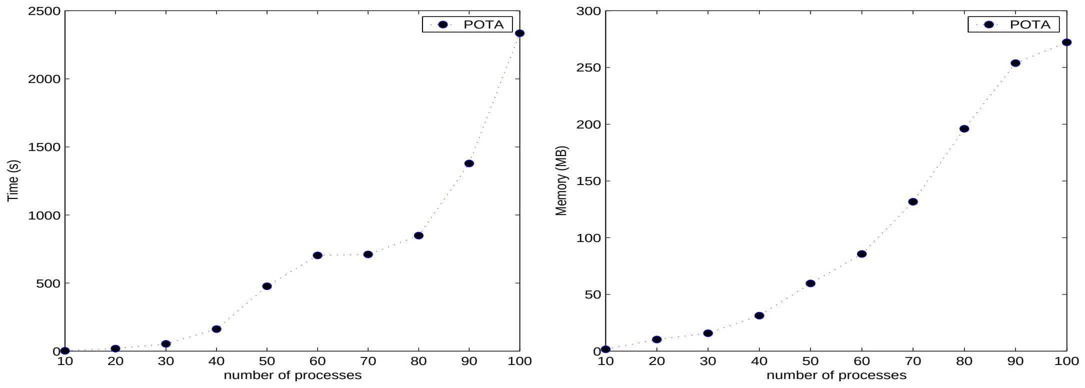
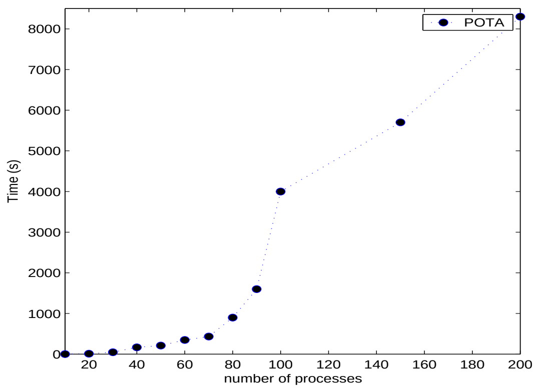


Figure 7.10: Mutual Exclusion (Message Passing) verification results for Property (2)

version.

1. Two processes cannot access the critical section simultaneously. We can check whether this property is violated by checking its complement, which is $\bigvee_{i,j\in 0\ldots(n-1)}(\mathsf{EF}(inCS_i \wedge inCS_j))$, where i and j denote processes. Figure 7.11 displays our results for this property. SPIN took 36 seconds and 375 MB to complete for 11 processes and it ran out of memory for more than 11

processes. Whereas, POTA took 8500 seconds and 30 MB to complete for 200 processes.



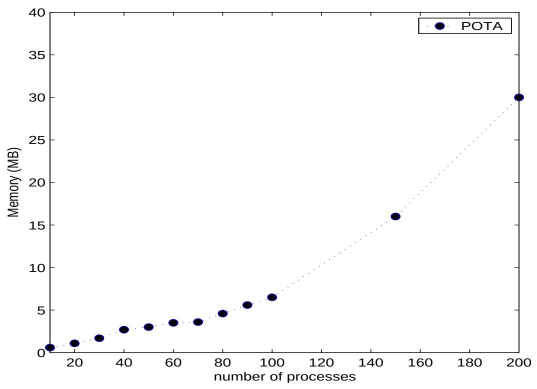


Figure 7.11: Mutual Exclusion (Shared Variable) verification results for Property (1)

7.3.6 Cache Coherence Protocol

The MSI (Modified Shared Invalid) cache coherence protocol is a protocol to maintain data consistency among a number of caches connected to a central directory structure in a multi-processor system. The protocol is a directory based scheme in which individual processes snoop on all other processors' activities over a shared directory. The details of the protocol can be found in [POT03].

1. The property we checked on the MSI protocol is the safety property, "two caches cannot be in the modified state simultaneously". The complement of the property is $\mathsf{EF}(modified_i \wedge modified_j)$, where i and j are cache identifiers. Figure 7.12 displays our results for this property. SPIN took 90 seconds and 468 MB to complete for 10 processes and it ran out of memory for more than 10 processes. Whereas, POTA took 1053 seconds and 44 MB to complete for

120 processes. Due to the overhead associated in generating traces, we stopped generating traces for more than 120 processes.

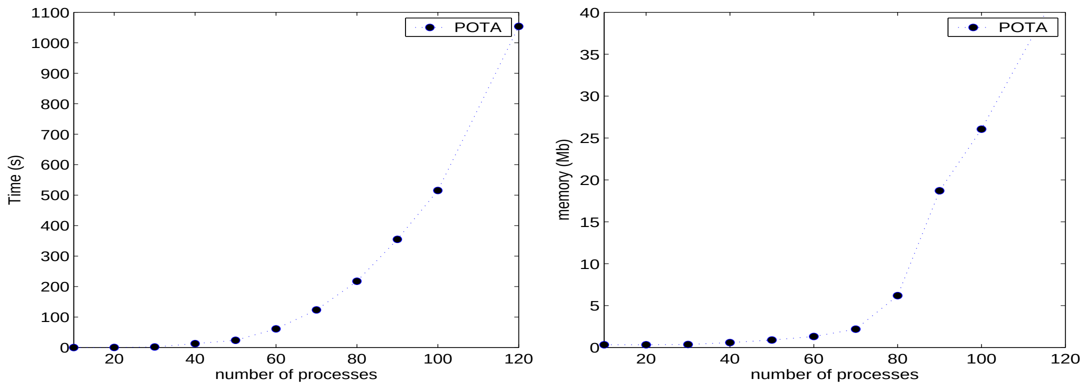


Figure 7.12: MSI verification results for Property (1)

7.3.7 General Inter-ORB Protocol (GIOP)

In this section, we present experimental results for the General Inter-ORB Protocol (GIOP) which was verified for a configuration with a small number of processes in [KL00] using SPIN.

The Common Object Request Broker Architecture (CORBA) [COR97] describes the architecture of a middleware platform that supports the implementation of applications in distributed and heterogeneous environments. The ORB is the key component of the CORBA programming model. An ORB is responsible for transferring operations from Clients to Servers. This requires the ORB to locate a Server implementation (and possibly activate it), transmit the operation and its parameters, and finally return the results back to the Client.

The General Inter-ORB Protocol (GIOP) is the abstract protocol which is used for communications between CORBA ORBs. It specifies the transfer syntax and a standard set of message formats for ORB interoperation over any connectionoriented transport protocol. GIOP is designed to be simple and easy to implement, while still allowing for reasonable scalability and performance. In order to allow server objects to move between different ORBs and have messages forwarded to them wherever they are, GIOP supports server migration.

Figure 7.13 displays the high level view of the Promela model of the GIOP protocol as depicted in [KL00]. The protocol consists of User, Client, Transport, Agent and Server processes.

