ABSTRACTION REFINEMENT FOR LARGE SCALE MODEL CHECKING

CHAO WANG
NEC Laboratories America
Princeton, New Jersey

GARY D. HACHTEL
University of Colorado
Boulder, Colorado

FABIO SOMENZI
University of Colorado
Boulder, Colorado

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

List of Figures

List of Tables

Preface

This book summarizes our research work conducted in the University of Colorado at Boulder from 2000 to 2004 while the first author was pursuing his Ph.D. degree in the Department of Electrical and Computer Engineering. The Ph.D. dissertation won the 2003-2004 ACM Outstanding Ph.D. Dissertation Award in Electronic Design Automation. This ACM award, established by ACM SIGDA, is given each year to an outstanding Ph.D. dissertation that makes the most substantial contribution to the theory and/or application in the field of electronic design automation.

Our research addresses the problem of applying automatic abstraction refinement to the model checking of large scale digital systems. Model checking is a formal method for proving that a finite state transition system satisfies a user-defined specification. The primary obstacle to its widespread application is the capacity problem: the state-of-the-art model checker cannot directly handle most industrial-scale designs. Abstraction refinement, an iterative process of synthesizing a simplified model to help verifying the original model, is a promising solution to the capacity problem. In this book, several fully automatic abstraction refinement techniques are proposed to efficiently reach or come close to the simplest abstraction.

First, a fine-grain abstraction approach is proposed to keep the abstraction granularity small. With the advantage of including only the relevant information, the fine-grain abstraction is proved to be indispensable in verifying systems with complex combinational logics. A scalable game-based refinement algorithm called GRAB is proposed to identify the refinement variables based on the systematic analysis of all the shortest counterexamples. Compared to single counterexample guided refinement methods, this algorithm often produces a smaller abstract model that can prove or refute the same property.

Second, a compositional SCC analysis algorithm called DnC is proposed in the context of LTL model checking to quickly identify unimportant parts of the state space in previous abstractions and prune them away before verification is applied to the next abstraction level. With a speed-up of up to two orders of magnitude over standard symbolic fair cycle detection algorithms, DnC demonstrates the importance of reusing information learned from previous abstraction levels to help verification at the current level.

Finally, BDD based symbolic image computation and Boolean satisfiability check are revisited in the context of abstraction refinement. We propose two new algorithms in order to improve the computational efficiency of BDD based symbolic fixpoint computation and SAT based bounded model checking, by applying the idea of abstraction and successive refinements inside the two basic decision procedures.

Analytical and experimental studies demonstrate that the fully automatic abstraction refinement techniques proposed in this book are the key to applying model checking to large systems. The suite of fully automatic abstraction refinement algorithms has demonstrated significant practical importance. Some of these BDD and SAT based algorithms have been adopted by various commercial/in-house verification tools in industry.

Chao Wang, Gary D. Hachtel, Fabio Somenzi

April 2006

Chapter 1

INTRODUCTION

Our society is increasingly dependent on various electronic and computer systems. These systems are used in consumer electronics, automobiles, medical devices, traffic controllers, avionics, and space programs, etc. Many of these systems can be classified as critical systems—safety-critical, mission-critical, or cost-critical. Design errors in these critical systems are generally intolerable, since they either cost a lot of money, or cost lives. However, designing a flawless computer system is becoming harder as the size of the system keeps getting larger. In the hardware design community, for instance, functional verification has been identified as the bottleneck in the entire design process. According to ITRS (the International Technology Roadmap for Semiconductors [ITR03]), two thirds of a typical ASIC design budget goes into verification, and verification engineers frequently outnumber design engineers in large project teams. Still, over 60% of the IC designs require a second "spin" due to logic and functional level errors. Similar problems also exist in the software community, especially in the design and implementation of embedded and safety-related software systems (device drivers, air traffic control systems, security protocols, etc.). The vast majority of verification experts believe that formal analysis methods are indispensable in coping with this "verification crisis."

Traditional verification techniques are simulation and testing. Simulation is applied to a model of the product, while testing is applied to the product itself. The basic idea of simulation and testing is feeding in some test vectors and then checking the output for correctness. The disadvantage of this "trial-and-error" based approach is that all the possible input conditions must be checked in order to make sure the design is correct. However, even for pure combinational circuits, it is infeasible to enumerate all the possible input conditions except for very small designs. For sequential circuits, there can be an infinite number of input conditions due to the possibly unbounded number of time instances. Therefore, although simulation and testing are very useful in detecting "bugs" in the early stages of the design process, they are not suitable for certifying that the design meets the specification.

To get a mathematical proof that the design satisfies a given specification under all possible input conditions, one needs formal verification techniques. Model checking and theorem proving are two representatives of the existing formal verification techniques. Given a finite state model and a property expressed in temporal logics, a model checker can construct a formal proof when the model satisfies the property [CE81, QS81]. If the property fails, the model checker can show how it fails by generating a counterexample trace. Model checking is fully automatic in the sense that the construction of proof or refutation does not require the user's intervention. This is in contrast to the formal techniques based on theorem proving, which rely on the user's expertise in logics and deductive proof systems to complete the verification.

Model checking has been regarded as a potential solution to the "verification crisis" in the computer hardware design community. It is showing promise for many other applications as well, including real-time system verification [AHH96], parameterized system verification [EK03], and software verification [VB00, BMMR01, MPC$^+$02].

The primary obstacle to the widespread application of model checking to real-world designs is the capacity problem. Since model checking uses an exhaustive search of the state space of the model to determine whether a specification is true or false, the complexity of model checking depends on the number of states of the model as well as the length of the specification. Due to its exponential dependence on the number of state variables or memory elements, the number of states of the model can be extremely large even for a moderate-size model. This is known as the *state explosion* problem. A major breakthrough in dealing with state explosion was symbolic model checking [BCM$^+$90, McM94] based on Binary Decision Diagrams (BDDs [Bry86]). However, even with these symbolic techniques, the capacity of model checking remains limited: The state-of-the-art model checkers still cannot directly handle most industry-scale designs. In fact, symbolic model checkers often lose their robustness when the model has more than 200 binary state variables; at the same time, hardware systems become more and more complex because of Moore's law and the increasing use of high level hardware description languages (HDLs)—models with thousands or tens of thousands of state variables may yet look modest.

1.1 Background

Abstraction is an important technique to bridge the capacity gap between the model checker and large digital systems. When a system cannot be directly handled by the model checker, abstraction can be used to remove information that is irrelevant to the verification of the given property. We then build an abstract model which hopefully is much simpler and apply model checking to it. In doing so, an abstract interpretation [CC77], or a relation between the abstract system and the concrete system is created. For it to be useful in model checking, abstraction must preserve or at least partially preserve the property to be verified. There exist automatic abstraction techniques under which a certain subclass of temporal properties are preserved. For instance, bi-simulation based reduction [Mil71, DHWT91] preserves the entire propositional μ-calculus. However, property-preserving abstractions are either very hard to compute or do not achieve a drastic reduction [FV99], and therefore are less attractive in practice. A more practical approach is called property driven abstraction, which preserves or partially preserves only the property at hand. Along this line, Balarin *et al.* [BSV93], Long [Lon93], and Cho *et al.* [CHM$^+$96a] have studied various ways of deriving an abstract model from the concrete system for model checking.

Abstraction refinement was introduced by Kurshan [Kur94] in the context of model checking linear properties specified as ω-regular automata. In this paradigm, verification is viewed as an iterative process of synthesizing a simplified model that is sufficient to prove or refute the given property. In COSPAN [HHK96], the initial abstraction contains only the state variables in the property and leaves the other variables unconstrained. Since unconstrained variables can take arbitrary values, the abstract model is an *over-approximation* in the sense that it contains all possible execution traces of the original model, and possibly more. Therefore, when a linear time property holds in the abstract model, it also holds in the concrete model; when the property fails in the abstract model, however, the result is inconclusive. In the case of inconclusive result, the abstract model is refined by adding back some relevant but previously unconstrained variables. The key issue in abstraction refinement is to identify in advance which variable is relevant and which is not. Note that an over-approximated abstraction is applicable not only to linear properties specified as ω-regular automata, but also to other universal properties including LTL [Pnu77] and ACTL [CE81, EH83], because over-approximation suffices to prove these properties true.

For practical reasons, it is important to keep the abstraction refinement process fully automatic. Manual abstraction techniques can be very powerful when they are carried out carefully by experienced users.

However, it often requires a significant amount of user's intervention and in-depth knowledge of the design. In fact, manual abstraction is very labor intensive and can be error-prone even for skilled users, making it hard for verification to keep up with the design schedule in real industry settings. Therefore, fully automated abstraction techniques are far more attractive in practice. In abstraction refinement, a procedure typically starts with a coarse initial abstraction and then automatically augments the abstract model by iterative refinement.

The main challenge in abstraction refinement is related to the ability of generating a final abstract model that is as simple as possible. The final abstraction, or *deciding abstraction*, is the one that decides the truth of the property to be verified. One can always start with a very coarse initial abstraction and keep refining it until the abstraction becomes deciding. Therefore, the effectiveness of the refinement algorithm is critical in keeping the final abstract model small. Existing refinement algorithms can be classified into the following categories. Some refinement algorithms rely on information about the structure of the model, e.g., the pair-wise latch relation [LPJ$^+$96] or the variable dependency graph [LNA99]. Some refinement algorithms rely on the analysis of the set of approximate satisfying states of the given property produced in a previous model checking run, e.g., the operation-based refinement methods [PH98, JMH00]. Some refinement algorithms are driven by spurious abstraction counterexamples produced in a previous model checking run [CGJ$^+$00, WHL$^+$01, CGKS02, CCK$^+$02, GKMH$^+$03, MH04]; in these methods, the goal of refinement is to remove the abstract counterexamples that do not correspond to any real path in the concrete model. Other refinement algorithms rely on the analysis of unsuccessful bounded model checking runs [MA03, LWS03, GGYA03, LS04, LWS05, ZPH04, ZPHS05]. In the latter cases, unsatisfiability proofs of these bounded model checking instances directly induce abstract models that are sufficient for disabling all counterexamples of a certain length.

The simplicity of the final abstract model is bounded ultimately by the degree of locality of the given property in the model. In general, a high degree of locality is necessary for the success of abstraction refinement. For a property whose proof or refutation relies on detailed knowledge of the entire system, it is clear that abstraction refinement is ineffective. In practice, however, the properties used in model checking are often partial specifications of the system behavior, and user-specified properties tend to depend on only part of the system. This is largely due to the structured programming or design style adopted by engineers. In this case, it is the refinement algorithm's responsibility to exploit fully the degree of locality of a given property. To measure the quality of different abstraction refinement algorithms, we define an important metric called *abstraction efficiency* as follows:

$$\eta = (1 - \frac{\text{final abstract model size}}{\text{original model size}}) \ .$$

For every pair of model M and property ϕ, there exists an optimum or maximum abstraction efficiency η^*. Note that η^* is a property of the specific verification problem $\langle M, \phi \rangle$, not a property of the abstraction refinement algorithm. As a heuristic principle, the closer to the optimum value it can achieves, the better a certain abstraction refinement algorithm is.

Another important metric for abstraction refinement is the *rate of convergence*. This characterizes how quickly a refinement algorithm converges from the initial abstract model to a deciding abstraction. In practice, this can be measured either by the number of refinement iterations or by the overall run time. We have observed cases for which some algorithms converge quickly to a near optimal abstraction while other algorithms spend a lot of time searching in vain for such an abstraction. In the ideal case, an algorithm should find, at each abstraction refinement iteration, a set of refinement variables that is a subset of an optimum deciding abstraction.

1.2 Our Contributions

This book deals with the main challenge in abstraction refinement, i.e., the ability to efficiently reach or come close to the optimum deciding abstraction. We propose several fully automatic abstraction techniques in order to improve the overall computation efficiency as well as the rate of convergence. Together, they address the following three problems that are critical in the abstraction refinement loop:

1 How to make the abstraction more concise?

2 How to identify and reuse critical information from previous abstraction levels?

3 How to make the basic decision procedures used in abstraction refinement more efficient?

In order to achieve a higher abstraction efficiency, it is crucial to keep the refinement granularity small so that only the relevant information is included in the abstract model. That is, each successive refinement should include only variables that are present in an optimal or near-optimal deciding abstraction. In previous work, the abstraction granularity is often limited at the state variable level: the entire fan-in combinational logic cone of a state variable is either included in or completely excluded from the abstract model. However, it is often the case that not every one of these fan-in logic gates is necessary for the verification of a certain property, even if the state variable itself is indeed necessary. Including these redundant logic gates often significantly increases the complexity of the abstract model—an abstract model with few state variables may end up containing a large number of logic gates.

In this book, we propose a fine-grain abstraction approach to push the granularity of abstraction beyond the usual state variable level. Boolean network variables are selectively inserted into large combinational logic cones to partition them into smaller pieces. In the abstraction as well as the successive refinements, Boolean network variables are given the same status as state variables—both are considered as *atoms.* With this approach, refinement strategies must search a two-dimensional space. Refinement in the *sequential* direction is comprised of the addition of new state variables only, which is typical of much of the prior art [LPJ+96, JMH00, CGJ+00, CGKS02]. Refinement in the *Boolean* direction is comprised of the addition of Boolean network variables only, which does not increase the number of abstract states but refines the transition relation among them. Although cut-set variables that are similar to Boolean network variables were used in the previous work of Wang *et al.* [WHL+01] and Glusman *et al.* [GKMH+03], these variables were not treated the same as state variables during refinement. We shall show that by separating the two refinement directions and carefully controlling the direction at each iteration, we can produce refinement variable sets that are significantly more concise.

Spurious counterexamples in an abstract model have been used in previous work to compute the set of refinement variables. With the exception of [GKMH+03], the prior art of counterexample based refinement relies exclusively on a single counterexample. In practice, however, there can be an extremely large number of spurious counterexamples when the property fails. In that case, arbitrarily picking up one counterexample and use it to drive the refinement is "a-needle-in-the-haystack" approach. In this book, we present a way to capture, for invariant properties, all the shortest counterexamples using a data structure called the Synchronous Onion Rings (SORs). A new refinement algorithm, called GRAB, is proposed to identify the refinement variables by systematically analyzing all the shortest counterexamples. GRAB has two novel features: First, it takes a *generation* of refinement steps to systematically eliminate all spurious counterexamples supported by a given set of SORs. Second, each refinement step in the current generation is computed using a scalable game-based strategy that depends solely on the current abstract model. Note that being able to compute the refinement without using the concrete model is crucial to the scalability of the algorithm, since the working assumption is that the concrete model is large and any computation on it is prohibitively expensive. In contrast, previous refinement methods in [CGJ$^+$00, CGKS02, CCK$^+$02] do not scale well, because they rely on computation in the concrete model.

Due to the global guidance from the SORs, and the quality and scalability of the game-based variable selection computation, GRAB demonstrates significantly advantages over these previous refinement algorithms – it can solve significantly larger problems, require less memory and less CPU time. Although the method in [GKMH$^+$03] is also driven by multiple counterexamples, it does not guarantee to capture each and every one of the shortest counterexamples. As a result, this refinement method is often less accurate than the SOR based refinement and is incapable of catching concretizable counterexamples at the earliest possible refinement step.

Proof based abstraction methods in [MA03, LWS03, GGYA03, LS04, LWS05, ZPH04, ZPHS05] captures implicitly all the shortest counterexamples. However, these are SAT based methods and rely on a SAT solver to produce the unsatisfiability proof of a SAT instance in the concrete model. In contrast, our core refinement variable selection algorithm is pure BDD based, even though we use SAT as well in concretization test and in predicting the refinement direction. We note that a small unsatisfiability proof, i.e., the one with a small subset of Boolean variables or clauses, does not automatically give a small refinement set [LS04, GGA05]. Both proof-based and counterexample based methods have their own advantages and disadvantages. A detailed experimental comparison of GRAB with a proof-based refinement algorithm can be found in [LWS05], showing that these two methods complement each other on the various test cases. Amla *et al.* [ADK$^+$05] also published results of their experimental evaluation of the various SAT based abstraction methods. There is also a trend of combining counterexample based methods and proof-based methods in abstraction refinement [AM04].

In abstraction refinement, we need to model check the abstract model repeatedly while it is gradually refined. Information gathered at previous abstraction levels can be carried on and be used to speed up the verification at the current level. In this book, we propose a compositional SCC (Strongly Connected Component) analysis algorithm, called DNC, to quickly identify unimportant parts of the state space and prune them away before going to the next abstraction. The search state space is also disjunctively decomposed into smaller subspaces that can be checked in isolation. Although there exist several symbolic SCC algorithms in the prior art [HTKB92, XB00, BGS00, GPP03] and some of them have been applied to model checking [FFK$^+$01, SRB02], these methods are not compositional, and would not be effective on the larger practical examples studied in this book. In this book, we also prove that the strength of an SCC or a set of SCCs decreases monotonically with refinement, which allows the model checking algorithm to tailor the proof to the strength of the SCC at hand. The concept of *automaton strength* was due to Kupferman and Vardi [KV98] and Bloem *et al.* [BRS99]. Although the strength of the automaton was used in [BRS99] to improve LTL model checking, we believe that DnC is the first to systematically exploit this important property in the context of abstraction refinement.

The idea of abstraction followed by the successive refinements is also applied to the two basic decision procedures used in model checking: BDD based symbolic image computation and Boolean Satisfiability (SAT) check. Image computation [CBM89a, GB94] accounts for most of the CPU time in BDD based symbolic model checking. The peak sizes of the BDDs produced during the computation are essential in determining whether or how fast image computation can be completed on a given computer. In this book, we propose a novel image computation algorithm called FarSide image, to reduce the peak BDD size inside image computation by minimizing the transition relation with over-approximated images as care sets. Exact and approximate reachable states have been widely used to improve image computation since the early work of Ranjan *et al.* [RAB$^+$95] and Moon *et al.* [MJH$^+$98]. However, BDD minimization was effective only when being applied to the *near side*, or present-state variables of the transition relation. The FarSide image algorithm is the first to achieve a significant performance gain by applying BDD minimization to the *far side*, or next-state variables of the transition relation. It may seem surprising that significant improvements to the low level BDD work routines can be obtained long after the time when BDD methods were a consistent focus in the relevant conferences and journals. From our discussion and presented results, it should be clear that these improvements are obtained only when compositional methods are applied to models that are much larger than previously considered.

Deciding the SAT problem of a Boolean formula is a fundamental computation in Bounded Model Checking (BMC [BCCZ99]). In BMC, we search for counterexamples of a finite length in the given model, and the existence of a finite-length counterexample is formulated into a Boolean formula that is satisfiable if and only if a counterexample exists. When the Davis-Longeman-Loveland recursive search procedure [DLL62] (implemented in many modern SAT solvers) is used to solve the SAT problem, the variable decision ordering affects the performance significantly. In this book, we propose a new algorithm to compute a good variable decision order for the series of SAT problems in BMC. The new algorithm exploits the fact that the SAT problems in BMC are highly correlated, and therefore information learned from previous problems can help solving the current problem. The new variable ordering is computed based on the analysis of the unsatisfiability proofs of previous SAT instances, and is gradually refined as the BMC unrolling depth keeps increasing. Shtrichman also studied in [Sht00] the use of static ordering to improve the SAT search in BMC. However, his method is based primarily on the unrolled circuit structure, and therefore is completely orthogonal to ours. Due to the strong correlation among different SAT instances in BMC, applying our new decision ordering can significantly reduce the sizes of the SAT search trees and therefore improve the overall performance of BMC.

To summarize, all the new techniques proposed in this book are fully automatic and are crucial at improving the performance of abstraction refinement. Their application to model checking can significantly increase the model checker's ability to handle large designs. Our experimental studies on real-world benchmark circuits indicate that these automatic abstraction refinement techniques are the key to applying model checking to industrial-scale systems.

1.3 Organization of This Book

This book has nine chapters. Chapter 2 is an introduction to the basic concepts and notations commonly used in model checking, including finite state models, temporal logics, Büchi automata, symbolic model checking, bounded model checking, and abstraction refinement. This chapter should be an easy reading for those who are familiar with model checking. We have also tried to make the materials easily accessible to readers who are in the general areas of computer science but not very familiar with model checking. From Chapter 3 to Chapter 8, we present our main research contributions in details.

In Chapter 3, we introduce the notion of abstraction granularity and present the FINE-GRAIN abstraction approach. We use the simulation relation between the abstract and concrete models to explain why model checking of the abstract system may be conservative. We present the data structure of the SORs to capture all the shortest abstract counterexamples. We show how to use SAT based multi-thread concretization test to decide whether the abstract counterexamples are real or not.

