The Dissertation Committee for Sujatha Kashyap
certifies that this is the approved version of the following dissertation:

Applications of Lattice Theory to Model Checking

Committee:

Vijay K. Garg, Supervisor

Jacob A. Abraham

Adnan Aziz

Craig M. Chase

E. Allen Emerson

Applications of Lattice Theory to Model Checking

by

Sujatha Kashyap, B. Tech., 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

Aug 2008

To my parents

Acknowledgments

My advisor, Dr. Vijay K. Garg, has been a wise and patient teacher, and an inspiring and unassuming role model. I owe him a tremendous debt of gratitude for teaching me how to be a researcher, for giving me interesting problems to work on and the confidence to explore new areas, and for his enthusiastic guidance throughout my Ph.D. journey.

I'd like to thank Dr. Jacob Abraham, Dr. Adnan Aziz, Dr. Craig Chase and Dr. Allen Emerson for graciously serving on my committee. I have been extremely fortunate to have had access to their insights, and have enormously enjoyed my interactions with them.

I am truly grateful to my employer, IBM Corp., for giving me the flexibility, support and resources required to successfully pursue this Ph.D. There are many people at IBM who put an extraordinary amount of effort into enabling this endeavor for me. I owe special thanks to Ms. Sandra Ellett-Salmoran, Mr. Bill Maron, Mr. John Makis, and Mr. Ray Young, among many others, for enrolling me into IBM's generous Academic Learning Assistance Program. I must thank my management and co-workers for being so accomodating of my schedule over all these years. They showed me that large corporations do have a heart.

My dear friend, Ruslan Bartsits, has been my anchor through this long journey. I am also thankful to my fellow students, Vinit Ogale and Selma Ikiz, for their camaraderie over these years.

And, of course, I must thank my parents for instilling dreams in me and shaping who I am, and my brother for unwittingly inspiring me to pursue a Ph.D. Their unwavering support and belief in me is what kept me going all along.

Sujatha Kashyap

The University of Texas at Austin

Aug 2008

Applications of Lattice Theory to Model Checking

Publication No. ____________________

Sujatha Kashyap, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Vijay K. Garg

Society is increasingly dependent on the correct operation of concurrent and distributed software systems. Examples of such systems include computer networks, operating systems, telephone switches and flight control systems. Model checking is a useful tool for ensuring the correctness of such systems, because it is a fully automatic technique whose use does not require expert knowledge. Additionally, model checking allows for the production of error trails when a violation of a desired property is detected. Error trails are an invaluable debugging aid, because they provide the programmer with the sequence of events that lead to an error.

Model checking typically operates by performing an exhaustive exploration of the state space of the program. Exhaustive state space exploration is not practical for industrial use in the verification of concurrent systems because of the well-known phenomenon of state space explosion caused by the exploration of all possible interleavings of concurrent events. However, the exploration of all possible interleavings is not always necessary for verification.

In this dissertation, we show that results from lattice theory can be applied to ameliorate state space explosion due to concurrency, and to produce short error trails when an error is detected.

We show that many CTL formulae exhibit lattice-theoretic structure that can be exploited to avoid exploring multiple interleavings of a set of concurrent events. We use this structural information to develop efficient model checking techniques for both implicit (partial order) and explicit (interleaving) models of the state space. For formulae that do not exhibit the required structure, we present a technique called predicate filtering, which uses a weaker property with the desired structural characteristics to obtain a reduced state space which can then be exhaustively explored. We also show that lattice theory can be used to obtain a path of shortest length to an error state, thereby producing short error trails that greatly ease the task of debugging.

We provide experimental results from a wide range of examples, showing the effectiveness of our techniques at improving the efficiency of verifying and debugging concurrent and distributed systems. Our implementation is based on the popular model checker SPIN, and we compare our performance against the state-of-the-art state space reduction strategies implemented in SPIN.

Contents

Part I

Introduction and Preliminaries

Chapter 1

Introduction

1.1 Motivation

Commercial software typically has 20 to 30 bugs for every 1,000 lines of code, according to Carnegie Mellon University's CyLab Sustainable Computing Consortium. Another commonly cited metric is that there are 1 to 10 *residual* defects per 1,000 lines of code. Residual defects are those that are found *after* the software has been released. To put this in perspective, consider that according to published sources, Windows Vista has 50,000,000 lines of code.

Testing, by its very nature, is more likely to find errors that occur with high probability, typically in frequently-used code paths. Once these frequently-used code paths have been adequately tested, the rate at which errors are found drops off with time, giving rise to the characteristic "S-curve" of software testing [KPM01] shown in Figure 1.1. Testing is typically halted once the rate at which errors are found falls below a certain threshold.

Tales abound of catastrophic software failures resulting in loss of life or fortune (*cf.* [KT07]). In [Wes89], errors are classified into *levels*, based on the number of independent factors that must occur in combination to cause the error. A level

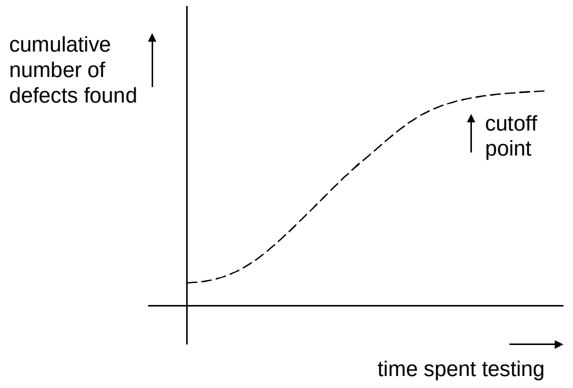


Figure 1.1: The S-curve of software testing.

one error has a single cause, a level two errors occurs when two or more independent causes occur in a certain combination, and so on. Holzmann [Hol01] hypothesizes that the level number of catastrophic errors, such as complete system failure, is usually high and requires multiple things to fail in combination. Thus, high impact failures have a lower probability of occurrence, and are therefore more likely to be missed in testing.

The term **formal verification techniques**, or **formal methods**, encompasses techniques that use mathematically rigorous reasoning to *prove* that an implementation satisfies all or part of its specifications. Formal verification techniques have a better chance of finding catastrophic errors, because they are not sensitive to the *probability* of an error, only its *possibility.* Unlike testing, formal methods can guarantee that a particular behavior never occurs in the system, because they examine *all* possible behaviors of the system.

Concurrent and distributed systems are, in particular, notoriously difficult to test adequately, because they exhibit a very large range of possible behaviors due to the different interactions possible between independently operating agents. Errors in such systems, such as race conditions, tend to have a higher level number, and are therefore prone to escape traditional testing. Consequently, formal methods are particularly useful in the verification of concurrent and distributed systems.

Formal verification techniques are broadly divided into two categories: **theorem proving** and **model checking**. In theorem proving, axioms and inference rules are used to construct proofs showing the correctness of the system. While parts of the theorem proving process are automated, it still usually requires significant human intervention by expert users to guide the proof-finding process. As a result, it is not well-suited for widespread industrial use. An advantage of theorem proving is that it can handle both finite and infinite-state systems.

Model checking [CE82, QS82], by contrast, is a completely automated technique for verifying finite-state systems through an exhaustive exploration of the state space of the system. Also, its use does not require expert knowledge on the part of the programmer. These qualities make it especially suitable for widespread industrial use. Another advantage of model checking is its ability to produce *counterexamples* when an error is detected. A counterexample is an execution path that leads to the error state. Counterexamples are invaluable in locating the source of subtle design errors whose cause would otherwise be difficult to pinpoint. For these reasons, the work in this dissertation focuses on model checking. While the restriction to finite-state systems may seem like a limiting factor, many real-world applications can be modeled as finite-state systems.

The primary obstacle to the widespread use of model checking in software verification is the well-known *state space explosion* problem, where the number of unique system states of most real-world systems is far too large to be exhaustively explored. Concurrency is a significant contributor to state space explosion, due to the large number of unique ways in which concurrent events can be interleaved. For example, the execution of n concurrent events requires the exploration of $n!$ different interleavings of these events in a brute force approach.

Concurrency contributes to state space explosion only if there is a need to explore all possible interleavings of concurrent events. The reason such a need exists is because the property being verified may hold true in one interleaving of concurrent events, but not in a different interleaving of the same events. For example, CTL and LTL formulae *can* distinguish between different interleavings of concurrent events. Therefore, to ensure that the entire state space has been adequately explored, traditional model checking algorithms explore all possible interleavings of concurrent events.

State space explosion due to concurrency can be due to one of two reasons. First, if an *explicit* state space representation is used, the size of the state space itself can be exponential in the number of concurrent events. An explicit state space representation is one in which all the reachable states of the program are explicitly constructed. The use of an *implicit* state space representation, such as partial order models, avoids this cause of state space explosion due to concurrency. However, implicit representations run into the second cause of combinatorial explosion due to concurrency - the verification algorithms can take time and space that is exponential in the size of the representation. For example, in [CG95, Hel99], it was shown that deciding reachability for a boolean formula on a partial order representation is NP-complete in the size of the representation. The problem of model checking a general CTL formula (containing nested temporal modalities) on a partial order state space representation is known to be PSPACE-complete in the size of the representation [Hel00, MMB08].

This dissertation shows how lattice theory [DP90] can be applied to improve the efficiency of verifying and debugging concurrent and distributed systems using model checking.

1.2 Contributions of This Dissertation

As noted above, partial order semantics can be used to obtain a compact representation of the state space. In this dissertation, we present a mechanism to represent a program by a set of finite partial orders, which covers the entire reachable state space of the program. We call this partial order representation of the program a *finite trace cover*. Our method uses Mazurkiewicz trace semantics, which is also the basis of the popular state space reduction mechanism known as partial order reduction (POR) [Val91b, Pel93, GW94b], which is implemented in the widely-used software model checker SPIN [Hol03]. This allows us to provide direct theoretical and experimental comparisons between our method and POR techniques.

While the finite trace cover allows for a compact (implicit) representation of the state space, in order to avoid state space explosion during the verification step, we also need efficient search algorithms that can operate on this representation. Here, we leverage a body of research done in the area of runtime verification, specifically, *predicate detection*. In [CG95], it was shown that for some restricted classes of predicates, reachability can be decided in polynomial time in the size of the partial order representation. We apply these polynomial-time algorithms to decide reachability of such predicates on the finite trace cover representation. This allows us to avoid state space explosion for state space representation *and* state space traversal.

In this dissertation, we focus on predicate classes for which the set of satisfying states exhibit a lattice structure. The search algorithms for these classes use sophisticated results from lattice theory to improve the efficiency of verification.

We explore the complexity of deciding whether a given predicate exhibits any *structure*. The ability to recognize structure in a given formula would make it possible to use the most efficient verification procedure for that formula. In particular, we explore the complexity of deciding whether a given formula belongs to an "efficient" predicate class, that is, a predicate class for which polynomial-time verification algorithms exist for partial order representations.

Computation Tree Logic (CTL) is one of the most popular logics used for specifying the properties to be verified on a program. We explore the structural characteristics exhibited by various CTL temporal and logical operators. We show that it is possible to build a logic containing several CTL temporal and logical operators, such that every formula yielded by the logic exhibits a certain lattice-theoretic structure. This structure can then be exploited to efficiently verify these formulae.

For improving the efficiency of verifying formulae that do not exhibit the required structural characteristics, we present the technique of *predicate filtering*. Predicate filtering uses a *weaker* property that exhibits exploitable structural characteristics, and produces a reduced state space which contains only those states of the original program that satisfy this weaker property. Clearly, the reduced state space also contains all the states that satisfy the original property. This reduced state space can then be explored exhaustively to verify the original property. Typically, only a fraction of the states of a program satisfy the weaker property, so predicate filtering is a useful state space reduction tool. Combinatorial explosion is avoided during generation of the reduced state space by exploiting the structural characteristics of the weaker predicate.

The verification algorithms used on the finite trace cover, and for predicate filtering, are limited to reachability detection for formulae that do not contain nested temporal modalities.

We also use lattice theory to develop model checking algorithms on an interleaving representation of the state space. These algorithms can verify formulae from a subset of CTL, which we call Crucial Event Temporal Logic (CETL). CETL contains the existential until and release operators of CTL, and the conjunction operator. The CETL model checking algorithm *does* encompass formulae containing nested temporal operators. We show that lattice theory can be used to achieve state space reduction and produce short counterexamples while model checking CETL formulae. The production of short counterexamples is of high practical relevance because it contains less extraneous information, making it easier for the programmer to pinpoint the source of the error.

Overall, this dissertation presents techniques that use principles from lattice theory to efficiently parse the concurrency information in a program, with the twin aims of ameliorating state space explosion during program verification and easing the task of debugging concurrent and distributed programs.

1.3 Related Work

1.3.1 Lattice-Theoretic Approaches

Lattice theory has previously been used in the area of runtime verification for the efficient verification of finite execution traces of distributed and concurrent programs. Unlike conventional testing strategies, which view an execution of a program as a single total ordering of events, runtime verification techniques treat the set of events that occur during an execution as a partial order, as proposed by Lamport [Lam78]. Formal methods are then applied to decide whether any interleaving consistent with this partial order can lead to a violation of the specified property. Thus, runtime verification sits at the crossroads between testing and formal verification. It increases the coverage of testing, but does not account for all behaviors of the program. The work presented in this dissertation does.

In [CG95], Chase and Garg introduced the concept of a *meet-closed formula*, and showed how the lattice-theoretic characteristics of such formulae could be exploited for reachability detection. They proposed an algorithm that could decide reachability for a meet-closed formula in a finite (partial order) trace in time that is polynomial in the number of events in the trace. In this dissertation, we apply their techniques to decide reachability in a program.

Computation Tree Logic (CTL) [CE82, EC82] is commonly used to specify the properties to be verified in a program. In [SG03a], Sen and Garg studied the lattice-theoretic characteristics exhibited by some CTL temporal and logical operators. They showed that the CTL operators of EG, EF and AG preserved meet-closure. In [GM01], it was shown that the logical operation of conjunction also preserved meet-closure. In this dissertation, we extend the work in [SG03a, GM01] to study all the temporal and logical CTL operators and provide a complete taxonomy of the CTL logical and temporal operators that do, and do not, preserve meet- and join-closure.

In [SG02], a polynomial-time algorithm was presented for verifying CTL formulae of the form $EG(p)$ or $AG(p)$ on a finite trace, when p is a meet-closed formula containing no temporal operators.

In [GM01], the technique of *computation slicing* was introduced as a way of applying lattice-theoretic principles to achieve state space reduction in runtime verification. Computation slicing was shown [MG01, MG03] to be a useful tool for state space reduction for deciding reachability for formulae that are not otherwise amenable to efficient verification algorithms. In this technique, a trace is *sliced* w.r.t. a given predicate ϕ, yielding a smaller trace (that is, one with fewer reachable states) containing all the states that satisfy ϕ, while eliminating most of the states that do not satisfy ϕ. In this dissertation, we apply the same techniques (which we call *predicate filtering*) for state space reduction in model checking.

The computation slicing algorithms in [MG01, MG03] were limited to formulae containing no nested temporal operators. In [SG03a], Sen and Garg presented computation slicing algorithms for a logic called RCTL (Regular CTL), which consists of the CTL temporal operators EF, EG, AG and the logical operation of conjunction. in [MSGA04], it was shown that a partial order trace can be sliced w.r.t. a predicate ϕ in polynomial time iff reachability for ϕ can be detected in polynomial time.

If ϕ belongs to a special class of predicates called *regular* predicates [GM01], then (and only then) the slice of a trace w.r.t. ϕ contains *exactly* all the states that satisfy ϕ. Thus, a slice for a regular predicate is a compact representation of exactly the set of all ϕ-satisfying states. We say "compact representation" because the slice is a partial order trace, not an explicit-state representation.

In [OG07], a method was presented to derive a compact representation of exactly the set of ϕ-satisfying states of a trace, when ϕ belonged to a logic called BTL, which contains the temporal CTL operator EF and the logical operations of negation, disjunction and conjunction. This compact representation, called a *basis*, has a size that is polynomial in the number of events in the trace. BTL formulae include predicates that are not regular. The algorithm presented in [OG07] to compute the basis of a predicate is polynomial in the number of events in the trace, although it is exponential in the length of the formula.

In the related work discussed so far, lattice-theoretic approaches were only applied to *finite* execution traces of a program. Further, these methods were only applied to a partial order representation of a trace. In this dissertation, we apply lattice-theoretic methods to complete programs, which can consist of both finite and infinite-length traces. Furthermore, we show that lattice theory can be used to improve the efficiency of model checking in both partial order and interleaving representations of the state space.

1.3.2 Concurrency in Interleaving Models

In an interleaving representation of the state space, the lattice-theoretic methods presented in this dissertation combat state space explosion due to concurrency by avoiding the exploration of multiple interleavings of concurrent events. In fact, the techniques we present explore only a *single* interleaving per set of concurrent events. A class of techniques called *partial order reduction* (POR) [Val91b, Pel93, GW94b] also combats state space explosion in an interleaving representation by avoiding the exploration of all interleavings of concurrent events.

POR techniques are altogether distinct from the techniques presented in this dissertation. POR techniques rely on the observation that when the property being verified cannot distinguish between different interleavings of a set of events, it is sufficient to explore a single one of these interleavings. However, when the property being verified does distinguish between two interleavings, both have to be explored. Our approach does not depend on the ability of the property being verified to distinguish between different interleavings. Instead, we use structural information about the property to pick a path that will lead to an error state iff an error state exists. A detailed comparison between our techniques and POR is presented in Chapter 6 (Section 6.7). The main drawback of POR techniques is that the amount of reduction achieved is highly dependent on the property being verified. In our approach, the amount of reduction achieved is not sensitive to the property being verified. Another drawback of POR techniques is that it tends to produce lengthy counterexamples. Our lattice-theoretic approach produces short counterexamples, as is discussed in Chapter 8.

1.3.3 Verifying Partial Order Models

Partial order models for representing the state space, such as Mazurkiewicz traces [Maz89] and Petri net unfoldings [McM92, ERV96], do not directly represent the global states of the system. This information is embedded or encoded in the representation, and must be retrieved by the verification algorithm. For most commonly-used temporal logics, verification algorithms for partial order models take exponential time in the size of the model. In [MMB08], Massart *et al.* show that for partial order traces, CTL model checking is PSPACE-complete, and LTL model checking is co-NP-complete. In [CG95], reachability checking was shown to be NP-complete for partial order traces. Similar results have been shown for Petri net unfoldings - Heljanko showed that CTL model checking is PSPACE-complete [Hel00] and reachability checking is NP-complete [Hel99] in the size of the finite complete prefix of a Petri net unfolding.

In this dissertation, we show that lattice theory can be exploited to derive polynomial-time model checking algorithms for reachability checking of some limited logics, on partial order models. The closest related work is by Esparza [Esp94], who presented an algorithm for reachability checking of properties expressible in a certain limited logic. This algorithm had polynomial running time for the class of 1-safe conflict-free nets. However, Esparza's algorithm applies to low-level Petri nets (Place/Transition nets), which are inherently unscalable. Low-level nets tend to be quite large even for very simple high level programs, making them impractical for use in the verification of real-world programs.

1.4 Organization of this Dissertation

This dissertation consists of the following five parts:

- **Part I: Introduction and Preliminaries**
  In the next chapter (Chapter 2), we present the system model used in this dissertation, including trace semantics. We also present relevant background concepts about partial orders and lattices. In Chapter 3, we discuss property specification logics such as CTL, and identify some predicate classes that exhibit exploitable structure. In particular, we define meet- and join-closed predicates, and discuss how structure is exploited in these predicate classes to improve the efficiency of verification.

- **Part II: Predicate Structure**

  In Chapter 4, we explore the complexity of deciding whether a given formula belongs to an efficient predicate class. In Chapter 5, we explore the lattice-theoretic characteristics exhibited by various CTL temporal and logical operators. In particular, we show that several CTL operators preserve meet- and join-closure, which can be exploited to design efficient verification algorithms.

- **Part III: Partial Order Semantics**

  In this part, we present applications of lattice theory to model checking partial order (implicit) representations of the state space. In Chapter 6, we introduce a partial order state space representation called the *finite trace cover.* We present a mechanism to convert a program into its finite trace cover representation, and show how efficient verification algorithms from the realm of predicate detection can be used to check reachability properties on this representation. In Chapter 7, we present the technique of *predicate filtering*, and show how it can be used to obtain state space reduction for checking reachability for formulae that do not belong to an efficient predicate class.

- **Part IV: Interleaving Semantics**

  In this part, we present applications of lattice theory to an interleaving (explicit) representation of the state space. In Chapter 8, we apply lattice-theoretic techniques to obtain a model checking technique that ameliorates state space explosion due to concurrency, while also producing short counterexamples when the property being verified is violated in the program.

- **Part V: Conclusion**

  We present concluding remarks and directions for future work in Chapter 9.

All chapters with technical content contain a section with bibliographic notes pertinent to the concepts presented in that chapter.

Chapter 2

System Model

2.1 Introduction

In this chapter, we present the system model and notational conventions used in this dissertation. We also present some background information, such as relevant concepts from the theory of partial orders and from lattice theory. Most of the notation used here is standard. A summary of the notation introduced is presented at the end of this chapter, in Table 2.1.

2.2 Programs

A **finite-state program** P is a triple (S, T, s_0), where:

- S is a finite set of states,
- T is a finite set of transitions,
- $s_0 \in S$ is the initial state of the program.

In real implementations, a program contains a countable set of **variables** of two kinds - **data variables** and **control variables**. A **data variable** can take on any value from a *data domain*, which is typically specified by the programming language. Examples of data variables include integers, pointers, lists and arrays. Message channels in concurrent and distributed programs are also considered data variables. A **control variable** assumes values corresponding to locations in the program. The program counter is an example of a control variable.

A **state** of a program is fully characterized by giving values to all of its (data and control) variables. The set of transitions that are executable from a given state $s \in S$ is denoted by $\mathit{enabled}(s)$. A transition $\alpha \in \mathit{enabled}(s)$ transforms the state s into a *unique* state s', denoted by $s' = \alpha(s)$.

A state s is said to be **reachable** in a program P iff it can be reached from s_0 by executing only enabled transitions at each state. The **full state space graph** of P is a directed, edge-labeled graph $\langle \mathcal{V}, \mathcal{E} \rangle$, such that:

- $\mathcal{V} \subseteq S$ is the *minimal* set of states of P satisfying:
  - $s_0 \in \mathcal{V}$, and
  - if $s \in \mathcal{V}$, $\alpha \in \mathit{enabled}(s)$, and $t = \alpha(s)$, then $t \in \mathcal{V}$.
- $\mathcal{E} = \{(s,t) | \exists \alpha \in \mathit{enabled}(s) : t = \alpha(s)\}$. In this case, the edge (s,t) is labeled with α.

In simple terms, the vertex set of the full state space graph of P is exactly the set of reachable states of P. An edge exists from vertex s to t iff $\exists \alpha \in \mathit{enabled}(s)$ such that $t = \alpha(s)$.

A **path** through the full state space graph consists of a (finite or infinite) sequence of states. Each path has a corresponding **transition sequence**, consisting of the edge labels along the path. Each occurrence of a transition in a transition sequence is called an *event*. For example, the transition sequence $\alpha\beta\alpha\beta$ consists of four events.

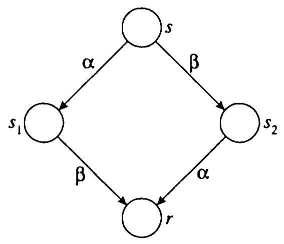


Figure 2.1: $(\alpha, \beta) \in I$.

Definition 2.1. *[Maz89, Pel94] An* **independence relation** $I \subseteq T \times T$ *is an irreflexive, symmetric relation such that* $(\alpha, \beta) \in I$ *iff* $\forall s \in S$*:*

- **Enabledness:** *If* $\alpha \in enabled(s)$*, then* $\beta \in enabled(s)$ *if and only if* $\beta \in enabled(\alpha(s))$*, and*
- **Commutativity:** *If* $\alpha, \beta \in enabled(s)$*, then* $(\alpha(\beta(s)) = \beta(\alpha(s)))$*.*

The enabledness condition states that the execution of α from any state does not affect the enabledness of β, and the commutativity condition states that executing α and β in either order results in the same state. Figure 2.1 illustrates these conditions.

The **dependency relation** D is the reflexive, symmetric relation given by $D = (T \times T) \setminus I$. We say that two events are dependent (correspondingly, independent) iff their corresponding transitions are dependent (independent).

2.3 Traces

Mazurkiewicz [Maz89] defined an equivalence relation on finite transition sequences, called **trace equivalence** and denoted by $\equiv$. The equivalence relation $\equiv$ is the *smallest* transitive relation that satisfies the following conditions, for all $u, v, w \in T^*$:

1. $v \equiv v$.

2. If $v = u_1\alpha\beta u_2$ and $w = u_1\beta\alpha u_2$ for some $u_1, u_2 \in T^*$ and $\alpha, \beta \in T$, such that $(\alpha, \beta) \in I$, then $v \equiv w$.

Informally, $v \equiv w$ iff v can be transformed into w by repeatedly commuting adjacent independent operations.

Example 2.1. *Let $\alpha_1\alpha_2\alpha_3\beta_1\beta_2$ be a transition sequence. Let the independence relation be given by $I = \{(\alpha_i, \beta_j)\}$. Then, $\alpha_1\alpha_2\alpha_3\beta_1\beta_2 \equiv \alpha_1\beta_1\alpha_2\beta_2\alpha_3$, from the following sequence of commuting independent events:*

$$\begin{array}{cll} & \alpha_1\alpha_2\alpha_3\beta_1\beta_2 & \\ \equiv & \alpha_1\alpha_2\beta_1\alpha_3\beta_2 & \{\textit{Commuting } \alpha_3 \textit{ and } \beta_1\} \\ \equiv & \alpha_1\alpha_2\beta_1\beta_2\alpha_3 & \{\textit{Commuting } \alpha_3 \textit{ and } \beta_2\} \\ \equiv & \alpha_1\beta_1\alpha_2\beta_2\alpha_3 & \{\textit{Commuting } \alpha_2 \textit{ and } \beta_1\} \end{array}$$

Trace equivalence for infinite transition sequences was first defined in [Kwi89], with the help of the relation $\preceq$. Let u, v be two finite or infinite transition sequences. That is, $u, v \in T^* \cup T^\omega$. We say that $u \preceq v$ iff for each finite prefix u' of u, there exists a prefix v' of v, and some w such that $v' \equiv w$, and u' is a prefix of w. We now extend the definition of $\equiv$ to infinite transition sequences as follows. For any $u, v \in T^* \cup T^\omega$, $u \equiv v$ if and only if $u \preceq v$ and $v \preceq u$.

Example 2.2. *Let $I = \{(b, c)\}$. We can show that $b(abc)^\omega \preceq (bac)^\omega$. Consider any prefix of $b(abc)^\omega$, such as $u' = babca$. The sequence $v' = bacbac$ is a prefix of $(bac)^\omega$. Now, $bacbac \equiv babcac$, since $(b, c) \in I$. Let $w = babcac$. Now, u' is a prefix of w. Similarly, we can show that $(bac)^\omega \preceq b(abc)^\omega$. For example, consider the prefix $u' = bacba$ of $(bac)^\omega$. We can pick the prefix $v' = babca$ of $b(abc)^\omega$. Now, v' can be transformed into $w = bacba$ by commuting the independent transitions b and c, an u' is a prefix of w. Thus, $b(abc)^\omega \preceq (bac)^\omega$ and $(bac)^\omega \preceq b(abc)^\omega$. Which means $b(abc)^\omega \equiv (bac)^\omega$.*

The equivalence relation $\equiv$ partitions the set of all transition sequences (correspondingly, paths) of a program P into equivalence classes called **traces** [Maz89].

Definition 2.2. *A* **trace** *is an equivalence class induced by the relation $\equiv$ over the set of all transition sequences of a program. We use the notation $\sigma = [s, v]$ to denote a trace σ with starting state s, and a representative transition sequence v. Clearly, $\sigma = \{u | u \equiv v\}$.*

Every transition sequence in a trace consists of the same set of events. By the commutativity property of independent events, it is easily shown [Pel94] that the same final state is reached upon executing any transition sequence of a trace. That is, each trace has a unique final state. We define the following operations on traces.

- The **concatenation** of a finite trace $\sigma_1 = [s, v]$ with a finite or infinite trace $\sigma_2 = [t, w]$ is defined when t is also the final state of σ_1, and is given by $\sigma_1.\sigma_2 = [s, v.w]$.
- We say that $\sigma_2 = [s, v]$ **subsumes** $\sigma_1 = [s, u]$, denoted $\sigma_1 \sqsubseteq \sigma_2$, iff $u \preceq v$. If σ_1 is finite, then $\sigma_1 \sqsubseteq \sigma_3$ iff there exists σ_2 such that $\sigma_3 = \sigma_1.\sigma_2$.

We say that a trace of a program is **maximal** iff no other trace subsumes it. In the following section, we discuss a correspondence between traces and partially ordered sets.

2.3.1 Traces and Posets

Definition 2.3. *A* **partially ordered set** *or* **poset** *is a set X together with a reflexive, antisymmetric and transitive binary relation $\leq$ on the elements of X. We use the notation $(X, \leq)$ to denote such a poset.*

If every pair of elements in X is comparable under the binary relation $\leq$, then $(X, \leq)$ is called a **totally-ordered** set. The notation $x < y$ is used when $x \leq y$ and

$x \neq y$. Finite posets can be represented diagrammatically using a directed acyclic graph called a **Hasse diagram** [DP90]. In the Hasse diagram representation of a finite poset $(X, \leq)$, the vertices are the elements of X, and an edge exists from a vertex $x \in X$ to a vertex $y \in X$ iff:

- $x < y$, and
- there is no $z \in X$ such that $x < z < y$ (*i.e.*, there are no in-between elements).

Further, when the Hasse diagram is drawn on the Euclidean plane, the vertex for x is drawn at a lower y-coordinate than the vertex for y if $x < y$.

Example 2.3. *Let $X = \{a, b, c, d\}$, in which $a < c$, $a < d$, $b < c$ and $b < d$. No other pairs of distinct elements are comparable. A Hasse diagram for $(X, \leq)$ is shown in Figure 2.2.*

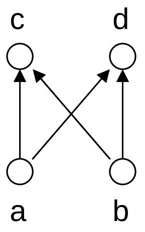


Figure 2.2: A Hasse diagram representing the poset in Example 2.3.

It is not possible to represent the whole of an infinite poset by a diagram, but if its structure is sufficiently regular, it can be suggested diagrammatically, as shown in Figure 2.3. In [AG07], Agarwal and Garg introduced the notion of *p-diagrams* to diagrammatically represent a class of infinite posets.

It is well-known [Win87, Maz89, Pra86] that a 1-1 correspondence exists between traces and posets. Let $\sigma = [s, v]$ be a trace, and E be the set of events in v. We can define a poset $(E, \rightarrow)$, where $\forall e, f \in E : e \rightarrow f$ iff $(e, f) \in D$ and either e occurs before f in v, or $e = f$. The relation $\rightarrow$ is the same as Lamport's

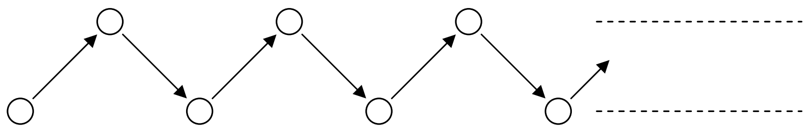


Figure 2.3: Diagram of an infinite poset with repeatable structure.

"happened-before" relation [Lam78], and expresses causal dependence. For instance, if an event e denotes the sending of a message, and f the corresponding receive event, then $e \to f$.

Definition 2.4. *A* **linear extension** *of a poset* $(X, \leq)$ *is any totally-ordered set* $(X, \leq_1)$ *such that for every* $x, y \in X$*, if* $x \leq y$ *then* $x \leq_1 y$.

A linear extension of a poset $(X, \leq)$ is often represented by a string (sequence) consisting of the elements of X, where an element x appears before y in the sequence if $x < y$. For example, the string *badc* is a linear extension of the poset in Figure 2.2.

Every transition sequence of σ is a linear extension of $(E, \to)$, and conversely every linear extension of this poset is a valid transition sequence of σ. We will use the notation $\sigma = (E, \to)$ to represent the poset corresponding to a trace σ.

The same state can be visited multiple times during the execution of a transition sequence, for example, in the case of a cycle in the state space graph. However, each *occurrence* of the state corresponds to a unique prefix of the transition sequence. If an event e is executed as part of a transition sequence, then the events that causally precede e must have been executed before e.

Definition 2.5. *A* **down-set** *of a poset* $(X, \leq)$ *is any subset* $Y \subseteq X$ *such that whenever* $y \in Y$*,* $x \in X$ *and* $x \leq y$*, we have* $x \in Y$.