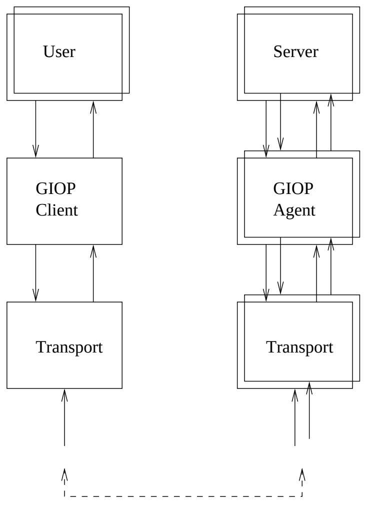


Figure 7.13: GIOP model

1. After sending a *URequest* message a User should eventually receive the corresponding *UReply* message.

   We check the complement of this property stated as:

   $\mathsf{EF}\,(\mathit{URequestSent}_i \wedge \mathsf{EG}(\neg \mathit{UReplyReceived}_i))$, for all users i. Figure 7.14 displays our results for this property. SPIN took 607.32 seconds 376.15 MB to complete for 10 processes and it ran out of memory for more than 10 processes. Whereas, POTA took 1761 seconds and 91 MB to complete for 250 processes. We stopped generating traces at 250 processes since this was the maximum number of processes allowed in SPIN.

2. After sending an *SRequest* the GIOP-Agent should eventually receive a corresponding *SReply*.

   We check the complement of this property stated as:

   $\mathsf{EF}(\mathit{SRequestSent}_i \wedge \mathsf{EG}(\neg \mathit{SReplyReceived}_i))$, for all agents i. Figure 7.15 displays our results for this property. SPIN took 324.71 seconds 305 MB to complete for 10 processes and it ran out of memory for more than 10 processes Whereas, POTA took 1177 seconds and 107 MB to complete for 250 processes.

3. If the user received no exception, its request was performed exactly once.

   We check the complement of the following predicate.

   $\mathsf{AG}(\neg \mathit{NoException}_i \vee (\bigvee_k \bigwedge_j \mathit{Server}_j\mathit{Processed}_i = m))$, where $m = 1$ if $k = j$ and $m = 0$ otherwise, for all users i and for all servers j, k.

   Figure 7.16 displays our results for this property. SPIN took 321.69 seconds 305 MB to complete for 10 processes and it ran out of memory for more than 10 processes. Whereas, POTA took 522 seconds and 369 MB to complete for 120 processes and it ran out of memory for more than 120 processes.

4. If the user received exception, its request was performed at most once.

   We check the complement of the following predicate.

   $\mathsf{AG}(\neg \mathit{SystemException}_i \vee (\bigvee_k \bigwedge_j \mathit{Server}_j\mathit{Processed}_i = m)$
   $\vee(\bigwedge_l \mathit{Server}_l\mathit{Processed}_i = 0))$, where $m = 1$ if $k = j$ and $m = 0$ otherwise, for all users i and for all servers j, k, l. Figure 7.17 displays our results for this property. SPIN took 319.21 seconds 305 MB to complete for 10 processes and it ran out of memory for more than 10 processes. Whereas, POTA took 520 seconds and 475 MB to complete for 120 processes and it ran out of memory for more than 120 processes. Observe the sudden increase in

time from 80 to 90 processes. The complexities of our algorithms in POTA are polynomial in the number of events. Since we generate traces for 20 seconds, the number of events may not be the same for traces of the same protocol with different number of processes. Hence, there may be a sudden increase as seen in the figure.

The full verification of GIOP by Kamel and Leue [KL00] even for the configuration in Figure 7.13 with 10 processes was not completed due to state space explosion. They could verify a simplified version of the protocol without server migration with 10 processes. To enable verification for larger number of processes, they used an approximation technique in SPIN called *bit-state hashing* where two bits of memory are used to store a reachable state. SPIN displays a state coverage number (hash-factor) at the end of a verification with bit-state hashing. With bit-state hashing, they could verify the unsimplified version of the protocol with 20 processes with 1.5 hash-factor, which means that the coverage was less than one percent since best coverage is obtained when the hash-factor is greater than 100.

We generated execution traces for a variety of GIOP architectures where we duplicated the User and Server blocks. In one case, we generated execution traces of the unsimplified version of GIOP protocol where the total number of processes was increased to 250 and we completed full verification of these traces.

7.3.8 Asynchronous Transfer Mode Ring (ATMR)

We present experimental results for the Asynchronous Transfer Mode Ring (ATMR) protocol which was verified for a configuration with a small number of processes in [PTK03] using SPIN.

ATMR protocol [ISO93] is an ISO standard based on a high-speed shared medium connecting a number of access nodes by channels in a ring topology. For controlling access to this type of shared medium, the ring is first initialized with a

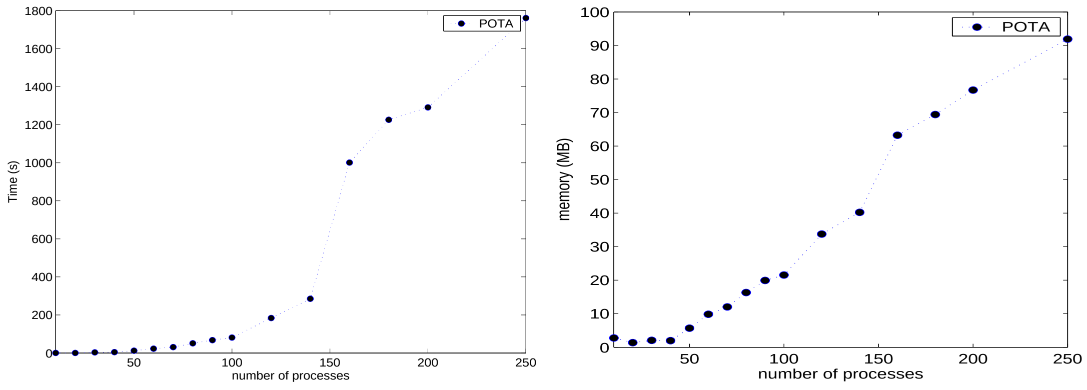


Figure 7.14: GIOP verification results for Property (1)

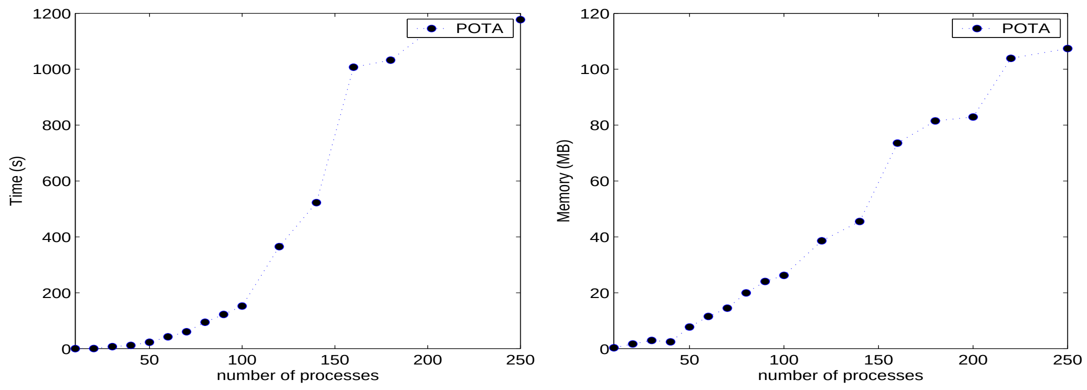


Figure 7.15: GIOP verification results for Property (2)

fixed number of ATM cells continuously circulating around the channel from one node to another. Within each access node there is an access unit which performs both the physical layer convergence function and the access control function. Access to the ring is requested by the client and controlled by a combination of a window mechanism and a reset procedure. The client can issue a sending request to the access unit and receive a data cell. The window mechanism limits the number of cells a node can transmit at a time, called the "credits" of this node. The reset procedure reinitializes the window in all access units to a predefined credit value. Figure 7.18