In Chapter 4, we present the Grab refinement algorithm for selecting refinement variables based on a two-player reachability game in the abstract model. At each refinement iteration, we show how to decide the appropriate refinement direction using a SAT check. In both refinement directions, a greedy generational minimization is used at the end to remove redundant refinement variables. Finally, we discuss the use of sequential don't cares to constrain the behavior of the abstract model.

In Chapters 5 and 6, we address the important problem of carrying on information from previous abstraction levels to the current level, and applying it to speed up model checking. We present a compositional SCC analysis algorithm called DnC for model checking LTL properties. Information learned from previous abstraction levels is used to restrict the search for fair cycles at the current abstraction level. We will explain the use of SCC strength reduction, disjunctive state space decomposition, and guided search for fair cycles in the general framework of abstraction refinement.

In Chapters 7 and 8, we apply the idea of abstraction and successive refinements to the basic symbolic computation algorithms in model checking. In Chapter 7, we focus on improving the performance of BDD based symbolic image computation and present the FarSide image computation algorithm. In Chapter 8, we discuss the variable decision ordering of a SAT solver based on the DLL procedure in the context of bounded model checking. We then present a new variable ordering algorithm to improve the performance of the SAT checks in BMC. In both chapters, we conduct experiments to demonstrate the effectiveness of the proposed techniques.

We conclude in Chapter 9 and point out some interesting research directions.

Chapter 2

SYMBOLIC MODEL CHECKING

Model checking [CE81, QS81] is an algorithmic method for proving that a digital system satisfies a user-defined specification. Both the system and the specification must be formally specified: The model of the system must have a finite number of states; the specification, or property, is often expressed in temporal logics. In the model checking literature, the model and the property are often represented by the Kripke structure and a temporal logic formula, respectively.

Given a model K and a property ϕ, model checking is used to check whether K models ϕ, denoted by $K \models \phi$. For properties specified in Computational Tree Logic (CTL [CE81, EH83]), the model checking problem can be solved by a set of least and/or greatest fixpoint computations [CES86]. For properties specified in Linear Time Logic (LTL [Pnu77]), model checking is often transformed into language emptiness checking in a generalized Büchi automaton. In this *automata-theoretic* approach [VW86], the negation of the given LTL formula is encoded into a Büchi automaton, which is then composed with the model. The LTL model checking problem is then decided by checking the language of the composed system—the model satisfies the property if and only if the language of the composed system is empty. Therefore, the underlying LTL model checking algorithms are usually variants of algorithms for computing Strongly-Connected Components (SCCs).

In this chapter, we first introduce the basic concepts and notations commonly used in model checking. We then review some of the fundamental algorithms in symbolic model checking, which includes BDD based symbolic fixpoint computation, SCC hull and SCC enumeration algorithms, SAT and bounded model checking, and iterative abstraction refinement.

2.1 Finite State Model

In model checking, we deal with a formal model of the given digital system, known as the Kripke structure. A Kripke structure is an annotated finite-state transition graph.

Definition 2.1 *A Kripke structure is a 5-tuple*

$$K = \langle S, S_0, T, A, \Lambda \rangle \ ,$$

where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, A is a finite alphabet for which a set P of atomic propositions is given and $A = 2^P$, and $\Lambda : S \to A$ is the labeling function.

We further require that the transition relation of a Kripke structure be complete; that is, every state has at least one successor. With this assumption, we can extend any finite state path in the state transition graph into an infinite one.

As the standard representation of models in the model checking literature, the Kripke structure has its origin in modal logic, the generalization of temporal logic. In modal logic, a certain formula is interpreted with respect to a state inside a universe, a domain of discourse, and a relation establishing how the validity of a predicate changes from state to state. Temporal logic is a special case of modal logic that allows us to reason about how predicates evolve over time. In temporal logic model checking, a node or state of the Kripke structure represents the "state" of the given system at a certain time, and the change from state to state represents a time change.

From an engineer's point of view, the Kripke structure is nothing but a labeled finite state machine (FSM). The additional features, i.e., the finite alphabet and a labeling function from states to sets of atomic propositions, make it possible to specify simple propositional properties on the finite state machine. These propositional properties, combined with some temporal operators, allow us to specify properties like "$\neg abort$ holds on all the states reachable from the initial states" or "from a state labeled *req* we will eventually reach a state labeled *ack*." We will introduce temporal logic operators in the next section. Now let us focus on propositional properties and take a look at the example FSM at the right-hand side of Figure 2.1.

The FSM in Figure 2.1 has four states, among which three are reachable from the single initial state a. Propositions p and q belong to the finite alphabet. With the labeling function and initial predicate indicated in Figure 2.1, the finite state machine is augmented into a Kripke

state	encoding	
	x_1	x_0
a	0	0
b	0	1
c	1	0
d	1	1

$$p = \neg x_0$$

$$q = x_1 \wedge x_0$$

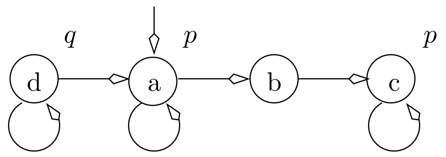


Figure 2.1. An example of the Kripke structure.

structure defined as follows:

$$\begin{array}{lcl}
S & = & \{a, b, c, d\} \\
S_0 & = & \{a\} \\
T & = & \{(a,a), (a,b), (b,c), (c,c), (d,d), (d,a)\} \\
P & = & \{p, q\}
\end{array}
\qquad
\begin{array}{lcl}
\Lambda(a) & = & \{p\} \\
\Lambda(b) & = & \{\ \} \\
\Lambda(c) & = & \{p\} \\
\Lambda(d) & = & \{q\}
\end{array}$$

Given a sequential circuit, the construction of the finite state machine from the system description is straightforward. A digital circuit is often defined as an entity with memory elements (latches and flip-flops), combinational logic gates, input signals, and internal wires. The transition functions of the memory elements are defined in terms of the current values of these memory elements and the input signals.

Figure 2.2 gives an example circuit, in which we use the variables x_1 and x_0 to represent the outputs of the two registers, and variables y_1 and y_0 to represent their data inputs. Note that after a clock cycle, the values of y_1 and y_0 will be propagated to the register outputs. Therefore, we often call x_1 and x_0 the present-state (or current-state) variables, and call y_1 and y_0 the next-state variables. In this example, we use the variable w_0 to represent the value of a primary input signal.

States in the corresponding FSM are mapped to the different valuations of the set of memory elements. Edges in the state transition graph correspond to the changes of states among different clock cycles. For the example in Figure 2.2, we can write out the transition functions of the

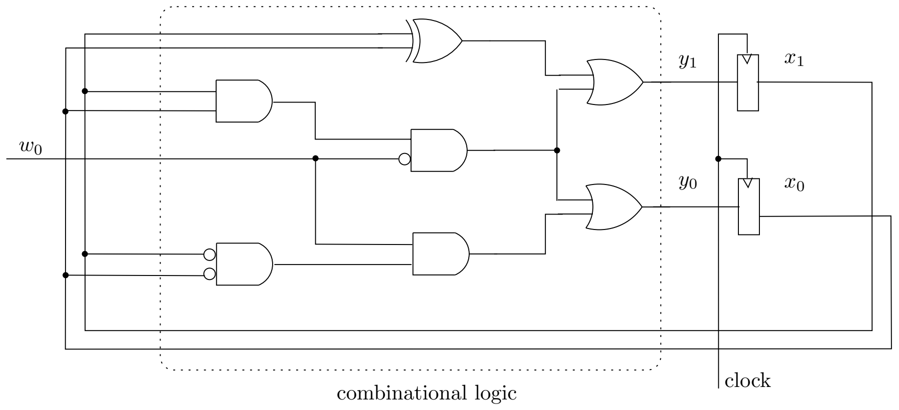


Figure 2.2. A sequential circuit example.

two registers as

$$\begin{aligned} y_1 &: \quad \neg x_1 \wedge x_0 \vee x_1 \wedge \neg x_0 \vee x_1 \wedge x_0 \wedge \neg w_0 \\ y_0 &: \quad \neg x_1 \wedge \neg x_0 \wedge w_0 \vee x_1 \wedge x_0 \wedge \neg w_0 \end{aligned}$$

Given the values of present-state variables and the input signal, the values of next-state variables are determined by their transition functions. When the current values of the two registers are $(x_1 = 0, x_0 = 0)$, for instance, their values at the next clock cycle will be $(y_1 = 0, y_0 = 0)$ for $w_0 = 0$, and $(y_1 = 0, y_0 = 1)$ for $w_0 = 1$. If we use the state encoding scheme and labeling functions described on the left-hand side of Figure 2.1, we will get the right-hand side Kripke structure in the same figure.

Since the number of memory elements in a sequential circuit is finite, there are only a finite number of states. However, There is a well-known *state explosion* problem. The total number of states in the FSM can be as large as 2^n for a system with n binary state variables. Due to its exponential dependence on the number of state variables, the number of states of the model can be extremely large even for a moderate-size system.

Some digital systems may have an infinite number of states. Software with recursive function calls and unbounded data structures, for instance, fall into this category. Other examples include timed systems and hybrid systems [ACH$^+$95, AH96], in which the state variables can be of unbounded integer or even real type. Since model checking requires the Kripke structure to be finite-state, before we can apply model checking, a certain degree of abstraction is needed to extract suitable verification models from these systems. In general, abstraction used for this purpose is either under-approximation or over-approximation. The process of mapping an infinite state space into a finite state space, by itself, is an important research topic, and is beyond the scope of this book. In the sequel, we assume that the finite state model of a given system, or the Kripke structure, is already available.

2.2 Temporal Logic Property

Propositional logic is the basis for specifying properties. A *proposition* is a declarative sentence about the Kripke structure that is either true or false. Propositions are represented by a set of propositional variables $p, q, \ldots$ plus the truth values true and false. A formula consisting of a propositional variable is called an *atomic proposition.* The evaluation of an atomic proposition maps to a set of states in the Kripke structure.

Propositional logic formulae are defined in terms of atomic propositions with the common logical connectives.

Definition 2.2 *A propositional logic formula is defined as follows:*

- *atomic propositions are propositional formulae;*
- *if ϕ is a propositional formula, then $\neg\phi$ is a propositional formula;*
- *if ϕ and ψ are propositional formulae, then $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$ are propositional formulae.*

In the set of logical connectives, the unary operator negation ($\neg$) and the binary operation logical AND ($\wedge$) constitute a minimal subset that is sufficient for defining propositional logic. Besides $\wedge$, there are 15 other binary logical connectives; however, all of them can be expressed in terms of $\neg$ and $\wedge$. For example, under the De Morgan's law the formula $\phi \vee \psi$ can be rewritten into $\neg(\neg\phi \wedge \neg\psi)$. The "implies" operator $\rightarrow$ means "only if", and therefore $\phi \rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$. Similarly, the formula $\phi \leftrightarrow \psi$ is equivalent to $\neg\phi \wedge \neg\psi \vee \phi \wedge \psi$.

Propositional logic is incapable of reasoning about the evolution of valuations over time. When the truth of a property depends on not only the present valuation, but also on the valuations in the past or in the future, we need temporal logics. The most common temporal logics to express system properties are Computational Tree Logic (CTL) and Linear Time Temporal Logic (LTL). CTL and LTL are subsets of the more general CTL*. In this book, we will focus on Linear Time Temporal Logic, but we will also briefly describe the Computational Tree Logic, since some of its operators will be used in our discussion of model checking algorithms.

There are two very different ways of modeling time in temporal logics. The linear time model assumes that each time instance has exactly one successor; the branching time model, on the other hand, allows several successors for each time instance. LTL is based on the linear time model. LTL formulae specify properties about the future of each individual execution trace such as the condition `ack` will eventually be true, or that the condition `busy` will be true until another condition `done` becomes true. Logics based on the branching time model, such as CTL, deal with all possible execution traces. CTL formulae can specify properties such as that if the condition `reset` is true then on all paths the condition `reset_done` will eventually be true.

LTL formulae are defined in terms of atomic propositions, the usual logic connectives, as well as linear time temporal operators. The two basic temporal operators in LTL are X and U, called *next* and *until*, respectively. The first operator is unary and the second is binary. The formula $\mathsf{X}\,\phi$ means that ϕ holds at the next point of time. The formula $\phi\,\mathsf{U}\,\psi$ means that ϕ has to hold until ψ becomes true, and ψ will eventually become true.

Definition 2.3 *A Linear Time Temporal Logic (LTL) formula is defined recursively as follows:*

- *atomic propositions are LTL formulae;*
- *if ϕ and ψ are LTL formulae, so are $\neg\phi$, $\phi \wedge \psi$, and $\phi \vee \psi$;*
- *if ϕ and ψ are LTL formulae, so are $\mathsf{X}\,\phi$ and $\phi\,\mathsf{U}\,\psi$;*

Besides X and U, there are other temporal operators including G for *globally*, F for *finally*, and R for *release*. The formula $\mathsf{G}\,\phi$ means that ϕ has to hold forever. The formula $\mathsf{F}\,\phi$ means that ϕ will eventually be true. The formula $\phi\,\mathsf{R}\,\psi$ means that ψ remains true before the first time ϕ becomes true (or forever if ϕ remains false). These three temporal operators can be expressed in terms of the two basic ones:

$$\begin{aligned}
\mathsf{F}\,\phi &= \mathsf{true}\,\mathsf{U}\,\phi \\
\mathsf{G}\,\phi &= \neg\,\mathsf{F}\,\neg\phi \\
\phi\,\mathsf{R}\,\psi &= \neg(\neg\psi\,\mathsf{U}\,\neg\phi)
\end{aligned}$$

The semantics of LTL formulae are defined for an infinite path $\pi = (s_0, s_1, ...)$ of the Kripke structure, where $s_i \in S$ is a state, s_0 is an initial state, and $T(s_i, s_{i+1})$ evaluates to true for all $i \geq 0$. The suffix of π starting from the state s_i is represented by π^i. We use $K, \pi^i \models \phi$ to represent the fact that ϕ holds in a suffix of path π of the Kripke structure K. The property ϕ holds for the entire path π if and only if $K, \pi^0 \models \phi$. When the context is clear, we will omit K and rewrite $K, \pi^i \models \phi$ into $\pi^i \models \phi$. The semantics of LTL formulae are defined recursively as follows:

$\pi \models \mathsf{true}$	always holds
$\pi \models \varphi$	iff $\pi^0 \models \varphi$
$\pi \models \neg\varphi$	iff $\pi \not\models \varphi$
$\pi \models \varphi \wedge \psi$	iff $\pi \models \varphi$ and $\pi \models \psi$
$\pi \models \mathsf{X}\ \varphi$	iff $\pi^1 \models \varphi$
$\pi \models \varphi\, \mathsf{U}\, \psi$	iff $\exists i \geq 0$ such that $\pi^i \models \psi$ and for all $0 \leq j < i$, $\pi^j \models \varphi$
$\pi \models \varphi\, \mathsf{R}\, \psi$	iff for all $i \geq 0$, $\pi^i \models \psi$; or $\exists j \geq 0$ such that $\pi^j \models \varphi$ and for all $0 \leq i \leq j$, $\pi^i \models \psi$

The Kripke structure K satisfies an LTL formula ϕ if and only if all paths from the initial states do. This means that all LTL properties are universal properties in the sense that we can add the path quantifier A as a prefix without changing the meaning of the properties. That is, $K \models \phi$ is equivalent to $K \models \mathsf{A}\,\phi$, where the path quantifier A means ϕ holds *for all computation paths*. Another path quantifier is E, which stands for *there exists a computation path*. E is not used in LTL, but both A and E are used in CTL.

An LTL formula is in the normal form if negation appears only in front of propositional formulae. For instance, the formula $\mathsf{F}\neg\mathsf{F}\,p$ is not in the normal form since negation is ahead of the temporal operator F; on the other hand, the equivalent formula $\mathsf{F}\,\mathsf{G}\,\neg p$ is in the normal form. We can always rewrite an LTL formula into normal form by pushing negation inside temporal operators. The following rules can be applied during the rewriting:

$$\begin{aligned} \mathsf{X}\,p &= \neg\,\mathsf{X}\,\neg p \\ \mathsf{G}\,p &= \neg\,\mathsf{F}\,\neg p \\ p\,\mathsf{U}\,q &= \neg(\neg q\,\mathsf{R}\,\neg p) \\ \mathsf{F}\,p &= \mathsf{true}\,\mathsf{U}\,p \end{aligned}$$

Since an LTL formula ϕ is a universal property and is equivalent to $\mathsf{A}\,\phi$, the negation of ϕ should be the existential property $\mathsf{E}\,\neg\phi$.

The two path quantifiers are an integral part of Computational Tree Logic (CTL), and are used explicitly to specify properties related to execution traces in the computation tree structure. A (for all computation paths) specifies that all paths starting from a given state satisfy a property; E (for some computation paths) specifies that some of these paths satisfy a property.

Definition 2.4 *A Computational Tree Logic (CTL) formula is defined recursively as follows:*

- *atomic propositions are CTL formulae;*
- *if φ and ψ are CTL formulae, then $\neg\varphi$, $\varphi \wedge \psi$, and $\varphi \vee \psi$ are CTL formulae;*
- *if φ and ψ are CTL formulae, then $\mathsf{EX}\,\varphi$, $\mathsf{E}\,\psi\,\mathsf{U}\,\varphi$, and $\mathsf{EG}\,\varphi$ are CTL formulae.*

A CTL formula is in the normal form if negation appears only in front of propositional formulae. Formula $\neg\,\mathsf{AX}\,p$ is not in the normal form since negation is ahead of the temporal operator AX; on the other hand, the equivalent formula $\mathsf{EX}\,\neg p$ is in the normal form. We can always rewrite a CTL formula into normal form by pushing negation inside temporal operators. The following rewriting rules can be applied during normalization:

$$\begin{aligned}
\mathsf{AX}\,p &= \neg\,\mathsf{EX}\,\neg p \\
\mathsf{AG}\,p &= \neg\,\mathsf{EF}\,\neg p \\
\mathsf{A}\,p\,\mathsf{U}\,q &= \neg(\mathsf{E}\,\neg q\,\mathsf{U}\,\neg p \wedge \neg q) \wedge \neg\,\mathsf{EG}\,\neg q \\
\mathsf{AF}\,p &= \mathsf{A}\,\mathsf{true}\,\mathsf{U}\,p \\
\mathsf{EF}\,p &= \mathsf{E}\,\mathsf{true}\,\mathsf{U}\,p
\end{aligned}$$

Many interesting properties in practice can be expressed in both LTL and CTL. However, there are also properties that can be expressed in one but not the other. The difference between an LTL formula and a CTL formula can be very subtle. For instance, the LTL formula $\mathsf{F}\,\mathsf{G}\,p$ holds in the Kripke structure in Figure 2.3, but the CTL formula $\mathsf{AF}\,\mathsf{AG}\,p$ fails. (In the Kripke structure, p and q are state labels.)

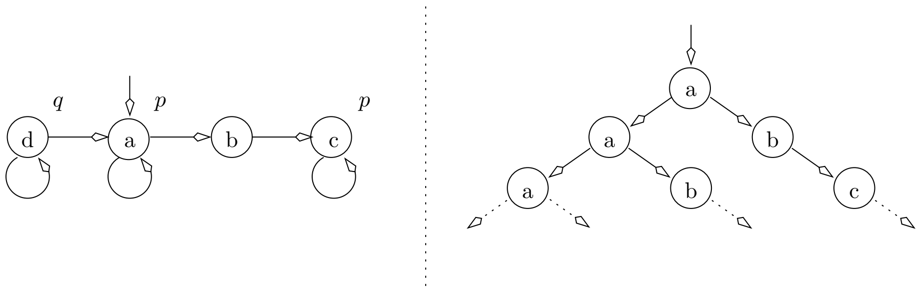


Figure 2.3. A Kripke structure and its computation tree.

The reason is that the LTL property is related to the individual paths, and on any infinite path of the given Kripke structure we can reach the state c from which p will holds forever. The CTL formula $\mathsf{AF}\,\mathsf{AG}\,p$, on the other hand, requires that on all paths from the state a we can reach a state satisfying $\mathsf{AG}\,p$. Note that the only state satisfying $\mathsf{AG}\,p$ is the state c; however, the Kripke structure does not satisfy $\mathsf{AF}\{c\}$—as shown in the right-hand side of the figure, the left most path of the computation tree is a counterexample. On this particular path, we can stay in the state a while reserving the possibility of going to the state b (where p does not hold). Therefore, $\mathsf{F}\,\mathsf{G}\,p$ and $\mathsf{AF}\,\mathsf{AG}\,p$ represent two very similar but different properties.