Example 2.4. *The down-sets of the poset in Figure 2.2 are:* $\emptyset$*,* $\{a\}$*,* $\{b\}$*,* $\{a, b\}$*,* $\{a, b, c\}$*,* $\{a, b, d\}$ *and* $\{a, b, c, d\}$.

In a trace $\sigma = (E, \rightarrow)$, there exists a correspondence between occurrences of states, and down-sets. Each occurrence of a state in σ corresponds to the execution of the set of events from some down-set of $(E, \rightarrow)$. Conversely, every state in σ can be reached by executing the events in some down-set of $(E, \rightarrow)$. For simplicity of presentation, in this dissertation we overload the term "down-set" to mean both a set of events, and an occurrence of a state.

Progress in a computation is measured by the execution of additional events from the current state. For down-sets G and H of a trace $(E, \rightarrow)$, $G \subseteq H$ iff H is reachable from G in the full state space graph. In the following section, we discuss the relationship between down-sets of a trace and lattices.

2.3.2 Traces and Lattices

Let $(X, \leq)$ be a poset, and $S \subseteq X$. An element $x \in X$ is an **upper bound** of S if for each $s \in S$, $s \leq x$. Dually, an element $y \in X$ is a **lower bound** of S if for each $s \in S$, $y \leq s$. An element $x \in X$ is the **least upper bound** of S if:

- x is an upper bound of S, and
- $x \leq x'$ for all upper bounds x' of S.

The least upper bound is also called the **supremum** or **join**. Dually, an element $y \in X$ is the **greatest lower bound** of S if:

- y is a lower bound of S, and
- $y' \leq y$ for all lower bounds y' of S.

The greatest lower bound is also called the **infimum** or **meet**.

Definition 2.6. *A poset $(X, \leq)$ is called a* **lattice** *if every pair of elements in X has a meet and a join that are also contained in X.*

The poset in Figure 2.2 is not a lattice, for the following reasons:

- c and d have no common upper bound,
- a and b have no common lower bound,
- a and b have no *least* upper bound, although c and d are both upper bounds,
- c and d have no *greatest* lower bound, although a and b are both lower bounds.

The poset in Figure 2.4 is a lattice.

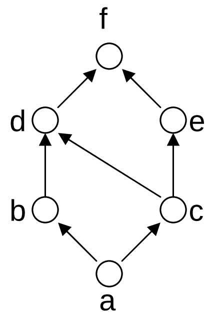


Figure 2.4: A lattice.

Let $\mathcal{P} = (X, \leq)$ be a poset, and $\mathcal{O}(\mathcal{P})$ be the set of all down-sets of $\mathcal{P}$. A well-known result in lattice theory [DP90] states that $(\mathcal{O}(\mathcal{P}), \subseteq)$ is a lattice. That is, the set of all down-sets of a poset forms a lattice under the subset relation. In particular, such a lattice is called a **down-set lattice**, and is actually a special kind of lattice, called a **distributive lattice**. Distributive lattices will be defined and further explored in Chapter 7. The meet and join operations on a down-set lattice are given by set intersection and set union, respectively. Figure 2.5 shows a poset and its corresponding down-set lattice.

Given a trace $\sigma = (E, \rightarrow)$, we use the notation $\mathcal{L}(\sigma)$ to denote its down-set lattice. It follows that, if G and H are down-sets of $(E, \rightarrow)$, then so are $G \cap H$ and $G \cup H$. Recall that down-sets of σ correspond to occurrences of states along some

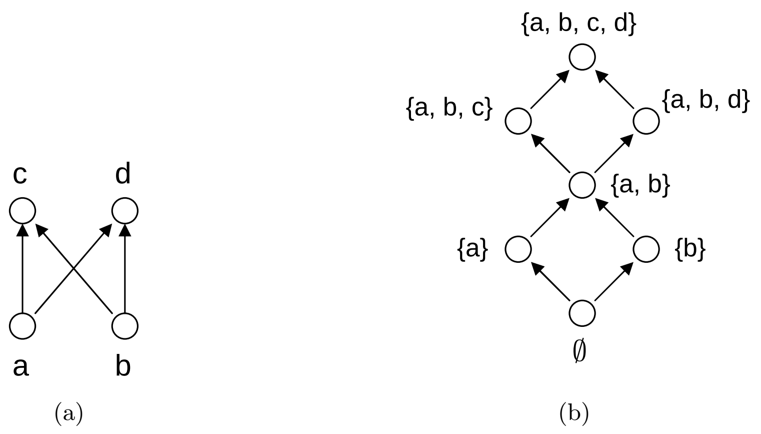
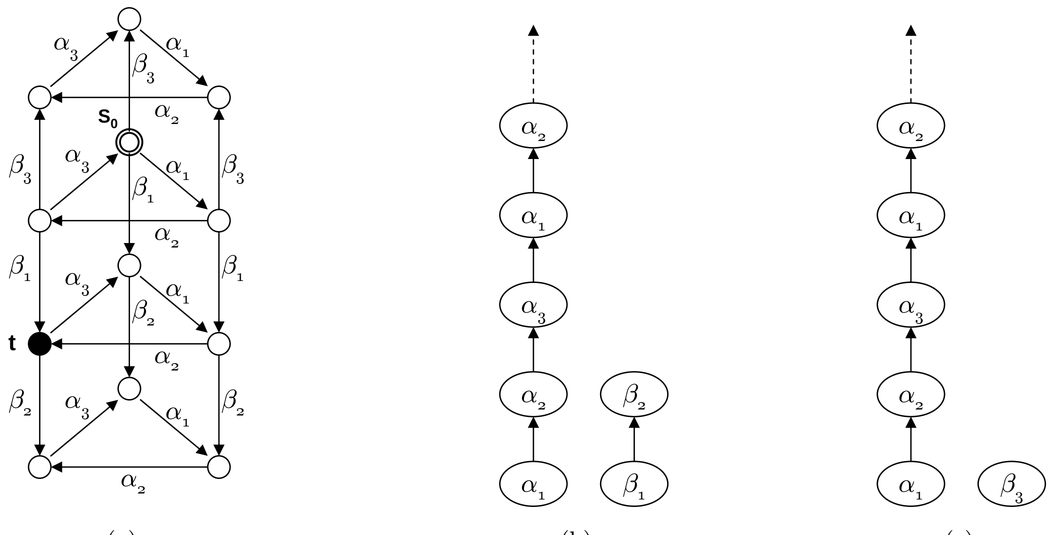


Figure 2.5: (a) A poset, and (b) its down-set lattice.

path in the full state space graph. This view of occurrences of states as elements of a lattice was previously explored in [Win87, Mat89, GM01], among others. Figure 2.6 shows an example that illustrates the relationship between the full state space graph, maximal program traces, and down-set lattices.

2.4 Processes

This dissertation focuses on the verification of concurrent and distributed programs, where the system is modeled as a set of processes, $\{P_1, ..., P_n\}$. Each process P_i has a finite set of transitions T_i, and a set of *local* variables V_i. The value of a local variable in V_i can only be changed by transitions in T_i. Additionally, a transition in T_i may also change the values of shared (global) variables in the program. Processes communicate with each other through shared variables, or by synchronous or asynchronous message passing. Synchronous message passing can be achieved through a handshake mechanism. Asynchronous message passing uses message channels or queues. A message channel can be shared among multiple processes, or can be point-to-point with a single sender and a single receiver.



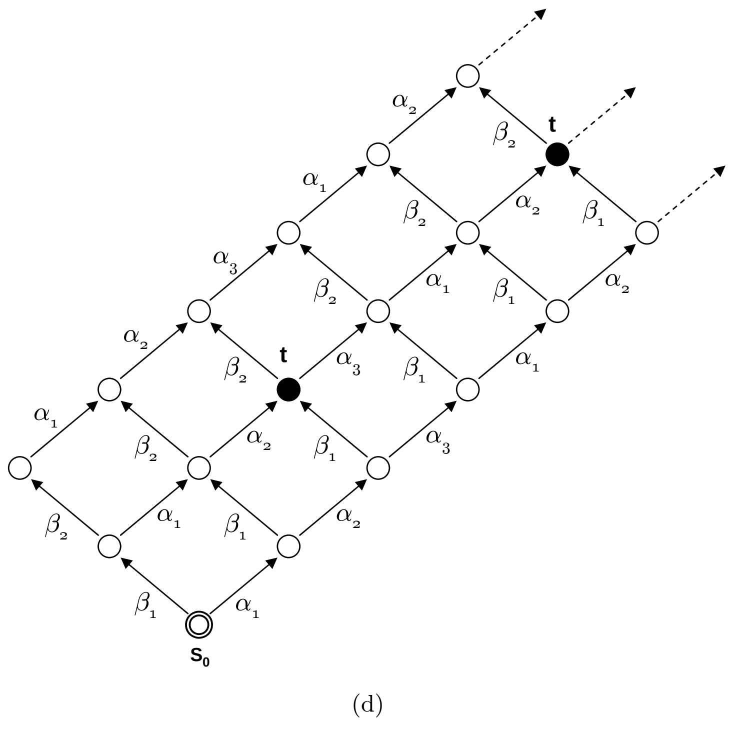


Figure 2.6: (a) The full state space graph of a program P. The independence relation $I = \{(\alpha_i, \beta_j) | 1 \leq i \leq 3, 1 \leq j \leq 3\}$. P has two maximal traces: (b) $\sigma_1 = [s_0, \beta_1\beta_2(\alpha_1\alpha_2\alpha_3)^\omega]$, and (c) $\sigma_2 = [s_0, \beta_3(\alpha_1\alpha_2\alpha_3)^\omega]$. (d) The down-set lattice $\mathcal{L}(\sigma_1)$. The dark circles show two separate occurrences of the state t, corresponding to distinct down-sets of $L(\sigma_1)$ but the same state in the full state space graph.

2.4.1 The Dependency Relation

In such a model of computation, dependency between transitions may arise either because of control flow (*e.g.*, updating the program counter for a process), or because of data (accessing common variables or message queues). In [KP92, God96], it was shown that a sufficient syntactic condition for two transitions α, β to be independent is that:

- α and β belong to different processes, and
- the set of objects (variables, message queues) accessed by α is disjoint from the set of objects accessed by β.

For our model of computation, we use the following dependency relation:

- Pairs of transitions that belong to the same process are dependent. That is, if $\alpha, \beta \in T_i$, then $(\alpha, \beta) \in D$.
- Pairs of transitions that access the same variable, which is changed by at least one of them, are dependent.
- Two send transitions that use the same message queue are dependent. This is because executing one may cause the message queue to fill, disabling the other. Also, the contents of the queue depends on their order of execution.
- Similarly, two receive transitions that use the same message queue are dependent.
- A send and receive transition on the same message queue are dependent. This is because execution of the send could enable the receive.

Table 2.1 summarizes the notations introduced in this chapter.

Notation	Description
P	A program.
S	The finite set of states of P.
T	The finite set of transitions of P.
$enabled(s)$	The set of all transitions that are executable from the state s.
$\alpha(s)$	The state reached by executing the transition α from the state s.
$(\alpha, \beta) \in I$	α and β are independent transitions.
$(\alpha, \beta) \in D$	α and β are dependent transitions.
$u \equiv v$	Transition sequences u and v belong to the same trace.
σ	A trace.
$\sigma = [s, v]$	A trace σ with starting state s, and representative transition sequence v.
$\sigma = (E, \rightarrow)$	The poset corresponding to a trace σ that has E as its set of events.
$\sigma_1.\sigma_2$	The concatenation of traces σ_1 and σ_2.
$\sigma_1 \sqsubseteq \sigma_2$	σ_2 subsumes σ_1.
$\mathcal{L}(\sigma)$	The down-set lattice of the trace σ.
P_i	A process.
V_i	The local variables of process P_i.
T_i	The set of transitions of process P_i.

Table 2.1: A summary of notation.

2.5 Bibliographic Notes

The definition of a program used here is the same as that used by Peled in his work on partial order reduction techniques, for example, [Pel93, Pel94]. Mazurkiewicz first proposed the theory of traces in 1977 [Maz77] as a tool for analyzing the behavior of Petri nets, and subsequently refined it in [Maz84, Maz85, Maz87, Maz89]. The terminology and notation for traces and the dependence and independence relation used in this dissertation is the same as that used in [Maz89], and was also used in [Pel93, Pel94].

The equivalence between traces and partial orders has been noted by various researchers, including [Lam78, Maz84, Win87, Pra86]. The correspondence between traces and lattices was also observed by Winskel in [Win87], and by Mattern in

[Mat89]. The notation ($\rightarrow$) we use for the partial order relation between events of a trace is based on Lamport's "happened-before" relation [Lam78]. The notation used for concepts from the theory of partial orders and lattice theory is standard, and follows the conventions in [DP90].

The syntactic conditions for deciding whether a pair of transitions is dependent is based on the chapter on partial order reductions in [CGP99]. These conditions are also implemented in the SPIN model checker [Hol03] in order to determine when two transitions are dependent. The advantage of using syntactic conditions for determining dependence is that they can be analyzed statically, at compile time, resulting in significant computational savings compared to detecting dependence between transitions during run-time. A further discussion on these issues can be found in [HP95].

2.6 Summary

In this chapter, we introduced our system model. We defined programs and traces, and discussed how the set of runs of a program is partitioned into traces by the independence relation. We also discussed the correspondence between traces, partial orders, and lattices. Finally, we presented some syntactic conditions for pairs of transitions to be considered independent, and showed how the dependency relation can be syntactically derived for a given program.

Chapter 3

Predicates

3.1 Introduction

In order to verify a program, it is first necessary to state the properties, called **predicates**, that the program must satisfy. For example, consider a program that implements mutual exclusion between two processes, P_1 and P_2. The following are some properties we would expect such a program to satisfy:

- P_1 and P_2 are never in the critical section at the same time.
- If P_1 wants to enter the critical section, it is guaranteed to eventually be able to do so. That is, P_1 is never starved.
- Similarly, P_2 is never starved.

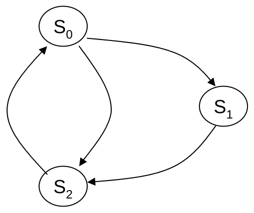
The properties to be verified on a program are typically expressed as **temporal logic** formulae. Temporal logic allows us to reason about changes in the behavior of a system over time, without explicitly mentioning specific instances of time. Instead, a formula may specify that some property *eventually* turns true, or *always* holds, or *never* turns true.

There are two prevalent types of temporal logic that are used in model checking: *linear time logic*, and *branching time logic*. **Linear time logic (LTL)** assumes that, at any given instant, there is only one possible future. LTL was first proposed by Manna and Pnueli in [MP79]. **Branching time logic** assumes that, at each instant, there are different possible futures, depending on the occurrence of (non-deterministic) events. The most prevalent branching time logic used in program verification is called **Computation Tree Logic** (CTL), which was first proposed by Emerson and Clarke in [CE82, EC82]. The lattice-theoretic principles we apply to program verification are more suited to a branching time logic. Therefore, we focus on CTL operators in this dissertation.

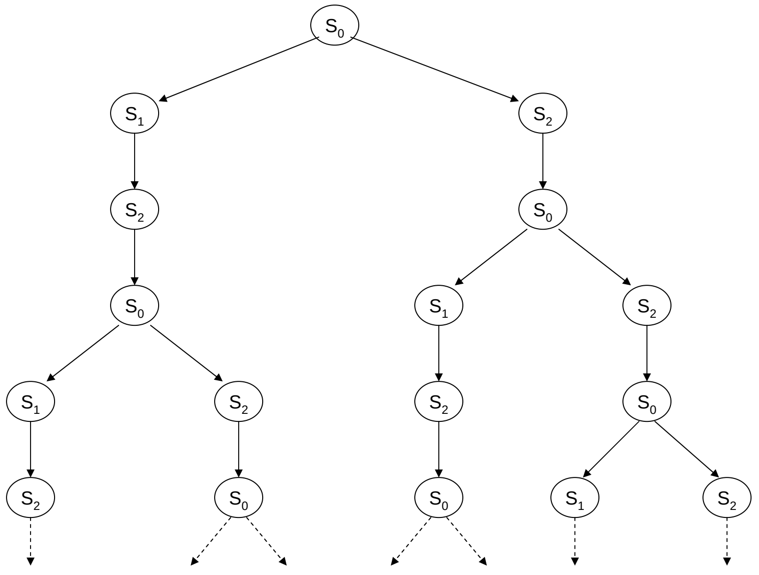
In this chapter, we discuss the syntax and semantics of CTL. We also discuss some predicate classes identified by other researchers, for which the set of satisfying states exhibit a certain structure which can be exploited to derive efficient verification algorithms. In particular, we focus on predicate classes where the set of satisfying states exhibit certain lattice-theoretic properties. We also discuss how predicate structure has been exploited in previous work by other researchers, to alleviate state space explosion during verification.

3.2 Computation Tree Logic

In Computation Tree Logic (CTL), time is assumed to have a tree-like structure. When applied to program verification, this is interpreted to mean that each state in the full state space graph can have several possible successor states. Thus, the computation can "branch" out in a tree-like structure from the current state. Different computational paths can arise from the current state, either due to non-deterministic choice in the program, or due to multiple ways of interleaving concurrent (independent) events. Figure 3.1 shows a program and its corresponding computation tree rooted at s_0.



(a)



(b)

Figure 3.1: (a) The full state space graph of a program P, and (b) its computation tree, from the initial state s_0.

The formal syntax of CTL is defined as follows. Let AP be the set of atomic propositions in a program.

1. Every atomic proposition $p \in AP$ is a CTL formula.
2. If p and q are CTL formulae, then so are $\neg p$, $(p \wedge q)$, $AX(p)$, $EX(p)$, $A[p\ U\ q]$ and $E[p\ U\ q]$.

The operator X stands for the *next time* operator. The operator A serves as a universal quantifier, and E as an existential quantifier. Intuitively, $AX(p)$ means that p holds in *all* immediate successors of the current state, and $EX(p)$ means that p holds at *some* immediate successor of the current state. $A[p\ U\ q]$ intuitively means that along every path starting from the current state, p holds continuously on the path until a state is reached at which q holds. $E[p\ U\ q]$ means that there is some path from the current state at which p holds continuously until a state is reached where q holds.

The semantics of CTL formulae are defined with respect to states in the full state space graph. A *full path* starting from a state s is a maximal (finite or infinite) path starting from s in the full state space graph. By maximal, we mean that the path is either infinite (contains a cycle), or terminates in a vertex with no outgoing edges. Let π^i denote the ith state on the path π. That is, $\pi = \pi^0\pi_1\pi_2....$ We use the notation $s \models p$ to indicate that the formula p holds at state s. The semantics of the temporal CTL operators are defined below:

- $s \models EX(p)$ iff there exists some full path π starting from s, such that $\pi^1 \models p$.
- $s \models AX(p)$ iff for every full path π starting from s, $\pi^1 \models p$.
- $s \models E[p\ U\ q]$ iff for some full path π starting from s, there exists a $j \geq 0$ such that $\pi^j \models q$ and for every i such that $0 \leq i < j$, $\pi^i \models p$.

- $s \models A[p\ U\ q]$ iff for every full path π starting from s, there exists a $j \geq 0$ such that $\pi^j \models q$ and for every i such that $0 \leq i < j$, $\pi^i \models p$.

In addition to the syntax above, the following *derived* operators are used:

- $EF(p) = E[true\ U\ p]$.

  $s \models EF(p)$ iff there exists some full path π starting from s and some $j \geq 0$ such that $\pi^j \models p$.

- $AF(p) = A[true\ U\ p]$.

  $s \models AF(p)$ iff for every full path π starting from s, there exists some $j \geq 0$ such that $\pi^j \models p$.

- $EG(p) = \neg AF(\neg p)$.

  $s \models EG(p)$ iff for there is some full path π starting from s, such that $\forall i \geq 0 :$ $\pi^i \models p$.

- $AG(p) = \neg EF(\neg p)$.

  $s \models EG(p)$ iff for every full path π starting from s, $\forall i \geq 0 : \pi^i \models p$.

- $E[p\ R\ q] = \neg A[\neg p\ U\ \neg q]$.

  $s \models E[p\ R\ q]$ iff there exists some full path π starting from s, such that for all $j \geq 0$ and $i < j$, if $\pi^i \not\models p$ then $\pi^j \models q$. In simple terms, q must hold along the path π up to and including the first state at which p holds.

- $A[p\ R\ q] = \neg E[\neg p\ U\ \neg q]$.

  $s \models E[p\ R\ q]$ iff for every full path π starting from s, for all $j \geq 0$ and $i < j$, if $\pi^i \not\models p$ then $\pi^j \models q$. In simple terms, along every full path π, q must hold on the path up to and including the first state at which p holds.

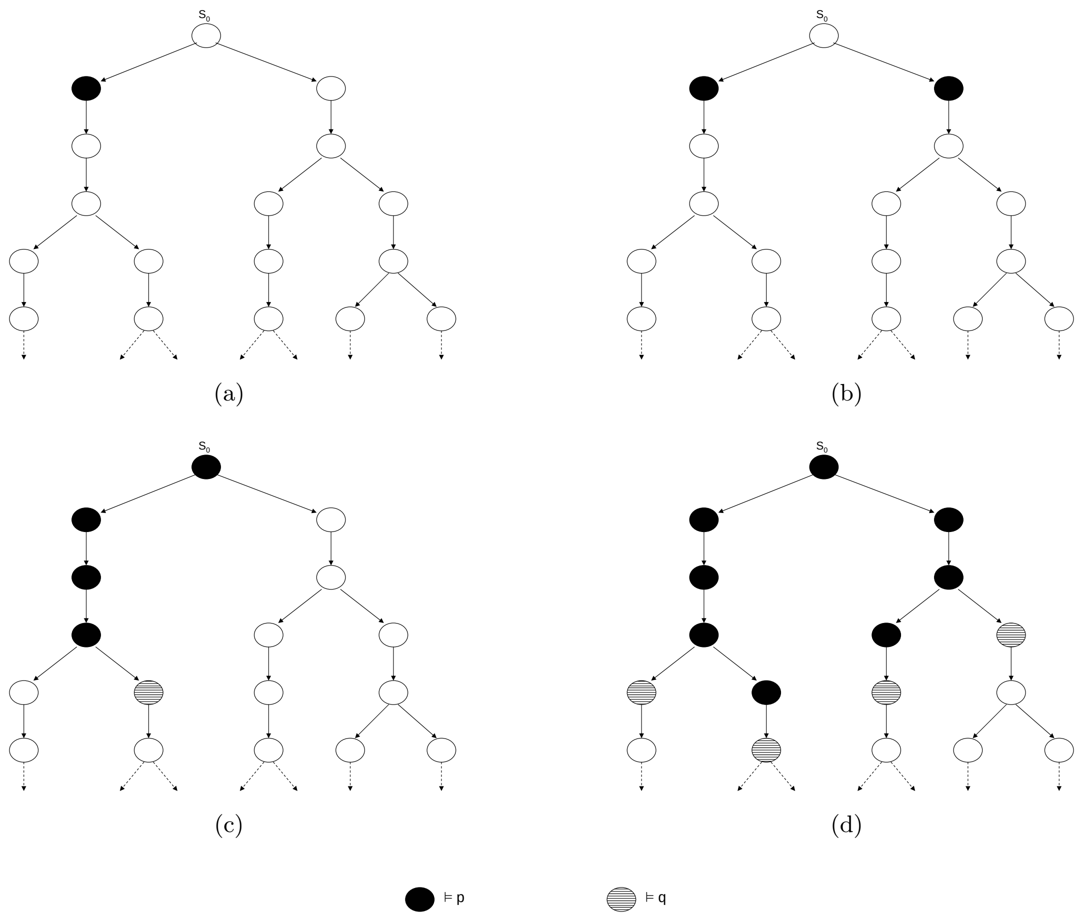


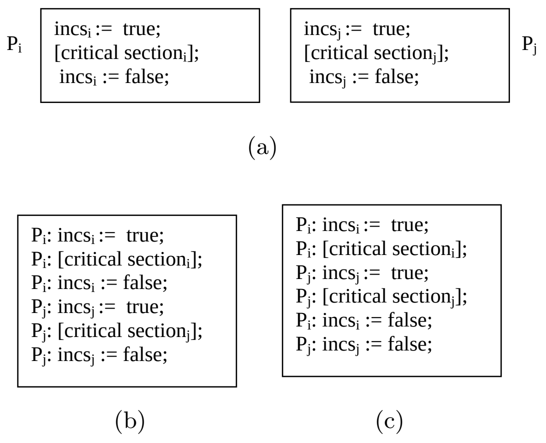
Figure 3.2: Basic CTL operators (a) $s_0 \models EX(p)$, (b) $s_0 \models AX(p)$, (c) $s_0 \models E[p\ U\ q]$, and (d) $s_0 \models A[p\ U\ q]$.

3.3 Predicate Structure

Several researchers have suggested exploiting specific characteristics of the property being verified as a means of reducing state space explosion during verification. In essence, these approaches identify certain classes of predicates for which the state space search can be specialized, usually by eliminating the need to explore multiple interleavings of concurrent events. For example, Chandy and Lamport [CL85] introduced the notion of a **stable** predicate: once the predicate turns true during the execution of a program, it stays true for the remaining duration of execution. Examples of stable predicates include termination ("the computation has terminated") and deadlock ("the system is deadlocked"). In order to decide whether a stable property ever turns true during the execution of a program, we simply need to check the final state of the execution.

Charron-Bost et al. [CBDGF95] introduced **observer-independent** predicates. An observer-independent predicate is one that holds in some maximal path in a trace iff it holds in *every* maximal path in the trace. This same class of predicates was termed **equivalence-robust** by Katz and Peled [KP87]. In order to decide whether an observer-independent (equivalence-robust) predicate ever turns true in a trace, it suffices to examine any one path in the trace. It was shown in [CBDGF95] that the class of stable predicates is a subset of the class of observer-independent predicates.

Distributed programs often suffer from errors due to insufficient synchronization, which leads to race conditions. Race conditions are usually transient errors - they do not fall under the class of observer-independent (equivalence-robust) predicates. For example, consider the improbably naive implementation of a mutual exclusion algorithm shown in Figure 3.3. A correct implementation of mutual exclusion requires synchronization between the two processes. The program shown in Figure 3.3 has only one maximal trace. Figure 3.3(b) shows an execution (path) that



P_i
```
incs_i := true;
[critical section_i];
 incs_i := false;
```

P_j
```
incs_j := true;
[critical section_j];
 incs_j := false;
```

(a)

```
P_i: incs_i := true;
P_i: [critical section_i];
P_i: incs_i := false;
P_j: incs_j := true;
P_j: [critical section_j];
P_j: incs_j := false;
```

(b)

```
P_i: incs_i := true;
P_i: [critical section_i];
P_j: incs_j := true;
P_j: [critical section_j];
P_i: incs_i := false;
P_j: incs_j := false;
```

(c)

Figure 3.3: A naive distributed mutual exclusion implementation. (a) Concurrent processes P_i and P_j, (b) an execution satisfying mutual exclusion, and (c) an execution that violates mutual exclusion.

does not cause mutual exclusion violation, while Figure 3.3(c) shows an execution (path) that does.

For program verification, **safety properties** are usually expressed as a predicate that must *never* hold in a program. That is, no reachable state of the program must satisfy the predicate. Mutual exclusion violation, and most race conditions, are examples of safety properties. Safety properties are usually not observer-independent (equivalence-robust).

In [CG95], Chase and Garg introduced the class of **meet-closed** predicates, which can be used to express many safety requirements. Meet-closed predicates exhibit lattice-theoretic properties that make them amenable to state space reduction strategies, as is explained in the following section.

3.4 Meet-Closed Predicates

We first formally define meet-closed predicates.

Definition 3.1. *A formula p is said to be* **meet-closed** *in a program P iff in every* *trace σ of P:*

$$\forall G, H \in \mathcal{L}(\sigma) : [(G \models p) \wedge (H \models p) \Rightarrow (G \cap H) \models p]$$

Informally, a formula p is meet-closed if, whenever any two states of a trace σ satisfy p, the state given by their meet in the down-set lattice also satisfies p. That is, in the down-set lattice $\mathcal{L}(\sigma)$, the set of all down-sets satisfying σ forms an inf-semilattice. Figure 3.4(a) shows an example of a meet-closed predicate. Meet-closed predicates imply the existence of certain "crucial" events, which can be used to prune the state space search. This concept of "crucial" events is explained in the following section.

3.4.1 Crucial Events

Let G be any down-set of a trace $\sigma = (E, \rightarrow)$. Let p be some meet-closed formula, and $G \not\models p$. Let $\mathcal{G}$ be the set of all p-satisfying states that are reachable from G in σ. That is:

$$\mathcal{G} = \{H \in \mathcal{L}(\sigma) | G \subseteq H \wedge H \models p\} \tag{3.1}$$

Now, $\mathcal{G}$ can be an infinite set. Let $\mathcal{H}$ be the set of elements of $\mathcal{G}$ that are minimal under $\subseteq$:

$$\mathcal{H} = \{H \in \mathcal{G} | \forall H' : H \subset H' \Rightarrow H' \notin \mathcal{H}\} \tag{3.2}$$

$\mathcal{H}$ is necessarily finite for finite-state programs. We now define:

$$K = \bigcap_{H \in \mathcal{H}} H \tag{3.3}$$

By the meet-closure of p, $K \models p$. Also, $G \subseteq K$. The following lemma is straightforward, from the properties of set intersection, and the fact that p is meet-closed.

Lemma 3.1. *If $\mathcal{H} \neq \emptyset$, then $\mathcal{H} = \{K\}$.*

That is, K is the unique and well-defined p-satisfying state that is reachable from G by executing the *fewest* events. In particular, $K \setminus G$ is the minimum set of events that *must* be executed along any path starting from G, in order to reach a p-satisfying state in σ. The events in $K \setminus G$ are called **crucial events** [CG95].

Note that if $\mathcal{H} = \emptyset$, then by the properties of nullary intersection we have $K = E$ (E is the set of all events in the trace). That is, if there is no p-satisfying state in σ, then every event in $E \setminus G$ is considered a crucial event. An alternative definition of crucial events follows.

Definition 3.2. Crucial event: *In a trace σ, an event e is said to be crucial from a state G with respect to a meet-closed formula p, denoted $e \in crucial(G, p, \sigma)$ iff:*

$$\forall H \in \mathcal{L}(\sigma) : (G \subseteq H) \wedge (G \not\models p) \wedge (H \models p) \Rightarrow (e \in H \setminus G)$$

In simple terms, a crucial event is one whose execution is necessary in order to reach a p-satisfying state from G in σ.

A transition sequence starting from G and comprising exactly of the events in $crucial(G, p, \sigma)$ gives us a path of shortest length from G to a p-satisfying state in σ. Such a path is called a **crucial path**. When $\mathcal{H} = \emptyset$, we have $K = E$ (the set of all events), and any maximal path starting from G in $\mathcal{L}(\sigma)$ constitutes a crucial path. A crucial path is of particular interest in model checking, because it gives us a witness path of the shortest length to a p-satisfying state. The following theorem is a direct consequence of the fact that a crucial path is a witness path of shortest length.

Theorem 3.2. *Let $\mathcal{H}$ be as defined in Equation (3.2). If $\mathcal{H} \neq \emptyset$, then a crucial path for p starting from G cannot contain a cycle.*

Recall that a down-set is an occurrence of a state. Suppose the down-set G is an occurrence of the state s. Executing the events in $crucial(G, p, \sigma)$ from s

will lead to a p-satisfying state in the full state space graph. The state s can have multiple occurrences in σ (for example, in Figure 2.6(c), the state t occurs multiple times in σ_2). Let G' be another down-set of σ that is also an occurrence of s. It is easy to see that $crucial(G, p, \sigma) = crucial(G', p, \sigma)$. Thus, every occurrence of s in σ has the same set of crucial events w.r.t. p. Based on this observation, we define:

$$crucial(s, p, \sigma) \overset{def}{\equiv} crucial(G, p, \sigma) \tag{3.4}$$

where G is any down-set of σ that is an occurrence of s.

Researchers have previously exploited meet-closure of formulae to derive efficient verification algorithms, as is discussed in the next section.

3.4.2 Exploiting Meet-Closure

Chase and Garg [CG95] used the notion of crucial events to derive an efficient algorithm for determining, for a given finite trace $\sigma = [s, v]$ and a meet-closed formula p, whether $s \models EF(p)$ in σ, that is, whether any reachable state of a finite trace σ satisfies p. Algorithm 3.1 shows the pseudocode for their approach. If p is a state formula involving no temporal operators, it takes $O(1)$ time to evaluate whether a given state t satisfies p. In line 4, a crucial event needs to be identified. If a crucial event for a formula can be identified in $O(|E|^k)$ time, where E is the event set of the given finite trace and $k \geq 0$ is some constant, the formula is said to satisfy the **efficient advancement property**[CG95]. Chase and Garg's algorithm requires traversing only a single crucial path through the trace. If p satisfies the efficient advancement property, Chase and Garg's algorithm can decide whether $s \models EF(p)$ in $O(|E|^{k+1})$ time [CG95].