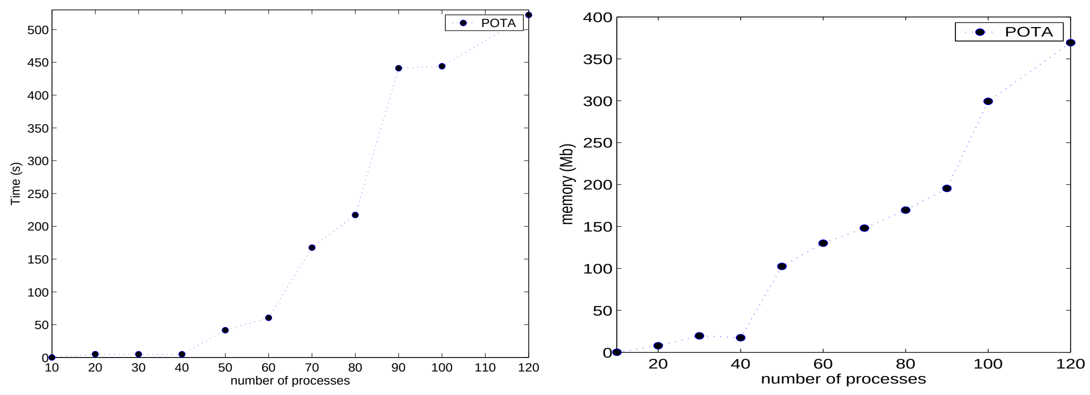


Figure 7.16: GIOP verification results for Property (3)

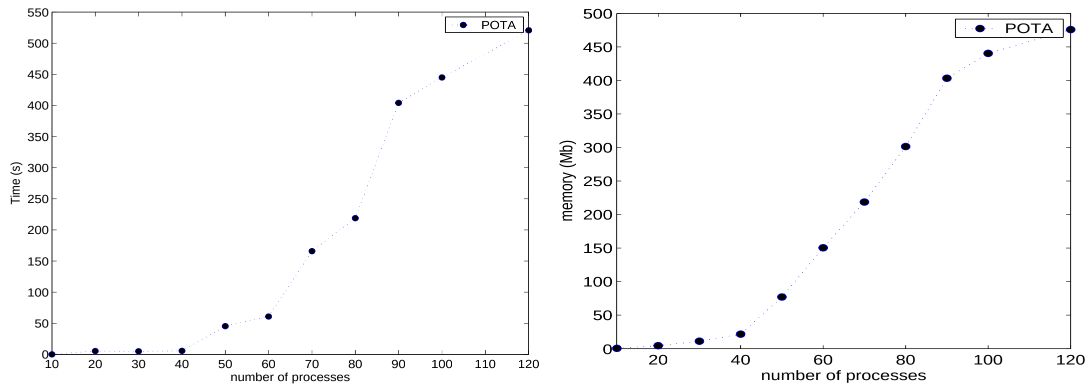


Figure 7.17: GIOP verification results for Property (4)

gives an example ring with five nodes connected via a channel transferring cells between the nodes as depicted in [PTK03].

We conducted experiments for the following predicates used in [PTK03].

1. Once an access unit exhausts its window size credit, the credit will eventually be renewed.

   We check the complement of this property stated as: $\mathsf{EF}\Big((credit_i == 0) \wedge \mathsf{EG}(\neg(credit_i == 6))\Big)$, for all access units i, where *credit* stands for the number of credits which is being held by an access unit and 6 is the preset max-

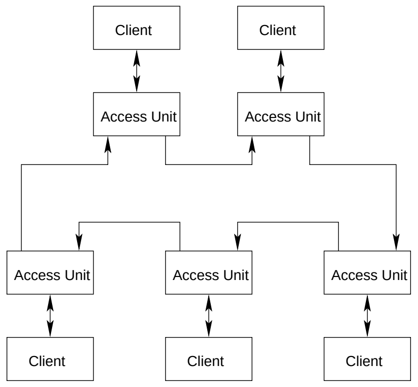


Figure 7.18: ATMR model

imum value. Figure 7.19 displays our results for this property. SPIN ran out of memory for more than 3 processes. Whereas, POTA took 5195 seconds and 460 MB to complete for 250 processes.

2. A client's request will be eventually acknowledged.
   We check the complement of this property stated as: $\mathsf{EF}\Big(req_i \wedge \mathsf{EG}(\neg ack_i)\Big)$, for all clients i, where *req* signal is generated by a cell sending request signal from a client to an access unit. If the requested cell has been sent out, the access unit will return an *ack* signal to the client. Figure 7.20 displays our results for this property. SPIN ran out of memory for more than 3 processes. Whereas, POTA took 3420 seconds and 357 MB to complete for 250 processes.

The full state space verification of ATMR by Peng et al. even for a configuration with 3 nodes was not completed due to state explosion. To enable verification for larger number of processes, they used the bit-state hashing technique which we explained in GIOP experiment. With bit-state hashing, they could verify up to 6 nodes on a 2GB memory machine with less than 98 percent coverage.

We generated execution traces for up to 250 nodes and completed full state

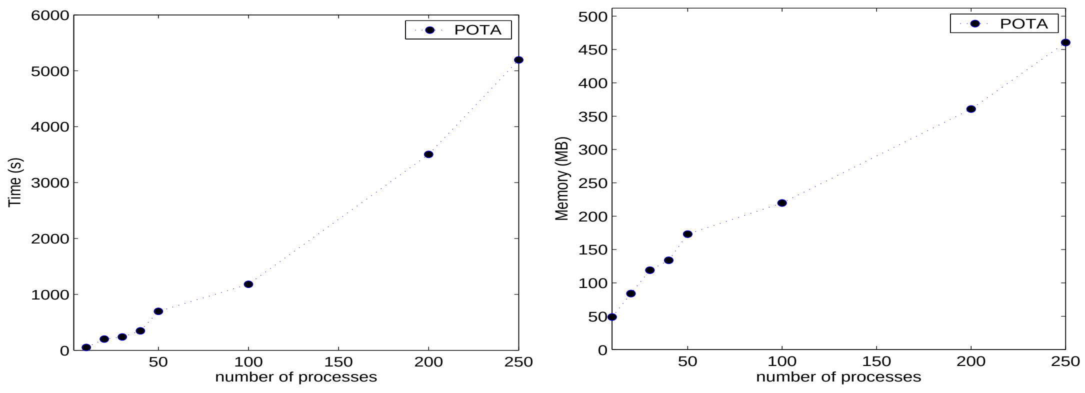


Figure 7.19: ATMR verification results for Property (1)