The above example shows that LTL and CTL have different expressing powers. Some LTL properties, like $\mathsf{F}\,\mathsf{G}\,p$, cannot be expressed in CTL. There are also CTL properties that cannot be expressed in LTL; an example in this category would be $\mathsf{AG}\,\mathsf{EF}\,p$. Both LTL and CTL are strict subsets of the more general CTL* logic [EH83, EL87]. The relationship among LTL, CTL, and CTL* is given in Figure 2.4. In this book, we focus primarily on LTL model checking. Readers who are interested in CTL model checking are referred to [CES86, McM94] or the book [CGP99].

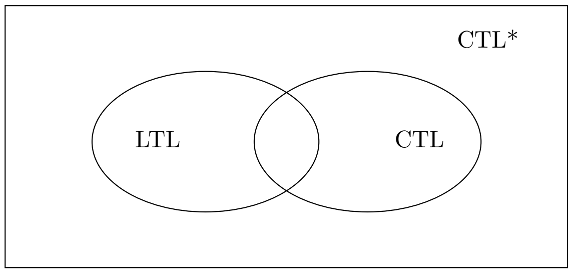


Figure 2.4. The relationship among LTL, CTL, and CTL*.

We have used the term *universal property* during previous discussions. Now we give a formal definition of universal and existential properties.

Definition 2.5 *A property ϕ is a universal property if removing edges from the state transition graph of the Kripke structure does not reduce the set of states satisfying ϕ. A property ψ is an existential property if adding edges into the state transition graph of the Kripke structure does not reduce the set of states satisfying ψ.*

It follows that all LTL properties and ACTL (the universal fragment of CTL) properties are universal. The existential fragment of CTL, or ECTL, is existential. For the propositional μ-calculus formulae, those that do not use EX and EY in their normal forms are universal.

Temporal logic properties can also be classified into the following two categories: *safety* properties and *liveness* properties. The notion of safety and liveness was introduced first by Lamport [Lam77]. Alpern and Schneider [AS85] later gave a formal definition of both safety and liveness properties. Informally, a safety property states that something bad will not happen during a system execution. Liveness properties are dual to safety properties, expressing that eventually something good must happen. The distinction of safety and liveness properties was originally motivated by the different techniques for proving them.

We can think of a property as a set of execution sequences, each of which is an infinite sequence of states of the Kripke structure. A property is called a safety property if and only if each execution violating the property has a finite prefix violating that property. In other words, a finite prefix of an execution violating the property (bad thing) is irremediable no matter how the prefix is extended to a infinite path. Safety properties can be falsified in a finite initial part of the execution, although proving them requires the traversal of the entire set of reachable states. The invariant property $\mathsf{G}\, p$ or $\mathsf{AG}\, p$, which states that the propositional formula p always holds, is a safety property. Other safety properties include mutual exclusion, deadlock freedom, etc.

A property is a liveness property if and only if it contains at least one good continuation for every finite prefix. This corresponds to the intuition that it is still possible for the property to hold (good thing to happen) after any finite execution. Liveness properties do not have a finite counterexample, and therefore in principle cannot be falsified after a finite number of execution steps. An example of liveness property is $\mathsf{G}(p \rightarrow \mathsf{F}\, q)$, which states that whenever the propositional formula p is true, the propositional formula q must become true at some future cycle although there is no upper limit on the time by which q is required to become true. Other liveness properties include accessibility, absence of starvation, etc.

2.3 Generalized Büchi Automaton

An LTL formula ϕ always corresponds to a Büchi automaton $\mathcal{A}_\phi$ that recognizes all its satisfying infinite paths. In other words, the Büchi automation $\mathcal{A}_\phi$ contains all the logic models of the formula ϕ. If we consider the Kripke structure as a language generator and the Büchi automaton $\mathcal{A}_\phi$ as a language recognizer, then we have $K \models \phi$ if and only if all infinite words generated by K are accepted by $\mathcal{A}_\phi$. Therefore, the LTL model checking problem can be translated to ω-regular language containment checking. Since checking language containment between two Büchi automata in general is PSPACE-complete [Kur94], it follows that LTL model checking is PSPACE-complete.

In practice, however, LTL model checking is often translated into language emptiness checking in a generalized Büchi automaton. This *automata-theoretic* approach [VW86] consists of the following three steps:

1 we negate the given property ϕ and translate it into a Büchi automaton $\mathcal{A}_{\neg\phi}$, which accepts all the infinite paths that do not satisfy ϕ;

2 we then compose the model K and the property automaton $\mathcal{A}_{\neg\phi}$ together. The system produced by parallel composition, denoted by $(K||\mathcal{A}_{\neg\phi})$, consists of only those infinite paths of K that are accepted by $\mathcal{A}_{\neg\phi}$;

3 finally, we check whether the language of the composed system is empty.

If the language is empty, then $K \models \phi$ since no infinite path in K is accepted by $\mathcal{A}_{\neg\phi}$. If the language is not empty, any accepting run in the composed system serves as a counterexample to $K \models \phi$.

LTL model checking via language emptiness has the same worst-case complexity bound as the language containment based approach, which is linear in the number of states of the model, but exponential in the length of the LTL formula. The exponential blow-up comes from the translation from LTL formulae to Büchi automata. However, this is often acceptable in practice, because user specified LTL formulae are usually small compared to the size of the model.

In the automata-theoretic approach, we can use the labeled generalized Büchi automata as a unified representation for the model K, the property automaton $\mathcal{A}_{\neg\phi}$, as well as the composed system$(K||\mathcal{A}_{\neg\phi})$. A labeled generalized Büchi automaton is simply a Kripke structure augmented by a set of acceptance conditions. In other words, we can view the model K as a special case of the labeled generalized Büchi automaton whose only acceptance condition is satisfied by all paths.

DEFINITION 2.6 *A labeled generalized Büchi automaton is a six-tuple*

$$\mathcal{A} = \langle S, S_0, T, A, \Lambda, \mathcal{F} \rangle \ ,$$

where S is the finite set of states, $S_0 \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the transition relation, A is a finite alphabet for which a set P of atomic propositions is given and $A = 2^P$, $\Lambda : S \to A$ is the labeling function, and $\mathcal{F} \subseteq 2^S$ is the set of acceptance conditions.

A *run* of $\mathcal{A}$ is an infinite sequence $\rho = s_0, s_1, \ldots$ over S, such that $s_0 \in S_0$ and for all $i \geq 0$, $(s_i, s_{i+1}) \in T$. A run ρ is *accepting* or *fair* if for each fair set $F_i \in \mathcal{F}$, there exists $s_j \in F_i$ that appears infinitely often in ρ. The automaton accepts an infinite word $\sigma = \sigma_0, \sigma_1, \ldots$ in A^ω if there exists an accepting run ρ such that for all $i \geq 0$, $\sigma_i \in \Lambda(\rho_i)$. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, is the subset of A^ω accepted by $\mathcal{A}$. Note that the language of $\mathcal{A}$ is nonempty if and only if $\mathcal{A}$ contains a reachable fair cycle—a cycle that is reachable from an initial state and intersects with all the fair sets.

We have defined the automata with labels on the states, not on the edges. The automata are called generalized Büchi automata because multiple acceptance conditions are possible. A state s is *complete* if for every $a \in A$, there is a successor s' of s such that $a \in \Lambda(s')$. A set of states, or an automaton, is complete if all of its states are. In a complete automaton, any finite state path can be extended into an infinite run. In the sequel all automata are assumed to be complete.

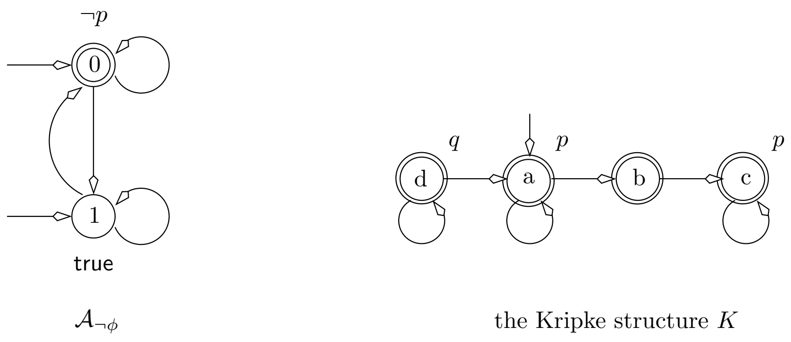
We define the concrete system $\mathcal{A}$ as the synchronous (or parallel) composition of a set of submodules. Composing a subset of these submodules gives us an over-approximated abstract model $\mathcal{A}'$. In symbolic algorithms, $\mathcal{A}$ and $\mathcal{A}'$, as well as the submodules, are all defined over the same state space and agree on the state labels. Communication among submodules then proceeds through the common state space, and composition is characterized by the intersection of the transition relations.

DEFINITION 2.7 *The composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ of two Büchi automata $\mathcal{A}_1$ and $\mathcal{A}_2$, where*

$$\mathcal{A}_1 = \langle S, S_{01}, T_1, A, \Lambda, \mathcal{F}_1 \rangle \ ,$$
$$\mathcal{A}_2 = \langle S, S_{02}, T_2, A, \Lambda, \mathcal{F}_2 \rangle \ ,$$

is a Büchi automaton $\mathcal{A} = \langle S, S_0, T, A, \Lambda, \mathcal{F} \rangle$ such that, $S_0 = S_{01} \cap S_{02}$, $T = T_1 \cap T_2$, and $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$.

In Figure 2.5, we give an example to show how the automata-theoretic approach for LTL model checking works. The model or Kripke structure in this example corresponds to the circuit in Figure 2.2. We are interested in checking the LTL property $\phi = \mathsf{F}\,\mathsf{G}\,p$; that is, eventually we will reach a point from which the propositional formula p holds for ever.



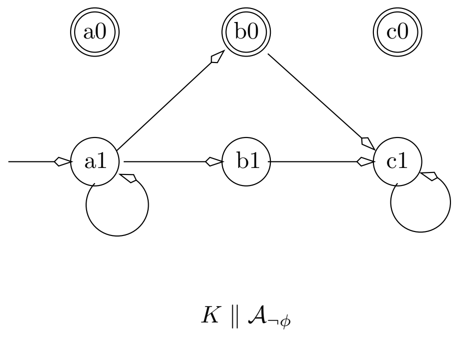


Figure 2.5. An LTL model checking example.

First, we create the property automaton $\mathcal{A}_{\neg\phi}$ that admits all runs satisfying the negation of ϕ, which is $\neg\phi$ or $\mathsf{G}\,\mathsf{F}\,\neg p$. Runs satisfying $\mathsf{G}\,\mathsf{F}\,\neg p$ must visit states labeled $\neg p$ infinitely often. There are existing algorithms to translate a general LTL formula to a Büchi automaton; readers who are interested in this subject are referred to [VW86, GPVW95, SB00]. Since our example is simple enough, we do not need to go through the detailed translation algorithm to convince ourselves that the automaton in Figure 2.5 indeed corresponds to $\mathsf{G}\,\mathsf{F}\,\neg p$. In Figure 2.5, states satisfying the acceptance condition are represented as double circles. The property automaton has only one acceptance condition $\{0\}$, meaning that any accepting run has to go through the state 0 infinitely often. We assume that all states in the Kripke structure K are accepting; that is, the acceptance condition of K is $\{a, b, c, d\}$.

After composing the property automaton with the model, we have the composed system at the bottom of Figure 2.5, whose acceptance condition is $\{a0, b0, c0\}$. Note that in the creation of $K||\mathcal{A}_{\neg\phi}$ we have used the parallel composition; that is, only transitions that are allowed by both parents are retained. The final acceptance condition is also the union of those of both parents. Since the one for K consists of the entire state space, it is omitted. Finally, we check language emptiness in the composed system by searching for a run that goes through some states in the fair set $\{a0, b0, c0\}$ infinitely often. It is clear that the language of that system is empty, because no run visits any of these states infinitely often. Therefore, the property ϕ holds in K.

Whether the language of a Büchi automaton is empty can be decided by evaluating the temporal logic property $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$ on the automaton. In other words, the language of the composed system is empty if and only if no initial state of the composed system satisfies $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$. The property is an existential CTL formula augmented with a set of Büchi fairness constraints; for our running example, $\mathsf{fair} = \{\{a0, b0, c0\}\}$. In a run satisfying this property, a state in every $F_i \in \mathcal{F}$ must be visited infinitely often. The CTL formula under fairness constraints can be decided by a set of nested fixpoint computations:

$$\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true} = \nu Z.\, \mathsf{EX} \bigwedge_{F_i \in \mathcal{F}} \mathsf{E}\, Z\, \mathsf{U}(Z \wedge F_i) \ ,$$

where ν denotes the outer greatest fixpoint computation, and EU represents the embedded least fixpoint computations. When a monotonically increasing transform function f is applied repeatedly to a set Z, we define $f^{(i)}(Z) = f(f(...f(Z)))$ and declare Z as a least fixpoint if $f^{(i)}(Z) = f^{(i+1)}(Z)$. Conversely, when a monotonically decreasing transform function g is applied repeatedly to a set Z, we define $g^{(i)}(Z) = g(g(...g(Z)))$ and declare Z as a greatest fixpoint if $g^{(i)}(Z) = g^{(i+1)}(Z)$. When we evaluate the above formula through fixpoint computation, the initial value of the auxiliary iteration variable Z can be set to the entire universe.

For our running example,

$$
\begin{aligned}
F_0 &= \{a0, b0, c0\} \\[1em]
Z^0 &= \{a0, b0, c0, a1, b1, c1, a2, b2, c2, a3, b3, c3\} \\[1em]
Z^1 &= \mathsf{EX}\,\mathsf{E}\,Z^0\,\mathsf{U}(Z^0 \wedge F_0) \\
&= \mathsf{EX}\,\mathsf{E}\{a0, b0, c0, a1, b1, c1, a2, b2, c2, a3, b3, c3\}\,\mathsf{U}\{a0, b0, c0\} \\
&= \mathsf{EX}\{a1, b0\} \\
&= \{a1\} \\[1em]
Z^2 &= \mathsf{EX}\,\mathsf{E}\,Z^1\,\mathsf{U}(Z^1 \wedge F_0) \\
&= \mathsf{EX}\,\mathsf{E}\{a1\}\,\mathsf{U}\{\ \} \\
&= \mathsf{EX}\{\ \} \\
&= \{\ \}
\end{aligned}
$$

Since no state in the composed system satisfies $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$, the language is empty. This method for deciding $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$ is known as the Emerson and Lei algorithm [EL86], which is the representative of a class of SCC hull algorithms [SRB02]. In general, the evaluation of EX and EU operators does not have to take the above alternating order; they can be computed in arbitrary orders without affecting the convergence of the fixpoint [FFK$^+$01, SRB02]. All SCC hull algorithms share the same worst-case complexity bound — they require $O(\eta^2)$ symbolic steps, where η is the number of states of the composed model. A symbolic step is either a pre-image computation (finding predecessors through the evaluation of EX) or an image computation (finding successors through the evaluation of EY, the dual of EX).

Another way of checking language emptiness is to find all the strongly connected components (SCCs) and then check whether any of them satisfies all the acceptance conditions. If there exists a reachable non-trivial SCC that intersects every $F_i \in \mathcal{F}$, the language of the Büchi automaton is not empty. An SCC consisting of just one state without a self-loop is called *trivial*. In our running example, the reachable non-trivial SCCs of the composed system are $\{a1\}$ and $\{c1\}$. Since none of the non-trivial SCCs intersects the fair set $\{a0, b0, c0\}$, the language of the system is empty.

An SCC is a maximal set of states such that there is a path between any two states. A reachable non-trivial SCC that intersects all acceptance conditions is called a fair SCC. An SCC that contains some initial states is called an initial SCC. An *SCC-closed set* of $\mathcal{A}$ is the union of a collection of SCCs. The complete set of SCCs of $\mathcal{A}$, denoted by $\Pi(\mathcal{A})$, forms a partition of the states of $\mathcal{A}$. Likewise, the set of disjoint

SCC-closed sets can also form a partition of the state space S. A SCC partition Π_1 of S is a refinement of another partition Π_2 of S if for every SCC or SCC closed set $C_1 \in \Pi_1$, there exists $C_2 \in \Pi_2$ such that $C_1 \subseteq C_2$.

An *SCC (quotient) graph* is constructed from a graph by contracting each SCC into a node, merging parallel edges, and removing self-loops. The SCC graph of $\mathcal{A}$, denoted by $\mathcal{Q}(\mathcal{A})$, is a directed acyclic graph (DAG); it induces a partial order: the minimal (maximal) SCC has no incoming (outgoing) edge. Reachable fair SCCs, by definition, contain accepting runs that make the language non-empty. Therefore, a straightforward way of checking language emptiness is to compute all the reachable SCCs, and then check whether any of them is a fair SCC.

Observation 2.8 *The language of a Büchi automaton is empty if and only if it does not have any reachable fair SCC.*

Tarjan's explicit SCC algorithm using depth-first search [Tar72] can be used to decide language emptiness. The algorithm can be classified as an explicit-state algorithm because it traverses one state at a time. Tarjan's algorithm has the best asymptotic complexity bound—linear in the number of states of the graph. However, for model checking industrial-scale systems, even the performance of such a linear time algorithm is not be good enough due the extremely large state space. A remedy to the search state explosion is a technique called "on-the-fly" model checking [GPVW95, Hol97], which avoids the construction of the entire state transition graph by visiting part of the state space at a time and constructing part of the graph as needed. Its fair cycle detection is based on two nested depth-first search procedures. Early termination, efficient hashing techniques, and partial order reduction can be used to reduce memory usage during the search and the number of interleavings that need to be inspected. The scalability issue in explict-state enumeration makes them unsuitable for hardware designs, although they have been successful in verifying controllers and software.

Symbolic state space traversal techniques are another effective way of dealing with the extremely large state transition graphs. Instead of manipulating each individual state separately, symbolic algorithms manipulate sets of states. This is accomplished by representing the transition relation of the graph and sets of states as Boolean formulae, and conducting the search by directly manipulating the symbolic representations. By visiting a set of states at a time (as opposed to a single state), symbolic algorithms can traverse a very large state space using a reasonably small amount of time and memory. Thousands or even millions of states, for instance, can be visited in one symbolic step.

An example of symbolic graph algorithms in the context of LTL model checking is the SCC hull algorithms introduced earlier (the Emerson and Lei algorithm [EL86] is a representative), where each image or pre-image computation is implemented as a symbolic step. There is also another class of SCC computation algorithms based on symbolic traversal, called the SCC enumeration algorithms [XB99, BRS99, GPP03]. Both SCC hull algorithms and SCC enumeration algorithms can be used for fair cycle detection and therefore can decide $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$; however, some of the enumeration algorithms have better complexity bounds than SCC hull algorithms. For example, the Lockstep algorithm by Bloem *et al.* [BRS99] runs in $O(\eta \; log \; \eta)$ symbolic steps, and the algorithm by Gentilini *et al.* [GPP03] runs in $O(\eta)$ symbolic steps.

2.4 BDD-based Model Checking

In symbolic model checking, we manipulate sets of states instead of each individual state. Both the transition relation of the graph and the sets of states are represented by Boolean functions called *characteristic functions*, which are in turn represented by BDDs. Let the model be given in terms of

- a set of present-state variables $x = \{x_1, ..., x_m\}$,
- a set of next-state variables $y = \{y_1, ..., y_m\}$;
- a set of input variables $w = \{w_1, ..., w_n\}$, and

the state transition graph can be represented symbolically by $\langle T, I \rangle$, where $T(x, w, y)$ is the characteristic function of the transition relation, and $I(x)$ is the characteristic function of the initial states. A *state* is a valuation of either the present-state or the next-state variables. For m state variables in the binary domain $\mathbb{B} = \{0, 1\}$, the total number of valuations is $|\mathbb{B}|^m$.

If a valuation of the present-state variables, denoted by $\tilde{x}$, makes the initial predicate $I(\tilde{x})$ evaluate to true, the corresponding state is an initial state. Let $\tilde{x}$, $\tilde{y}$, and $\tilde{w}$ be the valuations of x, y, and w, respectively; the transition relation $T(\tilde{x}, \tilde{w}, \tilde{y})$ is true if and only if under the input condition $\tilde{w}$, there is a transition from the state $\tilde{x}$ to the state $\tilde{y}$.