In [SG02], Sen and Garg exploited the properties of meet-closure to come up with an efficient verification algorithm for determining, given a finite trace $\sigma = [s, v]$ and meet-closed formula p, whether $s \models AG(p)$. Their algorithm identifies the

Algorithm 3.1: *EF_meet_closed*

```
input  : A finite trace $\sigma = [s, v]$, and a meet-closed predicate $p$.
output: $true$ if $s \models EF(p)$, $false$ otherwise.
begin
    $t := s$
    while $t \not\models p$ and $enabled(t) \neq \emptyset$ do
        let $\alpha \in crucial(t, p, \sigma)$
        $t := \alpha(t)$
    endw
    if $t \models p$ then
        return $true$
    endif
    else
        return $false$
    endif
end
```

set of **meet-irreducible** elements of the lattice $\mathcal{L}(\sigma)$. The concept of **meet-irreducibility** is analogous to that of prime numbers in arithmetic. Recall that every natural number can be expressed as the product of some prime numbers. Similarly, every element of the (finite) lattice $\mathcal{L}(\sigma)$ can be expressed as the meet of some meet-irreducible elements. Meet-irreducible events are discussed further in Chapter 7. As p is meet-closed, if every meet-irreducible element of $\mathcal{L}(\sigma)$ satisfies p, then $s \models AG(p)$. A famous result in lattice theory, known as **Birkhoff's representation theorem** [DP90], proves that the number of meet-irreducible elements of $\mathcal{L}(\sigma)$ is equal to the number of events in σ. In [GM01], Garg and Mittal presented an algorithm for identifying the set of meet-irreducible elements in $O(n^2.|E|)$ time, where n is the number of processes in the trace, and E is the event set of the trace. Assuming it takes constant time to decide if a given state satisfies p, Sen and Garg's algorithm runs in $O(n^2.|E|)$ time.

In addition to meet-closure, the property of **join-closure** has also been exploited for the development of efficient verification algorithms, as is discussed in the

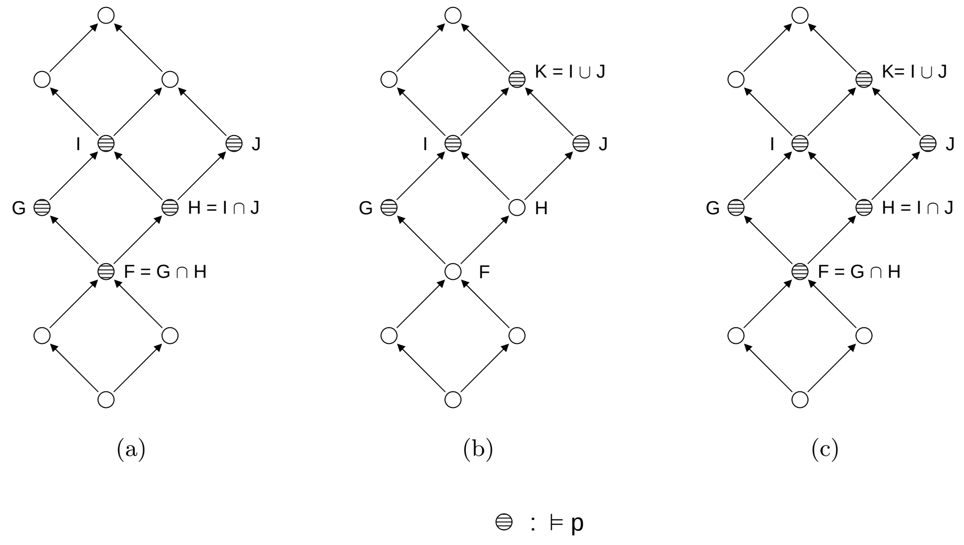


Figure 3.4: (a) p is meet-closed (b) p is join-closed and (c) p is regular.

next section.

3.5 Regular Predicates

Analogous to the concept of meet-closure, a formula may also exhibit **join-closure** in a program, as defined below.

Definition 3.3. *A formula p is said to be* **join-closed** *in a program P iff in every trace σ of P:*

$$\forall G, H \in \mathcal{L}(\sigma) : [(G \models p) \wedge (H \models p) \Rightarrow (G \cup H) \models p]$$

Informally, a formula p is join-closed if, whenever any two states of a trace σ satisfy p, the state given by their join in the down-set lattice also satisfies p. That is, in the down-set lattice $\mathcal{L}(\sigma)$, the set of all down-sets satisfying σ forms an sup-semilattice.

Join-closure was exploited by Sen and Garg in [SG02] to develop an efficient algorithm for deciding whether $s \models EG(p)$, for a finite trace $\sigma = [s, v]$ and join-closed formula p. Their algorithm shows that, as a consequence of the join-closure of p, it is possible to decide whether $s \models EG(p)$ by exploring a single path through the trace σ. Their algorithm runs in $O(n.|E|)$ time, where n is the number of processes in the trace, and E is the event set of the trace.

A formula that exhibits both meet- and join-closure is said to be **regular** [GM01].

Definition 3.4. *A formula p is said to be* **regular** *in a program P iff it is meet- and join-closed in P.*

Regular predicates were first introduced by Garg and Mittal in [GM01], where it was shown that, given a trace σ and regular predicate p, the set of down-sets of σ that satisfy p forms a sublattice of $\mathcal{L}(\sigma)$. This property of regular predicates was exploited for state space reduction through the concept of **slicing** in, for example, [GM01, MG01, SG03a]. It has been shown by Sen and Garg [SG03a] that several temporal CTL operators preserve regularity. In particular, if p is regular, so are $EG(p)$, $AG(p)$, and $EF(p)$. A further discussion on how regularity can be used for state space reduction is deferred until Chapter 7. For now, it suffices to say that regular predicates constitute another predicate class whose structure has been exploited for the development of efficient verification algorithms.

3.6 Bibliographic Notes

The use of temporal logic for specifying properties of concurrent programs was first proposed by Pnueli in [Pnu77]. Pnueli also proposed a temporal semantics for reasoning about concurrent programs in [Pnu81]. Some notable surveys about the role of temporal logic in computer science include those by Pnueli [Pnu86], Goldblatt

[Gol87], and Emerson [Eme90].

There has been much debate in the literature about the relative merits of linear versus branching time logics, going back to 1980. Vardi presents a discussion of these issues, together with an extensive list of references, in [Var01]. In practice, CTL tends to be more favored in industrial (hardware) verification, mainly because most of the early model checkers were CTL-based. Examples of these early CTL model checking tools include SMV [McM92] and its follower, VIS [BHSV$^+$96]. LTL has recently gained greater prevalence in software verification, largely due to the popularity of the LTL-based software model checker, SPIN [Hol03].

The logics and predicate classes discussed in this chapter all express properties on "global states" of the system. That is, the properties specify behaviors of states and paths in the full state space graph. The approaches that try to exploit predicate structure aim at reducing state space explosion by avoiding the exploration of multiple paths per trace. Another approach aimed at taming state space explosion due to concurrency involves defining logics that directly reason about the underlying partial order of events. An example is Pinter and Wolper's Partial Order Temporal Logic (POTL) [PW84]. However, these logics suffer from a complementary problem - in a pure partial order approach, there is no concept of a global state. Consequently, many interesting properties of systems, which tend to be global in nature, cannot be expressed in these logics, and as a result, they failed to catch on in the verification community.

A survey of the many applications of lattice theory to the verification of distributed systems appears in [GMS03]. However, the applications surveyed are limited to performing verification on *finite* program traces. Most of the previous work on applying lattice theory to distributed computing, such as [CG95, GM01, MG01, SG02, SG03a] *etc.*, is also limited to finite program traces. In this dissertation, we extend these concepts beyond finite program traces, to the verification of complete programs (albeit, finite-state programs).

3.7 Summary

In this chapter, we discussed the specification of properties to be verified on programs. We introduced the syntax and semantics of CTL. We also discussed how predicate structure can be a useful tool for alleviating state space explosion during verification. We discussed examples of predicate classes that have been exploited by previous researchers to reduce the state space searched during verification. In particular, we focussed on the classes of meet-closed, join-closed, and regular predicates. We showed that meet-closure implies the existence of certain crucial events in each program trace, whose execution is both necessary and sufficient for the predicate to turn true in that trace. We briefly discussed how meet- and join-closure has been exploited by other researchers to develop efficient verification algorithms.

Part II

Predicate Structure

Chapter 4

Predicate Recognition

4.1 Introduction

In Chapter 3, we mentioned that predicate structure has been exploited by various researchers to tame state space explosion during verification. These approaches typically reduce state space explosion by avoiding the exploration all possible interleavings of concurrent events. For example, for observer-independent predicates [CBDGF95], if the predicate turns true in any one maximal path of a trace, then it turns true in all maximal paths of the trace. Therefore, it suffices to explore any one maximal path in the trace. For meet-closed predicates, as discussed in Section 3.4.1, it suffices to explore any one **crucial path** in the trace. For a trace consisting of n events, these techniques have a *worst-case* time complexity of $O(n^k)$, for some constant $k \geq 0$.

Because these techniques do not perform an exhaustive exploration of the state space, they require the predicate being verified to adhere to a certain structure. If it is not known beforehand that the predicate exhibits the assumed structure, then the decision procedures in these algorithms are sound but not complete. Therefore, a problem of interest is whether we can determine if a given predicate exhibits a certain structure. Further, we must be able to make this determination without running into the same state space explosion problem that these specialized verification techniques are trying to avoid. In particular, given a trace with n events, we would like to determine in time that is *polynomial* in n, whether a given predicate belongs to a certain "efficient" predicate class. We call this the **predicate recognition problem**. This chapter addresses this problem.

4.1.1 Problem Statement

A **predicate class** is a set of predicates in which each member formula exhibits some common behavior. Meet-closed predicates, regular predicates and stable predicates are each examples of a predicate class. Let $\sigma = (E, \rightarrow)$ be a trace of a program, and C be a predicate class.

The Predicate Recognition Problem: Given a predicate p, can we decide if $p \in C$ in $O(|E|^k)$ time, for some constant $k \geq 0$?

4.1.2 Our Contribution

In this chapter, we answer the predicate recognition problem for various predicate classes. We show that the problem is co-NP-complete for the classes of meet-closed, join-closed and regular predicates. We also show that it is NP-hard for any predicate class for which $EF(p)$ can be decided in time that is polynomial in $|E|$, and co-NP-hard for any predicate class for which $AG(p)$ can be decided in time that is polynomial in $|E|$.

4.2 Recognizing Meet-Closure

As the focus of this dissertation is on exploiting lattice-theoretic properties exhibited by programs and predicates, we start by considering the problem of deciding whether a given predicate is meet-closed. In particular, we will focus on boolean predicates, that is, a predicate which is a boolean formula over the variables of a program.

Let $\sigma = (E, \rightarrow)$ be a trace of a program P, such that $|E| = n$. Let p be a boolean formula defined over the variables of P. We define the following decision problem:

MEET-CLOSURE: Is p meet-closed in σ?

A decision problem is said to be in **co-NP** [GJ90] if a counterexample for the decision problem exists, which can be verified in polynomial time. In other words, an efficiently verifiable proof of a "no" instance exists. A decision problem is said to be **co-NP-hard** if every problem in co-NP is polynomial-time reducible to it. That is, a decision problem C is co-NP-hard if there exists a deterministic Turing machine that can convert any other problem in co-NP into an instance of C, in polynomial time. A decision problem is said to be **co-NP-complete** iff it is in co-NP, and is co-NP-hard. A well-known co-NP-complete problem is TAUTOLOGY [Coo71], the problem of deciding whether a given boolean formula is a tautology, that is, whether the formula is always true for every possible valuation of its variables.

Theorem 4.1. *MEET-CLOSURE is co-NP-complete.*

Proof.

- MEET-CLOSURE is in co-NP.

  If the given predicate p is not meet-closed, then there exist down-sets $G, H \in \mathcal{L}(\sigma)$ such that $G \models p$ and $H \models p$, but $(G \cap H) \not\models p$. Since p is a boolean expression, its truth value at a state can be evaluated in polynomial time. So, we can verify in polynomial time that $G \models p$ and $H \models p$, but $(G \cap H) \not\models p$, and the down-sets G and H form the required counterexample for MEET-CLOSURE to be in co-NP.

- MEET-CLOSURE is co-NP-hard.

  We can transform an arbitrary instance of TAUTOLOGY into an instance of MEET-CLOSURE, as follows. Let f be a boolean expression involving variables $x_1, x_2, ..., x_n$. We construct a program P consisting of $n+2$ processes, $\{P_1, P_2, ..., P_n, P_{n+1}, P_{n+2}\}$. Each process P_i contains a single local variable, x_i. In the initial program state, $x_1, x_2, ..., x_n, x_{n+1}$ are all set to $false$, and x_{n+2} is set to $true$. The program consists of a single maximal trace σ, with $n+2$ independent (concurrent) events, as follows.

- $\forall i : 1 \leq i \leq n+1$, there is an event α_i that changes the value of x_i from $false$ to $true$, and
- an event α_{n+2} that changes the value of x_{i+2} from $true$ to $false$, and

Figure 4.1 shows the process transition graphs for the program described. It is evident that the transformation above can be performed in polynomial time. Note that any subset of $\{\alpha_1, \alpha_2, ..., \alpha_n + 2\}$ is a valid down-set of the trace σ, because for all $i \neq j : (\alpha_i, \alpha_j)$ are independent.

We define a boolean formula p as follows:

$$p = f \vee x_{n+1}x_{n+2} \vee \overline{x}_{n+1}\overline{x}_{n+2} \tag{4.1}$$

We claim that p is meet-closed if and only if f is a tautology. If f is a tautology, then p is trivially meet-closed, because every down-set of σ corresponds to a satisfying assignment for f, and consequently, a satisfying assignment for p.

Conversely, if f is not a tautology, then there exists some assignment for $x_1, ..., x_n$ for which f is false. This assignment corresponds to some subset, say G, of $\{\alpha_1, ...\alpha_n\}$. Clearly, G is also a subset of $\{\alpha_1, ..., \alpha_{n+2}\}$, and hence is a down-set of σ. Consider the following two subsets, G_1 and G_2, of $\{\alpha_1, ..., \alpha_{n+2}\}$:

$$G_1 = G \cup \{\alpha_{n+1}\} \tag{4.2}$$

$$G_2 = G \cup \{\alpha_{n+2}\} \tag{4.3}$$

The events in G_1 turn the clause $x_{n+1}x_{n+2}$ *true*, while the events in G_2 turns the clause $\overline{x}_{n+1}\overline{x}_{n+2}$ *true*. However, $G_1 \cap G_2 = G$, and p is *false* in G. Therefore, p is not meet-closed if f is not a tautology.

□

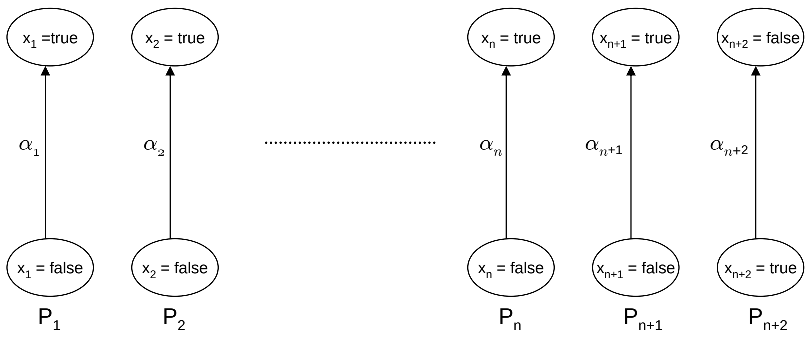


Figure 4.1: Process transition graphs for the transformation showing that MEET-CLOSURE is co-NP-hard.

4.3 Recognizing Regularity

Analogous to the decision problem MEET-CLOSURE, we define a similar decision problem for determining whether a given boolean formula p is regular (meet- and join-closed) in a trace σ:

REGULARITY: Is p regular in σ?

Again, we can show that REGULARITY is co-NP-complete in the number of events in the trace.

Theorem 4.2. *REGULARITY is co-NP-complete.*

Proof.
- REGULARITY is in co-NP.

  If the given boolean predicate p is not regular, then there exist down-sets $G, H \in \mathcal{L}(\sigma)$ such that $G \models p$ and $H \models p$, but either $(G \cap H) \not\models p$, or $(G \cup H) \not\models p$. Since p is a boolean expression, its truth value at a state can be evaluated in polynomial time. So, the down-sets G and H form the required counterexample for REGULARITY to be in co-NP.

- REGULARITY is co-NP-hard.

  The polynomial-time transformation from TAUTOLOGY to MEET-CLOSURE in Theorem 4.1 also serves as a transformation from TAUTOLOGY to REGULARITY. That is, as defined in Equation (4.1), p is regular iff f is a tautology. If f is a tautology, then p is trivially regular, because every down-set of σ corresponds to a satisfying assignment for f, and consequently, a satisfying assignment for p.

  If f is not a tautology then, as discussed in the proof of Theorem 4.1, both G_1 and G_2, from Equations (4.2) and (4.3) respectively, satisfy p, but $(G_1 \cap G_2)$ does not satisfy p, implying that p is not regular.

□

It is also worth noting that $(G_1 \cup G_2)$ also does not satisfy p. This implies that p is join-closed iff f is a tautology, which leads to the following corollary.

Corollary 4.3. *Given a trace σ and a boolean formula p, deciding whether p is join-closed in σ is co-NP-complete.*

So far, we have shown that there is no efficient algorithm for determining if a given predicate is meet- and/or join-closed. In the next section, we show that if the reachability problem can be solved for a predicate class in time that is polynomial in the number of events in a trace, then deciding membership for that predicate class is NP-hard.

4.4 Other Recognition Problems

Consider a boolean formula that is known to be non-satisfiable. Let us denote the class of all non-satisfiable boolean formulae by C_{false}. Then, no program can assign any formula $p_{false} \in C_{false}$ a satisfying assignment of values to its variables. That is, each $p_{false} \in C_{false}$ is meet-closed, join-closed, regular, stable, and observer-independent in any trace of a program.

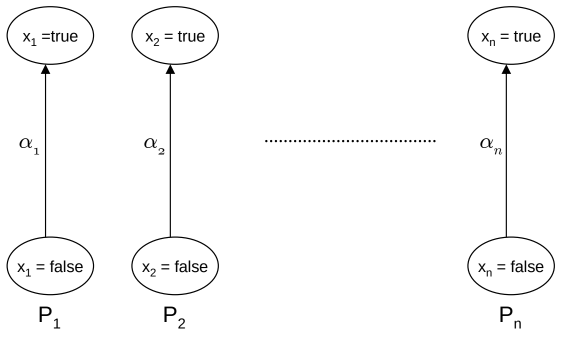


Figure 4.2: Process transition graphs for Theorem 4.4.

Theorem 4.4. *Given a predicate class $\mathcal{C}$ such that:*

- $C_{false} \subseteq \mathcal{C}$*, and*
- $\forall p' \in \mathcal{C} : s \models EF(p')$ *can be decided in* $O(n^k)$ *time for any trace* $[s, v]$*, where* $|v| = n$ *and* $k \geq 0$ *is some constant.*

It is NP-hard to determine whether a given boolean formula p is a member of the class $\mathcal{C}$, i.e., $p \in \mathcal{C}$.

Proof. We show that if a polynomial-time algorithm exists for deciding membership in $\mathcal{C}$, then there exists a polynomial time algorithm for deciding the boolean satisfiability problem, SAT. Recall that SAT is a well-known NP-complete problem.

The SAT decision problem asks whether, given a boolean formula, there is any assignment of values to its variables that turn the formula *true*. SAT was the first problem shown to be NP-complete, by Cook in [Coo71].

Let p be a boolean formula for which we wish to solve SAT. Let p involve the variables $x_1, x_2, ..., x_n$. We can create a program P consisting of n processes, where each process has a single local variable, x_i. In the initial program state, each x_i is set to $false$. The program contains n independent transitions, $\alpha_1, \alpha_2, ..., \alpha_n$, where each α_i changes the value of x_i from $false$ to $true$. The process transition graphs are shown in Figure 4.4. This program contains exactly one maximal trace, $\sigma = [s, \alpha_1\alpha_2...\alpha_n]$, where for each i and j such that $i \neq j$, (α_i, α_j) are indepenent.

Assume there exists an algorithm which can determine whether p is a member of the class $\mathcal{C}$, in time that is polynomial in n. If $p \notin \mathcal{C}$, then p has a satisfying truth assignment, because every non-satisfiable boolean formula p' is a member of $\mathcal{C}$. On the other hand, if $p \in \mathcal{C}$, then we can use the $O(n^k)$ algorithm for deciding whether $s \models EF(p)$ in σ. Clearly, p is satisfiable iff $s \models EF(p)$. Thus, if $p \in \mathcal{C}$ can be decided in polynomial time, then SAT can be solved in polynomial time.

$\square$

Dually, let us denote the class of all non-falsifiable (tautological) boolean formulae by C_{true}. Then, no program can assign any formula $p_{true} \in C_{true}$ an assignment of values to its variables that leads to the formula turning $false$. Again, each $p_{true} \in C_{true}$ is meet-closed, join-closed, regular, stable, and observer-independent for any trace of a program. The following theorem is the dual of Theorem 4.4.

Theorem 4.5. *Given a predicate class $\mathcal{C}$ such that:*

- $C_{true} \subseteq \mathcal{C}$, *and*
- $\forall p' \in \mathcal{C} : s \models AG(p')$ *can be decided in* $O(n^k)$ *time for any trace* $[s, v]$, *where* $|v| = n$ *and* $k \geq 0$ *is some constant.*

It is co-NP-hard to determine whether a given boolean formula p is a member of the class $\mathcal{C}$, i.e., $p \in \mathcal{C}$.

Proof. We show that if a polynomial-time algorithm exists for deciding membership in $\mathcal{C}$, then there exists a polynomial time algorithm for deciding TAUTOLOGY. Let p be a boolean formula for which we wish to solve TAUTOLOGY. Let p involve the variables $x_1, x_2, ..., x_n$. We create a program identical to the one in the proof of Theorem 4.4.

Assume there exists an algorithm which can determine whether p is a member of the class $\mathcal{C}$, in time that is polynomial in n. If $p \notin \mathcal{C}$, then p is not a tautology, because every tautology is a member of $\mathcal{C}$. On the other hand, if $p \in \mathcal{C}$, then we can use the $O(n^k)$ algorithm for deciding whether $s \models AG(p)$ in σ. Clearly, p is a tautology iff $s \models AG(p)$. Thus, if $p \in \mathcal{C}$ can be decided in polynomial time, then TAUTOLOGY can be solved in polynomial time.

□

4.5 Bibliographic Notes

The results presented in this chapter were published in [KG05b]. To the best of our knowledge, the predicate recognition problem has not been previously addressed by other researchers.

4.6 Summary

In this section, we presented some results on the predicate recognition problem. We showed that predicate recognition is co-NP-complete for meet-closed, join-closed, and regular predicates, is NP-hard for any predicate class for which $EF(p)$ can be decided efficiently, and is co-NP-hard for any predicate class for which $AG(p)$ can be decided efficiently.

Chapter 5

Regular CTL Operators

5.1 Introduction

In Chapter 4, we showed that, given an arbitrary boolean formula, the problem of deciding whether it is regular is co-NP-complete. In this chapter, we show that we can construct a grammar (*i.e.*, a set of syntactic rules) for a logic such that each formula yielded by the grammar is regular. This provides a mechanism by which we can specify properties for verification, while still taking advantage of efficient verification algorithms that can exploit the structure of regular predicates, such as the algorithms in [CG95, MG01, GM01, SG02].

In most logics used for property specification, including propositional and temporal logics, the simplest well-formed formulae of the logic are called **atomic propositions**. An atomic proposition is one that cannot be divided into smaller propositions, and its truth or falsity does not depend on any other proposition. In program verification, atomic propositions typically correspond to statements about valuations of program variables. For example, if p is a local variable on process P_i, then $p \geq 2$ is an atomic proposition. Logical or temporal operators are then applied recursively to build more complex formulae. In order to build a grammar for a logic that yields only regular formulae, we start with atomic propositions that exhibit regularity, and then add regularity-preserving logical and temporal operators to the grammar.

5.1.1 Our Contribution

This chapter addresses the question of what kinds of atomic propositions exhibit regularity, and which temporal and logical operators preserve it. Mittal and Garg [MG01] and Sen and Garg [SG03a] have also previously explored regularity-preserving operators. We summarize the results known so far in Table 5.1, which also lists the new results proved by us in this dissertation.

In some cases, an operator does not preserve regularity, but preserves a stronger property, called **biregularity**. A formula p is said to be **biregular** iff both p and $\neg p$ are regular. Figure 5.1 shows the relationship between the classes of meet-closed, join-closed, regular and biregular predicates. We also explore biregularity-preserving operators in this chapter.

Formula	Preserves regularity (p, q regular)	Preserves biregularity (p, q biregular)
$\neg p$	No, [GM01]	Yes, by definition
$p \wedge q$	Yes, [GM01]	No, example in Section 5.2
$p \vee q$	No, [GM01]	No, example in Section 5.2
$EF(p)$	Yes, [SG03a]	Yes, Theorem 5.2
$EG(p)$	Yes, [SG03a]	Yes, Theorem 5.4
$AF(p)$	No, [Sen04]	Yes, Theorem 5.4
$AG(p)$	Yes, [SG03a]	Yes, Theorem 5.2
$E[p\ U\ q]$	No, [Sen04]	No, example in Section 5.2
$E[p\ R\ q]$	Yes, Theorem 5.5	No, example in Section 5.2
$A[p\ U\ q]$	No, [Sen04]	No, example in Section 5.2
$A[p\ R\ q]$	No, example in Section 5.3	No, example in Section 5.2

Table 5.1: Closure properties preserved by the various CTL operators.

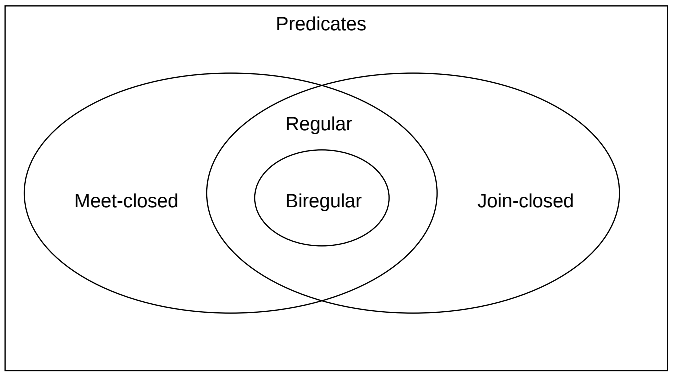


Figure 5.1: Relationship between predicate classes

5.2 Preserving Biregularity

A formula ϕ is called a *process-local state formula* iff its truth value is purely determined by the current values of the local variables V_i of some process P_i. Recall that the value of any local variable in V_i can only be changed by some transition from T_i. Thus, the truth or falsity process-local state formula can only be changed by transitions from T_i.

An example of a process-local state formula is "process i is in the critical section". In particular, this is a process-local state formula because the program counter is a local variable on process P_i, and can only be changed by a transition from T_i.

Theorem 5.1. *Process-local state formulae are biregular.*

Proof. Let σ be a trace, and p a process-local state formula defined on the local variables of process P_j. Since no two transitions from P_j are independent, no two transitions from P_j can commute with each other. So, the events from P_j must occur in the same sequence in every path of σ.

Let s be the starting state of σ, and v be a maximal path of $\mathcal{L}(\sigma)$. Let v_j

be the restriction of v to events from P_j, *i.e.*, v_j is obtained from v by deleting all events from processes other than P_j. Let G and H be any two down-sets of σ such that $G \models p$ and $H \models p$. Let u and w be any two paths in $\mathcal{L}(\sigma)$, leading, respectively, from s to G and s to H. Then, both u_j and w_j (derived in a similar fashion as v_j from v) are prefixes of v_j. Thus, either u_j is a prefix of w_j, or w_j is a prefix of u_j. WLOG, say u_j is a prefix of w_j.

Now, let u' be any path from s to $(G \cap H)$ in $\mathcal{L}(\sigma)$. Then, $u'_j = u_j$. Since the truth value of p is determined purely by events from process P_j, and $G \models p$, we have $(G \cap H) \models p$. Similarly, let w' be some path from s to $(G \cup H)$ in $\mathcal{L}(\sigma)$. Then, $w'_j = w_j$, hence $(G \cup H) \models p$.

Finally, the negation of a process-local state formula is also a process-local state formula. Thus, $\neg p$ is also regular, which implies that p is biregular. $\square$

Process-local state formulae will constitute the set of atomic propositions of our "regular" logic. We now consider some temporal CTL operators. In [SG03a], it was shown that if p is regular, then so are $EF(p)$ and $AG(p)$. The following theorem shows that the EF and AG operators preserve biregularity.

Theorem 5.2. *If p is biregular, then $EF(p)$ and $AG(p)$ are biregular.*

Proof. Since p is (bi)regular, from [SG03a], we know that $EF(p)$ and $AG(p)$ are regular. Now, we need to show that $\neg EF(p)$ and $\neg AG(p)$ are regular.

$$\begin{aligned}
& G \models \neg EF(p) \text{ and } H \models \neg EF(p) \\
\equiv\quad & \{\text{Since } \neg EF(p) = AG(\neg p)\} \\
& G \models AG(\neg p) \text{ and } H \models AG(\neg p) \\
\Rightarrow\quad & \{\neg p \text{ is regular, so } AG(\neg p) \text{ is regular}\} \\
& (G \cap H) \models AG(\neg p) \text{ and } (G \cup H) \models AG(\neg p) \\
\equiv\quad & \{\text{Since } AG(\neg p) = \neg EF(p)\} \\
& (G \cap H) \models \neg EF(p) \text{ and } (G \cup H) \models \neg EF(p)
\end{aligned}$$

Similarly:

$$\begin{aligned}
& G \models \neg AG(p) \text{ and } H \models \neg AG(p) \\
\equiv\quad & \{\text{Since } \neg AG(p) = EF(\neg p)\} \\
& G \models EF(\neg p) \text{ and } H \models EF(\neg p) \\
\Rightarrow\quad & \{\neg p \text{ is regular, so } EF(\neg p) \text{ is regular}\} \\
& (G \cap H) \models EF(\neg p) \text{ and } (G \cup H) \models EF(\neg p) \\
\equiv\quad & \{\text{Since } EF(\neg p) = \neg AG(p)\} \\
& (G \cap H) \models \neg AG(p) \text{ and } (G \cup H) \models \neg AG(p)
\end{aligned}$$

□

In [SG03a], it was also shown that $EG(p)$ is regular when p is regular. In [Sen04], it was shown that $AF(p)$ is join-closed for regular p, but is not meet-closed for regular p. Here, we show that $AF(p)$ is meet-closed when p is biregular.