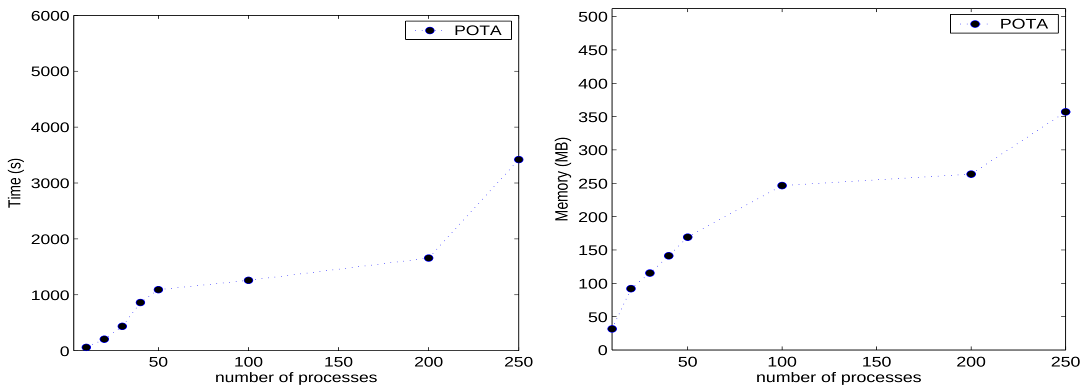


Figure 7.20: ATMR verification results for Property (2)

space verification of these traces. Whereas, SPIN failed to complete full state space verification for more than 3 nodes even when the input to SPIN were traces rather than the protocol.

7.3.9 $\mathsf{AF}(p)$ Experiments

We also performed experiments using algorithm $\mathsf{Algo}_{6.2}$ in Section 6.4 for $\mathsf{AF}(p)$ predicate detection.

We implemented the Chang-Roberts leader election algorithm in Java. In the leader election protocol the processes are arranged in a unidirectional ring. The algorithm ensures that the process with the maximum identifier gets elected as the leader. Every process sends messages only to its left neighbor and receives messages from right neighbors. A process can send *election* message along with its identifier to its left, if it has not seen any message with a higher identifier than its own identifier. It also forwards any message that has an identifier greater than its own; otherwise, it swallows that message. If a process receives its own message, then it declares itself as the leader by sending a *leader* message. The details of the Chang-Roberts algorithm can be found in [CR79].

1. We check $\mathsf{AF}(done_0 \wedge done_1 \wedge \ldots \wedge done_{n-1})$ which denotes that eventually a leader is chosen by every process. Our results are shown in Figure 7.21. The POTA results denote experiments performed by applying all of the conditions in Theorem 6.7, Theorem 6.10, and Theorem 6.16. Observe that our improvement in space and time performance is in the order of magnitude. SPIN took 15.8 seconds and 67.4 MB for 11 processes and it ran out of memory for more than 11 processes. Whereas, POTA took 54 seconds and 181 MB to complete for 17 processes.

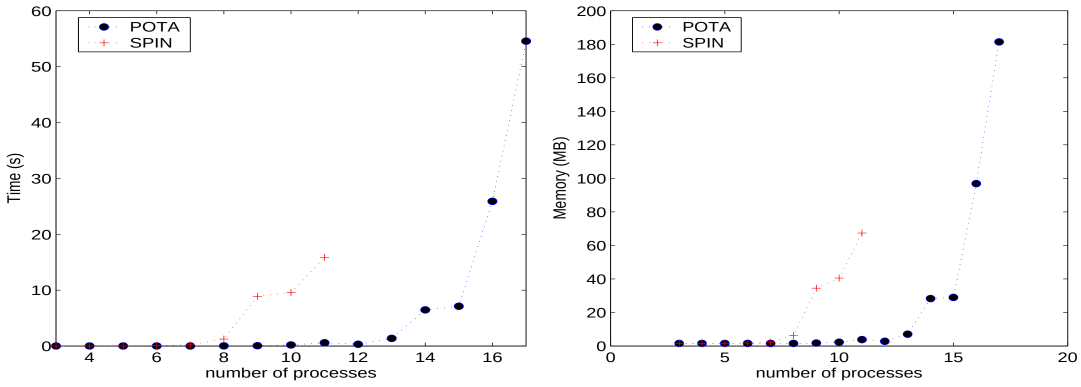


Figure 7.21: Leader Election verification results for Property (1)

7.4 Discussion

We have presented a partial order execution trace analysis tool (POTA) that implements our predicate detection algorithms. For problem sizes that preclude "exhaustive program verification", POTA proves to be an effective tool. Our technique is orthogonal to other reduction techniques, that is, one can use POTA to reduce the state space as long as we can exploit the specification for computation slicing.

We obtain three orders of magnitude speed up and state space reduction compared to partial order reduction with SPIN as shown in experiments. Using our slicing based technique we could verify up to 250 processes in some cases. Some protocols also violated the properties checked. With SPIN, even for bit-state hashing enabled verification, the faults could not be found because the state spaces were too large and the coverage was low. However, with POTA the faults were easily found.

Chapter 8

Related Work

8.1 Computation Slicing

The results in computation slicing presented in this dissertation were first published in [SG03a, SG03c, GMS03, MSGA04]. Computing the slice for an arbitrary predicate is known to be intractable, in general [MG01a]. However, by exploiting the structure of of the predicate, polynomial-time algorithms have been developed for non-temporal regular and linear predicates [GM01, MG01a, MG03]. Using our results, it is now possible to compute the slice efficiently for many more classes of non-temporal and temporal predicates. The non-temporal predicates include stable, co-stable, observer-independent, relational, and co-linear predicates and the temporal predicates include a subset of temporal logic CTL.

The algorithms described in earlier papers [GM01, MG01a] for computing a slice are all *off-line* in nature; they assume that the entire set of events is available *a priori*. While this is quite adequate for applications such as testing and debugging, for other applications such as software fault tolerance, it is desirable that the slice be computed incrementally in an *on-line* manner, that is, as and when a new event is generated, the current slice is updated to reflect its arrival. We developed an efficient algorithm to compute slices on-line.

The notion of a computation slice is similar to the concept of a program slice [Wei82]. Given a program and a slicing criterion, that is, a set of variables, a program slice consists of all statements in the program that may affect the value of the variables in the set at some given point. The criterion in program slicing has also been extended to some predicate classes such as atomic propositions in temporal logic LTL [DH99] and has been implemented in Bandera tool [CDH$^+$00]. Millett and Teitelbaum [MT00] have applied program slicing to the input language of the model checker SPIN [Hol97]. Program slicing has been shown to be useful in program debugging, testing, program understanding, and software maintenance [KR97, Wei82]. A program slice can significantly narrow the size of the program to be analyzed, thereby making the understanding of the behavior easier. We obtain similar benefits from a computation slice for predicate detection and furthermore our approach is orthogonal to program slicing. In other words, after finding a bug using computation slicing, one can use program slicing to reduce the program that needs to be analyzed for locating the bug.

8.2 Predicate Detection

We published our results on predicate detection in [SG02, SG03a, SG03b, SG03c]. Predicate detection in the partial order model is a hard problem. Detecting even a 2-CNF predicate under EF operator has been shown to be NP-complete, in general [MG01b]. Some examples of the predicates for which the predicate detection can be solved efficiently are: conjunctive [GW94, HMRS96], stable [CL85], observer-independent [CBDGF95], linear [CG98], relational [TG97], and non-temporal regular and linear [GM01, MG01a, MG03] predicates.