In our running example in Figure 2.2, the present-state variables, next-state variables, and inputs are $\{x_1, x_0\}$, $\{y_1, y_0\}$, and w_0, respectively. The next-state functions of the two latches, in terms of the present-state variables and inputs, are:

$$\begin{aligned}\Delta_1 &= (x_1 \oplus x_0) \vee (x_1 \wedge x_0 \wedge \neg w_0) \ , \\ \Delta_0 &= (\neg x_1 \wedge \neg x_0 \wedge \wedge w_0) \vee (x_1 \wedge x_0 \wedge \neg w_0) \ .\end{aligned}$$

Note that $\oplus$ denotes the exclusive OR operator. Boolean functions T and I are given as follows:

$$\begin{aligned}T &= (y_1 \leftrightarrow \Delta_1) \wedge (y_0 \leftrightarrow \Delta_0) \ , \\ I &= \neg x_1 \wedge \neg x_0 \ .\end{aligned}$$

When $(x_1 = 0, x_0 = 0)$, $(y_1 = 0, y_0 = 1)$, and $w_0 = 1$, for instance, the transition relation T evaluates to true, meaning a valid transition exists from the state $(0, 0)$ to the state $(0, 1)$ under this input particular condition.

Computing the *image* or *pre-image* is the most fundamental step in symbolic model checking. The image of a set of states consists of all the successors of these states in the state transition graph; the pre-image of a set of states consists of all their predecessors. In model checking, two existential CTL formulae, $\mathsf{EX}(D)$ and $\mathsf{EY}(D)$, are used to represent the image and pre-image of the set of states D under the transition relation T. With a little abuse of notation, we are using EX and EY as both temporal operators as sell as set operators. These two basic operations are defined as follows:

$$\begin{aligned}\mathsf{EX}_T(D) &= \{s \mid \exists s' \in D : (s, s') \in T\} \ , \\ \mathsf{EY}_T(D) &= \{s' \mid \exists s \in D : (s, s') \in T\} \ .\end{aligned}$$

When the context is clear, we will drop the subscripts and use EX and EY instead. Given the symbolic representation of the transition relation T and a set of states D, the image and pre-image of D are computed as follows:

$$\begin{aligned}\mathsf{EX}_T(D) &= \exists y, w \,.\, T(x, w, y) \wedge D(y) \ , \\ \mathsf{EY}_T(D) &= \exists x, w \,.\, T(x, w, y) \wedge D(x) \ .\end{aligned}$$

When we use them inside a fixpoint computation, we usually represent sets of states as Boolean functions in terms of the present-state variables only. Therefore, before pre-image computation and after image computation, we also need to simultaneously substitute the set of present-state variables with the corresponding next-state variables.

Many interesting temporal logic properties can be evaluated by applying EX and EY repeatedly, until a *fixpoint* is reached. The set of states that are reachable from I, for instance, can be computed by the least fixpoint computation as follows:

$$\mathsf{EP}\, I = \mu Z. I \cup \mathsf{EY}(Z) \ .$$

Here $\mathsf{EP}\, I$ denotes the set of reachable states and μ represents the least fixpoint computation. In this computation, we have $Z^0 = \emptyset$ and $Z^{i+1} = I \cup \mathsf{EY}(Z^i)$ for all $i \geq 0$. That is, we repeatedly compute the image of the set of already-reached states starting from the initial states I, until the result stops growing. Similarly, the set of states from which D is reachable, denoted by $\mathsf{EF}\, D$, can be computed by the least fixpoint computation

$$\mathsf{EF}\, D = \mu Z. D \cup \mathsf{EX}(Z) \ .$$

This fixpoint computation is often called the *backward reachability*.

The computation of $\mathsf{EG}\, D$, on the other hand, corresponds to a greatest fixpoint computation. ($\mathsf{EG}\, p$ means that there is a path on which p always holds—in a finite state transition graph, such a path corresponds to a cycle.) It is defined as follows:

$$\mathsf{EG}\, D = \nu Z. D \cap \mathsf{EX}(Z) \ ,$$

where ν represents the greatest fixpoint computation. In this computation, we have Z^0 set to the entire universe and $Z^{i+1} = D \cap \mathsf{EX}(Z^i)$ for all $i \geq 0$.

The computation of $\mathsf{EG_{fair}\,true}$ also corresponds to a set of fixpoint computations. As pointed out in the previous section, this is a CTL property augmented with a set of Bc̈hi fairness constraints, and it can be used to decide whether the language of a Büchi automaton is empty. The formula can be evaluated through fixpoint computations as follows:

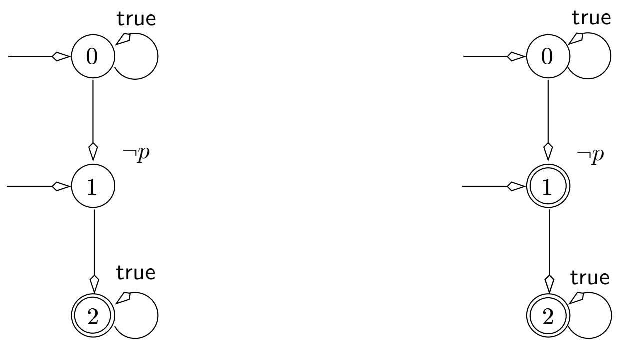
$$\mathsf{EG_{fair}\,true} = \nu Z.\,\mathsf{EX} \bigwedge_{F_i \in \mathcal{F}} \mathsf{E}\, Z\, \mathsf{U}(Z \wedge F_i) \enspace .$$

The evaluation corresponds to two nested fixpoint computations, a least fixpoint (EU) embedded in a greatest fixpoint ($\nu Z.\,\mathsf{EX}$). In the previous section, we have given a small example to illustrate the evaluation of this formula.

As mentioned before, we can use symbolic algorithms to enumerate the SCCs in a graph [XB99, BRS99, GPP03]. Conceptually, an SCC enumeration algorithm works as follows (here we take the algorithm in [XB99] as an example, for it is the simplest among the three and it serves as a stepping stone for understanding the other two). First, we pick an arbitrary state v as *seed* and compute both $\mathsf{EF}\,v$ and $\mathsf{EP}\,v$. $\mathsf{EF}\,v$ consists of states that can reach v and $\mathsf{EP}\,v$ consists of states reachable from v. We then intersect the two sets of states to get an SCC (and the intersection is guaranteed to be an SCC). If the SCC does not intersects with all the fair sets $F_i \in \mathcal{F}$, we remove it from the graph and pick another seed from the remaining graph. We keep doing that until we found an SCC satisfying all the acceptance conditions, or no state is left in the graph. Although SCC enumeration algorithms may have better complexity bounds than SCC hull algorithms, for industrial-scale systems, applying any of these algorithms directly to the concrete model remains prohibitively expensive.

For certain subclasses of LTL properties, there exist specialized algorithm that are more efficient than the evaluation of $\mathsf{EG_{fair}\,true}$. One way of finding these subclasses is to classify LTL properties by the strength of the corresponding Büchi automata. According to [BRS99], the strength of a property Büchi automaton can be classified as *strong*, *weak*, and *terminal*. If the property automaton is classified as strong, checking the language emptiness of the composed system requires the evaluation of the general formula $\mathsf{EG_{fair}\,true}$. Whenever the property automaton is weak or terminal, language emptiness checking in the composed system only requires the evaluation of $\mathsf{EF\,fair}$ or $\mathsf{EF\,EG\,fair}$, respectively. Note that the latter two formulae are much easier to evaluate since they corresponds to a single fixpoint computation or two fixpoint computations aligned in a row, rather than two fixpoint computations but with one nested inside the other as in $\mathsf{EG_{fair}\,true}$.

Let us take the invariant property $\mathsf{G}\,p$ as an example, whose corresponding property automaton is denoted by $\mathcal{A}_{\mathsf{F}\,\neg p}$. The property automaton is given at the left-hand side of Figure 2.6. The only acceptance condition of the automaton is $\{2\}$, or $\mathcal{F} = \{\{2\}\}$. The SCC $\{2\}$ is a fair SCC since it satisfies the acceptance condition; furthermore, the SCC is maximal in the sense that no out-going transition exists. For the automaton at the left-hand side, we can mark State 1 accepting as well; that is, $\mathsf{fair} = \{1, 2\}$. The automaton remains equivalent to the original one because both accept the same ω-regular language. However, this new property automaton can be classified as a *terminal* automaton although according to the definition in [BRS99], the original one is classified as weak. For a terminal property automaton, the language of the composed system is empty as long as no state in the fair SCC is reachable. This is equivalent to evaluating the much simpler formula $\mathsf{EF\,fair}$ (which has a similar complexity as the reachability analysis). Note also that when $\mathsf{EF\,fair}$ is used instead of $\mathsf{EG_{fair}\,true}$, we may end up producing a shorter counterexample.



Automaton for $(F\neg p)$ Same automaton, with more accepting states

Figure 2.6. Two terminal generalized Büchi automata.

Binary Decision Diagrams

Set operations encountered in symbolic fixpoint computations, including *intersection*, *union*, and *existential quantification*, are implemented as BDD operations. BDDs are an efficient data structure for representing Boolean functions. BDD in its current form is both *reduced* and *ordered*, called the Reduced Ordered BDD (RO-BDD). ROBDD was first introduced by Bryant [Bry86], although the general ideas of branching programs have been available for a long time in the theoretical computer science literature. Symbolic model checking based on BDDs, introduced by McMillan [McM94], is considered as a major breakthrough in increasing the model checker's capacity, leading to the subsequently widespread acceptance of model checking in the computer hardware industry.

Given a Boolean function, we can build a binary decision tree by obeying a linear order of decision variables; that is, along any path from root to leaf, the variables appear in the same order and no variable appears more than once. We further restrict the form of the decision tree by repeatedly merging any duplicate nodes and removing nodes whose *if* and *else* branches are pointing to the same child node. The resulting data structure is a directed acyclic graph. Conceptually, this is how we construct the ROBDD for a given Boolean function. In practice, BDDs are created directly in the fully reduced form without the need to build the original decision tree in the first place. BDDs representing multiple Boolean functions are also merged into a single directed graph to increase the sharing; such a graph would have multiple roots, one for each Boolean function.

The formal definition of a BDD is given as follows:

Definition 2.9 *A BDD is a directed acyclic graph* $(\Phi \cup V \cup \{\mathbf{1}\}, E)$ *representing a set of functions* $f_i : \{0,1\}^n \rightarrow \{0,1\}$*. The nodes* $\Phi \cup V \cup \{\mathbf{1}\}$ *are partitioned into three subsets.*

- $\mathbf{1}$ *is the only terminal node whose out-degree is 0.*
- V *is the set of internal nodes whose out-degree is 2 and whose in-degree is 1. Every node* $v \in V$ *corresponds to a Boolean variable* $l(v)$ *in the support of functions* $\{f_i\}$*; the* n *variables* $\{l(v)\}$ *in the entire graph are ordered as follows: if* v_j *is a descendant of* v_i*, then* $l(v_j) < l(v_i)$*.*
- Φ *is the set of function nodes whose out-degree is 1 and whose in-degree is 0; the function nodes are in one-to-one correspondence with the* f_i*'s.*

E is the set of edges connecting the nodes. The outgoing edge of function nodes may have the complement *attribute. The two outgoing edges for a node* $v \in V$ *are labeled T and E, respectively. The E edges may have the* complement *attribute. We write* $(l(v), T(v), E(v))$ *to indicate an internal node and its two outgoing edges.*

The set of functions represented by a BDD is defined as follows: (1) The function of the only terminal node, **1**, is `true`. (2) The function of a regular edge is the function of the head node; the function of a *complement* edge is the complement of the function of the head node. (3) The function of a node $v \in V$ is $l(v) \wedge f_T \vee \neg l(v) \wedge f_E$, where f_T and f_E are the functions of its T and E edges. (4) The function of $\phi \in \Phi$ is the function of its outgoing edge.

BDD provides a very compact representation for many Boolean functions found in practice, although in the worst case the size of a BDD may become exponential with respect to the number of support variables. (An example for the worst-case blowup is a multiplier, which is known to have an exponential number of BDD nodes regardless of the BDD variable order [Bry86].) In addition to the compactness, BDDs are also easy to manipulate. Efficient algorithms exist for almost all the common set-theoretic operations. For example, the intersection or union of two BDDs takes time proportional to the product of their respective sizes in the worst case.

BDDs are also a canonical representation in the sense that with a fixed variable ordering, every Boolean function has a unique BDD representation. Therefore, checking whether two Boolean functions are the same is reduced to a pointer comparison. Given a BDD, complementation or the validity check takes constant time as well.

The complexity of symbolic model checking depends on the size of the BDDs involved in the symbolic steps, such as the BDDs that represent the transition relation and sets of states. Because of this, the search for heuristics to avoid the BDD blow-up in the context of image computation and symbolic fixpoint computation has been one of the major research topics in formal verification.

CU Decision Diagram (CUDD) is a public-domain decision diagram package developed in the University of Colorado [Som]. CUDD is being used widely in industry and academia. The package provides a large set of operations to manipulate BDDs, Algebraic Decision Diagrams (ADDs) [BFG$^+$93], and Zero-suppressed Binary Decision Diagrams (ZDDs) [Min93]. The latter two are variants of BDDs. In particular, ADDs are used to represent function from $\mathbb{B}^m$ to an arbitrary set, as opposed to $\mathbb{B}$ in BDDs. ZDDs represent switching functions like BDDs; however, they are much more efficient than BDDs when the functions to be represented are characteristic functions of cube sets, or in general, when the ON-set of the function to be represented is very sparse. They are inferior to BDDs in other cases. The CUDD package also provides functions to convert BDDs into ADDs or ZDDs and vice versa, and a large assortment of variable reordering methods.

2.5 SAT and Bounded Model Checking

SAT based Bounded Model Checking (BMC) is a complementary technique to BDD based symbolic model checking. BMC was first introduced by Biere *et al.* in [BCCZ99]. Given a model and an LTL property, BMC searches in the given model for counterexamples of a finite length. The existence of a finite length counterexample is encoded as a Boolean formula, which is satisfiable if and only if the counterexample exists. The satisfiability problem of a Boolean formula can be decided by a SAT solver. Since modern SAT solvers often suffer less from the potential search space explosion, in practice, SAT based bounded model checking can handle some industrial-scale circuits that are beyond the reach of BDD based techniques.

In bounded model checking, one can keep increasing the counterexample length k (also called the unrolling depth) until either a counterexample is found or k exceeds a predetermined *completeness threshold*. A completeness threshold is a constant k_c such that if we cannot find any counterexample shorter than or equal to k_c, we have proved that the property holds. It is clear that $k_c \leq \eta$, where η is the number of states of the Kripke structure, since any finite run of the Kripke structure with distinct states cannot be longer than that. Therefore, BMC can be regarded as transforming the PSPACE-complete problem of LTL model checking into a finite number of Boolean satisfiability checks. Although each individual SAT problem in this process is NP-complete, the total number of SAT problems, or k_c, can be exponential with respect to the number of state variables. In practice, the SAT checks often slow down significantly after k goes beyond a few hundred steps.

A better completeness threshold than η would be the diameter of the state transition graph, i.e., the length of the longest shortest path between any two states. For safety properties, we can go one step further and use the reachable diameter of the graph, which is the length of the longest shortest path between an initial state and another state. While the diameter of a design may be exponential in the number of its state elements, Baumgartner and Kuehlmann [BK04] have observed that in practice it often ranges from tens to a few hundred regardless of design size. They also proposed a general approach for enabling the use of structural transformations to tighten the bounds obtained by arbitrary diameter approximation techniques. Despite this previous research work, computing a tight bound of the diameter of an extremely large graph remains very hard in practice. In the absence of a reasonably small completeness threshold, people use BMC primarily for detecting bugs rather than for proving properties.

The Boolean formula used to encode the existence of a finite length counterexample consists of two subformulae: $\Phi = \Phi_M \wedge \Phi_P$. The first subformula, denoted by Φ_M, captures all the length k execution traces that are possible in the Kripke structure, all of which starts from the initial states. The second subformula, denoted by Φ_P, captures the constraint for a length k path to violate the given property. The conjunction of the two subformula captures all the length k counterexamples of the property. Such a counterexample exists if and only if the Boolean formula has a satisfiable assignment.

First, we explain how to create the subformula Φ_M. We use V to represent the set of state variables (or latches) and U to represent the rest of the signals (primary inputs, outputs, and signals of internal logic gates). We then replicate these variables at every clock cycle: we use $V^i = \{v_1^i, \dots, v_n^i\}$ to represent the set of state variables at the i-th time frame and $U^i = \{u_1^i, \dots, u_m^i\}$ to represent the set of other signals at the i-the time frame. Now we can unroll the sequential circuit (by making multiple copies of the symbolic transition relation) into a combination circuit. When the BMC unrolling depth is k,

$$\Phi_M = I(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, U^{i-1}, V^i) \ ,$$

where $I(V^0)$ states that all paths must start from an initial state, and the rest is the conjunction of k copies of the transition relation.

The transition relation copy at the i-th time frame is denoted by $T(V^{i-1}, U^{i-1}, V^i)$, which is the conjunction of elementary transition relations,

$$T(V^{i-1}, U^{i-1}, V^i) = \bigwedge_{1 \le j \le n} (v_j^i \leftrightarrow u_j^{i-1}) \wedge \bigwedge_{1 \le j \le m} T_j(U^{i-1}, V^i) \ .$$

Each T_j is called a *gate relation* for it describes the behavior of a logic gate. For instance, if u_j is the output variable of a two-input AND gate with inputs u_l and u_r, then $T_j = u_j \leftrightarrow (u_l \wedge u_r)$. If u_j is a primary input to the circuit, then $T_j = 1$. Each term of the form $(v_j^i \leftrightarrow u_l^{i-1})$ equates a next-state variable to a combinational variable, describing that the output of a logic gate is fed to the data input of the j-th register. Adjacent transition relation copies are effectively connected together through the use of shared state variables in V^i.

Next we explain the creation of the subformula Φ_P, which states that a path of length k must violate the given property. To explain the basic idea, we use the invariant property $\mathsf{G}\, p$ as an example. For the encoding of a general LTL formula in BMC, the readers are referred to the original

BMC paper [BCCZ99]. Note that the property $\mathsf{G}\,p$ fails if and only if there is a path of length k from an initial state to a state labeled $\neg p$. Therefore,

$$\Phi_P = \neg P(V^k) \ ,$$

which indicates that the last state is a "bad" state. We use $P(V^k)$ to denote the predicate that the state V^k satisfies the propositional formula p. In general Φ_P should be the conjunction of $\neg P(V^i)$ for $0 \le i \le k$. However, if we start BMC with $k = 0$ and keep increasing the unrolling depth by 1 at a time, by the time it reaches k we know that the "bad" state cannot be found in the first $(k-1)$ depths; therefore, Φ_P can be simplified into $\neg P(V^k)$. To summarize, the entire BMC instance at the unrolling depth k is as follows,

$$\Phi = I(V^0) \wedge \bigwedge_{1 \le i \le k} T(V^{i-1}, U^{i-1}, V^i) \wedge \neg P(V^k) \ .$$

This formula can be viewed as a pure combinational circuit with some environmental constraints. Figure 2.7 illustrates such a view for the unrolling depth 2.

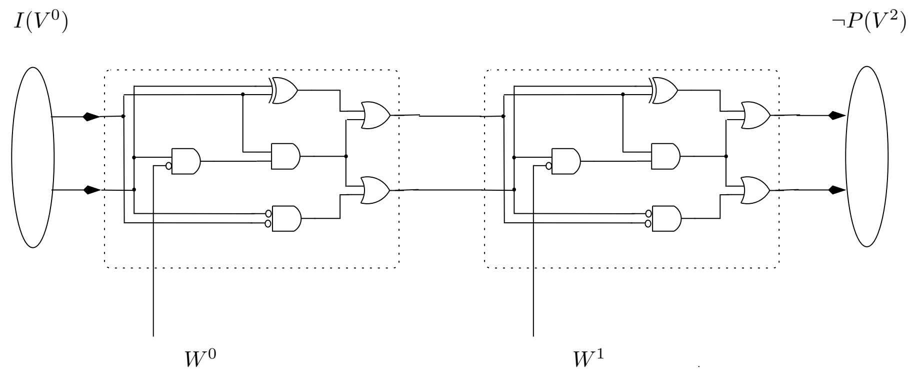


Figure 2.7. A bounded model checking instance.

Now, we explain how to convert the Boolean formula Φ into the Conjunctive Normal Form (CNF). This step is necessary because in practice, we often use an off-the-shelf Boolean SAT solver to decide Φ, and most of the modern SAT solvers accept the CNF input format. A CNF formula is the conjunction of a set of *clauses*, each of which is a disjunction of *literals*. A *literal* is the positive (or negative) phase of a Boolean *variable*. As an example, the following formula fragment

$$(a \vee \neg c) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \wedge \ldots$$

has three variables (a, b, and c), six literals (a, $\neg a$, b, $\neg b$, c, and $\neg c$), and three clauses.