Lemma 5.3. *$AF(p)$ is meet-closed for biregular p.*

Proof. Assume, for contradiction, that $G \models AF(p)$ and $H \models AF(p)$, but $(G \cap H) \models$

$\neg AF(p)$.

$$\begin{aligned}
& (G \cap H) \models \neg AF(p) \\
\equiv \quad & \{\text{Since } \neg AF(p) = EG(\neg p)\} \\
& (G \cap H) \models EG(\neg p) \\
\Rightarrow \quad & \{\text{Definition of } EG\ \} \\
& (G \cap H) \models \neg p \\
\Rightarrow \quad & \{p \text{ is biregular, so } \neg p \text{ is meet-closed }\} \\
& (G \models \neg p) \vee (H \models \neg p)
\end{aligned}$$

WLOG, let $G \models \neg p$. Since $(G \cap H) \models EG(\neg p)$, there exists a maximal path π starting from $(G \cap H)$ such that $\forall i : \pi^i \models \neg p$. Then, we can construct the following path ρ, starting from G, as follows:

$$\rho = G \cup \pi^0, G \cup \pi^1, G \cup \pi^2, ...$$

Recall that $\pi^0 = G \cap H$, so the path ρ starts from G , since $G \cup (G \cap H) = G$. From the properties of set union, and because π is a valid path, for each $i \geq 0$, ρ^{i+1} is either the same as ρ^i, or contains one additional event. Eliminating consecutive identical down-sets (states), we obtain a valid maximal path starting from G, such that no state along the path satisfies p. That is, ρ gives us a witness for $G \models EG(\neg p)$, which implies that $G \models \neg AF(p)$, which contradicts our initial assumption that $G \models AF(p)$. □

Theorem 5.4. *If p is biregular, then $AF(p)$ and $EG(p)$ are biregular.*

Proof. Given p is biregular. Then, from [SG03a], $EG(p)$ is regular. Also, from [Sen04], $AF(p)$ is join-closed. From Lemma 5.3, $AF(p)$ is meet-closed. Hence,

$AF(p)$ is regular. Now, we need to show that $\neg AF(p)$ and $\neg EG(p)$ are regular.

$$
\begin{array}{ll}
 & G \models \neg AF(p) \text{ and } H \models \neg AF(p) \\
\equiv & \text{Since } \neg AF(p) = EG(\neg p)\} \\
 & G \models EG(\neg p) \text{ and } H \models EG(\neg p) \\
\Rightarrow & \{\neg p \text{ is regular, so } EG(\neg p) \text{ is regular}\} \\
 & (G \cap H) \models EG(\neg p) \text{ and } (G \cup H) \models EG(\neg p) \\
\equiv & \text{Since } EG(\neg p) = \neg AF(p)\} \\
 & (G \cap H) \models \neg AF(p) \text{ and } (G \cup H) \models \neg AF(p)
\end{array}
$$

Similarly:

$$
\begin{array}{ll}
 & G \models \neg EG(p) \text{ and } H \models \neg EG(p) \\
\equiv & \text{Since } \neg EG(p) = AF(\neg p)\} \\
 & G \models AF(\neg p) \text{ and } H \models AF(\neg p) \\
\Rightarrow & \{\neg p \text{ is regular, so } AF(\neg p) \text{ is regular}\} \\
 & (G \cap H) \models AF(\neg p) \text{ and } (G \cup H) \models AF(\neg p) \\
\equiv & \{\text{Since } AF(\neg p) = \neg EG(p)\} \\
 & (G \cap H) \models \neg EG(p) \text{ and } (G \cup H) \models \neg EG(p)
\end{array}
$$

$\square$

We now consider operators that do not preserve biregularity. In Figure 5.2, p, q, r and s are each process-local state formulae, and hence biregular. Recall that, in order to be biregular, both the formula and its negation must be regular. The following counterexamples, based on the program trace in Figure 5.2, show that the remaining CTL logical and temporal operators do not preserve biregularity.

- **Conjunction**.

  $I \models \neg(q \wedge r)$ and $J \models \neg(q \wedge r)$, but $(I \cup J)$ does not. That is, $(I \cup J) \models (q \wedge r)$. So, $\neg(q \wedge r)$ is not regular.

- **Disjunction**.

  $G \models (p \vee q)$ and $H \models (p \vee q)$, but $(G \cap H) \not\models (p \vee q)$. So, $(p \vee q)$ is not regular.

- EX.

  $I \models EX(\neg s)$ and $J \models EX(\neg s)$, but $(I \cup J) \not\models EX(\neg s)$. So, $EX(\neg s)$ is not regular.

- AX.

  $I \models AX(q)$ and $J \models AX(q)$, but $G = (I \cap J)$ does not. In particular, I is a successor of G that does not satisfy q.

- EU.

  $G \models E[p\ U\ q]$, and $H \models E[p\ U\ q]$, but $(G \cap H)$ does not.

- AU.

  $G \models A[p\ U\ q]$, and $H \models A[p\ U\ q]$, but $(G \cap H)$ does not.

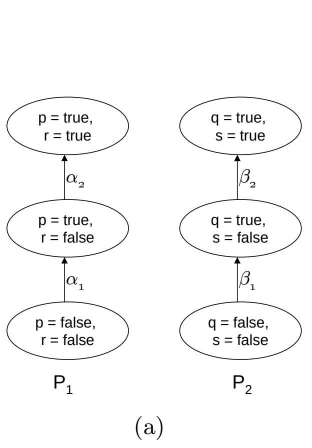
- ER.

  As shown above, $A[p\ U\ q]$ is not meet-closed. Therefore, $\neg E[\neg p\ R\ \neg q]$ is not meet-closed, so ER is also not biregular.

- AR.

  Similarly, as shown above, $E[p\ U\ q]$ is not meet-closed, so $\neg A[\neg p\ R\ \neg q]$ is not meet-closed, hence AR is not biregular.

We next consider the question of which operators preserve regularity.



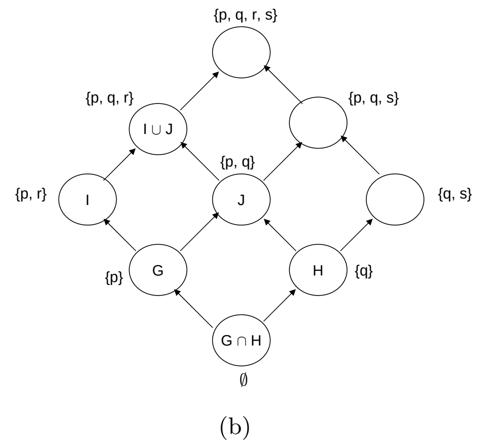


Figure 5.2: (a) Process transition graphs (b) the corresponding down-set lattice.

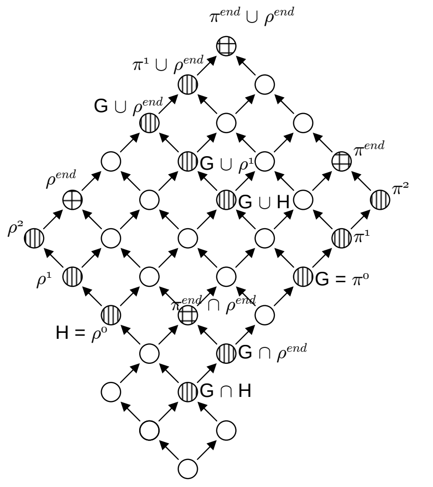
5.3 Preserving Regularity

Garg and Mittal [GM01] showed that the conjunction preserves regularity. That is, if p and q are regular, so is $p \wedge q$. They also showed that disjunction and negation do not preserve regularity. Sen and Garg [SG03a, Sen04] explored the question of which temporal CTL operators preserve regularity. In particular, they showed that the EF, EG and AF operators preserve regularity, while AF, AU, EU, EX and AX do not.

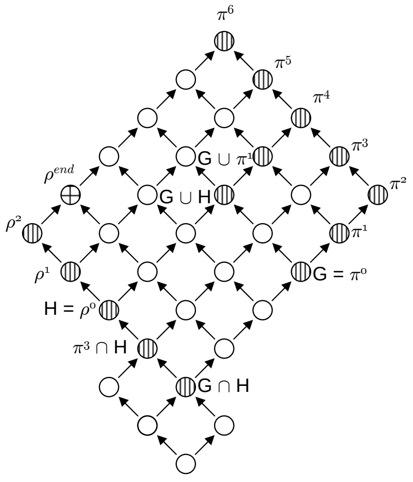
In this section, we show that the operator ER preserves regularity. Also, while the EU operator does not preserve regularity, we show that a variation of it does. In particular, we show that $E[p\ U\ (p \wedge q)]$ is regular when p and q are regular. In most cases, the system specification makes it equally valid to check for $E[p\ U\ (p \wedge q)]$ instead of $E[p\ U\ q]$.

Theorem 5.5. *The following temporal formulae are regular, if p and q are regular:*

- $E[q\ R\ p]$
- $E[p\ U\ (p \wedge q)]$



(a)



(b)

▥ : only p is true   ▤ : only q is true   ⊞ : p and q are both true   ○ : p and q are both false

Figure 5.3: Illustrating the construction of λ and ν in Theorem 5.5. (a) Case 1: $G, H \models E[p\ U\ (p \wedge q)]$ (b) Case 2: $G \models EG(p)$

Proof. We show that $E[q\ R\ p]$ is regular, for regular p and q. Let $G, H \in \mathcal{L}(\sigma)$ be two down-sets such that $G \models E[q\ R\ p]$ and $H \models E[q\ R\ p]$. Then, both G and H must satisfy p. Then, by the meet-closure of p, $(G \cap H) \models p$. Also, by the join-closure of p, $(G \cup H) \models p$.

Recall that $E[q\ R\ p] = E[p\ U\ (p \wedge q)] \vee EG(p)$.

- **Case 1:** Both G and H satisfy $E[p\ U\ (p \wedge q)]$.

  In the lattice $\mathcal{L}(\sigma)$, there exist finite paths π and ρ, starting from G and H respectively, such that $\pi^{end} \models q$ and $\rho^{end} \models q$, where π^{end} and ρ^{end} are the final states on π and ρ, respectively. From the meet- and join-closure of q, $(\pi^{end} \cap \rho^{end}) \models q$, and $(\pi^{end} \cup \rho^{end}) \models q$. We can construct a path λ starting from $(G \cap H)$ as follows:

  $$\lambda = G \cap H, G \cap \rho^1, G \cap \rho^2, ..., G \cap \rho^{end}, \pi^1 \cap \rho^{end}, \pi^2 \cap \rho^{end}, ..., \pi^{end} \cap \rho^{end}$$

  From the properties of set intersection, for each i, λ^i can either be the same as λ^{i-1} or contain one additional event. Eliminating consecutive identical down-sets, we get a valid path in which for each i, λ^i contains one event more than λ^{i-1}. From the meet-closure of p, it follows that λ is a witness for $E[p\ U\ (p \wedge q)]$. Similarly, we can construct ν starting from $(G \cup H)$:

  $$\nu = G \cup H, G \cup \rho^1, G \cup \rho^2, ..., G \cup \rho^{end}, \pi^1 \cup \rho^{end}, \pi^2 \cup \rho^{end}, ..., \pi^{end} \cup \rho^{end}$$

  From the properties of set union, for each i, either ν^i can be the same as ν^{i-1}, or contain one additional event. Eliminating consecutive identical down-sets, one obtains a valid path. From the join-closure of p, it follows that ν is a witness for $E[p\ U\ (p \wedge q)]$.

- **Case 2:** Either G or H satisfies $EG(p)$.

WLOG, let $G \models EG(p)$. Let π be a witness path starting from G, and v be its corresponding transition sequence. We first show that there exists a finite $k \geq 0$ such that $H \subseteq \pi^k$. Let s be the starting state of σ. Let u and w be transition sequences leading, respectively, from s to G and s to H in $\mathcal{L}(\sigma)$. Since $G \models EG(p)$, $u.v$ is a maximal transition sequence of σ, *i.e.*, $\sigma = [s, u.v]$. Therefore, $w \preceq u.v$. By the definition of $\preceq$, there exists a finite prefix u' of $u.v$ such that $u' \equiv w'$ and w is a prefix of w'. Let K be the final state of the transition sequence u'. Recall that H is the final state of the sequence w. Then, we have $H \subseteq K$. Now, K can occur either before or after G in the path corresponding to $u.v$. In either case, $K \subseteq \pi^k$ for some finite $k \geq 0$.

We use the above property to construct a path λ starting from $(G \cap H)$:

$$\lambda = G \cap H, \pi^1 \cap H, \pi^2 \cap H, ..., (\pi^k \cap H = H)$$

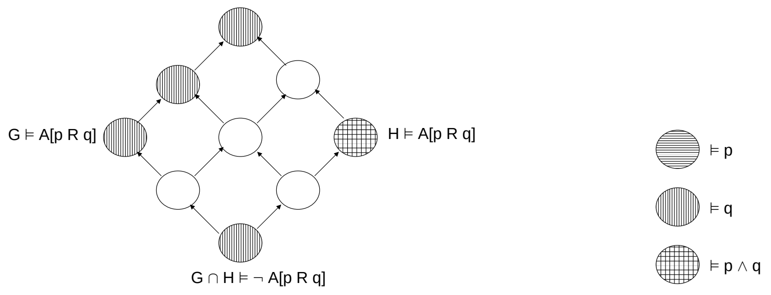
Eliminating consecutive identical down-sets, λ becomes a valid path. Since π is a witness for $G \models EG(p)$, every state along π satisfies p. Also, $H \models p$. Thus, by the meet-closure of p, every state on ρ satisfies p. Let ρ be the witness path for $E[q\ R\ p]$ starting from H. Then, the required witness path for $E[q\ R\ p]$ from $(G \cap H)$ is given by $\lambda.\rho$.

To demonstrate join-closure, we construct the following path ν starting from $(G \cup H)$:

$$\nu = G \cup H, \pi^1 \cup H, \pi^2 \cup H, ...$$

Removing consecutive identical down-sets, ν becomes a valid path. From the join-closure of p, it follows that ν is a witness path for $(G \cup H) \models EG(p)$.

The proof that $E[p\ U\ (p \wedge q)]$ is regular is the same as Case 1 above. $\square$



(a)

Figure 5.4: Counterexample showing that AR does not preserve regularity.

In [Sen04], Sen provided counterexamples to show that the following operators do not preserve regularity: negation, disjunction, AF, EU and AU. Figure 5.4 provides a counterexample showing that the operator AR also does not preserve regularity. In Figure 5.4, G and H both satisfy $A[p\ R\ q]$, but $(G \cap H)$ does not.

5.4 Bibliographic Notes

In [SG03a], Sen and Garg defined a logic called RCTL (Regular CTL), which contains only regular formulae. RCTL is a subset of CTL, in which the temporal operators are limited to EF, AG and EG. Every atomic proposition is required to be regular, and the only logical operator in RCTL is conjunction. In [SG03a], Sen and Garg showed that, for a finite program trace, an RCTL formula can be verified in time that is polynomial in the number of events in the trace. In particular, an RCTL formula f can be verified in $O(|f|.n^2.|E|)$ time, where n is the number of processes in the trace, E is the event set of the trace, and $|f|$ is the length of the formula (*i.e.*, the number of temporal and logical operators in the formula).

Sen and Garg's algorithm works on a partial order representation of a finite trace. In [KG08], we defined a subset of CTL called CETL (Crucial Event Temporal Logic), consisting of the temporal operators EU (albeit, the variation used in

Theorem 5.5) and ER, and the logical operation of conjunction. We presented a model checking algorithm for CETL that exploited the lattice-theoretic properties of regular formulae to achieve state space reduction while also resulting in short error traces. Our approach was applied to an interleaved model of the state space, and is presented in Chapter 8 of this dissertation.

The examination of properties preserved by various temporal operators allows us to build logics that are amenable to state space reduction techniques. This approach is also the basis of the *partial order reduction* (POR) techniques proposed by Peled [Pel93], Valmari [Val91b] and Godefroid [GW94b], among others. Partial order reduction techniques can be used for state space reduction when the properties to be verified are specified in a logic consisting solely of *stutter-invariant* formulae. A formula is said to be stutter-invariant if it cannot distinguish between a sequence of states and any sequence that results from replacing a single occurrence of a state with multiple copies of the same state. A formal definition can be found in [PW97]. Peled and Wilke [PW97] showed that removing the next-time operator from LTL (and/or CTL) produces a stutter-invariant logic. Further comparisons between POR techniques and our lattice-theoretic approach can be found in Chapters 6 and 8.

5.5 Summary

In this chapter, we addressed the question of which CTL logical and temporal operators preserve regularity and/or biregularity. We showed that when p is biregular, so are $\neg p$, $EF(p)$, $AF(p)$, $EG(p)$, and $AG(p)$. The remaining CTL logical and temporal operators do not preserve biregularity. We also showed that when p and q are regular, so are $E[p\ U\ (p \wedge q)]$ and $E[p\ R\ q]$. Previous work by Garg and Mittal [GM01] showed that conjunction preserves regularity, and previous work by Sen and Garg [SG03a] showed that $EF(p)$, $AG(p)$ and $EG(p)$ are regular when p is regular. The remaining CTL operators do not preserve regularity.

Part III

Partial Order Semantics

Chapter 6

Trace Covers

6.1 Introduction

Partial order representations of the state space have been advocated by several researchers as a means of capturing the true concurrency semantics of distributed systems. Examples of partial order representations include Lamport's *happened-before* or *causality* relation [Lam78], Mazurkiewicz's *trace* model [Maz89], Winskel's *event structures* [Win87], Pratt's *pomsets* [Pra86] and C. A. Petri's *Petri nets* [Pet62].

A common observation about distributed systems is that an interleaving model imposes an arbitrary total ordering on concurrent events. To avoid discriminating against any particular ordering, interleaving models represent all possible total orderings of concurrent events. This results in the state space representation being exponential in the number of events. Partial order models avoid this problem by representing the order between events as a poset. Thus, partial order models allow for a compact representation of the state space.

Most commonly-used temporal logics for specifying correctness properties of programs *can*, however, distinguish between different interleavings of concurrent events. Therefore, while a partial order representation of the state space is compact, we also need to avoid state space explosion during verification by avoiding the exploration of all linearizations of the partial order.

A class of techniques known as *partial order reduction* (POR) techniques [Val91a, Val91b, Val93, God91, GW92, GW94b, Pel93, Pel94] capitalizes on logics that cannot distinguish between different interleavings of concurrent events by defining an equivalence relation on the set of all interleavings, based on the formula being verified. The specified formula holds true in one interleaving of an equivalence class iff it holds true in all interleavings. An example of such an equivalence relation is *stuttering-equivalence* (described in Section 6.7 of this chapter). POR techniques still use an interleaving representation of the state space, but combat state space explosion by exploring a reduced set of interleavings, namely, one interleaving per equivalence class. However, this still results in the exploration of multiple interleavings per program trace. That is, the equivalence relation is stronger than trace-equivalence.

Another drawback of interleaving models is that they abstract away the fact that the distributed program in question is composed of independently computing processes, and models the program as purely sequential patterns of events. A partial order representation captures the notion of independently operating agents, making it easier to differentiate between program errors due to race conditions between processes, versus program errors due to a single incorrectly operating process. The ability to communicate this distinction to a programmer can ease the task of debugging distributed systems.

POR techniques, as they are based on an interleaving model, also suffer from the above drawback. Namely, once an interleaving is chosen for inclusion in the reduced state space graph, we lose all concurrency information between the events in the interleaving. Further, the reduced state space is generated with respect to the particular formula being verified. The subsequent verification of a different formula requires the construction of a different (reduced) state space.

6.1.1 Our Contribution

In this chapter, we show how a program can be decomposed into a set of partial orders. In particular, we present a mechanism to represent a (finite-state) program as a finite set of finite trace posets, called a *finite trace cover*. The finite trace cover represents all the reachable states of the program, and maintains all the concurrency information of the original program. In addition, the state space representation is independent of the formula being verified.

Like the POR techniques of [God91, GW92, GW94b, Pel93, Pel94], our approach uses trace semantics. This allows for a direct comparison between our approach and POR techniques. In fact, we exploit previous results from POR techniques to build the trace cover. We first generate a *single* representative transition sequence for each maximal program trace (therby avoiding state space explosion), then use a "vector timestamping" mechanism [Mat89, Fid88] to capture the concurrency information (specifically, the $\rightarrow$ relation) between events in the trace.

We also show how a restricted, but useful, class of formulae can be verified on the finite trace cover, while avoiding state space explosion. Currently, our approach is limited to the verification of formulae of the form $EF(\phi)$, where ϕ does not contain temporal operators. Specifically, these verification algorithms have running time complexity that is polynomial in the number of events in the trace cover. Experimental results are presented, comparing our approach to POR techniques. In our experiments, we detected safety violations in a leader election protocol in 53.53 seconds, compared to POR techniques, which took 547.41 seconds to detect the same violations.

6.2 Trace Covers

Let $States(\sigma)$ denote the set of all reachable states in a trace σ. That is, if $\sigma = [s, v]$ is a trace, then $t \in States(\sigma)$ iff there exists some path in σ that contains t.

Definition 6.1. *A set of traces Δ of a program $P = (S, T, s_0)$ is called a* **trace cover** *iff for every reachable program state $s \in S$, there exists a trace $\sigma \in \Delta$ such that $s \in States(\sigma)$.*

Theorem 6.1. *Given traces σ_1 and σ_2, $\sigma_1 \sqsubseteq \sigma_2 \Rightarrow States(\sigma_1) \subseteq States(\sigma_2)$.*

Proof. Let $t \in States(\sigma_1)$. Then, there exists some transition sequence u of σ_1 such that t occurs along u. In particular, there exists a prefix u' of u such that t is the final state reached after executing the events in u'. From the definition of the subsumes relation in Section 2.3, there exists some transition sequence w of σ_2 such that u' is a prefix of w. Therefore, t occurs while executing the transition sequence w, which means $t \in States(\sigma_2)$. Thus, $t \in States(\sigma_1) \Rightarrow t \in States(\sigma_2)$, which implies that $States(\sigma_1) \subseteq States(\sigma_2)$. □

Theorem 6.1 shows that it is sufficient to consider only traces that are maximal under the subsumes relation when constructing a trace cover.

6.3 Representative Transition Sequences

In this section, we present a method to generate a representative transition sequence for each maximal trace of the program, while avoiding state space explosion. In order to avoid state space explosion, we need to avoid exploring all interleavings of concurrent events. Ideally, we would like to explore only a single interleaving of events per maximal program trace.

In [Kwi89, PP94], it was shown that the set of transition sequences that belong to *maximal* program traces is exactly the same as the set of sequences that satisfy the following constraint:

(Constraint A) If a transition α is enabled at some state of a transition sequence, then a transition that is dependent on α (possibly α itself) must occur later (or immediately) in this sequence.

It was shown in [Pel94] that in order to construct at least one transition sequence per maximal program trace, it is sufficient to explore an ample set $ample(s)$ that satisfies the following condition:

(C1) Along every path starting from s in the full state space graph, a transition that is dependent on a transition from $ample(s)$ cannot be executed without a transition from $ample(s)$ occurring first.

Lemma 6.2. *The transitions in $enabled(s) \setminus ample(s)$ are all independent of those in $ample(s)$.*

Proof. Let $\alpha \in enabled(s) \setminus ample(s)$, and $\beta \in ample(s)$. Assume $(\alpha, \beta) \in D$. Since $\alpha \in enabled(s)$, there must be a path in the full state space graph that starts with α. Thus, there exists a path starting from s in the full state space graph in which a transition (α) that is dependent on a transition (β) in $ample(s)$ occurs before any transition from $ample(s)$ occurs. This contradicts condition (C1). Therefore, α and β must be independent. □

Theorem 6.3.

1. *Every transition sequence generated by Algorithm 6.1 is a valid transition sequence of the input program P.*
2. *Assuming that every queued sequence is eventually explored, Algorithm 6.1 produces a representative transition sequence for each maximal trace of P.*

Proof.

1. Obvious from the BFS construction.
2. The proof is by construction. Let $w = \alpha_1\alpha_2...$ be a representative transition sequence of some maximal trace of P, starting from the state s. Then, w

Algorithm 6.1: trace_cover

```
input  : A program $P$, with initial state $s_0$.
output: A transition sequence per maximal program trace
begin
    $enqueue(s_0, \varepsilon)$ /* (initial state, the empty string) */
    while $queue$ $is$ $not$ $empty$ do
        $(s, \tau) := dequeue()$
        $work\_set := ample(s)$
        while $work\_set \neq \emptyset$ do
            let $\alpha \in work\_set$
            $work\_set := work\_set \setminus \{\alpha\}$
            $t := \alpha(s)$
            $\tau_{new} := \tau.\alpha$
            $enqueue(t, \tau_{new})$
        endw
    endw
end
```

must satisfy Constraint A. We show that Algorithm 6.1 explores (constructs) a sequence that is trace-equivalent to w.

(a) **Case 1:** $\alpha_1 \in ample(s)$.

Then, the algorithm adds α_1 to the sequence τ, in line 10. Thus, the algorithm constructs a prefix of w, and the construction proceeds inductively from the state $\alpha_1(s)$.

(b) **Case 2:** $\alpha_1 \notin ample(s)$.

Let β be some arbitrary transition in $ample(s)$. Then, clearly, $\beta \in enabled(s)$. By Constraint A, w must contain some transition that is dependent on β. However, by condition (C1), a transition that is dependent on β cannot occur in w before some transition from $ample(s)$ occurs in w. Let α_k, where $k \geq 1$, be the first transition in the sequence w that belongs to $ample(s)$. By condition (C1), the events $\alpha_1, \alpha_2, ..., \alpha_{k-1}$ must all be independent of α_k, and thus can all commute with it. Therefore, the sequence $\alpha_k \alpha_1 \alpha_2 ... \alpha_{k-1} \alpha_{k+1} ...$ is trace-equivalent to w, and the algorithm constructs the prefix α_k of this trace-equivalent sequence in line 10. Construction proceeds inductively from the state $\alpha_k(s)$.

$\square$

6.4 Obtaining Posets From Sequences

In the previous section, we presented an algorithm that generates a representative transition sequence per maximal program trace. Recall that each transition sequence of a trace is a linear extension of the poset corresponding to the trace. In this section, given a transition sequence and the dependency relation, we present a method for retrieving the corresponding trace poset.

Our method generalizes the notion of vector timestamps introduced independently by Mattern in [Mat89] and Fidge in [Fid88], as a mechanism for representing the causality relation in a distributed computation. To each event in a trace, we assign an integer vector of dimension n, where n is the number of processes in the program. This integer vector is called a *vector timestamp*, The vector timestamp of an event α is denoted by $\alpha.\nu$, and the i^{th} component of $\alpha.\nu$ is denoted by $\alpha.\nu[i]$.

Given two n-dimensional vector timestamps, $\alpha.\nu$ and $\beta.\nu$, we compare them as follows:

$$\alpha.\nu = \beta.\nu \text{ iff } \forall i : \alpha.\nu[i] = \beta.\nu[i]$$
$$\alpha.\nu \leq \beta.\nu \text{ iff } \forall i : \alpha.\nu[i] \leq \beta.\nu[i]$$
$$\alpha.\nu < \beta.\nu \text{ iff } \alpha.\nu \leq \beta.\nu \text{ and } \alpha.\nu \neq \beta.\nu$$

Let a program P consist of n processes $\{P_1, ..., P_n\}$. We now describe an online mechanism for assigning timestamps to the events in a sequence. We assume that the empty sequence ε contains the empty event ϵ, and $\epsilon.\nu = [0, 0, 0...., 0]$. We assume that every event is dependent on ϵ. When an event α is concatenated to the sequence τ, it is assigned a timestamp as follows.

1. Calculate the set $dep(\alpha)$, where:

$$dep(\alpha) = \{\beta | (\beta \in \tau) \wedge (\alpha, \beta) \in D\}$$

2. For all $j \in \{1...n\}$, set:

$$\alpha.\nu[j] := max\{\beta.\nu[j] | \beta \in dep(\alpha)\}$$

3. Let $\alpha \in P_i$, where $1 \leq i \leq n$. Set $\alpha.\nu[i] := \alpha.\nu[i] + 1$.

To put it simply, when a new event α is added to a transition sequence, we first take the component-wise maximum of all the events that precede α the sequence and that α is dependent on. Then, we increment the component corresponding to the process to which α belongs.

Let τ be a representative sequence of the trace σ_E. The following theorem shows how our timestamping mechanism captures the poset $(E, \rightarrow)$.

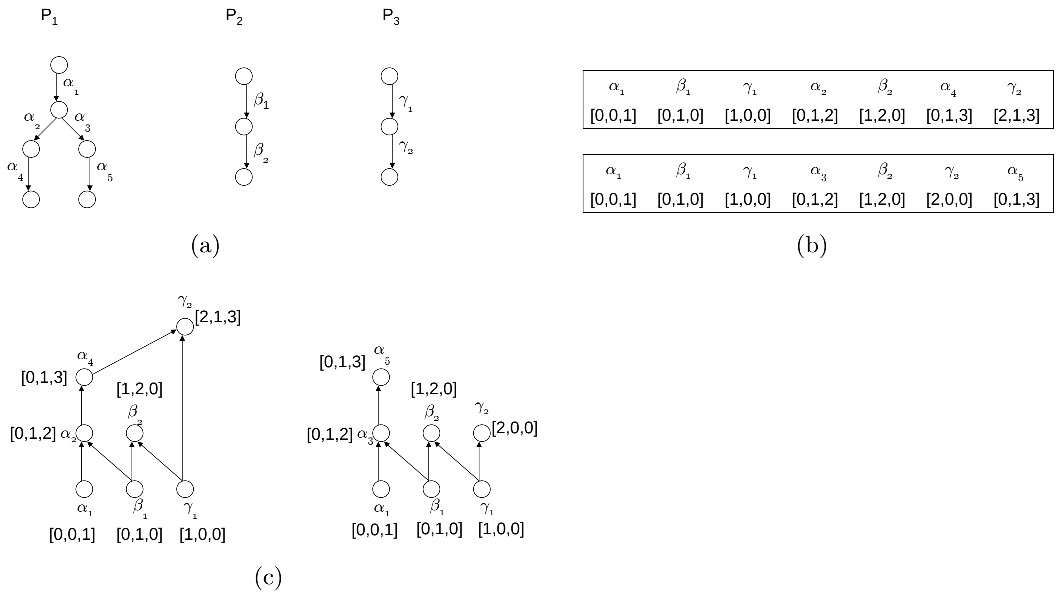
Theorem 6.4. *Given a trace $\sigma = (E, \rightarrow)$, and $\alpha, \beta \in E$:*

$$\alpha \rightarrow \beta \Leftrightarrow \alpha.\nu \leq \beta.\nu$$

Proof. Straightforward from the definition of $\rightarrow$ and the timestamping procedure.

□

Example 6.1. *Figure 6.1(a) shows the process transition graphs for three processes, P_1, P_2, P_3. We assume that all pairs of events on the same process are dependent.*



α_1	β_1	γ_1	α_2	β_2	α_4	γ_2
[0,0,1]	[0,1,0]	[1,0,0]	[0,1,2]	[1,2,0]	[0,1,3]	[2,1,3]

α_1	β_1	γ_1	α_3	β_2	γ_2	α_5
[0,0,1]	[0,1,0]	[1,0,0]	[0,1,2]	[1,2,0]	[2,0,0]	[0,1,3]

Figure 6.1: (a) Process transition graphs for processes P_1, P_2, P_3, (b) representative transition sequences and the vector timestamps assigned based on the dependency relation given in Example 1, and (c) the posets induced by the assigned timestamps.

In addition to these dependencies, we have the following dependencies for events from different processes: $\{(\beta_1, \alpha_3), (\gamma_1, \beta_2), (\beta_1, \alpha_2), (\alpha_4, \gamma_1)\}$. *It is easy to verify that the specified program has two maximal traces, and Figure 6.1(b) shows one interleaving sequence for each maximal trace. Under each interleaving sequence is the corresponding sequence of assigned vector timestamps. These vector timestamps induce a partial order on the events of each sequence. Figure 6.1(c) shows the partial order (trace) corresponding to each interleaving sequence.*

So far in this chapter, we have presented a method to generate a representative transition sequence for each maximal program trace, and a method to timestamp these sequences to capture the partial order relation of their corresponding traces. Note that the transition sequences generated in Algorithm 6.1 may be infinite in length. In the next section, we present a technique for generating a *finite* trace cover for a program. That is, a trace cover consisting of a finite number of traces, each of finite length.

6.5 Finite Trace Covers

For finite-state programs, it is possible to construct a *finite trace cover*, that is, a finite set of traces, where each trace is of finite length, which contains all the reachable states of the program. In this section, we present a modified version of Algorithm 6.1 which generates such a finite trace cover.

Definition 6.2. *Given a poset $(X, \leq)$ and an element $x \in X$, the* **principal down-set** *of x, denoted $\downarrow x$, is defined as:*

$$\downarrow x \overset{def}{\equiv} \{y \in X | y \leq x\}$$

In other words, $\downarrow x$ is the minimum set of events that must occur in any down-set containing x.

Lemma 6.5. *Let $\sigma = (E, \rightarrow)$ be a trace, and G be some down-set of σ that contains the event α. Then, G is reachable from $\downarrow \alpha$.*

Proof. From the definition of principal down-sets, it is clear that $\downarrow \alpha \subseteq G$. Thus, the state corresponding to G is reachable from the state corresponding to $\downarrow \alpha$ in the full state space graph. □

In [McM93], McMillan used Lemma 6.5 to develop a technique for "unfolding" a Petri net into a finite acyclic structure called a *finite complete prefix.* The finite complete prefix contains all the reachable states of the Petri net, and can be used to check reachability properties. McMillan's approach was applied to Petri net semantics. Here, we apply a similar approach to a system model based on trace semantics.

Let α and β be two events such that $\downarrow \alpha$ and $\downarrow \beta$ are different occurrences of the same state in the full state space graph. That is, even though $\downarrow \alpha$ and $\downarrow \beta$ may be different vertices in the lattice $\mathcal{L}(\sigma)$, they lead to the same vertex in the full state space graph. Clearly, any state reachable from $\downarrow \alpha$ is also reachable from $\downarrow \beta$. Therefore, when constructing a finite trace cover, if we explore the states reachable from $\downarrow \alpha$ then it is redundant to explore the ones reachable from $\downarrow \beta$.