In [SG02], we presented predicate detection algorithms for predicates of the form $\mathsf{EG}(p)$, $\mathsf{AG}(p)$ when p is linear. Our algorithms improve the complexity results for predicates $\mathsf{EG}(p)$ and $\mathsf{AG}(p)$ in [MG03] when p is non-temporal regular. We also developed detection algorithms for $\mathsf{E}(p \,\mathsf{U}\, q)$ when p is conjunctive and q is linear. Using this algorithm we obtained a simple algorithm to detect $\mathsf{A}(p \,\mathsf{U}\, q)$ when p and q are disjunctive predicates. Also, we are not aware of other efficient algorithms for temporal predicates $\mathsf{E}(p \,\mathsf{U}\, q)$ and $\mathsf{A}(p \,\mathsf{U}\, q)$.

Tarafdar and Garg [TG98b] proved that it is, in general, NP-complete to detect a predicate under EG operator. Since the problem of detecting a predicate under AF operator is the dual of the problem of detecting a predicate under EG operator, it is, in general, coNP-complete to detect a predicate under AF operator. We showed in [SG02] that the problem of detecting $\mathsf{EG}(p)$ when p is observer-independent is NP-complete and detecting $\mathsf{AG}(p)$ when p is observer-independent is co-NP-complete. Therefore, detecting a co-observer-independent predicate under AF operator is coNP-complete, in general.

Fromentin and Raynal [FR94] presented a polynomial time algorithm to solve the predicate detection problem for *proper* operator, which is a special case of AF operator. A computation satisfies $\mathit{proper} : p$ if all paths starting from the initial consistent cut and ending at the final consistent cut go through a *unique* cut that satisfies p.

The $\mathsf{AF}(p)$ detection problem has efficient solutions when the predicate p is disjunctive or conjunctive [GW92, GW96]. However, efficient algorithms are not known for regular p. In [SG03b], we presented efficient conditions to solve the problem for both arbitrary and regular predicates. We validate with experiments that these conditions are effective in reducing the state space.

The efficiency of predicate detection algorithms mentioned above depends on the fact that predicate p is a non-temporal predicate, therefore we could efficiently evaluate the satisfiability of p at a consistent cut (efficient predicate evaluation property). Those algorithms cannot detect predicates that contain nested temporal operators. For example, the predicate $\mathsf{EF}(p \wedge \mathsf{EG}(q))$, where p and q are conjunctive predicates, cannot be efficiently detected using only the algorithms for conjunctive predicates. With the use of our slicing algorithms [SG03a, SG03c], we can detect such nested temporal logic predicates efficiently. In particular we can detect predicates in a subset of CTL called $\mathsf{RCTL}+$.

Our predicate detection and computation slicing algorithms also exploit the distributive lattice property of the set of consistent cuts of a partial order execution trace. We present further applications of lattice theory in distributed computing in [GMS03].

8.3 Predicate Detection Environments

The idea of using temporal logic for analyzing execution traces at runtime (on-line predicate detection, also referred to as *runtime verification*) has recently been attracting a lot of attention. Tools that use temporal logic for checking execution traces are the commercial Temporal Rover tool (TR) [Dru00], the MaC tool [KKL$^+$01], the JPaX tool [HR01], and the JMPaX tool [SRA03]. TR allows the user to specify the temporal formula in programs. These temporal formula are translated into Java code before compilation. The TR, MaC and JPaX tools consider a totally ordered view of an execution trace and therefore can potentially miss bugs that can be deduced from the trace. LTL based verification of execution traces use automata generation [GH01, FS01] or rewriting [HR01]. JMPaX tool is closer to our tool POTA because of the partial order trace model. We work with shared variable and message passing programs, whereas JMPaX considers multithreaded shared variable Java programs only. We have both off-line and on-line algorithms, whereas the algorithms are on-line in JMPaX. JMPaX uses a subset of temporal logic LTL with safety properties where atomic propositions can be arbitrary. Whereas we use a subset of temporal logic CTL with both safety and liveness properties where atomic propositions are restricted. The complexity of the predicate detection algorithm in our approach is polynomial-time for RCTL predicates, whereas the complexity is exponential-time (proportional to the width of the lattice of consistent cuts) in JMPaX.

Since our partial order trace model results in a lattice structure with paths branching from an initial consistent cut (with several possible futures), we decided to use a branching temporal logic RCTL+ (a subset of CTL) to analyze these structures. Also, operators such as *possibly*, *definitely*, *invariant*, and *controllable*, which are traditionally used in predicate detection, can easily be expressed in RCTL+. Note that the complexity of checking CTL is linear in the size of the model (lattice of consistent cuts), which itself is exponential in the size of the trace description. More specifically, for a trace with n processes with at most k events on each process, the complexity of checking a formula p in CTL is $O(|p| \cdot k^n)$ (exponential in the number of processes). Whereas, the complexity of checking RCTL+ is $O(|p| \cdot n^2 k)$, (polynomial in the number of processes) when disjunction, negation and next-time operators are not allowed.

In case a total order trace is used for predicate detection, the linear time temporal logic LTL is a more suitable logic than CTL, since there is only a single possible future. The complexity of checking LTL on a total order trace with k events is $O(k)$ (linear in the number of events), whereas the complexity of checking LTL on a program (all possible traces) is PSPACE-complete (and on a partial order trace is exponential in the number of processes). Markey and Schoebelen [MS03] have an extensive complexity analysis for temporal logic model checking on a total order trace.

Partial order logics view each execution of a program as a set of partial order events as opposed to a total order as in LTL or CTL. We use CTL and reason about the linearizations of the partial order relation, which are paths in the lattice of consistent cuts. The operators of a partial order logic enable reasoning over the sets of partial orders. These logics are undecidable even when using AG, EF, and EX operators [AMP98, AP99]. However, these logics are more expressive in that they permit a direct representation of properties involving causality and concurrency, for example, serializability. There are extensive surveys on complexity analysis of temporal logic model checking in [Eme90, Pen95, Sch03]. Also [EFH$^+$03] contains an analysis of finite trace semantics for temporal logics.

Runtime verification tools are most useful in increasing the understanding of program behavior and finding bugs rather than proving programs correct. Therefore, POTA joins the arsenal of automatic verification tools but does not replace them.

8.4 Model Checking

Model checking [CE81, QS82, CGP00] is an automatic verification technique that takes a model and a temporal logic predicate and automatically checks whether the model satisfies the predicate. The model is in general a description of a program in a mathematical structure such as a Kripke structure or an automata. Some model checking tools are SMV [McM93], SPIN [Hol97], VIS [VIS96], Bandera [CDH$^+$00], and Java PathFinder (JPF) [HP00] to name a few. Our method differs from model checking approach in many aspects. First, model checking algorithms check whether a predicate is satisfied for all executions of a program, whereas we ascertain whether a predicate is satisfied for a *single* partial order execution of a program. This is because our objective is to develop fast algorithms for predicate detection and debugging of programs where a single execution trace of the program is observed. Second, even if model checking algorithms are used on a single finite partial order trace of a program as in our case, the complexity of detecting a temporal predicate would be, in general, proportional to the size of the global state space which is still exponential in the number of processes (state explosion problem). Whereas our algorithms have polynomial-time complexity since we do not generate the state space but work on the trace itself. Finally, we focus on specific predicate classes such as $\mathsf{RCTL}+$, whereas model checking deals with arbitrary predicates. As a result, model checking algorithms, although more general in their applicability, are much more expensive in terms of time and space. Several techniques have been developed to reduce the state explosion problem in model checking. We compare our approach with those techniques below.