Boolean formulae and combinational circuits can be converted into CNF formulae in linear time if we are allowed to add auxiliary variables. The result CNF formula is also linear with respect to the size of the input formula or the size of the input circuit. Since the transition relation of a Kripke structure can be represented as a network of logic gates, we only need to consider the problem of encoding the individual combinational logic gates as conjunctions of clauses. The solution to the latter problem is actually very straightforward. For instance, a two-input AND gate u_j with inputs u_l and u_r has the following set of clauses:

$$(u_l \vee \neg u_j) \wedge (u_r \vee \neg u_j) \wedge (\neg u_l \vee \neg u_r \vee u_j) \ .$$

Finally, we review an algorithm used in many modern SAT solvers to decide the satisfiability of a CNF formula. A formula is *satisfiable* if and only if there exists a set of assignments to the variables that makes the formula true. It is clear that for the entire CNF formula to be satisfiable, each individual clause must also be satisfiable. The SAT problem of a CNF formula can be solved by the Davis-Longeman-Loveland (DLL) recursive search procedure [DLL62]. Its basic steps are making decisions and propagating the implications of these decisions. Selecting a literal and making it true is called a *decision*. If a clause has only one unassigned literal and all the other literals are false, it is called a unit clause. Every unit clause triggers an implication—its only unassigned literal has to be true, otherwise, the clause is no longer satisfiable. The process of applying implications iteratively until no unit clause is left is called *Boolean Constrain Propagation (BCP)*.

A decision and the corresponding BCP restrict our attention into a subformula or a subset of the original clauses (since the rest of the clauses have been made true). If we keep making decisions on free variables and performing BCP until no subformula remains to be decided, the formula is proved to be satisfiable. However, if the remaining subformula is not satisfiable (i.e., some of its clauses become false), we need to backtrack and flip some of the previous assignments we made earlier.

The pseudo code of the DLL procedure is given in Figure 2.8. It makes decisions and then applies BCP inside the *while* loop. If all the variables have been assigned and no conflict occurs, the procedure MakeDecision will return false and the procedure SatCheck will return a complete set of variable assignments. If a conflict occurs after a partial set of assignments—some clauses become false inside BCP, it indicates that a previous decision is not appropriate. Inside the procedure ConflictAnalysis, the level of that decision is identified by conflict

```
SatCheck( ) {
    while (true) {
        if  ( MakeDecision( ) ) {
            while ( BCP( ) == CONFLICT ) {
                level = ConflictAnalysis( );
                if (level < 0)
                    return UNSAT;
                else
                    BackTrack(level);
        }}
        else
            return SAT;
    }
}
```

Figure 2.8. The DLL Boolean SAT procedure.

analysis [SS96], following which, the inappropriate decision is recovered by backtracking. Before backtracking, a conflict clause learned from this analysis is also added to the clause database (i.e., conjoined with the original formula) to prevent the search from repeating this mistake. When the backtrack level is less than 0, the given formula is proved to be unsatisfiable—there is a conflict before we make any decision.

2.6 Abstraction Refinement Framework

Abstraction refinement was first introduced by Kurshan [Kur94] in checking linear time properties specified by ω-regular automata. It is an important technique to bridge the capacity gap between the model checker and large digital systems. When a model cannot be directly handled by the model checker due to the limited capacity of the model checking algorithms, abstraction can be used to simplify the model by removing the information that is irrelevant to verification. To simplify verification, we want to retain only the relevant details. The key issue in abstraction refinement is identifying in advance which part of the model is relevant and which is not.

There are automatic techniques for computing a simplified model in which an entire class of properties are preserved. For instance, bi-simulation based reduction preserves the full propositional μ-calcus (hence the entire CTL since all CTL formulae can be evaluated through the translation to fixpoint computations in propositional μ-calcus). A nice feature of these techniques is that we only need to compute the reduction for a given model once, and then use the simplified model to check all kinds of properties in that class. In practice, however, bi-simulation and other property preserving abstractions are less attractive because they are either hard to compute, or do not achieve a drastic reduction. A previous study by Fisler and Vardi [FV99] has demonstrated that bi-simulation relation is often expensive to compute and bi-simulation based simplification does not speed up CTL model checking.

A more practical abstraction approach is called property driven abstraction, which often results in a significantly smaller model that preserves or partially preserves a given property (as opposed to a class of properties). This abstraction approach is frequently used in the iterative abstraction refinement loop. There are various ways of deriving such an abstract model [BSV93, Lon93, CHM$^+$96a]. Most of them create abstract models that are upper bounds or over-approximations of the exact system, which may have more behavior than the concrete model. They may produce *false negatives* when being used to verify universal properties such as $\mathsf{AG}\, p$: If a property holds in the abstract model, it also holds in the concrete model; however, if the property fails in the abstract model, it may still pass in the concrete model—in this case, the property is still undecided.

There are also lower bounds or under-approximations of the exact system. These abstractions are conservative as well because they may produce *false positives* when being used to check universal properties. Since the abstract models have less behavior than the exact system, if a counterexample is found in the abstract model, then it is also a counterexample in the exact system. However, if no counterexample exists in the abstract model, we cannot conclude that the property is true. In other words, these abstractions can only refute a property but cannot prove it. Note that one can also use under-approximations and over-approximations simultaneously in a single iterative refinement process, to tighten up abstraction from both ends.

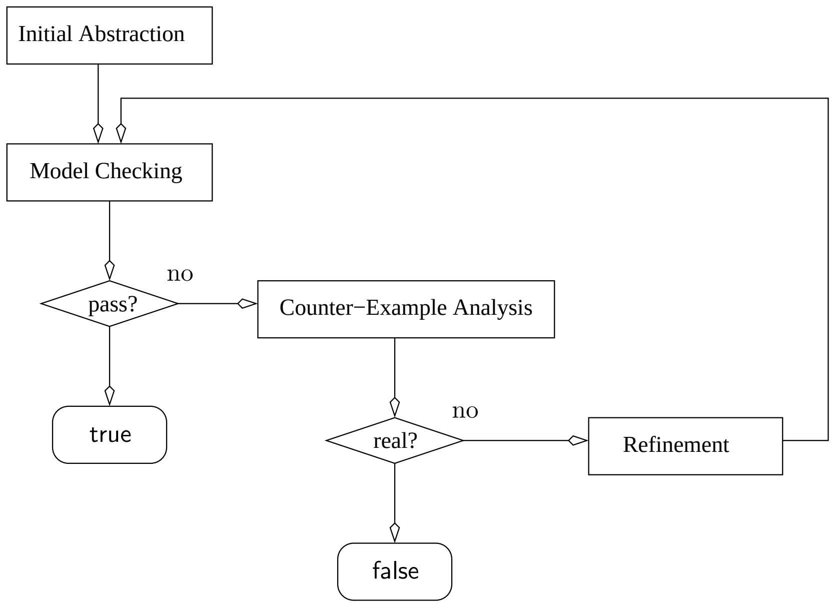


Figure 2.9. The abstraction refinement framework.

Since the property driven abstraction is conservative, we need an iterative process to refine the abstract model until it becomes deciding. In abstraction refinement, one seeks the simplest abstraction that can either prove or refute the given property. Such abstraction is called the *final* or *deciding* abstraction for the given model checking problem. Figure 2.9 shows the generic framework for iterative abstraction refinement. Given a model and an LTL property, we can build a very coarse initial abstraction than is an over-approximation of the concrete model. We then use a model checker to decide whether the abstract model satisfies the property. If the property is satisfied by the abstract model, it is also satisfied by the concrete model. If the property fails in the abstract model, there is no conclusive result yet.

At this point, the model checker returns an abstract counterexample showing how the property is violated. Inside counterexample analysis, we check whether this abstract counterexample contains a valid path in the concrete model. One way of doing that is using the *concretization test* to reconstruct the abstract path in the concrete model. If this is possible and we find a real path within the given abstract counterexample, the property is refuted. Otherwise, the abstract counterexample is declared as *spurious*, which means that some important information of the exact system is missing and therefore the current abstraction needs to be refined.

During refinement, we can use spurious counterexamples to guide the identification of missing information in the current abstract model. Often, the immediate goal in counterexample guided refinement is to remove the (set of) spurious counterexample(s). That is, one searches for a set of refinement variables such that, adding them into the current abstract model removes the spurious counterexample(s).

After computing the set of refinement variables, we can build the new abstract model by including their corresponding bit transition relations. We then start the model checker again. This iterative process terminates when either the property is decided, or the available computing resources (i.e., CPU time and memory) are depleted.

Chapter 3

ABSTRACTION

Abstraction is a key to model checking large real-world systems. The idea is using simplified model to help verify the original model. The definition of simplified model depends on the type of algorithms used in the verification process. In this chapter, abstraction is defined in terms of the simulation relation, and in such a way that the abstract models can be constructed directly from a high level description of the system, even before the concrete model of the system is available.

The abstraction granularity is very important in achieving a higher abstraction efficiency. In previous work, the abstraction granularity is often restricted at the state variable level—binary state variables, together with their transition bit-relations, are treated as *atoms*. This approach is too coarse for most industrial-scale circuits with large combinational logic cones. In this chapter, we propose a finer grain abstraction approach which goes beyond the usual state variable level. In the extreme case, we can treat every combinational logic gate as an atom for abstraction; refinement then becomes a process of synthesizing a final abstract model with the fewest logic gates.

Abstract models used in this chapter are over-approximations of the original model. If the property fails in an abstract model, we will systematically analyze the abstract counterexamples. For invariant properties, we propose a data structure called the synchronous onion rings (SORs) to symbolically capture all the shortest counterexamples and no other counterexamples. The SORs are then used to concretize all the shortest counterexamples simultaneously with a SAT solver, and to compute the refinement set.

3.1 Introduction

The concrete model is considered as a formal description of the complete behavior of the system. Abstraction preserves only part of the behavior that is relevant to the verification of the given property, in the hope that the simplified model is easier to verify. The definition of simplified model depends on the type of algorithms used in the verification process. For explicit state traversal algorithms whose complexity depends on the number of states, the simplification often aims at reducing the size of the state space. The complexity of symbolic state space traversal algorithms depends on the size of the symbolic representation, and therefore the abstract model must be simplified to provide a more compact BDD representation of the transition relation and the sets of states.

The properties under verification must be at least partially preserved during abstraction. Based on how well they preserve the properties, abstraction methods are classified into two categories: the *property-preserving* transformation and the *conservative* transformation. Let $\mathcal{A}$ be the original model, $\{\phi\}$ be a set of temporal logic properties, and $\widehat{\mathcal{A}}$ be the abstract model. Under a property-preserving transformation, for all the properties in $\{\phi\}$, $\widehat{\mathcal{A}} \models \phi$ if and only if $\mathcal{A} \models \phi$. Simplification based on bi-simulation and simulation equivalence [Mil71, DHWT91], for instance, are property-preserving transformations: bi-simulation based reduction preserves the entire propositional μ-calculas, while simulation equivalence based reduction preserves the entire LTL.

However, it is often the case that the more properties one wants to preserve, the less information of the system one can abstract away. Since the major concern in practice is to achieve a drastic simplification of the model, we are more interested in conservative transformations, even though properties may only be partially preserved. Simplification based on simulation relation is a conservative transformation.

Definition 3.1 *A Büchi automaton $\widehat{\mathcal{A}}$ simulates $\mathcal{A}$ (written $\mathcal{A} \preceq \widehat{\mathcal{A}}$) if there exists a simulation relation $R \subseteq S \times \widehat{S}$ such that*

1 *for every initial state $s \in S_0$, there is an initial state $\widehat{s} \in \widehat{S}_0$ such that $(s, \widehat{s}) \in R$;*

2 *$(s, \widehat{s}) \in R$ implies that $\Lambda(s) = \Lambda(\widehat{s})$, and if $(s,t) \in T$ then there exists $\widehat{t} \in \widehat{S}$ such that $(\widehat{s}, \widehat{t}) \in \hat{T}$, and $(t, \widehat{t}) \in R$.*

If $\mathcal{A} \preceq \widehat{\mathcal{A}}$, then $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\widehat{\mathcal{A}})$ [Mil71, DHWT91], where $\mathcal{L}(\mathcal{A})$ denotes the language accepted by $\mathcal{A}$. This is because a corresponding run in $\widehat{\mathcal{A}}$ exists for every run in $\mathcal{A}$—that is, $\widehat{\mathcal{A}}$ has all possible behavior of $\mathcal{A}$, and maybe more. This is the reason why abstract counterexamples may contain transitions that are allowed in $\widehat{\mathcal{A}}$ but not in $\mathcal{A}$, which produces "false negatives" when we are checking properties in $\widehat{\mathcal{A}}$. Conservative abstraction can be improved by successive refinements, as is done in the abstraction refinement framework. In practice, one often starts with a primitive abstract model that only partially preserves the property. Information about the concrete model are gradually added until the the false negative result is completely removed.

The mapping between abstract and concrete models can be described using the more general notion of Galois connection [BBLS92].

DEFINITION 3.2 *A Galois connection from S to $\widehat{S}$ is a pair of function $\alpha : 2^S \rightarrow 2^{\widehat{S}}$ and $\gamma : 2^{\widehat{S}} \rightarrow 2^S$ that are called the abstraction function and the concretization function, respectively. α and γ are both complete and monotonic, and must satisfy the following conditions:*

- $\forall X \in 2^S$, $\gamma \circ \alpha(X) \supseteq X$, *and*
- $\forall X \in 2^{\widehat{S}}$, $\alpha \circ \gamma(X) \supseteq X$.

Under the following condition, the Galois connection can be reduced to the simulation relation: if $\forall X \in 2^{\widehat{S}}$, $\alpha(\mathsf{EX}_T(\gamma(X))) \subseteq \mathsf{EX}_{\widehat{T}}(X)$, then $A \preceq \widehat{A}$. Although the Galois connection provides a more general framework, no easy and practical implementation has been proposed so far to exploit the flexibility provided by the extra generality.

We shall show in the sequel that when simulation relation is used, little overhead is required to construct abstract models from the concrete model. The idea is to directly abstract the transition relation. For sequential circuits, their abstract models can be constructed directly from a high level description of the system, even before the concrete model of the system is available. Abstraction used in this book relies on the simulation relation.

3.2 Fine-Grain Abstraction

Let the concrete model be represented as a generalized Büchi automaton $\mathcal{A} = \langle T, I \rangle$, where $T(x, w, y)$ is the characteristic function of the transition relation and $I(x)$ is the characteristic function of the set of initial states. The model is considered as the synchronous (parallel) composition of a set of submodules. In the simplest form, every binary state variable together with its bit transition relation is considered as a submodule. Let $J = \{1, \ldots, m\}$ be the a permutation of the indexes of the state variables; then,

$$T(x, w, y) = \bigwedge_{j \in J} T_j(x, w, y_j) \ ,$$

where $T_j(x, w, y_j)$ is the bit transition relation of the j-th binary state variable. Note that T_j depends on one next-state variable y_j but on potentially all present-state variables in x. Let $\Delta_j(x, w)$ be the next-state function of the j-th state variable, then $T_j = (y_j \leftrightarrow \Delta_j(x, w))$.

Over-approximations of T and I, denoted by $\widehat{T}$ and $\widehat{I}$, respectively, induce an abstract model that simulates $\mathcal{A}$. A straightforward way of building $\widehat{T}$ from T, which has been adopted by many existing algorithms, is to replace some T_j by tautologies. Note that this approach treats the bit transition relation T_j as an *atom* for abstraction—it is either included in or excluded from the abstract model completely. Since each T_j corresponds to a binary state variable (or latch), the abstraction granularity is at the state variable level. Assume that the abstract model $\widehat{\mathcal{A}}$ contain a subset of state variables $\widehat{J} = \{1, ..., k\} \subseteq J$, a subset $\hat{x} \subseteq x$ of present-state variables, and a subset $\hat{y} \subseteq y$ of next-state variables; then $\widehat{T}$ is defined as follows:

$$\widehat{T}(x, w, \hat{y}) = \bigwedge_{j \in \widehat{J}} T_j(\{\hat{x}, \check{x}\}, w, y_j) \ ,$$

where variables in $\hat{x}$ are called the *visible state variables*, and variables in $\check{x} = x \setminus \hat{x}$ are called the *invisible state variables.* Bit transition relations corresponding to the variables in $\check{x}$ are abstracted away. The set of initial states $\widehat{I}(\hat{x})$ is an existential projection of $I(x)$: an abstract state is initial if and only if it contains a concrete initial state.

There are two different view of the abstract model $\widehat{\mathcal{A}} = \langle \widehat{T}, \widehat{I} \rangle$:

1 The abstract model is defined in exactly the same concrete state space, only with more transitions among the states and possibly more states labeled as initial. While the number of states of the model

remains the same, the simplification is mainly in the size of the BDD representation of the transition relation. This interpretation appears natural when analyzing symbolic graph algorithms.

2 The abstract model is defined in a reduced state space: a set of concrete states in which any two states cannot be distinguished from each other under the simplified transition relation $\widehat{T}$ forms an equivalence class; mapping every equivalence class into a new state forms the abstract model in a reduced state space. The number of states in the reduced state space is also called the *number of effective states.* The worst-case complexity of the graph algorithms is determined by the number of effective states. It makes sense, especially for analyzing explicit graph algorithms, to consider the abstract model in a reduced state space. (In the analysis of symbolic graph algorithms, the number of effective states is also useful.)

Restricting the abstraction granularity at the state variable level, however, is not suitable for verifying industrial-scale systems with extremely large combinational logic cones. The bit transition relation T_j is either $(y \leftrightarrow \Delta_j)$ or the tautology, depending on whether the corresponding state variable is included or not, but it cannot be an arbitrary Boolean function in between. However, there are often cases where not all the logic gates in the combination logic cone are necessary for the verification, even though the state variable itself is necessary. In these cases, an abstraction $\widehat{T_j}$ of the bit transition relation such that $(y \leftrightarrow \Delta_j) \leq \widehat{T_j} \leq \mathbf{1}$ would be ideal; this, however, is not possible when we use the "coarse-grain" abstraction approach. Unnecessarily including the irrelevant information can make the abstract model too complex for the model checker to deal with.

We now give a finer grain approach for abstraction to push the abstraction granularity beyond the state variable level. We consider not only the state variables but also Boolean network variables. *Boolean network variables (BNVs)* are the intermediate variables selectively inserted into the combinational logic cones of latches to partition large logic cones so that a compact BDD representation of their transition relations is possible. Each Boolean network variable is associated with a small portion of the combinational circuit in its fan-in cone; similar to state variables, each BNV and its associated area have an elementary transition relation. The transition relation of the entire system is the conjunction of all these elementary transition relations. The following example shows how fine-grain abstraction works.