Algorithm 6.1 adds events to transition sequences to generate representative transition sequences of maximal program traces. Combined with the timestamping method presented in Section 6.4, Algorithm 6.1 can be viewed as adding events to traces, to eventually obtain the set of all maximal program traces. In the example of the previous paragraph, if Algorithm 6.1 adds an event α to a trace σ_1, then it does not need to add β to any other constructed trace σ_2, because all the states reachable from $\downarrow \beta$ will be part of $States(\sigma_1)$. The event β is called a **cutoff event** [McM93]. If an event β is marked as a cutoff event in σ_2, then no event γ such that $\beta \rightarrow \gamma$ needs to be added to the transition sequence representing σ_2 when constructing a finite trace cover. This is because any down-set that contains γ will correspond to some state that is contained in $States(\sigma_1)$.

In order to safely eliminate cutoff events from a trace, we need to ensure that we explore all the reachable states of *some* "equivalent" event. That is, if some event α is marked as a cutoff event, then all the reachable states from $\downarrow \alpha$ must be explored by some other trace. McMillan [McM93] showed that a sufficient condition for ensuring that no reachable states are ignored is that an event β is marked as a cutoff event iff there exists some event α such that $\downarrow \alpha$ and $\downarrow \beta$ correspond to the same state in the full state space graph and $|\downarrow \alpha| < |\downarrow \beta|$. That is, if an event is marked as a cutoff event, then there exists some "equivalent" event with a smaller principal down-set. Since principal down-sets cannot be made arbitrarily small (they are well-founded sets), there exists some equivalent event that cannot be marked as a cutoff event.

We can use cutoff events to prune the sequences generated by Algorithm 6.1. Recall that each pair of events from the same process are dependent. Therefore, if two events α, β belong to the same process and α occurs before β in a transition sequence, then $\alpha \rightarrow \beta$. If an event $\alpha \in P_i$ is identified as a cutoff event while constructing the sequence τ, further events from process P_i can be ignored when constructing sequences that contain τ as a prefix. That is, if α is identified as a cutoff event, then P_i is no longer considered an **eligible** process while picking an ample set to extend the sequence τ. Let $\mathit{eligible}(s)$ be the set of all enabled events from eligible processes at a state s in the current trace. Then, in addition to (C1), we impose the following constraint on $\mathit{ample}(s)$ while constructing a finite trace cover.

(C2) $\mathit{ample}(s) \subseteq \mathit{eligible}(s)$.

Algorithm 6.2 constructs a finite trace cover based on cutoff events. The key difference between Algorithms 6.1 and 6.2 is that Algorithm 6.2 chooses ample sets only from the set of eligible processes. In the worst case, if no reduction is found, $\mathit{ample}(s) = \mathit{eligible}(s)$. Initially, in line 3, all processes are considered eligible. If an event α is identified as a cutoff event, then the corresponding process is removed from the set of eligible processes (lines 15-17). Each event in the ample set is appended to the current transition sequence (lines 10-14), and is queued back for further exploration (line 21).

Two sets of traces are said to be **state-equivalent** iff they have the same set of reachable states. For a given program P, let Δ denote the trace cover (set of traces) produced by Algorithm 6.1, and Γ represent the set of traces produced by Algorithm 6.2.

Theorem 6.6. *Γ is state-equivalent to Δ.*

Proof. It is obvious that the set of reachable states in Γ is a subset of the set of

Algorithm 6.2: finite_trace_cover

```
input  : A program $P$, with initial state $s_0$.
output: Set of transition sequences corresponding to a finite trace cover
begin
    /* All processes are initially eligible */
    $eligible := \{P_1, P_2, ..., P_n\}$
    /* (initial state, the empty string, eligible procs) */
    $enqueue(s_0, \varepsilon, eligible)$
    while $queue$ is not empty do
        $(s, \tau, eligible) := dequeue()$
        /* ample set events restricted to eligible procs */
        $work\_set := ample(s)$
        while $work\_set \neq \emptyset$ do
            let $\alpha \in work\_set$
            $work\_set := work\_set \setminus \{\alpha\}$
            $t := \alpha(s)$
            $\tau_{new} := \tau.\alpha$
            if $\alpha$ is a cutoff event then
                let $\alpha \in P_i$
                $eligible_{new} := eligible \setminus P_i$
            else
                $eligible_{new} := eligible$ /* unchanged */
            endif
            $enqueue(t, \tau_{new}, eligible_{new})$
        endw
    endw
end
```

reachable state in Δ. We show the converse. Let G be a down-set belonging to some trace in Δ, such that G is not a down-set of any trace in Γ. Then, G contains a cutoff event, say α. Thus, there exists some event β belonging to a trace in Γ such that $|\downarrow \beta| < |\downarrow \alpha|$, and $\downarrow \alpha$ and $\downarrow \beta$ correspond to the same state in the full state space graph of the program. Then, Δ contains a down-set $H = \downarrow \beta \cup (G \setminus \downarrow \alpha)$, such that H and G correspond to the same state in the full state space graph. Note that $|H| < |G|$ because $|\downarrow \beta| < |\downarrow \alpha|$.

If H is also not a down-set of any trace in Γ then, by similar reasoning, there exists some down-set I belonging to a trace in Δ such that G, H and I all correspond to the same state in the full state space graph. Also, $|I| < |H| < |G|$. If I is also not a down-set of any trace in Γ, then we iterate this process of finding an "equivalent" down-set again. We cannot iterate infinitely because the order $<$ on the size of down-sets is well-founded. Therefore, there must exist some down-set J in Γ that corresponds to the same state as G. Thus, each reachable state in Δ is also reachable in some trace of Γ. □

By Theorem 6.6, Algorithm 6.2 also produces a trace cover for the given program. It now remains to be shown that it produces a *finite* trace cover.

Theorem 6.7. *1. Every trace in Γ is of finite length.*

2. There are a finite number of traces in Γ.

Proof. 1. Let N be the total number of distinct states in the given finite state program. Let $w = \alpha_1\alpha_2....$ be a transition sequence, starting from the initial state s_0, produced by Algorithm 6.2. We show that w cannot be of infinite length. Consider the first $N+1$ events in this sequence. Since there are only N states, there exist events α_i and α_j in w, where $1 \leq i < j \leq N+1$, such that the state corresponding to the down-set $\downarrow \alpha_i$ is the same as the state corresponding to $\downarrow \alpha_j$. Also, since w is a linearization of the trace partial order on $\alpha_1, \alpha_2...$, and $i < j$, we have $| \downarrow \alpha_i| < | \downarrow \alpha_j|$. Thus, α_i would be recognized as a cutoff event. Therefore, the length of any transition sequence produced by Algorithm 6.2 cannot be more than $N+1$.

2. Follows from the fact that the length of any trace in Γ is bounded by $N+1$, and $|enabled(s)|$ is finite for each state s.

□

In this section, we presented a technique to obtain a finite trace cover for a given finite-state program. The finite trace cover is a set of traces represented in partial order form, through the use of vector timestamps. In the next section, we discuss how we can apply lattice theory to develop efficient model checking algorithms for certain classes of predicates, on a finite trace cover.

6.6 Model Checking on Finite Trace Covers

The finite trace cover generated by Algorithm 6.2 is a set of finite traces that contains all the reachable states of the program. The **reachability** problem on a finite trace is defined as follows:

Reachability on a finite trace: Given a formula ϕ and a finite trace $\sigma = [s, v]$, does there exist some $t \in \mathit{States}(\sigma)$ such that $t \models \phi$? In other words, does $s \models EF(\phi)$?

In [CG95], it was shown that the reachability problem is NP-complete in the number of events in the trace for a general boolean formula ϕ. However, there are classes of predicates for which reachability can be solved in time that is polynomial in the number of events in the trace. Examples of such predicate classes were discussed in Section 3.3. Clearly, if an efficient algorithm exists for deciding reachability for a single finite trace, we can repeatedly invoke this algorithm on each trace in the finite trace cover. In this way, we can decide if any reachable state of the given program satisfies the predicate. In the following sections, we discuss two useful classes of predicates for which efficient reachability algorithms exist for a finite trace. We implemented these reachability detection algorithms, together with our algorithm for generating finite trace covers, in a model checking tool. Experimental results are provided in Section 6.8.

6.6.1 Meet-closed predicates

In Section 3.4, we introduced meet-closed formulae and the notion of crucial events. In Section 3.4.2, we presented Chase and Garg's algorithm [CG95] for determining, given a finite trace and a meet-closed formula, whether any reachable state of the trace satisfies the formula. In this section, we show how Chase and Garg's algorithm can be applied towards model checking meet-closed formulae in a program represented by a finite trace cover.

We restrict ourselves to checking formulae of the kind $EF(\phi)$, where ϕ is a conjunction of process-local state formulae, that is, ϕ is of the form $p_1 \wedge p_2 \wedge ...p_m$, where each p_i is a process-local state formula. Recall, from Table 5.1 in Chapter 5, that process-local state formulae are regular, and conjunction preserves regularity. Thus, ϕ is regular, and hence meet-closed.

Recall, from the discussion in Section 3.4.2, that given a trace $\sigma = [s, v]$ with E as its set of events, Chase and Garg's algorithm can determine whether $s \models EF(\phi)$ in time that is polynomial in E iff the crucial event from any given state can be identified in time that is polynomial in E. In the case where ϕ is a conjunction of process-local state formulae, if a given state t does not satisfy ϕ, then one of the conjuncts must be $false$ in that state. That is, there is some i such that p_i is $false$ in t. If p_i is a process-local state formula on process P_j, then clearly we must execute an event from P_j in order to turn p_i $true$. So, the next event in the trace that belongs to P_j is a crucial event. Clearly, we can identify the $false$ conjunct in $O(m)$ time if there are n conjuncts in ϕ. In that case, Chase and Garg's algorithm (Algorithm 3.1) for detecting whether $s \models EF(\phi)$ has running time complexity $O(m.|E|)$.

In order to decide whether any reachable state of the given program satisfies ϕ, we simply invoke Chase and Garg's algorithm on each trace in the finite trace cover. This restricted application of Chase and Garg's algorithm to verifying conjunctions of process-local state formulae was presented by Garg and Waldecker in [GW94a].

6.6.2 0-1 sum predicates

Another useful class of predicates are those of the form $x_1 + x_2 + ... + x_n > k$, where each x_i is a local variable on some process, and k is a constant. Such predicates were first introduced in [TG93], where they were called "relational predicates". The term "bounded-sum predicates" was used to describe them in [CG95]. A special case of bounded-sum predicates is where each x_i can only be equal to either 0 or 1. Such predicates are called **0-1 sum** predicates. 0-1 sum predicates can be used to detect mutual exclusion violation ($EF(\sum_i incs_i > 1)$). Or, to detect if there are more than k copies of a k-licensed software in use at once ($EF(\sum_i in\_use_i > k)$). In [TG97], it was shown that the problem of deciding whether any reachable state of a finite trace satisfies a given 0-1 sum predicate can be reduced to the problem of finding the **width** of a poset. Given a poset $(X, \leq)$, two elements $x, y \in X$ are said to be **incomparable** iff $x \not\leq y$ and $y \not\leq x$. The **width** of a poset $(X, \leq)$ is equal to the cardinality of the largest subset of A of X such that every pair of distinct elements x and y in A are incomparable.

We show how to derive a partial order relation from the trace poset (the $\rightarrow$ relation) such that the problem of verifying 0-1 sum predicates becomes equivalent to finding the width of the (derived) poset.

An event $\alpha \in P_i$ is called a *local event* iff it affects only the local variables of P_i (including the program counter at P_i). All other events are called *non-local events*. That is, the set of events E of a trace is partitioned into the set of local events, E_L, and the set of non-local events, E_{NL}. We now split each event $\beta \in E_{NL}$ into two sub-events, β_{nl} and β_l, where β_{nl} affects only non-local variables (including message channels), and β_l affects only local variables, including the program counter. Note that β_l is a local event, but is not a member of E_L. β can now be considered the sequential composition of β_{nl} and β_l, *i.e.*, $\beta \equiv \beta_{nl}.\beta_l$. The splitting process transforms E_{NL} into a set of sub-events, $\hat{E}_{NL}$.

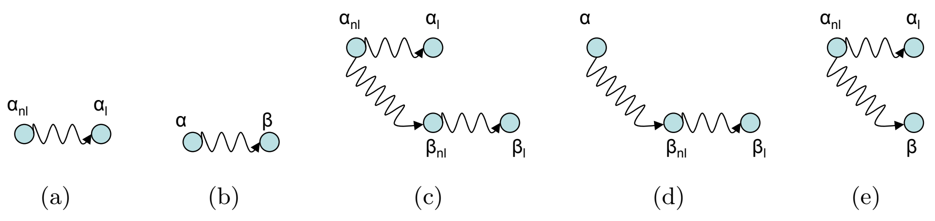


Figure 6.2: The relation $\rightsquigarrow$. (a) $\alpha \in E_{NL}$. (b)$\alpha, \beta \in E_L$, (c) $\alpha, \beta \in E_{NL}$, (d) $\alpha \in E_L, \beta \in E_{NL}$, and (e) $\alpha \in E_{NL}, \beta \in E_L$. Note that in (b) - (e), we also have $\alpha \rightarrow \beta$.

Let $\hat{E} = E_L \cup \hat{E}_{NL}$. We now transform the trace poset $(E, \rightarrow)$ into another poset $(\hat{E}, \rightsquigarrow)$, where the relation $\rightsquigarrow$ is the *smallest* transitive relation that satisfies each of the following (as shown in Figure 6.2):

(a) $\alpha \in E_{NL} \Rightarrow \alpha_{nl} \rightsquigarrow \alpha_l$

(b) $(\alpha, \beta \in E_L) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha \rightsquigarrow \beta$

(c) $(\alpha, \beta \in E_{NL}) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha_{nl} \rightsquigarrow \beta_{nl}$

(d) $(\alpha \in E_L) \wedge (\beta \in E_{NL}) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha \rightsquigarrow \beta_{nl}$

(e) $(\alpha \in E_{NL}) \wedge (\beta \in E_L) \wedge (\alpha \rightarrow \beta) \Rightarrow \alpha_{nl} \rightsquigarrow \beta$

For any $H \subseteq E$, let $\hat{H}$ denote the "expanded" set of events obtained by splitting the non-local events of H. That is,

$$\hat{H} \overset{def}{\equiv} \{\alpha | \alpha \in (H \cap E_L)\} \cup \{\alpha_{nl}, \alpha_l | \alpha \in (H \cap E_{NL})\}$$

Let $frontier(\hat{H})$ denote the set of maximal events, under $\rightsquigarrow$, from each process P_i in $\hat{H}$:

$$frontier(\hat{H}) \overset{def}{\equiv} \{\alpha | \alpha \in (P_i \cap \hat{H}) \wedge (\nexists \beta \in (P_i \cap \hat{H}) :: \alpha \rightsquigarrow \beta)\}$$

That is, $frontier(\hat{H})$ contains the "latest" event in $\hat{H}$ from each process P_i. From Figure 6.2, it is clear that $frontier(\hat{H})$ can contain only local events.

The following lemma is proved in [Mat89] and in [SL85].

Lemma 6.8. *G is a down-set of $(E, \rightarrow)$ iff:*

$$\forall \alpha, \beta \in frontier(\hat{G}) : (\alpha \not\rightsquigarrow \beta) \wedge (\beta \not\rightsquigarrow \alpha)$$

Let G be a down-set of $(E, \rightarrow)$ such that $G \models \varphi$. Let α_j be the (local) event from P_j in $frontier(\hat{G})$. Let $\ell(\alpha_j)$ denote the local state (*i.e.*, valuation of local variables) on P_j reached upon execution of α_j. Since $G \models \varphi$, there must exist a set Π of $(k+1)$ processes such that $\forall P_j \in \Pi$, $(x_j = 1)$ in the local state $\ell(\alpha_j)$. By Lemma 6.8, $\forall i, j \in \Pi : (\alpha_i \not\rightsquigarrow \alpha_j) \wedge (\alpha_j \not\rightsquigarrow \alpha_i)$.

Let $\mathcal{E} = \bigcup_i \{\alpha_i \in P_i | (x_i = 1) \text{ in } \ell(\alpha_i)\}$ That is, $\mathcal{E}$ is the set of all events that lead to a local state in which any x_i is set to 1. Thus, in order to detect $EF(\varphi)$, we simply need to determine whether the poset $(\mathcal{E}, \rightsquigarrow)$ has width greater than k, where $(\mathcal{E}, \rightsquigarrow)$ is the sub-poset of $(\hat{E}, \rightsquigarrow)$ induced by the relation $\rightsquigarrow$ on the set $\mathcal{E}$.

Tomlinson and Garg [TG97] presented an algorithm that solves this problem in $O(k.m.n(k + \log n))$ time, where $m = |\mathcal{E}|$ and n is the number of processes in the program. We can invoke Tomlinson and Garg's algorithm on each trace in the finite trace cover in order to determine if any reachable program state satisfies a given 0-1 sum predicates.

6.7 Comparison to POR Techniques

Our approach based on finite trace covers avoids state space explosion by exploring only a single path through each program trace to build a partial order model of the state space. That is, it uses a true partial order semantics. A class of techniques known in the literature as **partial order reduction** (POR) applies a similar notion to an interleaved model of the state space to cleverly obtain reduced state space graphs. This class of techniques also explores a subset of the set of all enabled

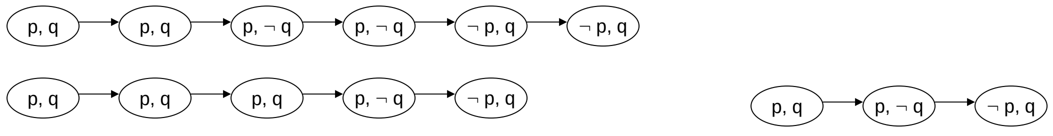


Figure 6.3: (a) Two stuttering-equivalent sequences, and (b) their corresponding collapsed sequence.

events at each state in order to construct a reduced state space graph. The term "POR techniques" encompasses the *stubborn set* method of Valmari [Val91a, Val91b, Val93], the *persistent sets* method of Godefroid and Wolper [God91, GW92, GW94b], and the *ample set* method of Peled [Pel93, Pel94]. These works contain similar ideas, although they differ in some details of choosing the subset of enabled events for reduction.

Unlike our approach, the amount of state space reduction achieved by POR techniques is sensitive to the property being verified. This is because POR techniques use the notion of *stuttering-equivalence* to generate a reduced state space graph. Assume the property being verified involves the subset of variables V' from the set of all variables of the program. A function that assigns a valuation to the variables in V' is called a *label*. Two states are said to be identically *labeled* if they have the same valuation for the variables in V'. Given a sequence of labeled states (path), any sub-sequence of consecutive identically-labeled states is called a *stutter*. A labeled sequence can be "collapsed" by replacing each stutter by a single state with the same label as the states in the stutter. Two state sequences are said to be *stuttering-equivalent* iff they can be collapsed into the same labeled state sequence (see Figure 6.3).

Peled and Wilke showed [PW97] that any LTL property expressible without the next-time operator X (this subset of LTL is called LTL$_{-X}$) cannot distinguish between stuttering-equivalent paths. That is, if π_1 and π_2 are two stuttering-equivalent state sequences, then an LTL$_{-X}$ formula ϕ holds in π_1 iff it holds in π_2. POR techniques exploit this observation to verify LTL$_{-X}$ formulae by constructing a reduced state space graph which includes at least one stuttering-equivalent path for each path in the full state space graph. In order to perform this reduction, they use the notion of *invisibility* of transitions. A transition is said to be *invisible* with respect to a given formula if its execution does not change the value of any variable used in the formula. That is, a transition α is said to be invisible with respect to a formula if s and $\alpha(s)$ have the same label, for every state s such that $\alpha \in \mathit{enabled}(s)$. A transition that is not invisible is said to be *visible*.

When choosing ample sets for exploration, POR techniques impose the following constraint (in addition to (C1) from Section 6.3):

Invisibility constraint: If $\mathit{ample}(s) \neq \mathit{enabled}(s)$, then every transition $\alpha \in \mathit{ample}(s)$ must be invisible.

As a result of the invisibility constraint, the reduction achieved by POR techniques is directly proportional to the number of invisible transitions in the program. Experimental results [CGMP99, PVK01] show that the effectiveness of reduction diminishes rapidly with an increase in the number of visible events. In contrast, our approach does not consider the invisibility of transitions, neither when constructing the finite trace cover, nor when performing model checking on the resulting partial order representation of the state space. As the experimental results presented in the next section show, this can result in significantly greater state space reduction than POR techniques.

6.8 Implementation and Experimental results

We implemented the approach presented in this chapter as an extension to the popular model-checker SPIN [Hol03, Hol07]. We chose SPIN for our implementation because it a widely-used software verification tool, and is especially effective for the verification of concurrent and distributed systems. Inspiring applications of SPIN have included the verification of mission-critical software for a number of space missions by NASA, including Deep Space 1 [HLP01], Cassini [SECH98], and the Mars Exploration Rovers [HJ04]. SPIN won the prestigious ACM System Software Award for 2001.

The input language for SPIN is called PROMELA (PROcess MEta LAnguage). The protocol to be verified is specified in the form of a PROMELA program. The syntax and semantics of PROMELA are documented in [Hol03, Hol91]. SPIN implements Peled's partial order reduction technique based on ample sets. The algorithm used is described in [HP95]. As part of its POR implementation, SPIN provides a mechanism for constructing ample sets that satisfy condition (C1) presented in Section 6.3. We modified this implementation to allow restricting the choice of ample sets to events from eligible processes, rather than all processes, in order to satisfy condition (C2) from Section 6.5.

For checking safety properties, SPIN allows a choice of using either breadth-first search (BFS) or depth-first search (DFS) for state space exploration. As the algorithms presented in this chapter use BFS, we compared our implementation against that of BFS search with POR in SPIN. Note that our algorithms can easily be adapted to use DFS, as well.

Our implementation is called TC-SPIN (”Trace Cover” SPIN). The experimental testbed was a single-CPU 2.8 GHz Intel Pentium 4 machine with 512 MB of memory, running Red Hat Enterprise Linux WS Release 4. Tables 6.1 and 6.2 present our experimental results from the verification of the following three protocols:

- Chandy and Misra's distributed dining philosophers protocol [CM84], with six philosophers ($N = 6$). We checked for the safety property that no pair of neighboring philosophers can ever eat simultaneously.

Protocol	Formula	Tool	Time (sec)	States	Memory (MB)
Dining philosophers	$EF(eating[i] \wedge eating[(i+1) mod N])$	SPIN, no reduction	***	***	***
		SPIN, POR	759.71	2116120	439.12
		TC-SPIN	0.03	83	1.25
Leader election	$EF(nr\_leaders > 1)$	SPIN, no reduction	***	***	***
		SPIN, POR	777.24	238569	64.74
		TC-SPIN	0.05	187	2.65
Mutual exclusion	$EF(incs > 1)$	SPIN, no reduction	25.31	652365	349.82
		SPIN, POR	2.51	46880	26.24
		TC-SPIN	0.05	187	2.65

*** denotes "ran out of memory"

Table 6.1: Experimental results in the absence of errors in the verified protocols.

- Dolev, Klawe and Rodeh's leader election protocol on a unidirectional ring [DKR82] of six processes. We used random initialization to assign id's to the processes in the ring. The safety property to be verified was that there was never more than one leader in the ring.

- Ricart and Agarwala's distributed mutual exclusion protocol [RA81] on five processes. The safety property to be checked was that there was at most one process in the critical section at any time.

The results in Table 6.1 are from verification runs where the protocols had no errors in them. For the second set of results, reported in Table 6.2, we introduced errors (safety violations) in each of the protocols, and measured the time and memory required to find these errors. In all our experiments, we compiled the verifier with the SPIN options -DBFS (for breadth-first search), and -DXUSAFE (to facilitate p.o. reduction). For SPIN, the safety properties were specified through a simple *assert*() statement placed in a separate monitor process. For TC-SPIN runs,

Protocol	Formula	Tool	Time (sec)	States	Memory (MB)
Dining philosophers	$EF(eating[i] \wedge eating[(i+1) mod N])$	SPIN, no reduction	41.86	1141680	257.05
		SPIN, POR	10.22	170619	43.34
		TC-SPIN	0.03	81	1.25
Leader election	$EF(nr\_leaders > 1)$	SPIN, no reduction	***	***	***
		SPIN, POR	547.41	159750	44.77
		TC-SPIN	53.53	87435	69.247
Mutual exclusion	$EF(incs > 1)$	SPIN, no reduction	19.61	510828	276.61
		SPIN, POR	1.59	26126	15.39
		TC-SPIN	0.01	181	2.65

*** denotes "ran out of memory"

Table 6.2: Experimental results in the presence of safety violations in the verified protocols.

we specified our predicates in a different file. Consequently, for TC-SPIN runs only, we had to disable SPIN's dataflow optimizations (-o1) during verifier generation because variables that were flagged as being "dead" by the SPIN preprocessor were actually being read in our predicates file, and used by our verification algorithms.

The results show that, while POR techniques do result in significant state space reduction compared to exhaustive state space search, the invisibility constraint (explained in Section 6.7) in POR techniques gives a greater advantage to TC-SPIN. In particular, the protocols being verified had a significant percentage of visible transitions, which greatly hampered the effectiveness of choosing ample sets using POR techniques.

On the downside, the overhead of storing vector timestamps for each event in the finite trace cover can constitute a significant memory overhead for TC-SPIN compared to SPIN. That is, for the same number of states explored, TC-SPIN consumes significantly more memory. When the number of visible transitions in a program is low, this gives the advantage to POR techniques, compared to our approach. For example, we verified a leader election protocol in 75.02 seconds, whereas partial order reduction techniques verified the same protocol in 777.24 seconds.

6.9 Bibliographic Notes

The work presented in this chapter was published in [KG05a].

The use of vector timestamps for capturing the partial order relation of a trace was pioneered by Fidge [Fid88] and Mattern [Mat89]. Since then, vector timestamps have been in widespread use in algorithms for distributed debugging [Fid88, GW94b], distributed simulation [Mat93, DWG97], and distributed recovery [SY85], among other applications. In [CB91], Charron-Bost showed that, in general, a trace consisting of n processes requires an integer vector of dimension at least n to capture the partial order relation (in the worst case). Garg and Skawratonand [GS01] showed the relation between vector timestamps and the *dimension* of a poset. The dimension of a poset is the smallest number of total orders whose intersection gives rise to the poset [DM41]. In [GS01], it was shown that, in order to capture the causality relation in a trace poset, it is necessary and sufficient to use an integer vector of dimension equal to the string dimension of the poset. In [AG05], Agarwal and Garg introduced the concept of *chain clocks*, which can be viewed as vector timestamps of variable dimension, where the *upper bound* on the vector dimension is equal to the number of processes. They showed that, in most real applications, the use of chain clocks results in far less memory consumption than n-dimension vector timestamps. Chain clocks can be applied to our finite trace cover algorithm to reduce memory consumption.

In [ERV96], Esparza *et al.* improved upon McMillan's [McM93] algorithm for identifying cutoff events, with the aim of reducing the size of the finite complete prefix. McMillan proposed using the $<$ relation on size of the "local configuration" of a Petri net marking to decide which events could be marked as cutoff events. This is analogous to the approach we used, which is the $<$ relation on the size of the principal down-set. Esparza *et al.* showed the sufficient conditions that such an order relation had to satisfy, and that the $<$ order on the size of the configuration was only one such relation satisfying these conditions. In particular, they showed the existence of weaker order relations, which could be used to reduce the size of the finite complete prefix. The results from [ERV96] can also be used to generate a smaller trace cover.

Chase and Garg [CG95] and, independently, Groselj [Gro93] presented an algorithm to detect reachability for bounded-sum predicates based on max-flow techniques. This algorithm works on the poset representation of a finite trace, and has a running time complexity of $O(|E|^2.n^2.log(|E|.n))$, where E is the set of events of the trace, and n is the number of processes. Thus, reachability of bounded-sum predicates can also be verified efficiently using finite trace covers.

6.10 Summary

In this chapter, we presented a method to decompose a program into a finite set of trace posets, while avoiding state space explosion. We presented verification algorithms that could be used for deciding reachability for some classes of formulae. Experimental results were presented, comparing our method against POR techniques. The results showed that we can achieve far greater state space reduction than POR techniques, due to the fact that we are not restricted by the invisibility constraint. For example, we verified a leader election protocol in 75.02 seconds, whereas partial order reduction techniques verified the same protocol in 777.24 seconds.

Chapter 7

Predicate Filtering

7.1 Introduction

In Chapter 6, we presented algorithms to efficiently verify restricted classes of formulae on a partial order representation of the state space. However, the techniques presented so far do not improve the efficiency of model checking for formulae that do not belong to one of our "efficient" predicate classes.

In this chapter, we discuss a technique called **predicate filtering**, which can be used for state space reduction for checking reachability for an extended class of formulae. Predicate filtering was introduced under the name "computation slicing" by Garg and Mittal in [GM01]. We use the term "predicate filtering" to avoid confusion with the completely distinct notion of **program slicing** [Wei81]. The distinction between program slicing and computation slicing is discussed in [MG05].

Given a finite trace σ and a predicate ϕ, we "filter" the trace w.r.t. ϕ, and produce a **filtrate**. The filtrate is the *smallest* trace, *i.e.*, the one with the fewest states, that contains *all* the states of σ that satisfy ϕ, while eliminating states that do not satisfy ϕ[1]. The process of producing the filtrate from a trace is called **predicate**

[1]As we will see in Section 7.3, depending on the predicate ϕ, the filtering process may not eliminate *all* the states of σ that do not satisfy ϕ.

filtering.

The filtrate of a trace w.r.t. ϕ can be computed in polynomial time if and only if reachability for ϕ can be decided in polynomial time on the trace [MSGA04]. Predicate filtering can be used for state space reduction as follows. Suppose we want to determine whether any reachable state of a program satisfies some predicate ϕ. Further, suppose that ϕ does not belong to any class of predicates for which efficient verification algorithms exist. We can specify a predicate ψ that is *weaker* that ϕ, such that ψ belongs to a class for which reachability can be decided efficiently. By *weaker*, we mean that whenever a state satisfies ϕ, it also satisfies ψ.

Now, we can filter each trace σ in the finite trace cover representation of a program w.r.t. ψ, producing a set of filtrates. Recall that a filtrate is the *smallest* trace containing all the states that satisfy ψ. Therefore, the filtrate of σ w.r.t. ψ should have far fewer states than the original trace σ. Traditional model checking techniques can now be used to verify ψ on these smaller traces.

7.1.1 Our Contribution

Predicate filtering has previously been used for state space reduction during the verification of finite traces in [MG03, SG03a], among others. However, these applications were limited to single, finite execution traces. In this chapter, we apply predicate filtering to reducing state space explosion in finite-state programs.

In our experiments, we could verify a leader election protocol using predicate filtering by constructing only one-third as many states as constructed by SPIN using partial order reduction.

7.2 Background

In this section, we present the relevant background material required for presenting the predicate filtering technique.

We represent the meet operation of a lattice by $\sqcap$, and the join operation by $\sqcup$.

Definition 7.1. *Let L be a lattice and $M \subseteq L$ be a non-empty subset of L. Then, M is a* **sublattice** *of L if:*

$$\forall a, b \in M : (a \sqcap b) \in M \text{ and } (a \sqcup b) \in M$$

where the $\sqcap$ and $\sqcup$ operations are the meet and join operations of L.

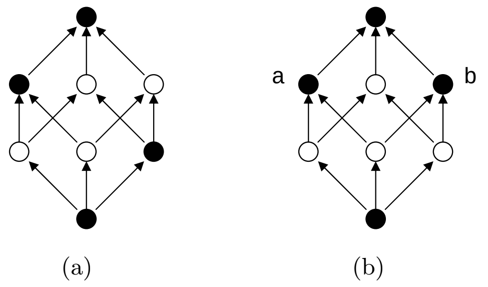


Figure 7.1: The shaded elements in (a) form a sublattice, while those in (b) do not.

Example 7.1. *In Figure 7.1(a), the shaded elements form a sublattice of the lattice shown. Figure 7.1(b) shows that a subset M of a lattice L can be a lattice in its own right, without being a sublattice of L. In particular, $a \sqcap b$ is not in M.*

Let 0 represent the least element of a lattice L.

Definition 7.2. *An element $x \in L$ is* **join-irreducible** *if:*

1. *$x \neq 0$, and*
2. *$\forall\, a, b \in L : x = a \sqcup b \implies (x = a) \vee (x = b)$.*

In simple terms, an element of L is join-irreducible if it is not the least element of L, and it cannot be expressed as the join of two elements, both different from itself. Pictorially, in a finite lattice, an element is join-irreducible iff it has exactly one incoming edge. Figure 7.2 shows an example. We will use the notation $J(L)$ to denote the set of join-irreducible elements of a lattice L.