In *symbolic model checking* [McM93], sets of consistent cuts are represented implicitly using boolean functions. Manipulating boolean functions can be done efficiently with Reduced Ordered Binary Decision Diagrams (ROBDD or BDD in short), a compact, canonical graph representation of boolean functions. The boolean functions required to represent the set of consistent cuts can grow exponentially, hence the complexity of storing and manipulating BDDs suffers from state explosion problem. A slice is similar to a BDD in that sets of consistent cuts that satisfy a predicate are represented implicitly using slices. However, unlike with BDDs, the size of the slice grows only polynomially with the number of processes.

In *symbolic trajectory evaluation* (STE) [SB95], sets of consistent cuts are represented explicitly using values from a multi-valued domain. Each component of a consistent cut has quaternary values represented by the lattice $L = \{0, 1, X, \top\}$ corresponding to false, true, under-constrained and over-constrained. The values on L are partially ordered $X \leqslant 0$, $X \leqslant 1$, $0 \leqslant \top$, and $1 \leqslant \top$. A trajectory is a totally ordered execution trace. Every element on a trajectory may correspond to a set of consistent cuts. Predicates are of the form $\mathsf{AG}(\mathit{Antecedent} \Rightarrow \mathit{Consequent})$, where antecedent and consequent contains only local predicates, boolean conjunction operator, and temporal next-time operator (negation and disjunction are not allowed).

Our temporal logic is more expressive than the temporal logic used in STE both in terms of the classes of atomic propositions and the temporal operators allowed.

In *bounded model checking* (BMC) [BCCZ99], sets of consistent cuts are represented implicitly using boolean functions expressed with propositional formulas and the algorithms are SAT based. Our approach is similar to BMC in that it cannot prove the absence of errors. However, there are several differences. First, given a finite bound S on the number of steps to reach a counter example (witness for the complement of the property), BMC generates all possible traces starting from an initial consistent cut such that each trace has length S. Whereas POTA considers a single partial order trace starting from an initial consistent cut (containing possibly exponential number of consistent cuts) such that the trace has finite length. Second, BMC algorithms work on the set of consistent cuts which is formulated as a propositional satisfiability problem, an NP-complete problem. Hence the BMC algorithm is inherently exponential, whereas we have polynomial-time algorithms. Third, BMC algorithms handle complements of LTL properties (since a SAT algorithm checks the existence of a solution to a propositional satisfiability problem), whereas we handle $\mathsf{RCTL}+$ properties.

In *partial order reduction* model checking, a subset of the lattice of consistent cuts is generated. This is due to executing only a subset of events enabled from the current consistent cut rather than executing all enabled events. The selection of the subset of events exploits the commutativity of concurrent events, which results in the same consistent cut when the events are executed in different orders. For example, read events of a shared variable by two different processes are commutative. The complexity of partial order reduction may still be exponential. The details of the partial order reduction can be found in [GW91, Val91, Pel93, Esp94, SUL00].

Message Sequence Charts (MSC) [MSC96] are a commonly used visual description of design requirements for concurrent systems such as telecommunications systems. An MSC depicts the desired exchange of messages among communication entries in distributed software systems. Also a shared variable version of MSCs has been developed [AG01]. An MSC corresponds to a single partial order execution of the system as in our case. There are several variants of MSCs expressing complex behavior including hierarchical MSCs. Model checking of MSCs has also been explored. When the specification is given as another MSC or an LTL property, model checking becomes co-NP-complete [AY99].

Chapter 9

Conclusions and Future Work

9.1 Summary

This dissertation has at its goal to develop formal verification procedures for analyzing program execution traces and also to ameliorate the effect of state explosion in analyzing such traces. This analysis is especially important for testing and debugging purposes. Common types of systems that exhibit state explosion include those that have large number of processes. This dissertation is focused on methods of verifying execution traces of such types of systems using temporal logic with a view to reducing state explosion. The contributions of this dissertation are summarized below.

Traditional formal verification techniques such as model checking and theorem proving capture the requirements of a program precisely and unambiguously but they do not scale well and generally work on an abstract model of a system. Traditional testing and simulation techniques scale well and work on the implementations of programs but lack a precise and unambiguous way of specifying the requirements of a system. We develop methods to analyze program traces using formal specifications while avoiding the pitfalls of traditional formal verification and testing.

Our representation of an execution trace by a partial order model gives an important advantage. Only those dependencies between processes that represent synchronization in the program execution are included in the model. Thus, errors can be detected even if they did not actually occur in the particular scheduling or interleaving of the program run.

Our focus in this dissertation has been on developing and unifying detection algorithms for temporal logic predicates on a partial order trace model. We present a subset of temporal logic CTL, called RCTL+, which we interpret on finite traces of programs. Our subset of temporal logic is powerful enough to specify various safety and liveness properties of concurrent and distributed systems as illustrated in our experimental work.

We present polynomial-time detection algorithms for predicates in RCTL+. Our predicate detection algorithms are based on the computation slicing technique. A slice is a concise representation of sets of global states in a trace that satisfy a given predicate. We extend computation slicing, which was earlier introduced for a specific class of non-temporal predicates (atomic propositions) [GM01, MG01a, MG03] to a larger class of non-temporal predicates that includes stable and co-stable predicates, observer-independent predicates, co-linear predicates, and relational predicates. We accomplish this by showing the equivalence of the two problems; computation slicing and predicate detection under EF temporal operator. A predicate of the form EF(p) is satisfied on a trace if there exists a global state that satisfies p. Specifically, given an algorithm to compute the slice for a predicate p, we can determine whether EF(p) is satisfied or not, and vice versa. We also extend computation slicing technique to temporal logic predicates in RCTL+. To that end, we prove that temporal predicates

$\mathsf{EF}(p)$, $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, and $\mathsf{EX}(p)[j]$ are regular, whereas $\mathsf{AF}(p)$, $\mathsf{EX}(p)$, $\mathsf{AX}(p)$, $\mathsf{E}(p\mathsf{U}q)$, and $\mathsf{A}(p \mathsf{U} q)$, in general, are not regular when p and q are regular. We present polynomial-time algorithms to compute slices for $\mathsf{EF}(p)$, $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, and $\mathsf{EX}(p)[j]$.

In this dissertation, we give efficient *on-line* algorithms for computing the slice of a trace, that is, we do not assume that the entire set of events in a trace is available *a priori*. Especially, for software fault-tolerance purposes, it is desirable to compute the slice in an on-line manner. Upon generation of an event in the system, the current slice is updated to accommodate the new event and the resultant slice is checked for an occurrence of a fault.

Another contribution of this dissertation is efficient predicate detection algorithms. We use our computation slicing algorithms to develop detection algorithms for predicates from temporal logic $\mathsf{RCTL}+$. In $\mathsf{RCTL}+$, the temporal operators are EF, EG, AG, $\mathsf{EX}[j]$, EX and the atomic propositions are regular, co-regular, linear, co-linear, stable, co-stable, observer-independent, and relational predicates. For temporal predicates that do not belong to $\mathsf{RCTL}+$ and that do not contain a nesting of temporal operators, we present algorithms that exploit the distributive lattice property of the set of global states. Such type of predicates have widely been studied in the context of distributed computing. Specifically, we provide several conditions for detecting predicates of type $\mathsf{AF}(p)$, when p is a regular predicate. We develop efficient predicate detection algorithms for $\mathsf{EG}(p)$, $\mathsf{AG}(p)$, when p is linear and for $\mathsf{E}(p \mathsf{U} q)$ when p is conjunctive and q is linear. Using the latter algorithm we obtain a simple algorithm to detect $\mathsf{A}(p \mathsf{U} q)$ when p and q are disjunctive predicates. We also show intractability results for detecting observer-independent predicates under EG and AG operators.