We use the example in Figure 3.1 to illustrate the difference between traditional (coarse-grain) abstraction and the fine-grain abstraction ap-

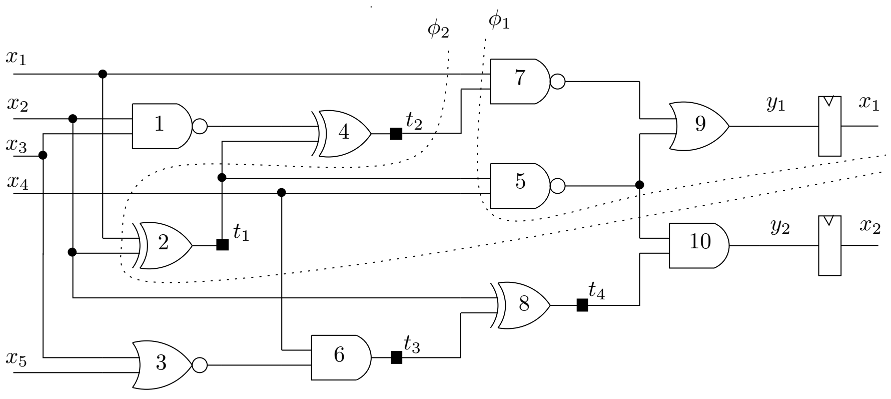


Figure 3.1. Illustration of fine-grain abstraction.

proach. In Figure 3.1, there are 10 gates in the fan-in combinational logic cones of the two latches. Variables y_1 and y_2 are the next-state variables, and $x_1, ..., x_5$ are the present-state variables. Let Δ_{y_1} be the output function of Gate 9 in terms of the present-state variables and inputs; similarly, let Δ_{y_2} be the output function of Gate 10. Δ_{y_1} and Δ_{y_2} are also called the transition functions of Latch 1 and Latch 2, respectively. According to the definition given above, the bit transition relation of Latch 1 is,

$$T_1 = y_1 \leftrightarrow \Delta_{y_1}(x_1, x_2, x_3, x_4) \ .$$

Boolean network variables are a selective set of internal nodes in the fan-in combinational cones of state variables. To illustrate this, we insert 4 BNVs, t_1, t_2, t_3, and t_4, into this circuit. We use δ_{t_i} to represent the output function of the signal t_i, but in terms of both present-state variables and BNVs (as opposed to present-state variables only). Similarly, we define for each BNV t_i the elementary transition relation $T_{t_i} = t_i \leftrightarrow \delta_{t_i}$. For the example in Figure 3.1, these new functions and transition relations are defined as follows:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\delta_{t_1} & = & x_1 \oplus x_2 & T_{t_1} & = & t_1 \leftrightarrow \delta_{t_1} \\
\delta_{t_2} & = & \neg(x_2 \wedge x_3) \oplus t_1 & T_{t_2} & = & t_2 \leftrightarrow \delta_{t_2} \\
\delta_{t_3} & = & \neg(x_3 \vee x_5) \wedge x_4 & T_{t_3} & = & t_3 \leftrightarrow \delta_{t_3} \\
\delta_{t_4} & = & x_2 \oplus t_3 & T_{t_4} & = & t_4 \leftrightarrow \delta_{t_4} \\
\delta_{y_1} & = & \neg(x_1 \wedge t_2) \vee \neg(x_4 \wedge t_1) & T_{y_1} & = & y_1 \leftrightarrow \delta_{y_1} \\
\delta_{y_2} & = & \neg(x_4 \wedge t_1) \wedge t_4 & T_{y_2} & = & y_2 \leftrightarrow \delta_{y_2}
\end{array}$$

Note that the state variable y_1 is now associated with δ_{y_1} instead of Δ_{y_1}. The bit transition relation of Latch 1 is a conjunction of three elementary transition relations

$$T_1 = T_{y_1} \wedge T_{t_1} \wedge T_{t_2} \ .$$

In coarse-grain abstraction methods where only state variables are treated as atoms, when Latch 1 is included in the abstract model, all the six fan-in gates (Gate 1, 2, 4, 5, 7, and 9) are also included; that is, $\widehat{T} = T_{y_1} \wedge T_{t_1} \wedge T_{t_2}$. In the new fine-grain abstraction approach, BNVs as well as latches are treated as atoms, which means that when Latch 1 is in the abstraction, only those gates covered by the elementary transition relation T_{y_1} are included. This is indicated in the figure by the cut ϕ_1, which contains Gate 5, 7, and 9.

In the successive refinements, only the clusters of logic gates that are relevant to a set of refinement variables are added. In the next chapter, an algorithm to identify which variables should be included in the refinement set will be presented. Meanwhile, assume that the current abstract model is not sufficient, and t_1 is added during refinement. The refined model is indicated by the new cut ϕ_2 in the figure, which corresponds to $\widehat{T} = T_{y_1} \wedge T_{t_1}$. The abstract model now contains Latch 1 and Gates 2, 5, 7, and 9. Continuing this process, we will add $y_2, t_4, \ldots$ until a proof or a refutation is found.

It is possible with the new fine-grain approach that gates covered by the transition cluster T_{t_2} (i.e., Gates 1 and 4) never appear in the abstract model, if they are indeed irrelevant to the verification of a given property. This demonstrates the advantage of fine-grain abstraction. In a couple of real-world circuits, we have observed that over 90% of the gates in some large fan-in cones are indeed redundant, even though the corresponding latches are necessary.

The granularity of the new abstraction approach depends on the size of the elementary transition relations, as well as the algorithm used to perform the partition. The partitioning algorithm is important because it affects the quality of the BNVs. In our own investigation, the *frontier* [RAB+95] partitioning method is applied to selectively insert Boolean network variables. The method was initially proposed in the context of symbolic image computation. It works as follows: First, the elementary transition function of each gate is computed from the combinational inputs to the combinational outputs, in a topological order. If the BDD size of an elementary transition function exceeds a given threshold, a Boolean network variable is inserted. For all the gates in the fan-outs of that gate, their elementary transition functions are computed in terms of the new Boolean network variable. Each Boolean network variable or state variable is then associated with a BDD δ_{t_k} or δ_{y_j} for describing its transition function.

Now we formalize the definition of a fine-grain abstract model. Let $t = \{t_1, \ldots, t_n\}$ denote the set of Boolean network variables. Assume that each variable t_k is associated with an elementary transition relation $T_{t_k} = (t_k \leftrightarrow \delta_{t_k})$, and each state variable is associated with a fine-grain bit transition relation $T_{y_j} = (y_j \leftrightarrow \delta_{y_j})$. Let $J = \{1, \ldots, m\}$ be a permutation of the indexes of state variables and $K = \{1, \ldots, n\}$ be a permutation of the indexes of BNVs. The concrete transition relation can be represented as

$$T(x,t,y) = \bigwedge_{j \in J} T_{y_j}(x,w,t,y_j) \wedge \bigwedge_{k \in K} T_{t_k}(x,w,t,t_k) \ .$$

Let the fine-grain abstraction consists of $m' \leq m$ state variables and $n' \leq n$ Boolean network variables, i.e., $\widehat{J} = \{1, \ldots, m'\} \subseteq J$ and $\widehat{K} = \{1, \ldots, n'\} \subseteq K$. Then,

$$\widehat{T}(\hat{x}, \{\check{x}, w\}, \hat{t}, \hat{y}) = \bigwedge_{j \in \widehat{J}} T_{y_j}(\hat{x}, \{\check{x}, w\}, \hat{t}, y_j) \wedge \bigwedge_{k \in \widehat{K}} T_{t_j}(\hat{x}, \{\check{x}, w\}, \hat{t}, t_k) \ .$$

Here $\hat{x} = \{x_j | j \in \widehat{J}\}$ is the subset of present-state variables in the abstract model, $\hat{y} = \{y_j | j \in \widehat{J}\}$ is the subset of next-state variables, and $\hat{t} = \{t_k | k \in \widehat{K}\}$ is the subset of BNVs. All the remaining (invisible) present-state variables and BNVs $(t \backslash \hat{t})$ go into $\check{x}$. We call the variables in $\check{x}$ *pseudo-primary inputs* since they are treated as inputs during symbolic model checking. The initial predicate $\widehat{I}$ is an existential projection of I—an abstract state is called initial if it contains a concrete initial state.

We can fine tune the actual abstraction granularity by controlling the BDD size threshold during the frontier partitioning. In the extreme case such that the BDD size threshold is set to 1 (i.e., a Boolean network variable is created for every logic gate), the optimum deciding (or final) abstraction, among all the possible final abstract models, is the one with the fewest logic gates. In this sense, we have built a connection between abstraction refinement and logic synthesis; abstraction refinement is an iterative process of synthesizing a small abstract model that can prove or refute the given property.

3.3 Abstract Counterexamples

counterexamples found in the abstract model may not be real paths, because some transitions that are responsible for the counterexamples may be forbidden in the concrete model. To check whether an abstract counterexample is real or not, we need to conduct a so-called concretization test. Conceptually, a *concretization test* is reconstructing an abstract execution trace in the concrete model. If the abstract trace cannot be reconstructed, the counterexample is called *spurious*.

When a property fails in the abstract model, there are often many counterexamples. We have observed, for instance, that the number of shortest counterexamples to an invariant property is 10^{45} in a model with 100 state variables. This suggests that analyzing them one-by-one through enumeration is not a good strategy; at the same time, arbitrarily choosing one counterexample is also "a-needle-in-the-haystack" approach, especially if we want to use a single counterexample to guide the computation of refinement variables. It is desirable to capture as many counterexamples as possible and analyzing them simultaneously. Note that the number of counterexamples to a general LTL property can be infinite (e.g., when the counterexamples contain cycles). Even if one focuses on the counterexamples of the shortest length, the number of counterexamples can still be extremely large. For safety properties, however, we can actually capture symbolically all the counterexamples of the shortest length.

Synchronous Onion Rings

All the shortest counterexamples can be captured by a data structure called the Synchronous Onion Rings (SORs). The SORs build upon the reachability onion rings, which have been used frequently in symbolic fixpoint computation. Consider model checking the invariant property $\mathsf{G}\,p$: forward reachability computation from the set I of initial states will produce a set of forward reachability onion rings $\{F^0, ..., F^n\}$, which is the sets of new states encountered during the breadth-first search. For the state transition graph in Part (1) of Figure 3.2, for instance, the set of forward reachability onion rings is represented by $F^0, F^1, ..., F^4$. In particular, $F^1 = \{3, 4, 5\}$ is the set of states that can be reached in one step from the initial states. Note that $\neg p$ is first reached at the 3rd step of the search. An analogous backward reachability analysis from the $\neg p$ state in the 3rd step would reach States $\{8, 9\}$ at the 2nd step, $\{5\}$ at the 1st step, and the initial state 2. That is the set of backward reachability onion rings. Once the forward and backward reachable onion rings are available, the intersection of the two sets of states at each corresponding step gives the synchronous onion rings,

$$S^0 = \{2\}, \quad S^1 = \{5\}, \quad S^2 = \{8, 9\}, \quad S^3 = \{12\} \ .$$

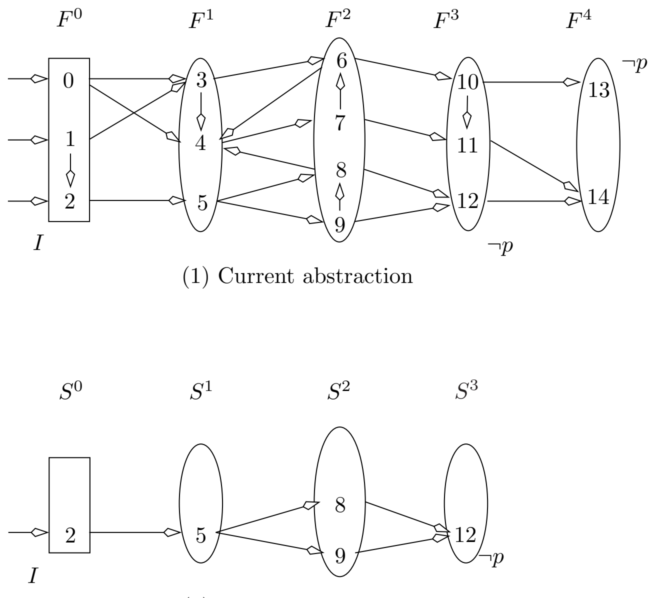


Figure 3.2. Ariadne's bundle of synchronous onion rings.

The term "Ariadne's bundle" is used to denote the subrelation T^B induced by considering only the transitions between a state at one step to another state in the next step in the SORs. It comprises the bundle of all shortest counterexamples, and no other counterexample. *(There is an interesting analogy between the abstract counterexamples and the magic Ariadne's thread: in the Greek mythology, Theseus needs the thread to navigate through the Labyrinthus to kill the monster Minotaurus; in abstraction refinement, one needs the guidance of abstract counterexamples to remove the "false negatives.")*

Note that the state transition graph of the Ariadne's bundle has significantly less states and transitions than the abstract model. In this simple case, there are two shortest counterexamples, both of length 3. In practice, however, the number of counterexamples in the SORs is typically large.

Multi-Thread Concretization Test

Once all the abstract counterexamples are captured in the SORs, we need to check whether they are real paths in the concrete model. This, unfortunately, cannot be accomplished by conventional simulation even if a single abstract path is being reconstructed. This is because an abstract path may not have a complete set of assignments to all the input variables—one abstract transition often corresponds to multiple concrete transitions. Therefore, the concretization test requires the use of symbolic simulation techniques.

Various symbolic techniques have been proposed for concretization test. In [CGJ+00], for instance, BDD based image computation was used in the concrete model to reconstruct all the concrete paths inside a single abstract path. In [WHL+01], the search of a concrete path inside a single abstract path was performed by an ATPG (automatic test pattern generation) engine. In [CGKS02, CCK+02], SAT solvers are used to perform the reconstruction. However, one thing is common to all these methods: concretization test deals with only a single abstract counterexample.

We want to simultaneously concretize of all the shortest counterexamples in the SORs. This multi-thread concretization test can also be formulated into a Boolean satisfiability problem, which can be solved by a SAT solver. Given a set of rings in the SORs $\{S^0, \cdots, S^L\}$, the SAT problem can be defined as $\Psi = \Psi_A \wedge \Psi_S$, where

$$\Psi_A = I(V^0) \wedge \bigwedge_{0 \le i < L} T(V^i, U^i, V^{i+1}) \ ,$$

$$\Psi_S = \bigwedge_{0 \le i \le L} S^i(V^i) \ .$$

Formula Ψ_A represents the unrolling of the concrete transition relation for exactly L time frames, and Ψ_S represents the constraints coming from the abstract SORs. V^i and U^i are the set of state variables and the set of combinational variables at the i-th time frame, respectively. The predicate $S^i(V^i)$ represents that states at the i-th time frame are in the i-th ring of the SORs. We give a graphic illustration of the multi-thread concretization test in Figure 3.3.

Formula Ψ is satisfiable if and only if there exists a concrete counterexample inside the abstract SORs. When Ψ is satisfiable, the set of assignments returned by the SAT solver represents a concrete path from an initial state to a $\neg p$ state.

In symbolic model checking, S^i initially is a BDD representing the set of states in the i-th ring of the SORs. To translate it into the CNF

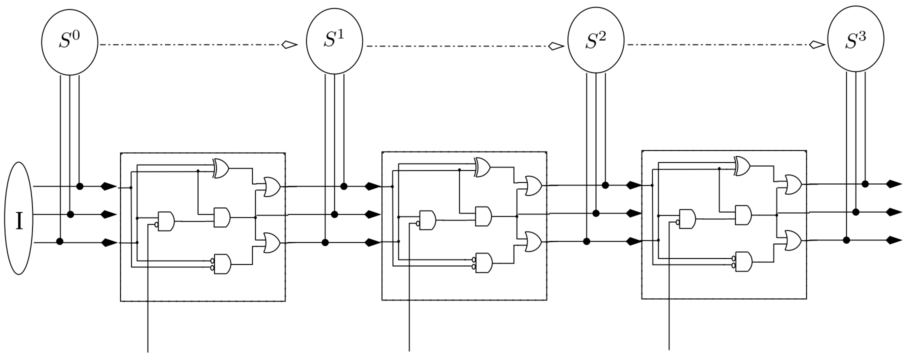


Figure 3.3. Multi-thread concretization test.

format, we need an encoding procedure that takes a BDD as input and produces a conjunction set of clauses. The translation, as illustrated in Figure 3.4, goes as follows: First, we translate the BDD into a combinational logic circuit by replacing every internal BDD node with a 2-to-1 *multiplexer*. Since each *multiplexer* consists of 3 AND gates, this translation is linear. Once the AND-INVERTER graph is built, encoding it into a CNF formula is straightforward. As we have explained before, the transition relation of each AND gate can be encoded as three clauses, if we are allowed to add auxiliary variables.

However, this linear encoding scheme requires the addition of a large number of auxiliary variables, one for each the logic gate. An alternative approach is to enumerate all the minterms (or cubes) of the complemented function $\neg S^i$, and convert the minterms (or cubes) into CNF clauses. A minterm (or cube) of a Boolean function corresponds to a root-to-leaf path in its BDD representation. This encoding scheme has been used in [CNQ03]. Although no auxiliary variable is required in this approach, the number of root-to-leaf paths in a BDD can be exponential with respect to the number of BDD nodes.

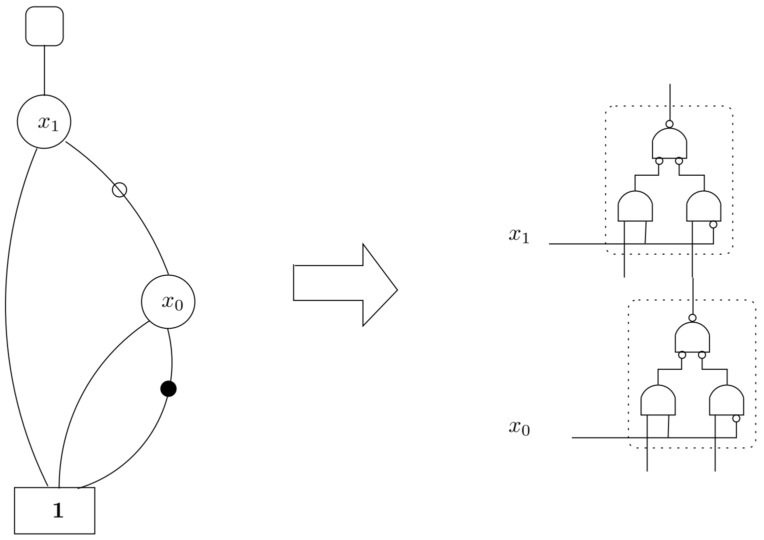


Figure 3.4. Translating a BDD into a combinational circuit.

3.4 Further Discussion

Since the introduction of the general abstraction refinement framework by Kurshan [Kur94], significant progresses have been made on the refinement algorithms and the concretization test, through the incorporation of latest development of BDD and SAT algorithms [LPJ$^+$96, JMH00, CGJ$^+$00, CGKS02, CCK$^+$02, GGYA03]. However, in most of these previous works, the abstraction granularity remains at the state variable level. The fine-grain abstraction approach described in this chapter is unique in the sense that it treats both state variables and Boolean network variables as abstraction atoms. With fine-grain abstraction, the refinement strategies must search in a two-dimensional space. Refinement in the *sequential* direction is comprised of the addition of new state variables only, which is typical of much of the pioneering prior art of Clarke *et al.* [CGJ$^+$00, CGKS02]. Refinement in the *Boolean* direction is comprised of the addition of Boolean network variables only. Boolean network variables belong to a special type of cut-set variables in the circuit network.

In [WHL$^+$01], Wang *et al.* proposed a *min-cut model* in abstraction refinement to replace the conventional coarse-grain abstract model. They first defined a free-cut set of signal as those at the boundary of the transitive fan-in and transitive fan-out of the visible state variables. They then computed a min-cut set of signals between the combinational inputs and the free-cut signals; the logic gates above this min-cut were included in the reduced abstract model. Since the transition relation is expressed in terms of a smaller set of signals, it often has a smaller BDD representation. However, the abstraction granularity of this method is still at the state variable level. In particular, logic gates above the free-cut are always in the abstract model, regardless of whether or not they are necessary for verification. In [CCK$^+$02], Chauhan *et al.* adopted a similar approach to simplify the coarse-grain model. In their method, the further reduction of abstract model was achieved by *pre-quantifying* pseudo-primary inputs dynamically inside each image computation step. This method shares the same drawback as that of [WHL$^+$01].

In [GKMH$^+$03], Glusman *et al.* computed a min-cut set of signals between the boundary of the current abstract model and the primary inputs, and included logic gates above this cut in the abstract model. Since an arbitrary subset of combinational logic gates in the fan-in cone of a state variable could be added, the abstraction granularity was at the gate level. However, there are significant differences between their method and the fine-grain abstraction. First, fine-grain abstraction aims at directly reducing the size of the transition relation by avoiding adding irrelevant elementary transition relations, while the method in [GKMH$^+$03] aims at reducing the number of cut-set variables. Second, with fine-grain abstraction, we can differentiate the two refinement directions and can control the direction at each refinement iteration, while the method in [GKMH$^+$03] does not differentiate the two directions. Third, in their method, logic gates added during refinement cannot be removed from the abstract model afterward – even if later they are proved to be redundant. Under fine-grain abstraction, the removal of previously added variables and intermediate logic gates is possible.

In [ZPHS05], Zhang *et al.* proposed a technique called "dynamic abstraction," which maintains at different time steps separate visible variable subsets. Their approach can be viewed as a finer control of abstraction granularity in the time axis, since they are using different abstract models for at different time frames. However, at each time frame, their abstraction atoms are still latches. Therefore, this approach is entirely orthogonal to our fine-grain abstraction.

Applying BDD constraints to help solving the series of SAT problems in bounded model checking has been studied by Gupta *et al.* in [GGW$^+$03a]. In their work, the BDD constraints were derived from the forward and backward reachability onion rings. They used such constraints primarily to improve the BMC induction proof by restricting induction to the (over-approximate) reachable state subspace (instead of the entire universe). In our multi-thread concretization test, we have used the same method as that of [GGW$^+$03a] for translating BDDs into CNF clauses. Another encoding scheme (from BDDs to CNF clauses) was studied by Cabodi *et al.* in [CNQ03]. The idea is to complete the given BDD and enumerate all the minterms (or cubes) of the BDD and convert each into a CNF clause. The same authors also proposed a hybrid encoding scheme that combines the aforementioned two encoding schemes and tries to make a trade-off between them.