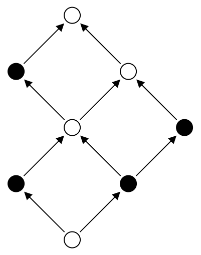


Figure 7.2: The shaded elements are join-irreducible.

A **meet-irreducible** element is defined dually. That is, an element of L is meet-irreducible if it is not the greatest element of L, and it cannot be expressed as the meet of two elements, both different from itself. Pictorially, in a finite lattice, a meet-irreducible element is one that has exactly one outgoing edge.

For a poset $P = (X, \leq)$, let $\mathcal{O}(P)$ represent the set of all down-sets of P. Recall that $(\mathcal{O}(P), \subseteq)$ is a lattice. A well-known result in lattice theory states that $(\mathcal{O}(P), \subseteq)$ is actually a special kind of lattice, called a *distributive* lattice [DP90].

Definition 7.3. *A lattice L is said to be* **distributive** *if it satisfies the following distributive law:*

$$\forall a, b, c \in L : a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$$

Predicate filtering is based on the following theorem by Birkhoff [DP90], which shows that any finite distributive lattice can be represented succintly by a poset.

Theorem 7.1. [Birkhoff's Representation Theorem for Finite Distributive Lattices]

Let L be a finite distributive lattice. Then, the map $f : L \to \mathcal{O}(J(L))$ defined by

$$f(a) = \{x \in J(L) | x \leq a\}$$

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

$$g(a) = \{x \in P | x \leq a\}$$

is an isomorphism of P onto $J(\mathcal{O}(P))$.

Birkhoff's Theorem establishes a 1-1 correspondence between a finite poset and a finite distributive lattice. Given a finite poset P, we can obtain the finite distributive lattice by considering the set of all its down-sets, $\mathcal{O}(P)$. Given a finite distributive lattice L, we can recover the corresponding finite poset by considering only the join-irreducible elements, $J(L)$. The number of join-irreducible elements in a lattice is typically exponentially smaller than the total number of elements in the lattice. For a finite distributive lattice, the number of join-irreducible elements is exactly equal to the length of the longest chain in the lattice [DP90]. In the case of a trace σ, it is easy to see that the number of join-irreducible elements is bounded by $|E|$, where E is the set of events in the trace. The size of the corresponding lattice $\mathcal{L}(\sigma)$ is bounded by $2^{|E|}$.

In the next section, we show how predicate filtering makes use of Birkhoff's Theorem (Theorem 7.1) to achieve state space reduction during verification.

7.3 Filtering a Trace

The notion of filtering a trace through a predicate is best introduced through an example. Figures 7.3(a) and (b), respectively, show a trace σ and its corresponding down-set lattice, $\mathcal{L}(\sigma)$. In the figure, each down-set in $\mathcal{L}(\sigma)$ is labeled by the maximal events (under the $\rightarrow$ relation) in the down-set. For example, the down-set $\{e_1, f_1, f_2, g_1\}$ is represented as $\{f_2, g_1\}$. Let ϕ be a given predicate. The shaded elements of the lattice $\mathcal{L}(\sigma)$ correspond to the states that satisfy ϕ.

Figure 7.3(c) shows the *smallest* sublattice of $\mathcal{L}(\sigma)$ that contains *all* the shaded elements. By "smallest", we mean the sublattice with the fewest elements. We denote this sublattice by L_ϕ. Note that, in order to make L_ϕ a *sublattice* of $\mathcal{L}(\sigma)$, we need to include some non-shaded elements (*i.e.*, down-sets that do not satisfy ϕ).

Every sublattice of a distributive lattice is also distributive [DP90]. So, we can apply Birkhoff's Representation Theorem (Theorem 7.1) to L_ϕ, and retrieve the poset $J(L_\phi)$. Figure 7.3(d) shows this poset. The set of down-sets of the poset in Figure 7.3(d) contains all the states of σ that satisfy ϕ. We call the poset induced by $J(L_\phi)$ the *filtrate* of σ with respect to ϕ.

Definition 7.4. *Let σ be a trace, and ϕ a predicate. Let L_ϕ be the smallest sublattice of $\mathcal{L}(\sigma)$ that contains all the down-sets of σ that satisfy ϕ. The poset $J(L_\phi)$ is called the* **filtrate** *of σ with respect to ϕ.*

As seen in Figure 7.3(d), an element of the poset $J(L_\phi)$ can correspond to multiple events from σ. That is, the filtrate can be viewed as a trace in which multiple events from σ are *merged* into a single (atomic) event.

The intersection of any two sublattices of a lattice L is also a sublattice of L [DP90]. Consequently, the *smallest* sublattice L_ϕ is unique and well-defined, which yields the following theorem.

Theorem 7.2. *The filtrate of a trace σ w.r.t. a predicate ϕ is unique and well-defined.*

A filtrate is said to be **pure** if each of its down-sets satisfies ϕ. It is easy to see that a filtrate is pure if and only if the set of all down-sets of σ that satisfy ϕ

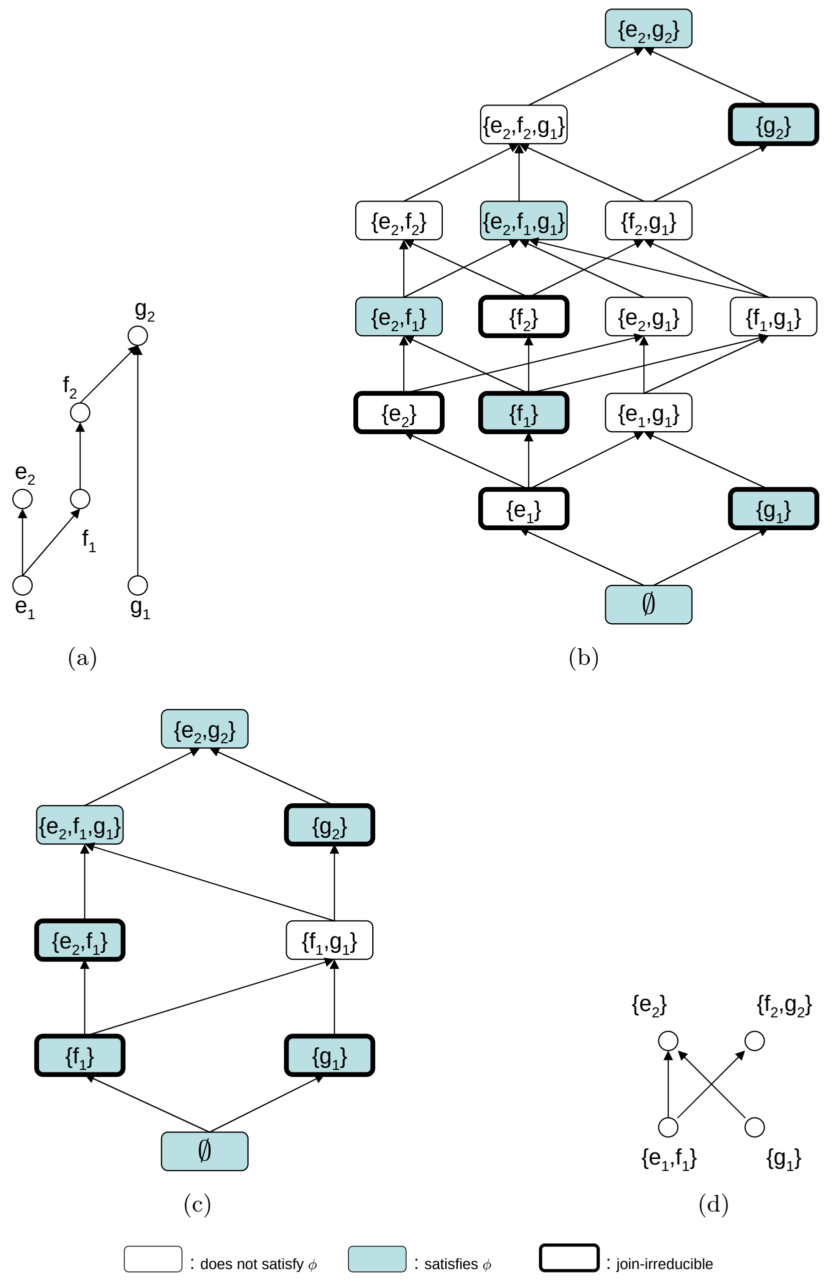


Figure 7.3: Predicate filtering. (a) A trace, σ, (b) its lattice of down-sets, $\mathcal{L}(\sigma)$, (c) the sublattice L_ϕ, and (d) the filtrate of σ w.r.t. ϕ.

forms a sublattice of $\mathcal{L}(\sigma)$. That is, the filtrate is pure iff L_ϕ contains *exactly* those down-sets of σ that satisfy ϕ. In [MG01], it was shown that the filtrate of σ w.r.t. ϕ is pure iff ϕ is a regular formula.

The filtrate can be viewed as a compact representation of the set of all ϕ-satisfying states of a trace. We say *compact* because the size of the filtrate is bounded by $|E|$, the number of events in the trace σ. In order to avoid state space explosion while *constructing* the filtrate, we need to avoid construction of the lattices $\mathcal{L}(\sigma)$ and L_ϕ, because their size could be exponential in the number of events in σ. In [MSGA04], it was shown that, given an algorithm A that can detect whether any reachable state of σ satisfies ϕ, we can derive an algorithm B to construct the filtrate of σ w.r.t. ϕ, such that the time complexity of B is within a polynomial factor of the time complexity of A. We explore this construction in the next section.

7.3.1 Constructing the Filtrate

For the purpose of constructing a filtrate, rather than modelling a trace as a poset, we model it as a directed graph. Note that a poset itself corresponds to a special kind of directed graph, namely, a directed acyclic graph (DAG). We extend the definition of down-sets of a poset to directed graphs as follows.

Definition 7.5. *Given a directed graph $G = (V, E)$, a subset $C \subseteq V$ is called a* **down-set** *of G iff for every $f \in C$, if there exists some edge $(e, f) \in E$, then $e \in C$.*

Note that a down-set of a directed graph either contains all the vertices in a given strongly connected component, or none of them. Let $\mathcal{O}(G)$ denote the set of all down-sets of a directed graph G. The following theorem is a generalization of the result in lattice theory that the set of down-sets of a poset forms a distributive lattice, and was shown in [MG01].

Theorem 7.3. *Given a directed graph G, $(\mathcal{O}(G), \subseteq)$ is a distributive lattice.*

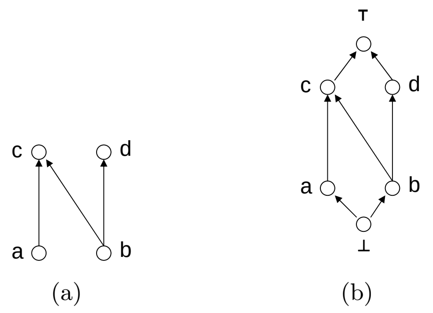


Figure 7.4: (a) A trace σ, and (b) its corresponding directed graph, G.

The empty set and the set of all vertices of G both trivially belong to $\mathcal{O}(G)$. We call them *trivial* down-sets. We can construct a graph G corresponding to a trace poset σ such that there is a 1-1 correspondence between the non-trivial down-sets of G and the down-sets of σ. We achieve this by adding two additional vertices to the Hasse diagram representing σ: $\bot$ and $\top$, where $\bot$ is the "smallest" vertex and $\top$ is the "largest" vertex (*i.e.*, there is a path from $\bot$ to every vertex and a path from every vertex to $\top$). An example of such a transformation is shown in Figure 7.4. Clearly, any non-trivial down-set of G will contain $\bot$ and not contain $\top$. As a result, every down-set of σ is a non-trivial down-set of G, and vice versa.

By definition, adding edges to a directed graph can only *reduce* the number of its down-sets. So, if G' is obtained by adding edges to G, then $\mathcal{O}(G')$ is a sublattice of $\mathcal{O}(G)$. In [Gar02], it was shown that *every* sublattice of $\mathcal{O}(G)$ can be derived by adding edges to G.

Algorithm 7.1, from [Gar02], computes the filtrate directly from the directed graph representation of a trace. It takes as input a directed graph G (derived from a trace σ) and a predicate ϕ. The algorithm constructs the filtrate by adding edges to G (lines 3-8), and finally returns the graph G_ϕ (line 9), which corresponds to the filtrate.

In line 2, the filtrate G_ϕ is initialized to G. For each pair of events e and f

Algorithm 7.1: construct_filtrate

```
    input  : A directed graph $G$ and a predicate $\phi$.
    output: The filtrate, $G_\phi$.
 1  begin
 2      $G_\phi := G$ /* Initialize filtrate to $G$ */
 3      for each pair of events $(e, f)$ do
 4          $Q := G$ with the additional edges $(f, \bot)$ and $(\top, e)$
 5          if $reachable(\phi, Q) == false$ then
 6              add edge $(e, f)$ to $G_\phi$
 7          endif
 8      endfor
 9      return $G_\phi$
10  end
```

in the trace σ, the algorithm constructs a graph Q by adding two additional edges to G: one from f to $\bot$, and the other from $\top$ to e (line 4). Any *non-trivial* down-set of Q cannot contain $\top$, so it cannot contain e. On the other hand, it must contain $\bot$, hence must contain f. Procedure *reachable*(ϕ, Q) checks whether any non-trivial down-set of Q satisfies ϕ. If *reachable*(ϕ, Q) returns $false$, that means no ϕ-satisfying down-set of the trace σ contains f but not e. Therefore, adding an edge from e to f in G will not eliminate any ϕ-satisfying down-sets, but it *will* create a sublattice of $\mathcal{L}(\sigma)$. We continue this procedure for all pairs of vertices. With each added edge, we produce an even smaller sublattice of $\mathcal{L}(\sigma)$, all the time retaining every ϕ-satisfying down-set of $\mathcal{L}(\sigma)$.

Note that the graph G_ϕ may not be acyclic, because adding edges to G can produce a cycle. We obtain a DAG (poset) from G_ϕ by collapsing each strongly-connected component of G_ϕ into a single node. This DAG (poset) is the filtrate of G w.r.t. ϕ. Figure 7.5 shows the directed graph G_ϕ returned by Algorithm 7.1 for the example in Figure 7.3, and the poset (filtrate) obtained by collapsing its strongly-connected components.

Assuming *reachable*(ϕ, Q) takes $O(t)$ time, Algorithm 7.1 has a time com-

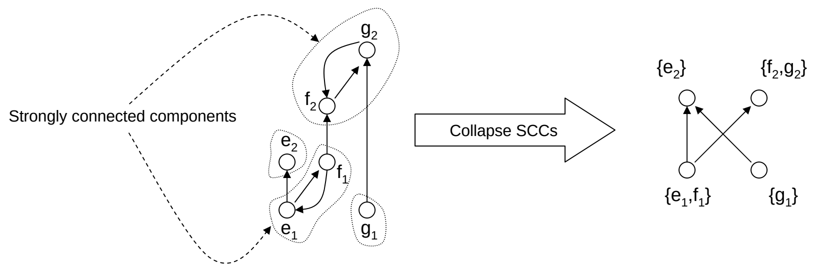


Figure 7.5: Converting the graph returned by Algorithm 7.1 into a poset.

plexity of $O(t.|E|^2)$, where E is the event set of the trace. Thus, if the reachability problem for ϕ can be solved in polynomial time on a trace poset, then the filtrate w.r.t. ϕ can be computed in polynomial time. The converse is also true, as stated in the following theorem, which was proved in [MSGA04].

Theorem 7.4. *The filtrate of a trace σ with respect to a predicate ϕ can be computed in polynomial time if and only if there exists a polynomial time algorithm to determine whether there exists a reachable state of σ that satisfies ϕ.*

Algorithm 7.1 constructs the filtrate w.r.t. *any* predicate ϕ. For special classes of predicates, more efficient filtering algorithms exist. For example, for the case where ϕ is a meet-closed predicate, an $O(n^2.|E|)$ algorithm for constructing the filtrate was presented in [MG03], where n is the number of processes in the trace, and E is the event set of the trace.

7.4 Filtering for State Space Reduction

To use predicate filtering as a state space reduction tool, we first decompose a program into its finite trace cover representation, using the algorithms presented in Chapter 6. To decide reachability for a formula ϕ, we find a weaker formula ψ, for which reachability can be decided in polynomial time on the finite trace cover representation. Then, we filter each trace in the trace cover w.r.t. ψ. This yields a set of filtrates, where each filtrate contains fewer states than the original trace. Exhaustive state space exploration techniques such as depth-first search or breadth-first search can then be used to decide reachability for ϕ.

In the following section, we present an experimental case study which demonstrates the effectiveness of predicate filtering as a state space reduction tool.

7.4.1 Case Study: Leader Election Protocol

We extended our implementation of TC-SPIN, presented in Chapter 6, to include Algorithm 7.1 to compute the filtrate of a trace w.r.t a given predicate. That is, TC-SPIN now takes a PROMELA program as input, generates a finite trace cover, then filters each trace in the finite trace cover with respect to a user-specified predicate. Each filtrate is written out as a PROMELA program.

For our experiments, we used a PROMELA specification of Dolev, Klawe and Rodeh's [DKR82] leader election protocol, with random initialization of processes. This PROMELA specification is available as part of the SPIN distribution [Hol07]. We added two local variables to each process: *know_leader*, which is set to 1 when the process knows the identity of the leader, and *leader_id*, which is set to the process id of the leader. The property being validated was that, once a leader is elected, every process is in agreement about the leader's identity:

$$\neg EF((\bigwedge_i know\_leader_i) \wedge (\nexists j : leader\_id_j \neq leader\_id_{(j+1)\%N}))$$

where N is the number of processes participating in the protocol. In the SPIN verification run, the property is specified by means of an *assert*() statement in a separate monitor process.

TC-SPIN executes in two passes - in Pass 1, it creates the finite trace cover and computes the filtrate of each trace w.r.t. the predicate $\bigwedge_i know\_leader_i$. This predicate is a conjunction of process-local state formulae, for which a polynomial time reachability detection algorithm exists [GW94b]. Hence, the filtrate can be computed efficiently. Each filtrate is written out as a PROMELA program. In Pass 2, SPIN is called on each filtrate. The property being verified in Pass 2 is $AG(\forall i : leader_i == leader_{(i+1)\%N})$. This property is specified by means of an *assert*() statement in a separate monitor process. Table 7.1 shows the number of states constructed (stored) by SPIN vs. TC-SPIN during verification. For TC-SPIN, the number of states in Pass 2 is the sum of all the states generated during verification of all the filtrates. In this example, the filtrates only created about 15 states each. The number of filtrates itself increased with the value of N, because the amount of non-deterministic choice in the program was directly proportional to N. The number of states in Pass 1 is the total number of states generated (stored) during construction of the finite trace cover **and** the filtrate computation.

# procs (N)	SPIN, P.O. reduction - # states	TC-SPIN - # states		
		Pass 1	Pass 2	Total
4	3985	2465	345	2810
5	47727	16721	1785	18506
6	408091	125755	9630	135385

Table 7.1: Number of states constructed during verification of the leader election protocol.

The time taken by SPIN vs. TC-SPIN was comparable in this example. However, our focus was on the amount of state space reduction achieved, because memory consumption is usually the larger concern during verification.

7.5 Bibliographic Notes

In [GM01], only pure filtrates were considered. That is, a filtrate was defined only with respect to a regular predicate. The definition of a filtrate w.r.t. *any* predicate was introduced in [MG01]. Predicate filtering has since been applied to a large range of problem domains. In [Gar02], filtering were used to provide mechanical (algorithmic) solutions to well-known combinatorial problems in the area of integer partitions, set families and permutations. In [MG03], filtering was used for state space reduction while verifying predicates of the form $EF(\phi)$, $AG(\phi)$ and $EG(\phi)$, where ϕ does not contain any temporal operators. That is, nesting of temporal operators was not allowed. Filtering a trace with respect to a formula containing nested temporal operators was first used in [SG03a] for state space reduction. A verification tool called the Partial Order Trace Analyzer (POTA) was built by Sen and Garg [SG03b], which incorporated filtering as a state space reduction mechanism. However, POTA required a specially-formatted trace as input, compared to TC-SPIN, which takes a PROMELA program as input.

The work presented in this chapter is also available as a technical report [KG06].

7.6 Summary

In this chapter, we introduced **predicate filtering**, which can be used on finite trace covers as a state space reduction tool for checking reachability properties. In our experiments, we could verify a leader election protocol using predicate filtering by exploring only one-third as many states as those explored by SPIN using p.o. reduction.

Part IV

Interleaving Semantics

Chapter 8

Producing Short Counterexamples

8.1 Introduction

In Part III, we showed how results from lattice theory could be applied to improve the efficiency of model checking on a partial order representation of the state space. A lively point of debate in the verification community is whether it is preferable to use partial order semantics or interleaving semantics for representating the state space of a program. An interleaving semantics is traditionally considered easier to work with, as it lends itself to a simpler theory of specification and verification of concurrent systems, *e.g.*, with finite state machines. In this chapter, we show that lattice-theoretic concepts can also be used to improve the efficiency of model checking using interleaving semantics.

The ability to produce counterexamples (error trails) is one of the great strengths of model checking. Counterexamples help in the task of debugging by giving the programmer a way to reproduce the sequence of actions that lead to an error. Shorter error trails greatly reduce debugging effort, because they are easier to comprehend. Additionally, the ability to find errors at shorter depths can make it possible to verify larger models, by finding the error state before the model checker runs out of available computational resources, such as time or memory.

Traditional model checking algorithms based on depth-first or breadth-first search perform an “uninformed” state space exploration. That is, the order in which nodes are expanded is arbitrary. This can result in a lot of effort being wasted exploring “uninteresting” portions of the state space, that is, portions of the state space that are not relevant to the property being verified. Additionally, the error trails produced can be needlessly lengthy.

A class of techniques, known as *directed model checking* (DMC) techniques [YD98, ELL01, ELLL04, TAC$^+$04] use heuristic search strategies to guide state space exploration towards those portions of the state space that are most likely to contain errors. The aim is to generate short counterexamples by reaching error states quickly. Heuristic search techniques calculate a cost function for each outgoing transition from the current state, then explore these transitions in the order of increasing cost. Lower cost transitions are considered to be “closer” to the error state. However, in the absence of errors, heuristic search techniques do nothing to reduce state space explosion, because they simply change the order in which nodes are expanded, without reducing the number of nodes to be expanded. Thus, these methods are not suitable for complete validation of the model.

POR techniques [Val91b, Pel93, GW94b] have proved to be highly successful in tackling state space explosion. However, it has been observed that they produce lengthier error trails than even “blind” search strategies. The reason for this can be traced back to the **invisibility constraint**, which was discussed in Section 6.7. In POR techniques, *preference* is given to executing invisible events. However, invisible events are less likely to be relevant to the formula being verified, as illustrated in Figure 8.1.

In Figure 8.1, we are interested in determining whether any reachable program state satisfies q. The shortest path from the initial state to a q-satisfying state is shown in Figure 8.1(d). One possible reduced graph generated by POR is shown in Figure 8.1(c). While this reduced graph is significantly smaller than the full graph in Figure 8.1(b), it still contains several events that do not contribute to reaching the target state. In particular, the events α_1, α_2 and α_3 are irrelevant w.r.t. the property being verified. While there are other possible outcomes for the reduced state graph generated by POR, the main drawback remains the same, namely, POR cannot distinguish between necessary and unnecessary *events*. It only distinguishes between necessary and unnecessary (redundant) interleavings.

There has been some effort to combine heuristic search techniques with state space reduction techniques [LLEL02, Laf03]. However, the combination can interfere with the efficiency of the individual techniques, either resulting in less reduction [Laf03], or lengthier error trails [LLEL02]. To the best of our knowledge, there is currently no single technique that achieves both objectives - state space reduction for complete validation, while narrowing down on error states quickly to produce short error trails. We present such a technique in this chapter.

8.1.1 Our Contribution

In this chapter, we show that lattice theory can be used to produce short error trails while providing state space reduction comparable to POR techniques, even in the absence of errors.

Our approach exploits the properties of meet-closure. Recall, from Section 3.4.1, that for a meet-closed formula, there exists a unique minimum set of **crucial events** per maximal program trace, whose execution is both *necessary and sufficient* to lead to a state satisfying the formula in that trace. Executing crucial events in any order consistent with the dependency relation of the trace results in the same

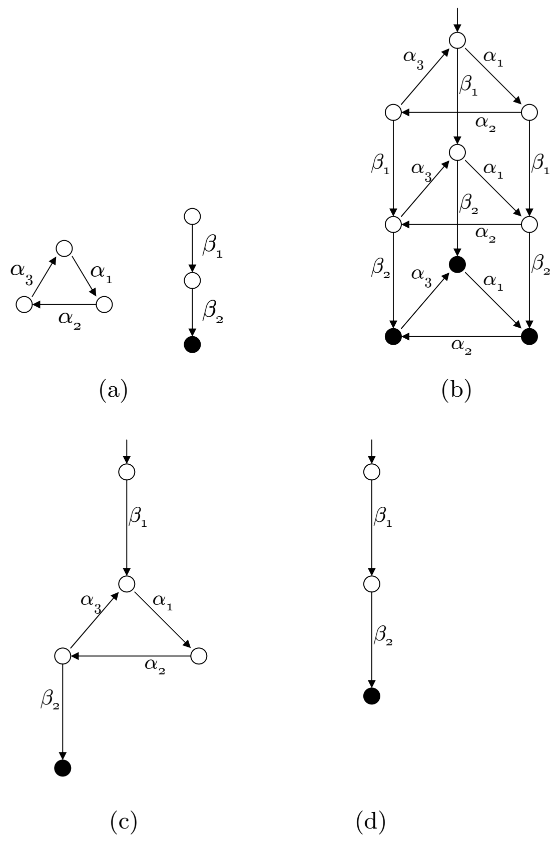


Figure 8.1: The property to be verified is whether any reachable state satisfies q. (a) Two concurrent processes. β_2 is visible, and changes the value of q to true. All other transitions are invisible. (b) The full state space graph. (c) A reduced state space graph, generated through POR. (d) The shortest path to a state satisfying q.

state [God96]. A path consisting only of crucial events is called a **crucial path**.

For a single trace, it is sufficient to explore any one crucial path through the trace. If an error state exists, a crucial path will lead to it through the fewest possible transitions. If the explored crucial path does not encounter an error state, then there is no error state in the trace.

We identify a subset of CTL, which we call Crucial Event Temporal Logic (CETL), which consists only of meet-closed formulae. CETL includes the existential until[1] and release operators of CTL, and allows conjunction. Atomic propositions are limited to process-local variables. CETL does not allow negation, except at the level of atomic propositions, nor does it allow disjunction. Despite these limitations, CETL can express many reachability, safety, liveness and response properties. In fact, of the 131 properties in the BEEM database [Pel07], which is a large repository of benchmarks for explicit-state model checkers, 101 (77%) can be expressed in CETL.

We present an explicit-state model checking algorithm for CETL formulae, and show how crucial events can be identified. While the problem of identifying a crucial event for a general CETL formula remains open, there are many cases where we *can* identify crucial events.

We have implemented our approach as an extension to the popular model checker SPIN [Hol03]. We call our tool SPICED (Simple PROMELA Interpreter with Crucial Event Detection). We provide experimental results from a wide range of examples from the BEEM database [Pel07], as well as from the SPIN distribution [Hol07]. We ran experiments on 76 different variations (with differences in problem sizes and the location of errors) of 16 different models from the BEEM database. In 100% of our tests, the error trails produced by SPICED were at least as short as those produced by SPIN. SPICED achieved trail reduction greater than 1x in 93% of

[1] We use a modified version of the *EU* operator, $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$, as discussed in Section 5.3.

the cases, greater than 10x in 55% of the cases, and greater than 100x in 19% of the cases. We completed verification faster than SPIN (with POR) in 44% of the cases, with a 10x reduction in time in 9% of the cases. For 3 of the 15 models, we were able to verify problem sizes for which SPIN ran out of resources. We also provide experimental results that show that we achieve state space reduction comparable to POR techniques even in the absence of errors.

8.2 Crucial Event Temporal Logic (CETL)

We define a grammar for a subset of CTL, called Crucial Event Temporal Logic (CETL), such that every formula generated by the grammar is regular. In Section 5.2, we showed that process-local state formulae are regular. In Section 5.3, we showed that if p and q are regular, so are $E[p\ U\ (p \wedge q)]$ and $E[q\ R\ p]$. Based on these results, we define the following grammar for CETL.

Definition 8.1. Crucial Event Temporal Logic (CETL) *A CETL formula is one that can be generated from the following rules:*

1. *The trivial propositions $true$ and $false$ are CETL formulae.*
2. *Every process-local state formula is a CETL formula.*
3. *If ϕ_1 and ϕ_2 are CETL formulae, so are $(\phi_1 \wedge \phi_2)$, $E[\phi_2\ R\ \phi_1]$, and $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$.*

Definition 8.2. *Let ϕ be a CETL formula. The set $sub(\phi)$ of subformulae of ϕ is defined as follows:*

- *If ϕ is a process-local state formula, or $true$ or $false$, then $sub(\phi) = \{\phi\}$.*
- *If ϕ is $\phi_1 \wedge \phi_2$, $E[\phi_2\ R\ \phi_1]$ or $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$, then $sub(\phi) = \{\phi\} \cup sub(\phi_1) \cup sub(\phi_2)$.*

The length of a CETL formula ϕ is equal to the cardinality of $sub(\phi)$.

If G is a down-set of $\mathcal{L}(\sigma)$, and H is an immediate successor of G in $\mathcal{L}(\sigma)$, we denote this by $G \triangleright H$. Formally, if $G, H \in \mathcal{L}(\sigma)$, and $\exists e \notin G$, and $H = G \cup \{e\}$, then $G \triangleright H$. The notation $G \trianglerighteq H$ means $(G \triangleright H) \vee (G = H)$. The following two lemmas are used in the proofs presented in Sections 8.4.1 and 8.4.2, and are from [SG02].

Lemma 8.1. *[SG02] Given a trace σ, and down-sets $C, D, F \in \mathcal{L}(\sigma)$, if $C \triangleright F$ and $D \subseteq F$, then $(C \cap D) \trianglerighteq D$.*

Proof. From the definition of $\triangleright$, $\exists e \notin C : C \cup \{e\} = F$. If $e \notin D$, then $D \subseteq C$, so $(C \cap D) = D$. If $e \in D$, then $(C \cap D) = D \setminus \{e\}$. That is, $(C \cap D) \triangleright D$.

□

Lemma 8.2. *Given a trace σ, and down-sets $C, D, F \in \mathcal{L}(\sigma)$, if $F \triangleright C$ and $F \subseteq D$, then $D \trianglerighteq (C \cup D)$.*

Proof. Let $C = F \cup \{e\}$. If $e \in D$, then $C \subseteq D$, so $(C \cup D) = D$. If $e \notin D$, then $C \cup D = D \cup F \cup \{e\}$. Since $F \subseteq D$, $(C \cup D) = D \cup \{e\}$, which implies $D \triangleright (C \cup D)$.

□

We now show how lattice-theoretic concepts can be used to prune the state space *and* produce short counterexamples while model checking CETL formulae. We will start by presenting a "baseline" model checking algorithm in Section 8.3, then enhance it with techniques based on lattice theory in Sections 8.4 and 8.5.

8.3 Baseline Algorithm

The method we present in this chapter is applicable to any *recursive, local* model checking algorithm. By *recursive*, we mean that the truth value of a subformula is decided at a state before starting the search for determining the truth value of

```
Procedure check_CETL(s, φ)
    pre : $info(s,\phi)$ is $true$, $false$ or $\star$.
    post: $info(s,\phi) \neq \star$.
begin
    if $info(s,\phi) \neq \star$ then return
    if $\phi$ is a process-local state formula then
        if $s \models \phi$ then $info(s,\phi) := true$
        else $info(s,\phi) := false$
    endif
    if $\phi$ is $(\phi_1 \wedge \phi_2)$ then
        $check\_CETL(s,\phi_1)$
        if $info(s,\phi_1) = false$ then $info(s,\phi) := false$
        else
            $check\_CETL(s,\phi_2)$
            $info(s,\phi) := info(s,\phi_2)$
        endif
    endif
    if $\phi$ is $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$ then
        new stack($stk$)  /* Create a new stack, with id $stk$ */
        $push(s, stk)$
        $check\_EU\_ER(s,\phi,stk)$
        $pop(stk)$
    endif
end
```

the top-level formula at that state. A *local* model checking algorithm is one that decides, for a specific state s and formula ϕ, whether $s \models \phi$. This is in contrast to a *global* model checking algorithm which decides whether $s \models \phi$ for every state s of the program. An advantage of local model checking is that we do not explore the parts of the state space that are not relevant to the formula being evaluated. This often translates to reduced memory consumption in practical use. An example of a recursive, local model checking algorithm for CTL is ALMC [VL93], which has an asymptotic time complexity that is linear in the size of the state space graph and the length of the formula being verified. Our baseline algorithm is based on ALMC.