We have developed a prototype system Partial Order Trace Analyzer (POTA), which implements our slicing and predicate detection algorithms. We performed several experiments on scalable and industrial protocols including CORBA's GIOP, ISO's ATMR, cache coherence and mutual exclusion. Our experimental results indicate that slicing can lead to exponential reduction over existing techniques both in time and space.

9.2 Future Work

9.2.1 Computation Slicing

As a future work, slicing can be extended to the remaining subset CTL with temporal operators such as AF, AX, EU and AU. Note that, such operators can be obtained directly from the fix-point characterization of CTL operators that uses the EX operator [CGP00]. However, this procedure may not result in polynomial-time complexity.

Since our theory relies heavily on lattice theory, concepts such as composition of partial orders can be exploited. Using the compositionality, we can first compute the slice for smaller traces and then combine these slices in an appropriate way to obtain the slice for the composition. Such concepts will extend the effectiveness and efficiency of slicing in predicate detection.

9.2.2 Predicate Detection

By the nature of working on a single finite trace, our technique is more suitable for finding bugs than proving programs correct. An important direction in extending our work is to develop a better understanding of safety and liveness properties in the context of finite traces. For certain systems such as finite state systems, we can reason about infinite traces by considering only finite traces. For safety properties, if a finite prefix of an execution violates the property then the execution itself violates the property. Our current technique enables us to detect safety violations. For liveness properties, if a finite prefix of an execution that leads to a loop violates the property then the execution itself violates the property. Hence, if we can detect whether a global state exists more than once in a finite trace (a loop), we can detect the violation of a liveness property. Our current technique does not find such loops, hence when we report a violation of a liveness property, the property may indeed hold for the program. It would be useful to develop efficient techniques to find such loops in finite execution traces. Also, liveness properties can be turned into safety properties using translation techniques [BAS02]. In general, liveness properties are specified in the context of fairness properties. Another research topic is to investigate the importance of fairness in the context of a finite trace.

In this dissertation, we give polynomial-time conditions (either necessary or sufficient conditions) to detect a regular predicate under AF operator. It still remains an open problem whether there exists an efficient algorithm for $\mathsf{AF}(p)$ when p is a regular or a 2-CNF predicate.

It is important to develop efficient program instrumentation techniques. These techniques may instrument the program at the source code or object code level. Also, the instrumentation overhead can be substantially reduced if only a small part of the program (for example, one that concerns only the relevant events) is instrumented. We presented several experiments demonstrating the applicability of our techniques on software and hardware programs. However, all these experiments concern asynchronous systems. How can we instrument synchronous hardware programs in order to generate partial order traces?

9.2.3 Extending the Models and Coverage

It is important to develop metrics to determine whether "sufficient" number of traces have been explored or not. We can explore coverage-metrics to increase confidence on verification results. It is also important to develop a *theory of test coverage and generation* based on temporal logics and other formal methods.

Currently, our partial order trace model assumes a total order of events on each process. This allows us to obtain efficient predicate detection algorithms. It is important to study the effects of computation slicing and predicate detection on other models such as potential causality diagrams [TG98a], which allows a partial order of events on each process. It is clear that our slicing algorithms can work on a directed graph, hence is applicable to such causality diagrams. However, the efficiency of the algorithms may decrease in this new model. In particular, even computing the slice for a conjunctive predicate in such a new model becomes intractable because detecting a conjunctive predicate under EF operator in the potential causality model is NP-complete.

Although our partial order trace model captures possibly an exponential number of total order traces, it would still be useful to devise a model that encodes multiple partial order traces rather than a single one. Such a model will have immediate coverage impact. For programs with atomic regions and locks, the notion of a hierarchical (2-level) partially ordered set can be used. A 2-level poset is a poset where each event may correspond to an atomic region with a total order of events such that no interleaving of the events in an atomic region is allowed.

Also, we know that for any partial order trace the set of global states forms a distributive lattice. What additional properties are satisfied when the partial order is given for a synchronous system?

While formal specification languages have so far been mostly investigated for model checking, predicate detection can reveal new logics. It is important to investigate and develop such new logics. Furthermore, temporal logics may not be the best way to write specifications formally. Can we write specifications in an easier way? For example, we can represent specifications as partial order traces similar to Message Sequence Charts [MSC96]. One can then define a notion of refinement among such finite partial order traces. It is important to explore formalisms that go beyond behavioral properties. This includes, but certainly is not limited to performance and security properties, survivability and fault tolerance.

We have demonstrated that integration of formal methods with testing and an implicit representation such as slicing is a successful approach and should further be researched because of the pressing need for formal but still computationally efficient techniques.

Bibliography

[AG01] R. Alur and R. Grosu. Shared Variable Interaction Diagrams. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, San Diego, California, 2001.

[AMP98] R. Alur, K. McMillan, and D. Peled. Deciding Global Partial-Order Properties. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, number 1443 in Lecture Notes in Computer Science, pages 41–52. Springer-Verlag, 1998.

[AP99] R. Alur and D. Peled. Undecidability of Partial Order Logics. *Information Processing Letters*, 69(3), 1999.

[AS85] B. Alpern and F. B. Schneider. Defining Liveness. *Information Processing Letters*, 21:181–185, 1985.

[AV01] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, August 2001.

[AY99] R. Alur and M. Yannakakis. Model Checking of Message Sequence Charts. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR)*, number 1664 in Lecture Notes in Computer Science, pages 114–129. Springer-Verlag, 1999.

[BAS02] A. Biere, C. Artho, and V. Schuppan. Liveness Checking as Safety Checking. In *Proceedings of the 7th International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, volume 66 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2002.

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, number 1579 in Lecture Notes in Computer Science, pages 193–207. Springer-Verlag, 1999.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, Orlando, 1979.

[CBDGF95] 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.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubachand, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.

[CG95] C. Chase and V. K. Garg. Efficient Detection of Restricted Classes of Global Predicates. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[CG98] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CGP00] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.

[CL85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[COR97] Object Management Group (OMG): The Common Object Request Broker: Architecture and Specification, August 1997.

[CR79] E. J. H. Chang and R. Roberts. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM (CACM)*, 22(5):281–283, 1979.

[DH99] M. B. Dwyer and J. Hatcliff. Slicing Software for Model Construction. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 105–118, 1999.

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

[Dru00] D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proceedings* *of the 7th SPIN International Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330, 2000.

[EFH$^+$03] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1990.

[Esp94] J. Esparza. Model Checking Using Net Unfoldings. *Science of Computer Programming*, 23(2):151–195, 1994.

[Fid91] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FR94] E. Fromentin and M. Raynal. Inevitable Global States: A Concept to Detect Unstable Properties of Distributed Computations in an Observer Independent Way. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 242–248, Los Alamitos, CA, USA, 1994.