Chapter 4

REFINEMENT

If the abstract counterexamples are all spurious, we need to refine the current abstraction by identifying a subset of currently invisible variables and restoring their bit transition relations. To improve abstraction efficiency, it is crucial to identify those variables that have larger impact on removing *false negatives*. In most of the previous counterexample driven refinement methods [CGJ$^+$00, WHL$^+$01, CGKS02, CCK$^+$02], refinement variable selection was guided by a single spurious counterexample. This is "a-needle-in-the-haystack" approach, since in practice the number of counterexamples tends to be extremely large.

In this chapter, we propose a refinement algorithm driven by all the shortest counterexamples in the synchronous onion rings. The new algorithm, called GRAB, does not try to kill the entire bundle of spurious counterexamples in one shot. Instead, it identifies critical variables that are in the local support of the current abstraction by viewing refinement as a two-player reachability game in the abstract model. Due to the global guidance provided by the SORs and the quality and scalability of the game-based variable selection computation, GRAB has demonstrated significantly performance advantages over previous refinement algorithms—it often produces a smaller final abstract model that can prove or refute the same property.

At the end of each refinement generation, i.e., a set of refinement steps that are responsible for the removal of the spurious counterexamples of a certain length, we also apply a SAT based greedy minimization to the refinement set in order to remove redundant variables; the result is a minimal refinement set that is sufficient to kill the entire bundle of abstract counterexamples.

4.1 Generational Refinement

In this section, we will illustrate the generic process of abstraction refinement by using the simple example in Figure 4.1. Here we consider the concrete model $\mathcal{A}$ as the synchronous (parallel) composition of three submodules: M_1, M_2, and M_3. That is,

$$\mathcal{A} = M_1 \parallel M_2 \parallel M_3 \ \ .$$

Each component M_i has one state variable v_i. The state variable v_1 can take four values and thus must be implemented by two binary variables; the other two state variables (v_2, v_3) are binary variables. Variable v_4, which appears in the edge labels in M_1, is a primary input. Assume that the property of interest is $\mathsf{G}(v_1 \neq 3)$, i.e., State 3 in M_1 is not reachable in the concrete model $\mathcal{A}$.

The right-hand side of Figure 4.1 is the state transition graph of the concrete model. It is clear that the given property fails in the concrete model, as shown by the bold edges, which exhibit a concrete counterexample of length 4: (000, 111, 200, 211, 300).

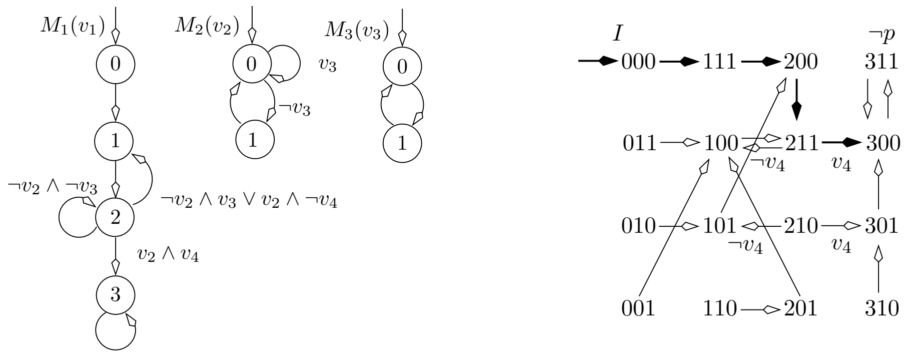


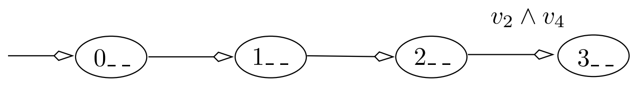
Figure 4.1. An example for abstraction refinement.

The initial abstract model is $\widehat{\mathcal{A}} = M_1$, which preserves only the state variable appearing in the given property, and all the other state variables are treated as pseudo-primary inputs. There is an abstract counterexample of length 3: ($0\_\_$, $1\_\_$, $2\_\_$, $3\_\_$). This counterexample is spurious because it is not *concretizable* in $\mathcal{A}$; in particular, no direct transition is possible from 200 to $3\_\_$.

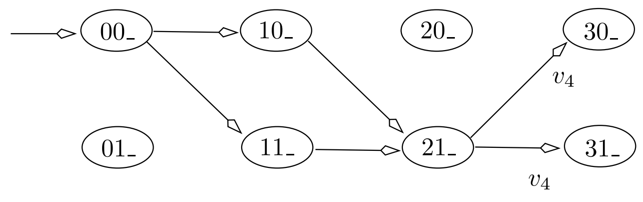
Although this example is simple, it demonstrates an important aspect of the abstraction refinement process. Refinement algorithms like those in [CGJ$^+$00, CGKS02], since they are actuated by a single counterexample, may pick only variable v_2 for refinement. However, after this refinement, an abstract counterexample of the same length still exists—for instance, it can be (00_ , 10_ , 21_ , 30_). Therefore, the refinement set $\{v_2\}$ is not a sufficient set to kill the abstract counterexample (0_ _ , 1_ _ , 2_ _ , 3_ _), although it can separate the set of *deadend states* $\{200\}$ from the set of *bad states* $\{211, 210\}$. (In [CGJ+00], deadend states are a subset of concrete states inside an abstract state $\hat{s_i}$ that are reachable from the initial states but can not reach any concrete states in $\hat{s_{i+1}}$; bad states are a subset of concrete states inside $\hat{s_i}$ that can reach some concrete states in $\hat{s_{i+1}}$.) This is suggestive of the danger of placing too much refinement emphasis on a single abstract counterexample. Of course, it is much more of a problem when the SORs contain an extremely large number of counterexamples. In the presence of many counterexamples, computing the set of refinement variables based on one arbitrarily chosen counterexample can be ineffective.

We now illustrate the SOR based generational refinement framework using the above example. In building the SORs, we have pruned away self loops and back edges to focus on the shortest counterexamples in the current abstract model. When $\widehat{\mathcal{A}} = M_1$, the SORs contain just the four states of M_1, which are connected by the four forward edges. The initial SORs are shown in Part (a) of Figure 4.2. Since the first *generation* of shortest counterexamples are of length 3, the refinement process starts by dealing with the SORs of length 3 until all paths in them are killed. Note that in M_1 only edges from State 2 to States 1, 2, and 3 are labeled. We will discuss below in Section 4.2 that these edge labels cause the variable selection routine to pick v_2 for the first refinement iteration, and as a result, the refined model is $\widehat{\mathcal{A}} = M_1 \parallel M_2$. However, $\widehat{\mathcal{A}}$ is not constructed in the naive way of building the transition relation for M_1 and M_2 again and conjoining them together, but by a more efficient two-step process.

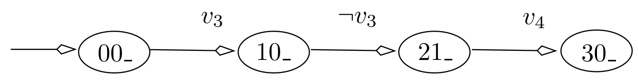
First, we split the states of M_1 according to the labels on their outgoing edges, as shown in Part (b) of Figure 4.2. Because of the label $v_2 \wedge v_4$, the last abstract edge (2_ _,3_ _) is split into two rather than four refined edges. State 20_ is now backward unreachable from $\neg p$, so the two incoming edges, (10_,20_) and (11_,20_), are removed. The outgoing edges from the state 01_ are removed because the state 01_ is not an initial state. States like 20_ are called the *deadend* states. The concept of deadend states is critically involved in the refinement variable selection algorithm, as discussed below in Section 4.2. Note that all splits that make the SORs change from the one in Part (a) to the one in Part (b) are done before M_2 is brought in. The second step is to actually take the composition of M_2 with the remaining edges of the SORs. This kills



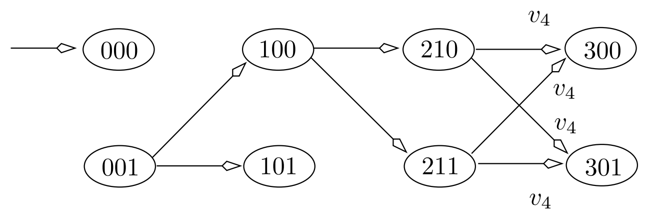
(a) original SORs



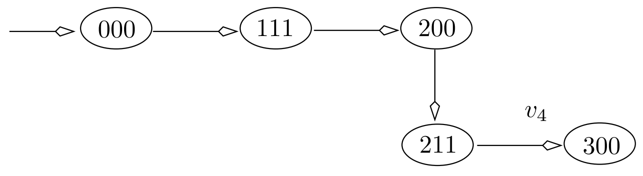
(b) split states



(c) account for TR of M_2



(d) split states again



(e) compute length-4 SORs

Figure 4.2. The generational refinement process.

the edges ($11\_$,$21\_$) and ($21\_$,$31\_$), and leads to the reduced SORs in Part (c).

After the above refinement step, the number of length-3 spurious counterexamples is decreased, but they have not been removed completely. Now v_3 is selected as the next refinement variable. We then proceed to again take the first part of the two-step refinement process, as illustrated in Part (d). The result is a disconnection of I from $\neg p$, because there is no outgoing edge from the sole remaining initial state. At this point, it has been proved that no concrete counterexample of length 3 exists, so this generation of refinements is complete.

During the two refinements in the first generation, i.e., adding v_2 and adding v_3, the SORs are updated *incrementally* since each ring of the SORs is a subset of the corresponding ring of the previous SORs. The BDD don't cares associated with this incremental process lend critical efficiency to the SORs refinement process.

Next, we build from scratch the new SORs, which are of length 4 as is shown in Part (e). This final set of SORs contains a single counterexample that is concretizable in $\mathcal{A}$, as discussed above in reference to Figure 4.1. Therefore, the given property fails.

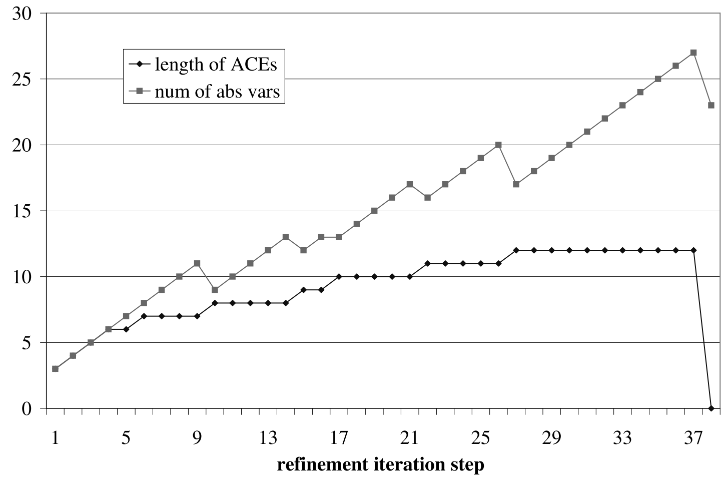


Figure 4.3. The effect of generational refinement, with refinement minimization.

To summarize, the proposed generational refinement algorithm does not try to remove all the spurious abstract counterexamples in one shot.

Instead, it identifies local variables that are critical to refinement by exploiting the global guidance provided by the synchronous onion rings. It may take a set of refinement steps, called a *generation* of refinements, to remove all the abstract counterexamples of a given length.

The effect of generational refinement is illustrated in Figure 4.3. The data are obtained from a real-world example in which the given invariant property holds in the concrete model. The upper curve represents the number of state variables in the abstract model at the different refinement steps, and the lower curve represents the length of the abstract counterexample (ACE). Each flat segment of the lower curve corresponds to a generation of refinement steps. A generation consists of a number of consecutive refinement steps, all with SORs of the same length. Note that within the same generation, the size of the abstract model keeps increasing; every time the length of the shortest abstract counterexample changes (between generations), the number of abstract variables may decrease, due to the greedy refinement minimization procedure that tries to keep the abstraction as small as possible by removing redundant variables. Our experience shows that this greedy minimization is critical in achieving a high abstraction efficiency.

The GRAB Algorithm

We now give the pseudo code of our abstraction refinement algorithm in Figure 4.4. It is called the algorithm Grab, for *Generational Refinement of Ariadne's Bundle*. Grab accepts as inputs a concrete model $\mathcal{A}$ and a property Φ of the form $\Phi = \mathsf{G}\, p$.

The initial abstract model $\widehat{\mathcal{A}}$ contains only those state variables that appear in the local support of the property. In Figure 4.1, for example, the initial abstract model contains only variable v_1, because the property is $\mathsf{G}(v_1 \neq 3)$. Let $\{S^0, S^1, \ldots, S^L\}$ be the length-L synchronized onion rings, where S^0 is a subset of initial states, S^L is a subset of states satisfying $\neg p$, and S^j is a set of states on the shortest abstract paths from S^0 to S^L.

The outer loop is over the length, L, of the current generation of SORs. With the abstract model being gradually refined, L is guaranteed to grow monotonically in the outer loop. The action starts in Line 3, where BDD based forward reachability analysis is used to compute the *forward onion rings* from the initial states to $\neg p$ states. If $\neg p$ cannot be reached in $\widehat{\mathcal{A}}$, it cannot be reached in $\mathcal{A}$ either. In this case of early termination, Grab returns true. Otherwise, the first set of SORs are computed.

A SAT based concretization test is then performed in the concrete model. Here, it simultaneously tries to concretize all the abstract coun-

```
GRAB($\mathcal{A}, \Phi$)    {
    $\widehat{\mathcal{A}}$ = INITIALABSTRACTION($\mathcal{A}, \Phi$);
    **while** (true) {        //Loop over SORs with different length
        $\{S^l\}$ = COMPUTESORS($\widehat{\mathcal{A}}, \Phi$);
        **if** ( $\{S^l\}$ is empty )
            **return** true;
        CCE = MULTITHREADCONCRETIZATION($\mathcal{A}, \{S^l\}$);
        **if** (CCE not empty)
            **return** (false, CCE);
        $\{S^l_R\} = \{S^l\}$;
        **while** (true) {        //Loop at the current length
            $\widehat{\mathcal{A}}$ = REFINEABSTRACTION($\widehat{\mathcal{A}}, \{S^l_R\}$);
            $\{S^l_R\}$ = REDUCESORS($\widehat{\mathcal{A}}, \{S^l_R\}$);
            **if** ($\{S^l_R\}$ is empty)
                **break** ;
        }
        $\widehat{\mathcal{A}}$ = REFINEMENTMINIMIZATION($\widehat{\mathcal{A}}, \{S^l\}$);
}  }

REFINEABSTRACTION($\widehat{\mathcal{A}}, \{S^l\}$)    {
    $w_S = \{\ \}, w_E = \widehat{w}$;
    **while** ($|w_S| <$ threshold) {
        $v$ = PICKBESTVAR($\widehat{\mathcal{A}}, \{S^l\}$);
        $w_S = w_S \cup \{v\}, w_E = w_E \setminus \{v\}$;
    }
    **return** COMPUTEABSTRACTION($\widehat{\mathcal{A}}, w_S$) ;
}
```

Figure 4.4. The GRAB abstraction refinement algorithm.

terexamples in the SORs by one satisfiability instance. If any of these counterexamples can be concretized, the property Φ is proved to be false, and the concrete counterexample (CCE) is returned. If no concrete counterexample exists, we start the inner loop over the refinements in this generation.

In the inner loop, although the number of abstract counterexamples in the SORs decreases monotonically, the length of the SORs does not change. Since all the abstract counterexamples have been proved spurious at the very beginning, no concretization test is needed inside this loop. We use $\{S_R^l\}$ to represent the set of "reduced SORs." Each time the abstract model is refined, the synchronous onion rings are reduced (Line 12) until all the spurious counterexamples disappear. Typically a few passes through the inner loop produce the break-out, which implies that the set of refinement variables added in the current generation constitutes a *sufficient set*—a set of refinement variables, when added, kill the entire length-L SORs.

Termination of the GRAB procedure is guaranteed by the finiteness of the model. The game based heuristic for picking refinement variables will be presented in the next section, followed by a SAT based greedy minimization of the refinement set.

Prior art in abstraction refinement algorithms [CGJ$^+$00, CGKS02, CCK$^+$02] can also be described with a similar framework of pseudo code, however, these algorithms are all based on the analysis of a single counterexample. As we have pointed out earlier, even an optimal refinement algorithm based on a single counterexample cannot necessarily guarantee a good overall refinement. We will compare GRAB with these alternative methods later in the experimental results section.

4.2 Refinement Variable Selection

We consider the refinement variable selection problem as a two-player reachability game in the abstract model. Given the abstract model $\widehat{\mathcal{A}}$ and a target predicate $\neg p$, the model checking of $\mathsf{G}\,p$ with respect to a model $\hat{\mathcal{A}}$ can be viewed as a two-player concurrent reachability game [EJ91, JRS02]. The two players of this game are the *hostile environment* and the *abstract system*; they play by controlling the values of the pseudo-primary inputs of the abstract model. In $\widehat{\mathcal{A}}$, the pseudo-primary inputs, denoted by $\check{x}$, are the set of invisible variables. We need to partition the set $\check{x}$ into two subsets, $\check{x} = w_E \cup w_S$, among which w_E is controlled by the *environment (player)*, and w_S is controlled by the *system (player)*.

The positions of the game correspond to the states of the abstract model. Let $\hat{X}$, a valuation of the set of present-state variables $\hat{x}$, be a position (similarly, let capital values of other vector names stand for their valuations). From one position $\hat{X}$, the *environment (player)* chooses values for the variables in w_E and simultaneously the *system (player)* chooses values for variables in w_S. The new position is the unique $\hat{Y}$ determined by the abstract transition relation $\widehat{T}(\hat{X}, \check{X}, \hat{Y})$. Note that we are assuming that the model $\langle \widehat{T}, \widehat{I} \rangle$ is a closed system. This is not a problem at all since any open system can be transformed into a closed system by treating inputs as free state variables. The goal of the *environment (player)* is forcing the abstract model to go through spurious paths and reach a state labeled $\neg p$ in spite of the opposition of the *system (player)*. A (memoryless) strategy for the *environment* is a function that maps each state of $\widehat{\mathcal{A}}$ to one valuation of the variables in w_E. Likewise, a strategy for the *system* is a function that maps each state of $\widehat{\mathcal{A}}$ to one valuation of the variables in w_S.

To relate this reachability game to our refinement problem, we consider the *environment* the hostile player (and we want the *system* to win). Next, we define the winning positions for the hostile environment.

Definition 4.1 *A position $\hat{X}$ in $\widehat{\mathcal{A}}$ is a* winning position *for the hostile environment if there exists an environment strategy such that, for all system strategies, $\neg p$ is eventually satisfied.*

The concept of winning position is closely related to the refinement problem. Before the abstract model is refined, there are spurious paths from the initial states to states labeled $\neg p$. This corresponds to the partition $(w_E = \check{x}, w_S = \{\ \})$—the hostile environment controls all the invisible variables. Assume that $\widehat{\mathcal{A}}$ is deterministic, then the *environment* always has a winning strategy because it can force any transition by controlling all the variables in $\check{x}$. In refinement, our task is to remove some variables from w_E and put them into w_S. We wants to identify a small subset of variables that, once being removed from w_E, will significantly reduce the number of winning positions for the hostile environment.

Therefore, the refinement problem can be stated as follows: among all the possible partitions of $\check{x} = w_E \cup w_S$, choose the one that gives the *environment* the least number of winning positions. Once the partition is identified, variables in w_S together with their elementary transition relations are added into the abstract model. In this two-player reachability game, the partition that favors the hostile environment the least also favors the abstract system most.

Given an input variable partition $\check{x} = \{w_E, w_S\}$ and the spurious counterexamples in the SORs $\{S^0, S^1, ..., S^j, ..., S^L\}$, the *environment*'s winning positions inside S^j are computed as follows:

$$\exists w_E.\ \forall w_S.\ \exists \hat{y}.\ [S^j(\hat{x}) \wedge \widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y})]\ ,$$

which is the subset of S^j states from which the *environment* can force the transition to S^{j+1} despite the opposition of the *system.*

The normalized number of winning positions for the hostile environment inside S^j is computed as follows:

$$N_j^{\{w_E,w_S\}} = \frac{|\exists w_E.\forall w_S.\exists \hat{y}.[S^j(\hat{x}) \wedge \widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y})]|}{|S^j(\hat{x})|}\ .$$

Given a set S, we use $|S|$ to denote the cardinality of the set. The normalized number of winning positions, denoted by N_j, is a good indicator of the impact of refining with respect to the variables in w_S. For the purpose of refinement, we prefer the partition that gives N_j the lowest value.