As in ALMC, we use a function $info()$, which uses a hash table to implement the function:

$$info : S \times sub(\phi) \mapsto \{true, false, \star\}$$

$info()$ keeps track of all the subformulae evaluated so far at any state s. If $info(s, \phi_1) = \star$, then we do not yet know whether $s \models \phi_1$. If ϕ_1 has already been evaluated at s, then $info(s, \phi_1) = true$ if $s \models \phi_1$, and $false$ otherwise. Of course, initially $info(s, \phi_1) = \star$ for all state-subformula pairs.

Procedure *check_CETL*() is the main routine of our model checking algorithm, and is self-explanatory for process-local state formulae (lines 3-6) and conjunctions (lines 7-14). For temporal subformulae, we offload the work to Procedure *check_EU_ER*(), passing it a clean stack to use for its state space search (lines 15-20). Procedure *check_EU_ER*(s, ϕ) performs a depth-first search starting from state s, with the stack *stk* maintaining the current search path. The depth first search only explores the events returned by Procedure *ample*(s, ϕ) (line 15) from each state s. We call the set of events returned by Procedure *ample*() an "ample set", which is a term borrowed from Peled's partial order reduction technique [Pel93]. In the non-reduced (baseline) case, *ample*(s, ϕ) is equal to *enabled*(s).

We are interested in finding a witness for either $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$, or $EG(\phi_1)$ (if $\phi = E[\phi_2\ R\ \phi_1]$). In either case, every state of the witness path must satisfy ϕ, and also ϕ_1. Consequently, we abandon the current search path (by backtracking) if we encounter a state s' that either does not satisfy ϕ (line 3), or does not satisfy ϕ_1 (lines 5-8).

The search stops with success in one of three cases: (1) a state satisfying $(\phi_1 \wedge \phi_2)$ is found (line 11), which is the final state of a witness path for $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$, or (2) some state t satisfying ϕ is reached (line 3, 28-30), in which case we can transitively deduce that $s \models \phi$, or (3) if $\phi = E[\phi_2\ R\ \phi_1]$, and a cycle is found consisting entirely of ϕ_1-satisfying states (lines 18-24). If a witness is found, then we use the fact that *every* state on the witness path also satisfies ϕ to set

Procedure `check_EU_ER`(s, ϕ, stk)

```
begin
    /* $\phi = E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$ */
    if $info(s,\phi) \neq \star$ then return
    $check\_CETL(s,\phi_1)$
    if $info(s,\phi_1) = false$ then
        $info(s,\phi) := false$
        return
    endif
    $check\_CETL(s,\phi_2)$
    if $info(s,\phi_2) = true$ then
        $info(s,\phi) := true$ /* $s \models (\phi_1 \wedge \phi_2)$ */
        return
    endif
    /* $s \models (\phi_1 \wedge \neg\phi_2)$ */
    $working\_set := ample(s,\phi)$
    for each $\alpha \in working\_set$ do
        $t := \alpha(s)$
        if $on\_stack(t, stk)$ then
            /* Found a cycle */
            if $\phi$ is $E[\phi_2\ R\ \phi_1]$ then
                $info(s,\phi) := true$ /* Cycle demonstrates $EG(\phi_1)$ */
                return
            endif
        else
            $push(s, stk)$
            $check\_EU\_ER(t,\phi)$
            $pop(stk)$
            if $info(t,\phi) = true$ then
                $info(s,\phi) := true$ /* $(s \models \phi_1) \wedge (t \models \phi) \Rightarrow (s \models \phi)$ */
                return
            endif
        endif
    endfor
    $info(s,\phi) := false$ /* No successors satisfy $\phi$, backtrack */
    return
end
```

$info(s', \phi) = true$ for every state s' on the current search path (lines 28-31).

Note that $check\_EU\_ER(s, \phi)$ not only evaluates whether $s \models \phi$, but also evaluates the truth value of ϕ at every state visited during the search. This gives our baseline model checking algorithm an asymptotic time complexity that is linear in the length of the formula and the size of the full state space graph, similar to ALMC.

The baseline algorithm does not exploit any lattice-theoretic properties. We now show how we can use meet-closure to select only a *subset* of the enabled events at each state as our ample set. We start with the narrower problem of model checking a CETL formula in a single trace of a program, then extend this approach to model checking the entire program.

8.4 Model Checking CETL in a Trace

The following example highlights the basic principle of our approach. Let $\sigma = [s, v]$ be some trace, and ϕ a meet-closed formula. Suppose we are interested in finding out whether $s \models EF(\phi)$. If $s \models \phi$, then we are done. If $s \not\models \phi$, then there exists a crucial path for ϕ in σ, starting from s. Starting from s, we simply need to choose an ample set that consists of a single crucial event at each state in order to build this crucial path. This approach results in state space reduction by choosing a singleton ample set, and the crucial path built is the shortest witness for $s \models EF(\phi)$. In this section, we show how crucial paths can act as witnesses for the temporal operators in CETL.

8.4.1 Existential Until Operator (EU)

Let G_0 be some down-set of σ that satisfies $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$. Let π be the corresponding witness path with $\pi^l = H$ as its final state. Then, $\forall j : 0 \leq j \leq l : \pi^j \models \phi_1$, and $H \models (\phi_1 \wedge \phi_2)$. Let $\mathcal{J}$ be the set of all down-sets of σ that are reachable from G_0, are minimal under $\subseteq$[2], and satisfy $(\phi_1 \wedge \phi_2)$. Define:

$$G = \bigcap_{J \in \mathcal{J}} J \tag{8.1}$$

Since $(\phi_1 \wedge \phi_2)$ is regular, $G \models (\phi_1 \wedge \phi_2)$.

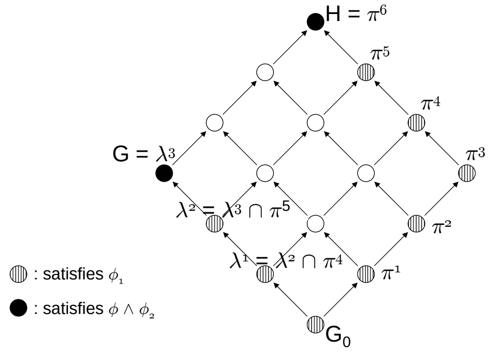

H = π^6
π^5
π^4
π^3
G = λ^3
$\lambda^2 = \lambda^3 \cap \pi^5$
π^2
$\lambda^1 = \lambda^2 \cap \pi^4$
π^1
: satisfies ϕ_1
: satisfies $\phi \wedge \phi_2$
G_0

Figure 8.2: Example illustrating the construction of Theorem 8.3

Theorem 8.3. *There exists a path from G_0 to G such that every state along the path satisfies ϕ_1.*

Proof. We will construct a path λ from G_0 to G, consisting entirely of ϕ_1-satisfying states. We construct this path backwards, starting from $\lambda^k = G$, towards $\lambda^0 = G_0$. Figure 8.2 illustrates this construction through an example.

We show that, if $\lambda^i \models \phi_1$ for any $1 \leq i \leq k$, there exists a $G' \triangleright \lambda^i$ such that $G' \models \phi_1$. We can then extend λ with $\lambda^{i-1} = G'$, and proceed with our construction. For the base case, we have $\lambda^k = G$, and $G \models \phi_1$.

Let $1 \leq j \leq l$ be the *least* j such that $\lambda^i \subseteq \pi^j$. First, we show that such a j must exist. Recall that $\pi^l = H$, and $\lambda^i \subseteq G \subseteq H$. Therefore, for some $j \leq l$, $\lambda^i \subseteq \pi^j$. Also, $\pi^0 = \lambda^0 = G_0$, so $\forall i : i \geq 1 : \lambda^i \not\subseteq \pi^0$. Therefore, $j \geq 1$. Since j is the least such, we have:

$$\lambda^i \not\subseteq \pi^{j-1} \tag{8.2}$$

[2]This ensures that $\mathcal{J}$ is finite

So, we have $\pi^{j-1} \triangleright \pi^j$, and $\lambda^i \subseteq \pi^j$. From Lemma 8.1, this implies $(\lambda^i \cap \pi^{j-1}) \trianglerighteq \lambda^i$. We cannot have $(\lambda^i \cap \pi^{j-1}) = \lambda^i$, because this would imply $\lambda^i \subseteq \pi^{j-1}$, which violates (8.2). Therefore, $(\lambda^i \cap \pi^{j-1}) \triangleright \lambda^i$. Set $G' = (\lambda^i \cap \pi^{j-1})$. Since $\lambda^i \models \phi_1$, and $\pi^{j-1} \models \phi_1$, by the meet-closure of ϕ_1, $G' \models \phi_1$. $\square$

Theorem 8.3 tells us that if $G_0 \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$, then a crucial path for $(\phi_1 \wedge \phi_2)$ can act as a witness. Since $G_0 \models \phi_1$, and every state along the witness path satisfies ϕ_1, it is easy to see that $\mathit{crucial}(G_0, (\phi_1 \wedge \phi_2), \sigma) = \mathit{crucial}(G_0, \phi_2, \sigma)$. The following theorem shows how we can construct this path "forward", that is, starting from G_0.

Theorem 8.4. *We can construct the path of Theorem 8.3 as follows. Starting from G_0, at each state H we execute a single enabled event α that satisfies the following two conditions:*

- *$\alpha \in \mathit{crucial}(H, \phi_2, \sigma)$, and*
- *$H \cup \{\alpha\} \models \phi_1$.*

Proof. Let G be as in (8.1). From Theorem 8.3, there exists *some* path λ such that $\lambda^0 = G_0$, $\lambda^k = G$, and $\forall j : 0 \leq j \leq k : \lambda^j \models \phi_1$. We need to show that we can construct such a path by choosing, at each state, any crucial event that leads to a ϕ_1-satisfying successor.

Clearly, if every event along our path is crucial for ϕ_2, then our path will lead to G. It remains to be shown that, at any state H along our constructed path, there exists a successor J such that $J \models \phi_1$. To begin with, $H = G_0$. Of course, our construction ends when $H = G$, so any H for which a successor needs to be found must be a strict subset of G.

Let $0 \leq i < k$ be the *greatest* i such that $\lambda^i \subseteq H$. We first show that such an i exists. Note that $\lambda^0 = G_0 \subseteq H$. Thus, for some $i \geq 0 : \lambda^i \subseteq H$. Also, $\lambda^k = G$, and $H \subset G$. Therefore, $\lambda^k \not\subseteq H$, so $i < k$. Since i is the greatest such, we have:

$$\lambda^{i+1} \not\subseteq H \tag{8.3}$$

Now, $\lambda^i \rhd \lambda^{i+1}$, and $\lambda^i \subseteq H$. By Lemma 8.2, $H \unrhd (\lambda^{i+1} \cup H)$. If $H = (\lambda^{i+1} \cup H)$, then $\lambda^{i+1} \subseteq H$, which violates (8.3). Therefore, $H \rhd (\lambda^{i+1} \cup H)$. Also, $H \models \phi_1$, and $\lambda^{i+1} \models \phi_1$, so by the join-closure of ϕ_1, $\lambda^{i+1} \cup H \models \phi_1$. Hence, $J = \lambda^{i+1} \cup H$ is the required successor for H.

□

8.4.2 Existential Release Operator (ER)

Recall that $E[\phi_2 \ R \ \phi_1] \overset{def}{\equiv} E[\phi_1 U(\phi_1 \wedge \phi_2)] \vee EG(\phi_1)$. Theorem 8.4 showed how to construct a witness for $G_0 \models E[\phi_1 U(\phi_1 \wedge \phi_2)]$. The following theorem shows how to construct a witness for $G_0 \models EG(\phi_1)$.

Theorem 8.5. *Let $G_0 \in \mathcal{L}(\sigma)$ such that $G_0 \models EG(\phi_1)$ in σ. We can construct a witness path as follows. Starting from G_0, at each state H, we execute a single enabled event α such that $H \cup \{\alpha\} \models \phi_1$.*

Proof. We simply need to show that, for every state H on the constructed path, there exists a ϕ_1-satisfying successor state. The proof for this is exactly the same as shown in Theorem 8.4.

□

Procedure *ample_trace*() constructs an ample set in accordance with Theorems 8.4 and 8.5. We assume the existence of a function $find\_crucial(s, \phi_2, \sigma)$, which returns a (possibly empty) subset of *enabled*$(s) \cap$ *crucial*(s, ϕ_2, σ). The implementation of this function is deferred till Section 8.6. In lines 4-8, we try to find some α such that $\alpha \in$ *crucial*(s, ϕ_2, σ) and $\alpha(s) \models \phi_1$. If such an event is found, then it satisfies the requirements of both Theorems 8.4 and 8.5, so our ample set is

Procedure `ample_trace(`s, ϕ`)`

```
begin
    /* $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$ */
    $working\_set := find\_crucial(s, \phi_2, \sigma)$
    for each $\alpha \in working\_set$ do
        $t := \alpha(s)$
        $check\_CETL(t, \phi_1)$
        if $info(t, \phi_1) = true$ then return $\{\alpha\}$
    endfor
    return $enabled(s)$
end
```

a singleton consisting of this event. If such an event is not found, then we explore all enabled events (line 9). The following theorem is straightforward.

Theorem 8.6. *Procedure ample_trace() returns an ample set that is sufficient for model checking CETL formulae in a single trace of a program.*

We now extend our approach beyond a single trace, to model checking a program.

8.5 Model Checking CETL in a Program

For model checking CETL in a single trace, we simply needed to explore a single crucial path for the formula through the trace. We achieved this by exploring a crucial successor event at each state during our depth first search. To model check CETL in a program, we need to explore a crucial path in each maximal trace of the program. That is, at each state s encountered during DFS, our ample set must contain a crucial event for every trace that starts from s.

Let $ample(s, \phi)$ denote the ample set at state s, for the CETL formula ϕ. In [Pel93], it was shown that if $ample(s, \phi)$ satisfies the following condition (C1), then it generates a successor in each maximal trace starting from s.

(C1) Along every path starting from s in the full state space graph, a transition that is dependent on a transition from $ample(s, \phi)$ cannot be executed without a transition from $ample(s, \phi)$ occurring first.

Theorem 8.7. *[Pel93] If $ample(s, \phi)$ satisfies condition (C1), then for every maximal trace σ starting from s, there exists some $\alpha \in ample(s, \phi)$ such that $[s, \alpha] \sqsubseteq \sigma$.*

Recall that Condition (C1) was also used in Section 6.3 to generate a representative transition sequence per maximal program trace. If $ample(s, \phi)$ satisfies (C1), then it contains a successor event for each trace starting from s. A single event $\alpha \in ample(s, \phi)$ can be a successor in *multiple* traces starting from s. For example, executing α may enable β and γ, where $(\beta, \gamma) \in D$. Thus, α is a successor in both $[s, \alpha\beta]$ and $[s, \alpha\gamma]$. In order to construct a crucial path per maximal trace, α must be crucial in every trace in which it is a successor:

Definition 8.3. Universally crucial event: *An event α is said to be universally crucial from a state s for a meet-closed formula ϕ_2, denoted $\alpha \in ucrucial(s, \phi_2)$, iff for every trace σ such that $[s, \alpha] \sqsubseteq \sigma$, $\alpha \in crucial(s, \phi_2, \sigma)$.*

Recall that Procedure $check\_EU\_ER(s, \phi, stk)$ calls Procedure $ample(s, \phi)$, passing it a formula ϕ of the form $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$. Procedure $ample(s, \phi)$ tries to construct an ample set that satisfies the following three conditions: (1) If $\alpha \in ample(s, \phi)$, then $\alpha \in ucrucial(s, \phi_2)$ (line 3), (2) If $\alpha \in ample(s, \phi)$, then $\alpha(s) \models \phi_1$ (lines 4-8), and (3) $ample(s, \phi)$ satisfies condition (C1) (line 9). If any of these conditions is violated, then $ample(s, \phi) = enabled(s)$ (lines 7, 10). We discuss the implementation of $find\_ucrucial()$ in the next section. For now, it suffices to say that $find\_ucrucial(s, \phi_2)$ returns a (possibly empty) subset of $enabled(s) \cap ucrucial(s, \phi_2)$. The function $satisfies\_C1()$ is the same as that used in the implementation of p.o. reduction in SPIN [Hol03].

Procedure `ample`(s, ϕ)

```
begin
    /* $\phi$ is $E[\phi_1 U(\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$ */
    $candidate := find\_ucrucial(s, \phi_2)$
    for each $\alpha \in candidate$ do
        $t := \alpha(s)$
        $check\_CETL(t, \phi_1)$
        if $info(t, \phi_1) = false$ then return $enabled(s)$
    endfor
    if $(candidate = \emptyset)$ or $(\neg satisfies\_C1(candidate))$ then return
    $enabled(s)$
    else return $candidate$
end
```

Theorem 8.8. *Procedure ample() returns an ample set that is sufficient for model checking CETL in a program.*

Proof. It is straightforward to see that if $check\_EU\_ER(s, \phi, stk)$ finds a witness path, then $s \models \phi$. We show the other direction. Assume $s \models \phi$, where ϕ is $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$. Then, either $s \models E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$, or $s \models EG(\phi_1)$.

- **Case 1:** $s \models E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$. Let σ be the maximal program trace to which the witness path belongs. By Theorem 8.3, there exists a *crucial* witness path for $s \models \phi$ in σ. We will construct a crucial witness path using only transitions in $ample(s, \phi)$.

  Let u denote the transition sequence of the witness path constructed so far, and s' be the final state reached after executing u from s. In our construction, we will maintain the invariant that every event in u is in $crucial(s, \phi_2, \sigma)$, and for every state s' in the path, $s' \models \phi_1$. By Theorem 8.4, these two invariants ensure that at each state s' along the constructed path, there exists some $\alpha \in enabled(s')$ such that $\alpha \in crucial(s', \phi_2, \sigma)$, and $\alpha(s') \models \phi_1$. We will show that $ample(s', \phi)$ contains such an event α. Initially, $u := \epsilon$ (the empty string), and $s' := s$. Since $s \models \phi$, we know that $s \models \phi_1$.

  - **Case 1.1:** The candidate set picked in line 3 of Procedure *ample*() does not satisfy (C1) or is empty. Then, $ample(s', \phi) = enabled(s')$ (line 9). As discussed in the previous paragraph, $enabled(s')$ must contain an event α that satisfies the two conditions of Theorem 8.4, so we set $u := u.\alpha$, and continue construction.
  - **Case 1.2:** The candidate set picked in line 3 is non-empty and satisfies (C1). We can express σ as the concatenation $[s, u].\sigma'$, for some σ'. By Theorem 8.7, there exists some $\alpha \in ample(s', \phi)$ such that $[s', \alpha] \sqsubseteq \sigma'$. That is, $[s, u.\alpha] \sqsubseteq \sigma$. Since α is in $ucrucial(s', \phi_2) \cap enabled(s')$ (line 3), we have $\alpha \in crucial(s', \phi_2, \sigma)$, and $\alpha(s') \models \phi_1$, thus satisfying the conditions of Theorem 8.4. We set $u := u.\alpha$, and continue construction.

- **Case 2:** $s \not\models E[\phi_1 \; U \; (\phi_1 \wedge \phi_2)]$ and $s \models EG(\phi_1)$. Again, let σ be the maximal program trace containing the witness path in the full state space graph. Using arguments identical to those in Case 1, we can show that $ample(s', \phi)$ always contains an event from σ that satisfies the conditions of Theorem 8.5. Thus, we can construct a witness path using the technique of Theorem 8.5, with only the transitions returned by Procedure *ample*().

□

Next, we provide an implementation for the function $find\_ucrucial()$, which is used by Procedure *ample*().

8.6 Finding Universally Crucial events

In this section, we identify some sufficient conditions for an event to be universally crucial. Procedure $find\_ucrucial()$ takes as input a state s and a CETL formula

ϕ_2, and returns a subset of $ucrucial(s, \phi_2) \cap enabled(s)$.

First, note that $find\_ucrucial(s, \phi_2)$ is called by our model checking routine only when $s \not\models \phi_2$. This assertion can be verified by navigating the procedure call chain of our model checking algorithm. Procedure $check\_EU\_ER(s, \phi)$ calls Procedure $ample(s, \phi)$ (line 22), where ϕ is $E[\phi_1\ U\ (\phi_1 \wedge \phi_2)]$ or $E[\phi_2\ R\ \phi_1]$. The call to $ample(s, \phi)$ is only made after verifying that $s \not\models \phi_2$ (line 16). Procedure $ample(s, \phi)$ then calls $find\_ucrucial(s, \phi_2)$ (line 3). The following theorem both explains Procedure $find\_ucrucial()$ and shows its correctness.

Procedure `find_ucrucial(`s, ϕ_2`)`

```
input  : State $s$ and CETL formula $\phi_2$, where $s \not\models \phi_2$.
output: A subset of $ucrucial(s, \phi_2) \cap enabled(s)$.
1  begin
2      if $\phi_2$ is a process-local state formula on process $P_i$ then
3          return $enabled(s) \cap T_i$  /* $T_i$ is the set of transitions
           of $P_i$ */
4      endif
5      if $\phi_2$ is $(\psi_1 \wedge \psi_2)$ then
6          $check\_CETL(s, \psi_1)$
7          if $info(s, \psi_1) = false$ then  return $find\_ucrucial(s, \psi_1)$
8          else return $find\_ucrucial(s, \psi_2)$
9      endif
10     if $\phi_2$ is $E[\psi_1\ U\ (\psi_1 \wedge \psi_2)]$ or $E[\psi_2\ R\ \psi_1]$ then
11         $check\_CETL(s, \psi_1)$
12         if $info(s, \psi_1) = false$ then return $find\_ucrucial(s, \psi_1)$
13         else
14             if $\neg\psi_1$ is meet-closed then return $find\_ucrucial(s, \neg\psi_1)$
15             else return $\emptyset$
16         endif
17     endif
18 end
```

Theorem 8.9. *Procedure $find\_ucrucial(s, \phi_2)$ returns a subset of $ucrucial(s, \phi_2) \cap enabled(s)$.*

Proof. We show that Procedure $find\_ucrucial()$ returns only enabled, universally crucial events for each formula type.

- **(Lines 2-4):** ϕ_2 is a process-local state formula on process P_i.

  Only transitions from T_i can change the truth value of ϕ_2. Since $s \not\models \phi_2$, in order to reach a ϕ_2-satisfying state from s, we must execute some transition from $T_i \cap enabled(s)$. Now, for any $\alpha, \beta \in T_i \cap enabled(s)$, $(\alpha, \beta) \in D$. Further, execution of α disables β and vice-versa. Therefore, each $\alpha \in T_i \cap enabled(s)$ is a crucial event in any trace that subsumes $[s, \alpha]$. Thus, $T_i \cap enabled(s) \subseteq ucrucial(s, \phi_2) \cap enabled(s)$.

- **(Lines 5-9):** $\phi_2 = \psi_1 \wedge \psi_2$.

  This case is straightforward. If $s \not\models \psi_1$, then clearly we first need to get to a state that satisfies ψ_1. Similarly for ψ_2. Therefore, if $s \not\models \psi_1$ then $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$, else $ucrucial(s, \psi_2) \subseteq ucrucial(s, \phi_2)$.

- **(Lines 10-17):** $\phi_2 = E[\psi_1 \; U \; (\psi_1 \wedge \psi_2)]$ or $\phi_2 = E[\psi_2 \; R \; \psi_1]$.

  - **(Line 12)**: $s \not\models \psi_1$. Clearly, any state that satisfies ϕ_2 must satisfy ψ_1. Therefore, $ucrucial(s, \psi_1) \subseteq ucrucial(s, \phi_2)$.

  - **(Lines 13-16)**: $s \models \psi_1$. Let t be some state reachable from s such that $t \models \phi_2$. Let w be a witness for $t \models \phi_2$. Since $s \not\models \phi_2$, along every path v from s to t, there must exist some state s' such that $s' \not\models \psi_1$ (otherwise, $v.w$ would be a witness for $s \models \phi_2$). That is, we must first reach a state that satisfies $\neg\psi_1$ in order to reach any state that satisfies ϕ_2. If $\neg\psi_1$ is meet-closed, then there exist crucial events for $\neg\psi_1$, so $ucrucial(s, \neg\psi_1) \subseteq ucrucial(s, \phi_2)$.

$\square$

8.7 Experimental Results

We have implemented the CETL model checking algorithms presented in this chapter as an extension to the SPIN model checker [Hol03, Hol07], called SPICED (Simple PROMELA Interpreter with Crucial Event Detection). Our implementation of SPICED, along with detailed experimental results, is available at:
`http://maple.ece.utexas.edu/spiced`.

We ran SPICED against a large set of examples from the BEEM database [Pel07]. The BEEM database is a large collection of benchmarks for explicit-state model checkers. The database includes PROMELA[3] models with errors injected into them, and lists the properties to be verified on these models. Of the 131 properties included in the BEEM database for verification, 101 (77%) can be expressed in CETL. All experiments were performed on a single-cpu 2.8 GHz Intel Pentium 4 machine with 512 MB of memory, running Red Hat Enterprise Linux WS Release 4.

For our experiments, we specified the formulae to be verified in CETL for the SPICED runs, and in LTL for the SPIN runs. Table 8.1 lists the formulae verified for each of the models used in our experiments.

BEEM contains multiple problem sizes for each model. We verified 16 different protocols, and several problem sizes for each of the verified protocols. Overall, we verified over 80 models, with 76 of these models containing errors in them. The errors were already present within the models in the BEEM database - we did not inject the errors ourselves.

For SPIN, *never claims* [Hol03] were used for the verification of LTL properties, and simple $assert()$ statements were used for specifying reachability properties. The SPIN runs use POR techniques for state space reduction, and an automata-theoretic approach, called on-the-fly model checking [VW86, CVWY92], for the

[3]Recall that PROMELA is the input language for SPIN.

Model	Description	Logic	Formula
anderson	Anderson's mutual exclusion [And90]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
at	Alur-Taubenfeld mutual exclusion [AT96]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
bakery	Bakery mutual exclusion [Lam74]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
driving_phils	Driving philosophers [PBG04]	CETL LTL	$EF(P_0.req \wedge EG(!P_0.grant))$ $\neg\Box(req0 \Rightarrow \Diamond grant0)$
fischer	Fisher's mutual exclusion [Fis85]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
frogs	2D Toads and Frogs puzzle [BC87]	CETL LTL	$EF(Check.done)$ local assert()
gear	Gear controller [LPY01]	CETL LTL	$EF(Clutch.err\_open)$ local assert()
lamport	Lamport's mutual exclusion [Lam87]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
loyd	Sam Lloyd's fifteen square puzzle	CETL LTL	$EF(Check.done)$ local assert()
mcs	MCS mutual exclusion [MCS91]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
msmie	Multiprocessor Shared-Memory Information Exchange	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
needham	Needham-Schroeder encryption [NS78]	CETL LTL	$EF(init0.fin \wedge resp0.fin)$ $\neg\Diamond(init\_fin \wedge resp\_fin)$
peterson	Peterson's mutual exclusion [Pet83]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$
phils	Dining philosophers [Dij72]	CETL LTL	$EF(P_0.req \wedge EG(!P_0.grant))$ $\neg\Box(req0 \Rightarrow \Diamond grant0)$
POTS	Plain Old Telephony Service [KL00]	CETL LTL	$EF(P0.dial \wedge P1.idle \wedge EG(!P0.connect))$ $\neg\Box((P0\_dial \wedge P1\_idle) \Rightarrow \Diamond P0\_connect)$
szymanski	Szymanski mutual exclusion [Szy90]	CETL LTL	$EF(P_0.wait \wedge EG(!P_0.cs))$ $\neg\Box(wait0 \Rightarrow \Diamond cs0)$

Table 8.1: Formulae being verified on various models.

verification of LTL formulae. For SPICED, the CETL formulae were specified a separate file, and fed directly as input to our model checking algorithm.

8.7.1 Length of Error Trails

Table 8.2 shows the largest reduction in trail length achieved for each of the 16 protocols verified, compared to SPIN with POR. As seen in the table, for 3 of these protocols, SPIN was unable to complete verification for the larger problem sizes.

SPICED consistently achieves dramatic reductions in the size of the produced error trail, compared to SPIN with p.o. reduction. In many instances, this also results in a significant reduction in the number of states visited during verification, which in turn resulted in less memory consumption and faster run times.

Over all our experiments, SPICED produced error trails that were at least as short as SPIN's in 100% of the cases, were at least 10x shorter in 55% of the cases, and at least 100x shorter in 19% of the cases. For 44% of the cases, SPICED completed verification faster than SPIN, with at least a 10x reduction in time in 9% of the cases. Although CETL is a branching-time logic, in these examples, the properties were in LTL $\cap$ CETL, so the error trails were non-branching. The error trails were produced in the same format as those of SPIN's, and can be examined using SPIN's guided simulation feature.

The complete set of results for all of the verified models is presented in Table 8.4.

8.7.2 State Space Reduction

Table 8.3 shows the state space reduction achieved by SPICED, compared to SPIN with p.o. reduction, in the absence of errors. The examples in Table 8.3 are from the SPIN distribution [Hol07], and have previously been used to showcase the effectiveness of p.o. reduction [CGMP99]. For SPIN, no LTL properties were specified

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
phils.7	SPICED	0.01	15	3.15	6	N/A
	SPIN	**Could not complete**			-	
szymanski.9	SPICED	0.02	256	3.15	43	N/A
	SPIN	**Could not complete**			-	
fischer.18	SPICED	0.02	28	3.15	19	N/A
	SPIN	**Could not complete**			-	
mcs.5	SPICED	0.09	30227	4.89	14	403.29
	SPIN	0.03	2821	2.72	5646	
anderson.7	SPICED	0.03	65387	7.03	82	382.79
	SPIN	0.13	15692	6.63	31389	
peterson.7	SPICED	0.09	29080	4.89	159	125.69
	SPIN	0.1	9992	9.93	19984	
lamport.7	SPICED	0.06	6850	3.45	30	44.33
	SPIN	0.02	665	2.62	1330	
at.7	SPICED	0.02	19	3.15	11	33.64
	SPIN	0.01	182	2.62	370	
bakery.6	SPICED	0.01	69	3.15	46	18.61
	SPIN	0.02	896	2.62	856	
POTS	SPICED	0.89	153775	21.58	41	5.39
	SPIN	0.38	40161	6.72	221	
gear.2	SPICED	0.03	4185	3.13	5056	3.84
	SPIN	0.13	22386	5.5	19396	
needham.4	SPICED	0.01	27	2.72	15	3.47
	SPIN	0.04	4003	3.03	52	
msmie.2	SPICED	0.02	83	2.72	63	3.4
	SPIN	0.01	370	2.62	214	
loyd.2	SPICED	0.19	50931	9.24	52597	1.6
	SPIN	0.63	166133	17.61	84418	
driving_phils.4	SPICED	0.01	212	3.15	123	1.38
	SPIN	0.01	85	2.62	170	
frogs.3	SPICED	0.41	190318	16.45	261	1
	SPIN	0.38	190315	13.99	261	

Table 8.2: Trail reduction with SPICED, compared to SPIN with p.o. reduction.

Model	Tool	Time (sec)	States	Memory (MB)	Formula
sort	SPIN, no reduction	1.19	107713	20.64	-
	SPIN, p.o. reduction	0.1	135	2.62	-
	SPICED	0.1	148	2.72	$EG(!left.tstvar)$
leader	SPIN, no reduction	0.17	15779	3.35	-
	SPIN, p.o. reduction	0.01	97	2.62	-
	SPICED	0.05	104	2.72	$EG(!node[4].tstvar)$
eratosthenes	SPIN, no reduction	0.52	49790	9.07	-
	SPIN, p.o. reduction	0.02	3437	3.03	-
	SPICED	0.02	2986	3.13	$EG(!sieve[0].tstvar)$
snoopy	SPIN, no reduction	0.53	81013	11.34	-
	SPIN, p.o. reduction	0.06	14169	4.06	-
	SPICED	0.4	58081	9.69	$EF(cpu0.tstvar)$

Table 8.3: State space reduction with SPICED.

during verification, which is optimal for maximizing the effectiveness of p.o. reduction. Since our algorithm is based on choosing crucial events, it requires the specification of a property. For each example, we chose a property that is never satisfied in the program, and forces exhaustive validation. As the results in Table 8.3 show, we achieve state space reduction comparable to POR techniques.

8.7.3 Complete List of Results

Table 8.4 lists the complete set of results for all of the verified models that had errors in them. The table contains 76 models, and shows that SPICED consistently produces significantly shorter error trails than SPIN, often resulting in shorter run times and lower memory consumption than SPIN.