[FS01] B. Finkbeiner and H. B. Sipma. Checking Finite Traces using Alternating Automata. In *Proceedings of the 1st International Workshop Runtime Verification (RV)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2001.

[Gar96] V. K. Garg. *Principle of Distributed Systems.* Kluwer Academic Publishers, 1996.

[Gar02] V. K. Garg. *Elements of Distributed Computing.* John Wiley & Sons, 2002.

[Gar04] V. K. Garg. *Concurrent and Distributed Computing in Java.* John Wiley & Sons, 2004.

[GH01] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE)*, pages 412– 416, San Diego, California, 2001.

[GJ91] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1991.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[GM01] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.

[GMS03] V. K. Garg, N. Mittal, and A. Sen. Applications of Lattice Theory to Distributed Computing. *ACM Special Interest Group on Algorithms and Computation Theory (SIGACT) News*, 34(3):40–61, 2003.

[GS01] V. K. Garg and C. Skawratananond. String Realizers of Posets with Applications to Distributed Computing. In *Proceedings of* *the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 72–80, Newport, Rhode Island, August 2001.

[GW91] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[GW92] V. K. Garg and B. Waldecker. Detection of Unstable Predicates in Distributed Programs. In *Proceedings of the 12th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 652 of *Lecture Notes in Computer Science*, pages 253–264, New Delhi, India, 1992.

[GW94] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[GW96] V. K. Garg and B. Waldecker. Detection of Strong Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.

[Har98] S. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, 1998.

[HMRS96] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient Distributed Detection of Conjunctions of Local Predicates in Asynchronous Computations. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 588–594, New Orleans, October 1996.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP00] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[HR01] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of the 1st International Workshop Runtime Verification (RV)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2001.

[ISO93] ISO: Specification of the Asynchronous Transfer Mode Ring (ATMR) Protocol, January 1993.

[KKL$^+$01] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Proceedings of the 1st International Workshop Runtime Verification (RV)*, volume 55 of *ENTCS*. Elsevier Science Publishers B. V. (North-Holland), 2001.

[KL00] M. Kamel and S. Leue. Formalization and Validation of the General Inter-ORB Protocol (GIOP) Using Promela and SPIN. *International Journal on Software Tools for Technology Transfer*, 2(4):394–409, April 2000.

[KMM00] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[KR97] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In Mariam Kamkar, editor, *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, Linköping, Sweden, May 1997.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[LL96] S. Leue and P.B. Ladkin. Implementing and Verifying MSC Specifications Using Promela/XSpin. In *Proceedings of the 2nd SPIN International Workshop*, 1996.

[LP85] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–107, New York, January 1985. ACM.

[Mat89] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MG01a] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[MG01b] N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, April 2001.

[MG03] N. Mittal and V. K. Garg. Software Fault Tolerance of Distributed Programs Using Computation Slicing. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 105–113, Providence, Rhode Island, May 2003.

[MS03] N. Markey and Ph. Schnoebelen. Model Checking a Path. In *Proceedings of the 14th International Conference on Concurrency Theory (CONCUR)*, volume 2761 of *Lecture Notes in Computer Science*, pages 251–265, Marseille, France, September 2003. Springer-Verlag.

[MSC96] ITU-T recommendation Z. 120: Message Sequence Chart (MSC), 1996.

[MSGA03] N. Mittal, A. Sen, V. K. Garg, and R. Atreya. Finding Satisfying Global States: All for One and One for All. Technical Report TR-PDS-2003-006, PDSL, ECE Dept. Univ. of Texas at Austin, 2003.

[MSGA04] N. Mittal, A. Sen, V. K. Garg, and R. Atreya. Finding Satisfying Global States: All for One and One for All. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[MT00] L. I. Millett and T. Teitelbaum. Issues in Slicing Promela and its Applications to Model Checking, Protocol Understanding, and Simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, April 2000.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Sci-*

ence, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[Pel93] D. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proceedings of the 5th International Conference on Computer-Aided Verification (CAV)*, pages 409–423, Berlin, Heidelberg, 1993.

[Pen95] W. Penczek. Branching Time and Partial Order in Temporal Logics. In L. Bolc and A. Szalas, editors, *Time and Logic: A Computational Approach*. UCL Press Ltd, London, 1995.

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57. IEEE Computer Society Press, 1977.

[POT03] Partial Order Trace Analyzer (POTA) web site, 2003. http://maple.ece.utexas.edu/˜sen/POTA.html.

[PR94] G. Preusse and F. Ruskey. Generating Linear Extensions Fast. *SIAM J. Comput., 23, 373-386*, 1994.

[PTK03] H. Peng, S. Tahar, and F. Khendek. Comparison of SPIN and VIS for Protocol Verification. *International Journal on Software Tools for Technology Transfer*, 4(2):234–245, 2003.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer-Verlag.

[RA81] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM (CACM)*, 24, 1981.

[SB95] C. H. Seger and R. E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[Sch03] P. Schnoebelen. The Complexity of Temporal Logic Model Checking. *Advances in Modal Logic*, 4:437–459, 2003.

[SG01] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. Technical Report TR-PDS-2001-003, PDSL, ECE Dept. Univ. of Texas at Austin, September 2001.

[SG02] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, April 2002.

[SG03a] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, December 2003.

[SG03b] A. Sen and V. K. Garg. On Checking Whether a Predicate Definitely Holds. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, volume 2931 of *Lecture Notes in Computer Science*, pages 15–29, Montreal, Quebec, July 2003. Springer-Verlag.

[SG03c] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proceedings of the 3rd International Workshop Runtime Verification (RV)*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 2003.

[SL98] S. D. Stoller and Y. A. Liu. Efficient Symbolic Detection of Global Properties in Distributed Systems. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 357–368. Springer-Verlag, 1998.

[SRA03] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, 2003.

[SUL00] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279, July 2000.

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring Functions on Global States of Distributed Programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, March 1997.

[TG98a] A. Tarafdar and V. K. Garg. Addressing False Causality while Detecting Predicates in Distributed Programs. In *Proceedings of the 9th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 94–101, Amsterdam, The Netherlands, May 1998.

[TG98b] A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, 1998.

[TG99] A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[Val91] A. Valmari. A Stubborn Attack On State Explosion. In *Proceedings of the 3rd International Conference on Computer-Aided Verification (CAV)*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Berlin, Germany, 1991.

[VIS96] VIS. A system for Verification and Synthesis. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV)*, number 1102 in Lecture Notes in Computer Science, pages 428–432, New Brunswick, NJ, July 1996.

[Wei82] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.

Vita

Mehmet Alper Sen was born on September 12, 1973 in Diyarbakır, Turkey, the son of Ferhunde and Ömer Şen. After graduating in 1991 from Ankara Science High School, he studied Electrical and Electronics Engineering at the Middle East Technical University at Ankara, graduating with a Bachelor of Science degree in July 1995 and a Master of Science degree in May 1997. Thereafter, he joined the graduate program at the University of Texas at Austin. He has been employed at Sun Microsystems during the summer of 1998 and at Intel Corporation during the summers of 2000, 2001 and the spring of 2004.

Permanent Address: c/o Mrs. Ferhunde Şen
Kartaltepe Mah. Karacaoğlan Sok. Ulukışla Sit. No:6
Bakırköy, 34740 İstanbul, Turkey

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.