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
anderson.1	SPICED	0.01	99	3.15	39	67.56
	SPIN	0.02	1317	2.62	2635	
anderson.3	SPICED	0.01	246	3.15	58	51.48
	SPIN	0.02	1492	2.62	2986	
anderson.5	SPICED	0.03	6664	3.45	74	413.89

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
	SPIN	0.12	15312	4.61	30628	
anderson.7	SPICED	0.03	65387	7.04	82	382.79
	SPIN	0.13	15692	6.63	31389	
at.1	SPICED	0.02	16	3.15	8	18.88
	SPIN	0.01	74	2.62	151	
at.2	SPICED	0.02	16	3.15	8	18.88
	SPIN	0.01	74	2.62	151	
at.3	SPICED	0.02	17	3.15	9	24.89
	SPIN	0.01	110	2.62	224	
at.4	SPICED	0.02	18	3.15	10	29.70
	SPIN	0.01	146	2.62	297	
at.5	SPICED	0.02	18	3.15	10	29.70
	SPIN	0.01	146	2.62	297	
at.6	SPICED	0.02	18	3.15	10	29.70
	SPIN	0.01	146	2.62	297	
at.7	SPICED	0.02	19	3.15	11	33.64
	SPIN	0.01	182	2.62	370	
bakery.1	SPICED	0.02	2601	3.25	510	1.97
	SPIN	0.02	2071	2.62	1006	
bakery.2	SPICED	0.01	45	3.15	28	5.71
	SPIN	0.01	163	2.62	160	
bakery.3	SPICED	0.02	2257	3.25	55	6.51
	SPIN	0.01	179	2.62	358	
bakery.4	SPICED	0.01	57	3.15	37	16.49
	SPIN	0.03	748	2.62	610	
bakery.5	SPICED	1.23	401575	25.57	158	8.30
	SPIN	0.02	656	2.62	1312	
bakery.6	SPICED	0.01	69	3.15	46	18.61
	SPIN	0.02	896	2.62	856	
bakery.7	SPICED	5.80	1740000	100.84	670	2.88
	SPIN	0.02	964	2.62	1928	

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
bakery.8	SPICED	5.98	1500000	92.85	77	5.66
	SPIN	0.01	218	2.62	436	
driving_phils.1	SPICED	0.01	85	3.15	53	1.28
	SPIN	0.01	34	2.62	68	
driving_phils.3	SPICED	0.01	747	3.25	426	1.86
	SPIN	0.01	397	2.62	794	
driving_phils.4	SPICED	0.01	212	3.15	123	1.38
	SPIN	0.01	85	2.62	170	
fischer.1	SPICED	0.01	23	3.15	14	6.07
	SPIN	0.01	48	2.62	85	
fischer.2	SPICED	0.01	24	3.15	15	15.33
	SPIN	0.01	156	2.62	230	
fischer.3	SPICED	0.01	26	3.15	17	91.88
	SPIN	0.02	1237	2.62	1562	
fischer.4	SPICED	0.01	27	3.15	18	297.83
	SPIN	0.05	6454	3.03	5361	
fischer.5	SPICED	0.01	28	3.15	19	268.37
	SPIN	0.04	4744	2.93	5099	
fischer.6	SPICED	0.01	28	3.15	19	777.37
	SPIN	0.15	21860	5.12	14770	
fischer.7	SPICED	0.01	28	3.15	19	497.37
	SPIN	0.07	8016	3.24	9450	
fischer.18	SPICED	0.02	28	3.15	19	N/A
	SPIN	**Could not complete**			-	
frogs.1	SPICED	0.01	1436	2.72	86	1.00
	SPIN	0.01	1433	2.62	86	
frogs.2	SPICED	0.04	9859	3.24	30	1.00
	SPIN	0.03	9856	3.03	30	
frogs.3	SPICED	0.41	190318	16.45	261	1.00
	SPIN	0.38	190315	13.99	261	
gear.1	SPICED	0.01	1060	2.83	1256	2.27

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
	SPIN	0.03	3186	2.93	2846	
gear.2	SPICED	0.03	4185	3.13	5056	3.84
	SPIN	0.13	22386	5.50	19396	
lamport.1	SPICED	0.01	230	3.15	28	10.86
	SPIN	0.01	152	2.62	304	
lamport.2	SPICED	0.01	92	3.15	14	21.71
	SPIN	0.01	152	2.62	304	
lamport.3	SPICED	0.01	63	3.15	17	5.41
	SPIN	0.01	46	2.62	92	
lamport.5	SPICED	0.01	1221	3.15	29	28.90
	SPIN	0.02	419	2.62	838	
lamport.6	SPICED	0.01	483	3.15	19	37.05
	SPIN	0.01	352	2.62	704	
lamport.7	SPICED	0.06	6850	3.45	30	44.33
	SPIN	0.02	665	2.62	1330	
lamport.8	SPICED	0.01	1087	3.15	25	4.32
	SPIN	0.01	54	2.62	108	
loyd.1	SPICED	0.01	47	2.72	42	1.00
	SPIN	0.01	363	2.62	42	
loyd.2	SPICED	0.19	50931	9.24	52597	1.60
	SPIN	0.63	166133	17.61	84418	
mcs.1	SPICED	0.01	345	3.15	12	22.33
	SPIN	0.01	133	2.62	268	
mcs.2	SPICED	0.01	118	3.15	29	2.48
	SPIN	0.01	35	2.62	72	
mcs.3	SPICED	0.02	2710	3.25	13	91.15
	SPIN	0.01	591	2.62	1185	
mcs.4	SPICED	0.01	771	3.15	30	2.30
	SPIN	0.01	33	2.62	69	
mcs.5	SPICED	0.09	30227	4.89	14	403.29
	SPIN	0.03	2821	2.72	5646	

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
mcs.6	SPICED	0.03	8871	3.66	36	2.22
	SPIN	0.01	38	2.62	80	
msmie.2	SPICED	0.02	83	2.72	63	3.40
	SPIN	0.01	370	2.62	214	
needham.1	SPICED	0.01	21	2.72	11	3.36
	SPIN	0.01	296	2.62	37	
needham.2	SPICED	0.01	24	2.72	13	3.23
	SPIN	0.01	1313	2.72	42	
needham.3	SPICED	0.01	27	2.72	15	3.13
	SPIN	0.03	4001	2.93	47	
needham.4	SPICED	0.01	27	2.72	15	3.47
	SPIN	0.04	4003	3.03	52	
peterson.1	SPICED	0.01	592	3.15	46	6.13
	SPIN	0.01	141	2.62	282	
peterson.2	SPICED	0.01	93	3.15	59	6.51
	SPIN	0.01	192	2.62	384	
peterson.3	SPICED	0.01	50	3.15	37	8.16
	SPIN	0.01	151	2.62	302	
peterson.4	SPICED	0.02	3004	3.25	95	16.17
	SPIN	0.02	768	2.62	1536	
peterson.5	SPICED	0.01	175	3.15	110	56.20
	SPIN	0.04	3091	2.93	6182	
peterson.6	SPICED	0.01	75	3.15	59	97.29
	SPIN	0.03	2870	2.83	5740	
peterson.7	SPICED	0.09	29080	4.89	159	125.69
	SPIN	0.10	9992	9.93	19984	
phils.1	SPICED	0.02	15	3.15	6	4.83
	SPIN	0.01	15	2.62	29	
phils.2	SPICED	0.01	13	3.15	5	98.80
	SPIN	0.01	969	2.62	494	
phils.3	SPICED	0.01	15	3.15	6	80.00

Model	Tool	Time (sec)	States	Memory (MB)	Trail length	Trail reduction factor
	SPIN	0.01	253	2.62	480	
phils.4	SPICED	0.01	13	3.15	5	16687.60
	SPIN	0.36	41719	9.09	83438	
phils.5	SPICED	0.01	15	3.15	6	8106.17
	SPIN	0.21	24319	6.36	48637	
phils.6	SPICED	0.04	15	3.15	6	141948.50
	SPIN	3.83	425846	86.22	851691	
phils.7	SPICED	0.01	15	3.15	6	N/A
	SPIN	**Could not complete**			-	
POTS	SPICED	0.89	153775	21.58	41	5.39
	SPIN	0.38	40161	6.72	221	
szymanski.1	SPICED	0.01	80	3.15	31	17.68
	SPIN	0.01	274	2.62	548	
szymanski.2	SPICED	0.06	19006	4.07	25	9.28
	SPIN	0.01	116	2.62	232	
szymanski.3	SPICED	0.01	138	3.15	37	197.35
	SPIN	0.04	3651	2.83	7302	
szymanski.4	SPICED	4.98	1	70.42	28	56.14
	SPIN	0.01	786	2.62	1572	
szymanski.5	SPICED	0.02	256	3.15	43	1811.53
	SPIN	0.31	38948	9.19	77896	
szymanski.9	SPICED	0.02	256	3.15	43	N/A
	SPIN	**Could not complete**			-	

Table 8.4: Complete list of results for 76 models, each containing an error.

8.8 Bibliographic Notes

The representation of concurrent programs by transition graphs was advocated by Lamport [Lam83]. The modeling of concurrency by interleaving was first used by Dijkstra [Dij65]. The term "interleaving" was also coined by Dijkstra [Dij72].

While the production of counterexamples has always been an integral part of model checkers, until recently, they were largely treated as a mere by-product of model checking. During the last few years, however, counterexamples have received renewed interest in their own right as useful debugging tools. A survey of recent contributions in the area of counterexamples appears in [CV04].

Research on counterexample minimization can be divided into two categories: (1) finding a short path to an error *state*, and (2) finding short *cycles* (loops) as counterexamples for liveness properties. For example, LTL formulae are defined over infinite paths. In finite-state systems, a failing LTL property has a lasso-shaped counterexample $u.v^{\omega}$, where u is the stem and v is the loop [VW86]. Minimizing the length of the stem u falls under category (1) above, while minimizing the length of the loop v falls under the latter category. The problem of minimizing the length of the loop in the presence of fairness constraints was shown to be NP-complete in [CGMZ95].

Our approach falls under the former category (minimizing the length of the stem), as do the directed model checking techniques presented in [YD98, ELL01, ELLL04, TAC$^+$04]. In [YD98], heuristics such as Hamming distance are used in a symbolic model checker to estimate the distance of the current state from an error state, and thereby guide state space exploration. In [ELL01, ELLL04], best-first search based on the A* algorithm [HNR68] is used in an explicit-state model checker to guide the search towards an error state.

Approaches that attempt to minimize the length of the loop in an explicit-state model checker include a BFS-based approach in [HK06] and a DFS-based approach in [GM07].

The work presented in this chapter is also published in [KG08].

8.9 Summary

In this chapter, we presented a model checking algorithm for a subset of CTL, called CETL, which produces short counterexamples, while simultaneously achieving state space reduction for the exhaustive validation of programs. Experimental results confirm that our approach can significantly outperform SPIN in the presence of errors, while providing state space reduction comparable to POR techniques. The effectiveness of our approach is highly dependent on the ability to identify crucial events during state space exploration. We have shown how crucial events can be identified in many cases, but the problem of finding crucial events for a general CETL formula remains open.

Part V

Conclusion

Chapter 9

Conclusion and Future Work

9.1 Conclusion

The focus of this dissertation has been on applying results from lattice theory to the problem of model checking concurrent and distributed systems.

The limiting factor in model checking large concurrent and distributed systems is state space explosion. In this dissertation, we established the usefulness of lattice theory in ameliorating state space explosion during the verification of concurrent and distributed systems. We showed that it is possible to exploit the *structure* exhibited by the the formulae being verified, to selectively explore a portion of the state space, rather than the entire state space. Specifically, we showed that the set of states satisfying formulae belonging to some "efficient" logics exhibit a lattice structure. We used this knowledge to devise efficient algorithms for state space traversal, which avoid construction of the entire state space.

The ability to produce a counterexample illustrating how an error state is reached is a central feature of model checking tools. Counterexamples are generated for human consumption, to aid in the task of debugging. Consequently, the production of *short* counterexamples is highly desirable. We have shown that lattice theory can be used to produce short counterexamples, containing only events that are both necessary and sufficient to lead to an error state.

A model checking framework has three basic elements [McM92]:

1. A formal language for specifying the properties to be verified.
2. A mathematical model of the system to be verified.
3. A set of algorithms that can be mechanically applied to prove these properties on the model.

This dissertation makes contributions in each of the above three areas.

Previous researchers [CL85, KP87, CBDGF95, CG95, GM01] have identified various classes of properties based on the structure exhibited by states that satisfy these properties in a computation, and used the exhibited structure to avoid state space explosion during verification. In this dissertation, we have shown that the problem of deciding membership in these classes is co-NP-complete in the size of the trace. In particular, we showed that the decision problem of identifying whether a given formula is meet-closed [CG95] or regular [GM01] is co-NP-complete, and the problem of deciding whether a given formula is stable [CL85] or observer-independent [KP87, CBDGF95] is NP-complete.

We extended the work of Sen and Garg [SG03a, Sen04] in studying the lattice-theoretic characteristics of CTL operators. We showed that the existential until (EU) and existential release (ER) operators of CTL preserve regularity. We also showed that the CTL operators EF, AF, EG and AG preserve biregularity. These results allow us to build a specification logic that yields formulae whose structure can then be exploited to derive efficient verification algorithms.

We showed how Mazurkiewicz trace semantics [Maz89] can be used in conjunction with a vector timestamping mechanism [Fid88, Mat89] to obtain a set of partial orders, called a *trace cover*, that encode all the reachable states of the program. This provides a compact representation of the state space of the program. Our method involves exploring only a single interleaving per maximal program trace to obtain the trace cover, thereby avoiding state space explosion during model construction.

We applied algorithms developed for the verification of reachability properties on partial order representations to finite trace covers, in order to decide reachability in the program. In particular, we applied Chase and Garg's [CG95] algorithm for reachability detection of meet-closed predicates to verify safety properties on several famous distributed protocols. We also applied Tomlinson and Garg's [TG97] algorithm for deciding reachability for 0-1 sum predicates to verify properties on these protocols. These algorithms avoid state space explosion by efficiently extracting the necessary information from the partial order trace representation, without constructing the entire state space. We provided experimental results from an implementation based on SPIN, which corroborated the effectiveness of these algorithms in achieving state space reduction.

We showed that *predicate filtering* can be used on a finite trace cover to reduce the state space for verifying classes of properties for which efficient verification algorithms do not exist. Predicate filtering allows us to take advantage of polynomial-time reachability algorithms to generate a reduced state space w.r.t. a weaker property than the one to be verified. The reduced state space contains all the states of the program that satisfy the weaker property (and hence, the original property as well), but is typically exponentially smaller than the full state space of the program.

We showed that lattice theory can also be used for state space reduction in an interleaving representation of the state space. We developed efficient model checking algorithms for a subset of CTL, which we call CETL, that consists entirely of regular formulae. Not only do these algorithms provide significant state space reduction by avoiding the exploration of multiple interleavings of concurrent events, but they also result in the production of short error trails, thereby reducing debugging effort. Our model checking algorithms explore only those events which are both necessary and sufficient to verify the specification. Further, it explores only a single interleaving of such events per maximal program trace. CETL consists of the existential until and release operators of CTL, and the conjunction operator. CETL can express most safety and liveness properties. We implemented these algorithms as an extension to SPIN, which we call SPICED (Simple PROMELA Interpreter with Crucial Event Detection). Experimental results compared our performance against that of SPIN with partial order reduction, and demonstrated that we consistently produced significantly shorter error trails than SPIN. This ability to find errors at shorter depths also resulted in faster run times and less memory consumption, compared to SPIN with POR. Furthermore, the amount of state space reduction achieved was comparable to POR techniques.

9.2 Future Work

Our model checking algorithms rely on the ability to identify crucial events. In particular, we need to be able to efficiently identify a crucial event to execute from the current state. In [KG05b], we showed that unless RP=NP, there is no polynomial-time algorithm for determining a crucial event for a meet-closed formula. In other words, there is unlikely to be a *deterministic* polynomial-time algorithm for finding a crucial event for a general meet-closed formula.

For some kinds of formulae, such as conjunctions of local predicates, the crucial event can be identifed in time that is linear in the size of the formula. We also identified cases where crucial events could be identified for CETL formulae. However, the time complexity of identifying a crucial event for a general CETL formula remains an open problem, and is a direction for future research.

Our algorithms for verification on partial order models are currently limited to deciding reachability, that is, deciding whether the initial state of a program satisfies $EF(\phi)$, where ϕ is a formula that contains no temporal operators. A future direction of research is to extend these algorithms to include other CTL operators, such as EG and AF, to enable the efficient verification of liveness properties on partial order models. The finite trace cover model succintly encodes all the reachable *states* of a program. However, this representation loses information about complete *paths* in the original program, because of the use of cutoff events to truncate state space construction. The verification of liveness properties requires information about infinite paths, which is not preserved in the finite trace cover model. Thus, in order to verify liveness properties, the finite trace cover model needs to be enhanced to include information that allows a verification algorithm to traverse complete paths, including cycles, in a computation. Beyond the verification of additional CTL operators, future work on verification using finite trace covers would entail the verification of formulae containing nested temporal operators.

For an interleaving state space representation, we presented model checking algorithms for the logic CETL. While CETL can express many interesting safety and liveness properties, it is strictly less expressive than CTL. In particular, CETL does not contain the disjunction operator, and negation can only be applied to atomic propositions. A future direction of research would be to apply lattice-theoretic methods to improve the efficiency of model checking the full CTL logic.

In this dissertation, we used the mechanism of predicate filtering for state space reduction. Another potential application of this technique lies in the area of program repair. There has recently been some research interest in the use of automated techniques to find the *cause* of an error encountered during verification. For example, Ball *et. al.* [BNR03] examine all executions that lead to the same control location (program counter) as that of error state, and attribute the error to those transitions that are present only in an incorrect execution, and never in a correct one. A similar approach was also proposed by Groce and Visser [GV03].

In predicate filtering, additional dependencies are added between events in a trace, thereby reducing the number of possible interleavings in the trace. The aim of adding these additional dependencies is to produce a sublattice of the down-set lattice corresponding to the original trace. Given a predicate ϕ, predicate filtering tries to produce a sublattice that contains only all the states that satisfy ϕ. Suppose a property ϕ is required to be an invariant in a trace, that is, it is a correctness requirement that ϕ must hold in every reachable state of the trace. If ϕ is a regular formula, filtering each maximal trace of P w.r.t. ϕ retains all the states that satisfy ϕ, while eliminating all the states that do *not* satisfy ϕ. In other words, all undesirable behaviors are eliminated, while all correct behaviors are maintained. Exploring this application of predicate filtering for program repair is another direction for future research.

Bibliography

[AG05] Anurag Agarwal and Vijay K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *PODC '05: Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, pages 19–28, New York, NY, USA, 2005. ACM.

[AG07] Anurag Agarwal and Vijay K. Garg. Predicate detection on infinite computations. Technical Report TR-PDS-2006-001, ECE Dept., University of Texas at Austin, http://maple.ece.utexas.edu/TechReports/2006/TR-PDS-2006-001.ps, 2007.

[And90] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[AT96] Rajeev Alur and Gadi Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

[BC87] W. W. Rouse Ball and H. S. M. Coxeter. *Mathematical Recreations and Essays*. Dover, 1987.

[BHSV^{+}96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A.

Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 428–432, London, UK, 1996. Springer-Verlag.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–105, New York, NY, USA, 2003. ACM Press.

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.

[CBDGF95] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1):157–179, 1995.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[CG95] Craig M. Chase and Vijay K. Garg. Efficient detection of restricted classes of global predicates. In *WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 303–317, London, UK, 1995. Springer-Verlag.

[CGMP99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[CGMZ95] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 427–432, New York, NY, USA, 1995. ACM.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, 1984.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[CV04] E. M. Clarke and H. Veith. Counterexamples revisited: Principles, algorithms, applications. *Verification: Theory and Practice, Lecture Notes in Computer Science*, 2772:208–224, 2004.

[CVWY92] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Dij72] E. W. Dijkstra. Hierarchical ordering of sequential processes. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques*, New York, NY, 1972. Academic Press.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3:245–260, 1982.

[DM41] B. Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63:600–610, 1941.

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

[DWG97] Om P. Damani, Yi-Min Wang, and Vijay K. Garg. Optimistic distributed simulation based on transitive dependency tracking. *SIGSIM Simulation Digest*, 27(1):90–97, 1997.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 57–79, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[ELLL04] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 6(4), 2004.

[Eme90] E. Allen Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072, 1990.

[ERV96] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. In *TACAS '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 87–106, London, UK, 1996. Springer-Verlag.

[Esp94] Javier Esparza. Model checking using net unfoldings. In *TAPSOFT '93: Selected papers of the colloquium on Formal approaches of software engineering*, pages 151–195, Amsterdam, The Netherlands, The Netherlands, 1994. Elsevier Science Publishers B. V.

[Fid88] C. J. Fidge. Partial orders for parallel debugging. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 183–194, New York, NY, USA, 1988. ACM.

[Fis85] Michael J. Fischer. Real-time mutual exclusion. Unpublished manuscript, 1985.

[Gar02] Vijay K. Garg. Algorithmic combinatorics based on slicing posets. In *FSTTCS '02: Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science*, pages 169–181, London, UK, 2002. Springer-Verlag.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[GM01] Vijay K. Garg and Neeraj Mittal. On slicing a distributed computation. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 322–329, Phoenix, AZ, USA, May 2001.

[GM07] Paul Gastin and Pierre Moro. Minimal counterexample generation for SPIN. In *SPIN 2007: Proceedings of the 14th International SPIN Workshop on Model Checking Software*, pages 24–38, 2007.

[GMS03] Vijay K. Garg, Neeraj Mittal, and Alper Sen. Applications of lattice theory to distributed computing. *ACM SIGACT Notes*, 34(3):40–61, sep 2003.

[God91] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Vabderlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

[Gol87] Robert Goldblatt. *Logics of time and computation.* Center for the Study of Language and Information, Stanford, CA, USA, 1987.

[Gro93] Bojan Groselj. Bounded and minimum global snapshots. *IEEE Parallel and Distributed Technology*, 1(4):72–83, 1993.

[GS01] Vijay K. Garg and Chakarat Skawratananond. String realizers of posets with applications to distributed computing. In *PODC '01: Proceedings* *of the twentieth annual ACM symposium on Principles of distributed computing*, pages 72–80, New York, NY, USA, 2001. ACM.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, Oregon, May 9–10, 2003.

[GW92] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 332–342, London, UK, 1992. Springer-Verlag.

[GW94a] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(3):299–307, 1994.

[GW94b] Patrice Godefroid and Pierre Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305–326, 1994.

[Hel99] Keijo Heljanko. Deadlock and reachability checking with finite complete prefixes. Research Report A56, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, Espoo, Finland, December 1999.

[Hel00] Keijo Heljanko. Model checking with finite complete prefixes is pspace-complete. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 108–122, London, UK, 2000. Springer-Verlag.

[HJ04] Gerard J. Holzmann and R. Joshi. Model-driven software verification. In *SPIN '04: Proceedings of the 11th International SPIN Workshop* *on Model Checking of Software*, pages 77–92, Barcelona, Spain, April 2004. Springer-Verlag, LNCS 2989.

[HK06] Henri Hansen and Antti Kervinen. Minimal counterexamples in $O(n\ log\ n)$ memory and $O(n^2)$ time. In *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 133–142, Washington, DC, USA, 2006. IEEE Computer Society.

[HLP01] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hol91] Gerard J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Hol01] Gerard J. Holzmann. Economics of software verification. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 80–89, New York, NY, USA, 2001. ACM.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, September 2003.

[Hol07] Gerard Holzmann. On-the-fly LTL model checking with SPIN. `http://spinroot.com/spin/`, 2007.

[HP95] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Con-*

ference on Formal Description Techniques VII, pages 197–211, London, UK, UK, 1995. Chapman & Hall, Ltd.

[KG05a] Sujatha Kashyap and Vijay K. Garg. Exploiting predicate structure for efficient reachability detection. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 4–13, New York, NY, USA, 2005. ACM.

[KG05b] Sujatha Kashyap and Vijay K. Garg. Intractability results in predicate detection. *Information Processing Letters*, 94(6):277–282, June 2005.

[KG06] Sujatha Kashyap and Vijay K. Garg. State space reduction using predicate filters. Technical Report TR-PDS-2006-003, ECE Dept., University of Texas at Austin, http://maple.ece.utexas.edu/TechReports/2006/TR-PDS-2006-003.pdf, 2006.

[KG08] Sujatha Kashyap and Vijay K. Garg. Producing short counterexamples using crucial events. In *CAV '08: Proceedings of the 17th International Conference on Computer Aided Verification (to appear)*. Springer-Verlag, 2008.

[KL00] Moataz Kamel and Stefan Leue. VIP: A visual editor and compiler for v-PROMELA. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 471–486, London, UK, 2000. Springer-Verlag.

[KP87] Shmuel Katz and Doron Peled. Interleaving set temporal logic. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 178–190, New York, NY, USA, 1987. ACM.

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[KPM01] S. H. Kan, J. Parrish, and D. Manlove. In-process metrics for software testing. *IBM Systems Journal*, 40(1):220–241, 2001.

[KT07] Danny Kopec and Suzanne Tamang. Failures in complex systems: case studies, causes, and possible remedies. *SIGCSE Bull.*, 39(2):180–184, 2007.

[Kwi89] Marta Z. Kwiatkowska. Event fairness and non-interleaving concurrency. *Formal Aspects of Computing*, 1:213–228, 1989.

[Laf03] Alberto Lluch Lafuente. Symmetry reduction and heuristic search for error detection in model checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam78] L. Lamport. Time, clock and the ordering of events in a distributed system. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[Lam83] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[LLEL02] Alberto Lluch-Lafuente, Stefan Edelkamp, and Stefan Leue. Partial order reduction in directed model checking. In *Proceedings of the 9th*

International SPIN Workshop on Model Checking of Software, pages 112–127, London, UK, 2002. Springer-Verlag.

[LPY01] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gearbox controller. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Distributed Algorithms*, pages 215–226, 1989.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[Maz77] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI PB 78, Aarhus University, Aarhus, 1977.

[Maz84] Antoni W. Mazurkiewicz. Traces, histories, graphs: Instances of a process monoid. In *Proceedings of the Mathematical Foundations of Computer Science 1984*, pages 115–133, London, UK, 1984. Springer-Verlag.

[Maz85] Antoni W. Mazurkiewicz. Semantics of concurrent systems: a modular fixed-point trace approach. In *Proceedings of the European Workshop on Applications and Theory in Petri Nets, covers the last two years which include the workshop 1983 in Toulouse and the workshop 1984 in Aarhus, selected papers*, pages 353–375, London, UK, 1985. Springer-Verlag.

[Maz87] A Mazurkiewicz. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[Maz89] Antoni W. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 285–363, London, UK, 1989. Springer-Verlag.

[McM92] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[McM93] Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV '92: Proceedings of the Fourth International Workshop on Computer Aided Verification*, pages 164–177, London, UK, 1993. Springer-Verlag.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[MG01] Neeraj Mittal and Vijay K. Garg. Computation slicing: Techniques and theory. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 78–92, London, UK, 2001. Springer-Verlag.

[MG03] Neeraj Mittal and Vijay K. Garg. Software fault tolerance of distributed programs using computation slicing. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing*

Systems, page 105, Providence, Rhode Island, USA, 2003. IEEE Computer Society.

[MG05] Neeraj Mittal and Vijay K. Garg. Techniques and applications of computation slicing. *Distrib. Comput.*, 17(3):251–277, 2005.

[MMB08] Thierry Massart, Cédric Meuter, and Laurent Van Begin. On the complexity of partial order trace model checking. *Information Processing Letters*, 106(3):120–126, 2008.

[MP79] Zohar Manna and Amir Pnueli. The modal logic of programs. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 385–409, London, UK, 1979. Springer-Verlag.

[MSGA04] Neeraj Mittal, Alper Sen, Vijay K. Garg, and Ranganath Atreya. Finding satisfying global states: all for one and one for all. In *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 66–75, April 2004.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[OG07] Vinit Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *DISC '07: Proceedings of the 21st International Conference on Distributed Computing*, pages 420–434, London, UK, 2007. Springer-Verlag.

[PBG04] B. Pochon, S. Baehni, and R. Guerraoui. The driving philosophers. In *TCS'04: Proceedings of the 3rd IFIP International Conference on Theoretical Computer Science*, 2004.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[Pel07] Radek Pelanek. BEEM: BEnchmarks for Explicit Model checkers. `http://anna.fi.muni.cz/models/index.html`, 2007.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universitt Bonn, Schriften des Instituts fr Instrumentelle Mathematik Nr. 3., 1962.

[Pet83] Gary L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):56–65, 1983.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pnu81] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu86] A Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. *Current trends in concurrency. Overviews and tutorials*, pages 510–584, 1986.

[PP94] Doron Peled and Amir Pnueli. Proving partial order properties. *Theoretical Computer Science*, 126(2):143–182, 1994.

[Pra86] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[PVK01] Doron Peled, Antti Valmari, and Ilkka Kokkarinen. Relaxed visibility enhances partial order reduction. *Formal Methods for System Design*, 19(3):275–289, 2001.

[PW84] Shlomit S. Pinter and Pierre Wolper. A temporal logic for reasoning about partially ordered computations (extended abstract). In *PODC '84: Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 28–37, New York, NY, USA, 1984. ACM.

[PW97] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[RA81] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[SECH98] Francis Schneider, Steve M. Easterbrook, John R. Callahan, and Gerard J. Holzmann. Validating requirements for fault tolerant systems using model checking. In *ICRE '98: Proceedings of the 3rd International Conference on Requirements Engineering*, pages 4–13, Washington, DC, USA, 1998. IEEE Computer Society.

[Sen04] Alper Sen. *Techniques for Formal Verification of Concurrent and Distributed Program Traces.* PhD thesis, University of Texas at Austin, May 2004.

[SG02] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates on the happened-before model. In *IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, pages 76–83, Washington, DC, USA, April 2002. IEEE Computer Society.

[SG03a] Alper Sen and Vijay K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *OPODIS '03: Proceedings of the 7th International Conference on Principles of Distributed Systems*, pages 171–183, La Martinique, France, December 2003.

[SG03b] Alper Sen and Vijay K. Garg. Partial Order Trace Analyzer (POTA) for distributed programs. In *RV'03: Proceedings of the Workshop on Runtime Verification*, pages 105 – 113, Boulder, Colorado, USA, 2003. Electronic Notes on Theoretical Computer Science (ENTCS), vol. 89.

[SL85] Fred B. Schneider and Leslie Lamport. Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985 Munich*, pages 431–480, London, UK, 1985. Springer-Verlag.

[SY85] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[Szy90] B. K. Szymanski. Mutual exclusion revisited. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 110–119, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[TAC$^+$04] Jianbin Tan, George S. Avrunin, Lori A. Clarke, Shlomo Zilberstein, and Stefan Leue. Heuristic-guided counterexample search in FLAVERS. *SIGSOFT Softw. Eng. Notes*, 29(6):201–210, 2004.

[TG93] Alexander I. Tomlinson and Vijay K. Garg. Detecting relational global predicates in distributed systems. In *PADD '93: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, pages 21–31, New York, NY, USA, 1993. ACM.

[TG97] Alexander I. Tomlinson and Vijay K. Garg. Monitoring functions on global states of distributed programs. *J. Parallel Distrib. Comput.*, 41(2):173–189, 1997.

[Val91a] Antti Valmari. A stubborn attack on state explosion. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 156–165, London, UK, 1991. Springer-Verlag.

[Val91b] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, London, UK, 1991. Springer-Verlag.

[Val93] Antti Valmari. On-the-fly verification with stubborn sets. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 397–408, London, UK, 1993. Springer-Verlag.

[Var01] Moshe Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22, London, UK, 2001. Springer-Verlag.

[VL93] Bart Vergauwen and Johan Lewi. A linear local model checking algorithm for CTL. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 447–461, London, UK, 1993. Springer-Verlag.

[VW86] Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS '86: Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wes89] C. H. West. Protocol validation in complex systems. *SIGCOMM Computer Communication Review*, 19(4):303–312, 1989.

[Win87] G. Winskel. Event structures. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 325–392, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[YD98] C. Han Yang and David L. Dill. Validation with guided search of the state space. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 599–604, New York, NY, USA, 1998. ACM.

Vita

Sujatha Kashyap was born in Srinagar, Jammu & Kashmir, India in 1977. She received her Bachelor of Technology degree in Computer Engineering from the National Institute of Technology, Karnataka in 1998. She graduated with a Master of Science degree in Computer Science from Texas A&M University in 2000. She started her Ph.D. program at the University of Texas at Austin in 2002. She has been an employee of IBM Corporation since 2000.

Permanent Address: 32, Jalvayu Vihar,
Kammanahalli Main Road,
Bangalore, India - 560043

This dissertation was typeset with $\LaTeX\,2_\varepsilon$[1] by the author.

[1]$\LaTeX\,2_\varepsilon$ 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, James A. Bednar, and Ayman El-Khashab.