We use universal quantification ($\forall$) is mimic the impact of parallel composition on the abstract model, since both reduce the number of enabled edges. It can be shown that when an edge label has an essential variable—a variable which factors out of its label (all the edges in Figure 4.5 except the edge from state 5 to state 7), composing that variable with the abstract model splits the abstract edge into two edges (instead of four). Furthermore, among the two new tail states created by the splits, one has no fan-out—that is, it is a *deadend split.*

The abstract model in Figure 4.5 has $S^0 = \{1, 2, 5, 6\}$, $S^1 = \{3, 4\}$, and $\check{x} = \{g, f\}$. When the partition of $\check{x}$ is such that $w_E = \{g\}$ and $w_S = \{f\}$, the set of winning positions for the hostile environment is $\{1, 2\}$. State 1 is a winning position because when the hostile environment makes the assignment $g = 1$, the system player will be forced to a

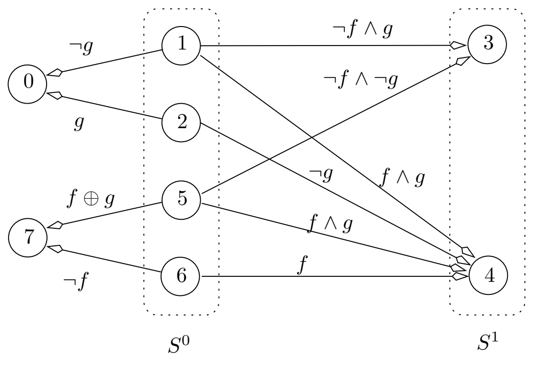


Figure 4.5. Illustration of the winning positions.

$\neg p$ state (either 3 or 4) no matter what value it assigns to f. A similar argument applies to State 2 as well. According to the definition of N_j,

$$\begin{aligned}
N_0^{\{\{g,f\},\{\}\}} &= 1.0,\\
N_0^{\{\{g\},\{f\}\}} &= 0.5,\\
N_0^{\{\{f\},\{g\}\}} &= 0.25,\\
N_0^{\{\{\},\{g,f\}\}} &= 0.0.
\end{aligned}$$

It indicates that g is a better candidate than f for refinement, because putting g alone in w_S gives the hostile environment one winning position, while putting f alone in w_S gives it two winning positions.

A further explanation of the heuristic via state splitting is shown by the two examples in Figure 4.6 and Figure 4.7. In the first example, g is an *essential* variable to the label on the spurious transition $2 \rightarrow 4$, and f is an irrelevant variable. A variable v is essential to a function $f(v)$ if and only if either $f(0) = 0$ or $f(1) = 0$. By intuition, one would prefer refining with g, because f is irrelevant. This is the right choice because it will split State 2 into two new states, ($\neg g$,2) and (g,2), only one of which has an out-going edge to state 4. Therefore, it is *possible* to remove this spurious edge—in the case when State ($\neg g$,2) becomes unreachable after refinement. Refining with f, however, does not have such an impact since both of the two new states, ($\neg f$,2) and (f,2), will have out-going edge to State 4. This is consistent with the game based analysis—State 2 is a winning position for the hostile environment if it controls g.

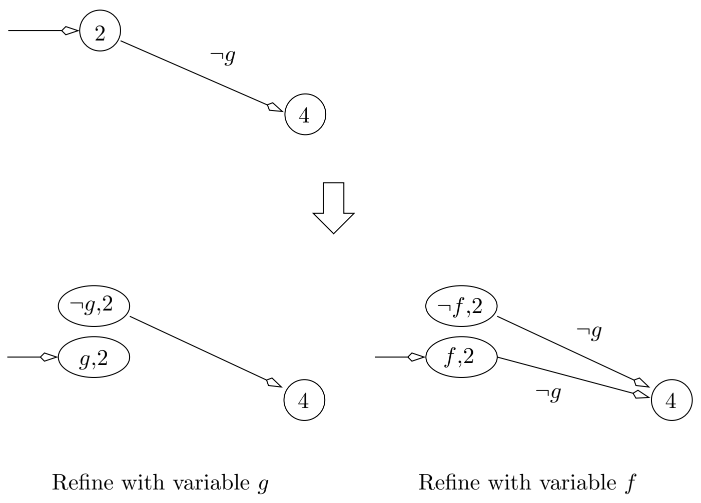


Figure 4.6. An example for state splitting: g is a better refinement candidate.

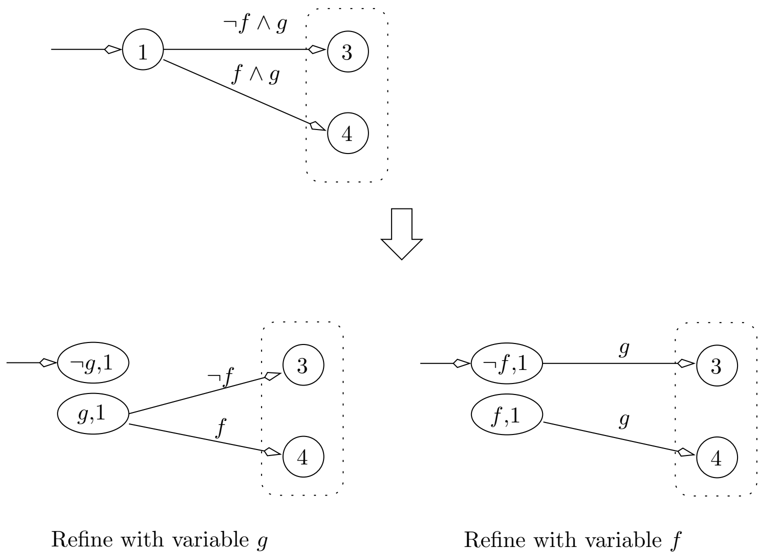


Figure 4.7. Another example for state splitting: g is still a better refinement candidate.

In the second example (Figure 4.7), it is no longer straightforward to figure out that g is a better refinement candidate than f, because both g

and f appear in the edge labels. However, the game based analysis tells us that State 1 is a winning position for the hostile environment if it controls g. Refining with g produces a similar deadend split—only one of the two new states has an out-going edge to the next ring. Therefore, it is *possible* to remove this spurious edge—in the case that State (g,1) becomes unreachable after refinement. Refining with f, however, leaves the spurious edges intact, since both of the two new states have outgoing edges to the next ring. This is also consistent with the game based analysis—There is no winning position for the hostile environment if the abstract system controls g.

To summarize, our refinement algorithm selects a small subset of invisible variables into w_S such that the partition $\{w_E, w_S\}$ minimizes the

$$\sum_{0\le j\le l} N_j^{\{w_E,w_S\}}, \ \ \forall\{w_E, w_S\} \ .$$

This is greedily approximated inside REFINEABSTRACTION: the one variable that minimizes the above number is repeatedly picked (Line 19 in Figure 4.4).

The computation of winning positions is similar to BDD based pre-image computation. A normal pre-image would have $\exists w_E.\ \exists w_S.\ \exists \hat{y}.$ instead of $\exists w_E.\ \forall w_S.\ \exists \hat{y}.$. Since $S^j(\hat{x})$ does not have any quantified variable in its support, we can pull it out of the quantification operators. Furthermore, the following common intermediate result

$$\exists \hat{y}.[\widehat{T}(\hat{x}, \check{x}, \hat{y}) \wedge S^{j+1}(\hat{y})]$$

can be shared among various partitions of $\check{x}$. Combining these simplifications with the conventional early quantification techniques, we can make the computation of N_j very efficient. Another thing we would like to point out is that $w_E \cup w_S$ contains only invisible variables that are in the local support of the current abstract model, not necessarily the entire set of invisible variables.

The refinement method in [GKMH$^+$03] also used more than one counterexample. It was based on the classification of invisible variables into *strong 0/1 signals* and *conditional 0/1 signals*. A strong 0/1 signal was defined as in all counterexamples, the value of the signal at the given step of the trace is always 0 or always 1, respectively. In other words, only when a variable is *essential* with respect to the labels of all the abstract edges from S^j to S^{j+1}, will it be classified as a strong 0/1 signal. Otherwise, it will be classified as a conditional 0/1 signal. In practice, however, strong 0/1 signals as defined in [GKMH$^+$03] are very rare cases. In fact, both f and g in Figure 4.5 are not strong 0/1 signals; according to [GKMH$^+$03], both would be classified as conditional 0/1 signals, and therefore are assigned the same weight in refinement variable selection. In contrast, Grab can tell that g is actually a better refinement candidate than f. In general, Grab is often more accurate in identifying important refinement variables.

4.3 Keep the Refinement Set Small

In fine-grain abstraction, there are two types of elementary transition relations: one is associated with state variables, while the other is associated with Boolean network variables. The addition of these variables to the abstract model indicates two different directions for the refinement. In the *sequential direction*, we only add state variables (or latches), which results in a potentially larger state space in the refined model. In the *Boolean direction*, we only more logic gates in the fan-in cones of the visible state variables, which means the state space will stay the same but some transitions will be removed. Our experience shows that if one makes no distinction between these two types of variables, the final abstract model may contain many redundant state variables. This suggests that the refinement procedure needs some guidance on the proper direction.

Choosing the Refinement Direction

At every refinement step, the proper refinement direction needs to be predicted. If going in the Boolean direction (i.e., without adding any state variable) can remove the spurious counterexamples, then the Boolean direction should be chosen to avoid a potentially larger state space. A satisfiability check similar to concretization test can be used to predict the refinement direction. We can formulate the SAT problem as a constrained BMC instance such that it is satisfiable if and only if the entire fan-in cones of the visible state variables makes the instance unsatisfiable. This SAT problem differs from multi-thread concretization test in that an *extended abstract model*, instead of the concrete model, is unrolled exactly L time frames.

Definition 4.2 *Given a fine-grain abstract model* $\langle \widehat{T}, \widehat{I} \rangle$*, the extended abstract model* $\langle \widehat{T}_\epsilon, \widehat{I} \rangle$ *is defined as the one that contains all the visible state variables of the fine-grain model and the complete fan-in logic cones of these state variables.*

Refer to Figure 3.1, when the current (fine-grain) abstract model contains Latch 1 and Gate 5, 7, and 9, the extended abstract model contains Latch 1, and Gate 1, 2, 4, 5, 7, and 9.

Let $\widehat{T}_\epsilon$ be the transition relation of the extended abstract model; then, the refinement direction can be decided by solving the SAT problem of

$\Psi^\epsilon = \Psi_E \wedge \Psi_S$, defined as follows:

$$\Psi_E = I_0(X^0) \bigwedge_{0 \le i < L} \widehat{T}_\epsilon(X^i, W^i, X^{i+1}) \ ,$$
$$\Psi_S = \bigwedge_{0 \le i \le L} S^i(V^i) \ .$$

Formula Ψ_E enables only paths of length L that are allowed by the extended abstract model, and Ψ_S enables only paths embedded in the abstract SORs. If Ψ^ϵ is unsatisfiable, it means that the abstract counterexample in the SORs do not exist in the extended abstract model. In other words, it is possible to kill all the length-L counterexamples by adding some logic gates that are in the fan-in cones of the visible state variables; in this case, we opt for the Boolean direction. On the other hand, if Ψ^ϵ is satisfiable, it means that even adding all the logic gates in the Boolean direction cannot kill the spurious counterexamples. In this latter case, more state variables need to be added by going in the sequential direction.

Minimizing the Refinement Set

Once the entire set of spurious counterexamples in the SORs are gone, all the newly added variables forms a sufficient refinement set—that is, they are enough to remove all the length L spurious counterexamples. However, this refinement set may not be minimal. In previous work [WHL$^+$01, CCK$^+$02], a trial-and-error based greedy minimization has been used to remove redundant variables from the refinement set. This kind of greedy minimization can also be applied here in Grab. With fine-grain abstraction, however, the minimization must be applied in both refinement directions, with respect to the entire SORs instead of a single counterexample.

For the spurious counterexamples of a certain length, refinement is performed first in the sequential direction. As soon as a sufficient set of state variables is added, it is minimized with respect to the entire bundle of counterexamples before refinement shifts to the Boolean direction. When a set of state variables is being minimized, the extended abstraction model induced by these state variables is unrolled to form the SAT formula Ψ^ϵ (referred to the previous section). We go through all the state variables of the refinement set, one at a time, to check if any of them is redundant. This is done by temporarily removing a state variable from the abstract model and check whether Φ^ϵ is still unsatisfiable. Every time a state variable is removed from the refinement set, all the Boolean network variables that are relevant only to this state variable are also pruned away. If removing a state variable does not cause any spurious counterexample to reappear, then the variable is redundant and will be dropped permanently. However, if after removing a variable, the new formula Ψ^{ϵ} becomes satisfiable, we need to add the variable back and proceed to the next variable.

Note that a sufficient set of *state variables* only means that the abstract counterexamples do not appear in the extended abstract model ($\widehat{T}_{\epsilon}$), but may still exist in the fine-grain abstract model ($\widehat{T}$). After the greedy minimization in the sequential direction, we will stay in the current refinement generation and switch to the Boolean direction. After refinement shifts to the Boolean direction, only Boolean network variables will be added until the same set of SORs is removed again. At this point, the set of newly added Boolean network variables is also a sufficient set and will be greedily minimized. The minimization procedure for BNVs is similar to that for state variables, with the only difference being that now we are using the fine-grain abstract model (with $\widehat{T}$ instead of $\widehat{T}_{\epsilon}$).

We define the greedy refinement procedure more formally as follows:

Definition 4.3 *Given a sufficient set of refinement variables and the SORs, the refinement minimization problem can be defined as finding the minimal subset of refinement variables that can kill the spurious counterexamples.*

Our minimization uses a SAT solver and the satisfiability checks are similar to the multi-thread concretization test. BDDs are translated into CNF formulae, as described in the previous chapter, to constrain the SAT problem. In concretization test, the bounded model is the unrolling of the concrete model, and it captures all the length-L paths allowed by the concrete model. In refinement minimization, the bounded model is the unrolling of an abstract model (extended abstract model for minimizing state variables, and fine-grain abstract model for minimizing Boolean network variables. Since the satisfiability checks are conducted in the abstract models, which can be arbitrarily smaller than the concrete model, these SAT problems are usually much easier to solve.

4.4 Apply Sequential Don't Cares

Previous work in abstraction refinement often divides the set of variables (state variables and BNVs) of the concrete model into two parts: a set of visible variables and a set of invisible variables. Model checking is applied to the abstract model that contains only the elementary transition relations of visible variables. The elementary transition relations of invisible variables, on the other hand, are completely ignored. Since their transition constraints are removed, the invisible variables are treated as inputs during model checking—they can take arbitrary values at all times. This explains why counterexamples may exist in an abstract model but not in the concrete model. The valuations of some of these inputs that are responsible for triggering these counterexamples may not be allowed in the concrete system.

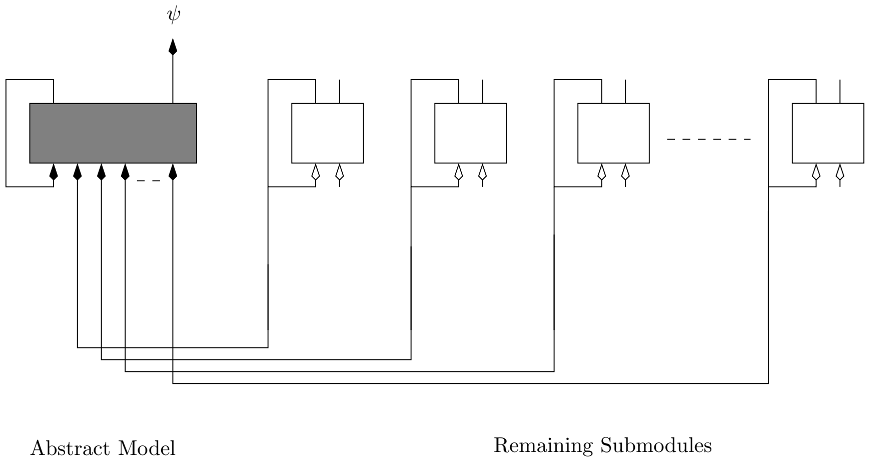


Figure 4.8. Sequential Don't Cares from remaining submodules.

In this section, we show that with some additional analysis of the invisible part of the system, we can further constrain the valuations of these invisible variables. As illustrated in Figure 4.8, the invisible part of the system is further decomposed into a series of submodules, each of which contains a subset of the invisible latches. We then perform an over-approximate reachability analysis of the set of submodules. Approximate reachable states of the invisible part can be computed by first analyzing each submodule in isolation, assuming that the other submodules are in any states that have already been estimated to be reachable, and then propagating the result to other submodules to improve the reachability analysis on them. If there are circular connections, the reachability analysis will be iterated Machine-By-Machine (MBM) [CHM+96a, MKSS99] until a fixpoint is reached.

The set of approximate reachable states of the invisible part is an upper bound on the set of exact reachable states. If certain valuations of the invisible variables are not even in the set of approximate reachable states, they will never appear in the original system. Therefore, this set can be used to constrain the behaviors of the invisible variables, or pseudo-primary inputs, of the abstract model. Conceptually, constraints can be added by conjoining the set of approximate reachable states with the transition relation of the abstract model. During symbolic model checking of the abstract model, certain valuations of pseudo-primary input will be disabled.

In the current implementation, we apply the machine decomposition algorithm as suggested by [CHM+96b] to the entire model, and then use the LMBM [MKSS99] approximate state space traversal algorithm to compute the approximate reachable states. We compute this set of approximate reachable states only once before the abstraction refinement loop starts. Inside the abstraction refinement loop, we use the set of approximate reachable states at every iteration to constrain the forward reachability analysis of the abstract models. Specifically, the BDD operation *constrain* [CM90] is used to remove spurious transitions from $\widehat{T}$, by using the approximate reachable states as the *care set*. This often results in a smaller BDD representation than conjoining the care set with $\widehat{T}$.

Constraints on the behavior of the abstract model due to the neighboring submachines prevent some spurious abstract counterexamples from occurring, possibly leading to the decision of a property earlier in the refinement cycle. A more systematic integration of machine decomposition and approximate reachability analysis into the abstraction refinement paradigm is possible. The result will be a multi-way partition refinement process. Partitioning of the model into submachines can be done so that the abstract model is one of the many submachines. In this new context, refinement will be considered as merging the abstract model with some other submachines. We leave this as an interesting future work.

4.5 Implementation and Experiments

The Grab algorithm and two competing refinement algorithms have been implemented in VIS-2.0 [B+96, VIS]. In the implementation, CUDD was used for BDD related computations and Chaff [MMZ+01] was used as the back-end SAT solver. Our experiments were conducted on 26 hardware verification test cases, coming from both industry and the VIS verification benchmarks [VVB]. The D-designs were kindly provided by the authors of [CCK+02]. All the experiments in this section were run under Linux on an IBM IntelliStation with a 1.7 GHz Intel Pentium 4 CPU and 2 GB of RAM.

Comparisons of Refinement Algorithms

We first compare two variants of the Grab algorithm against the default invariant checking algorithm in VIS (CI), Bounded Model Checking (BMC), the SepSet algorithm [CGKS02], a variant of SepSet called SepSet+, and the conflict analysis algorithm of [CCK+02]. The results are presented in Table 4.1. The CI experiments consist of forward reachability analysis with early termination. For BMC, only the times for failing properties are reported. (BMC in VIS checks for inductive invariants, but none of these invariant properties can be proved by induction.) The variant of Grab denoted by Grab– does not perform refinement minimization. The variant SepSet+ differs from SepSet because it minimizes the number of variables in the separation set, instead of the size of the separation tree. In this section, we focus on comparing the performance of the various refinement algorithms only. For the purpose of this controlled experiment, the same coarse-grain abstraction and concretization test are used for all abstraction and refinement methods.

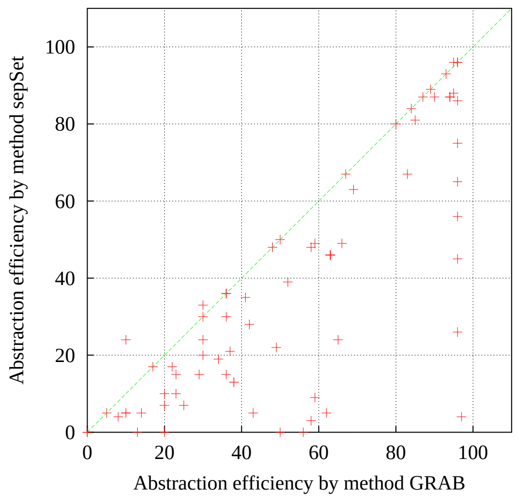
Each model checking run was limited to 8 hours. Dynamic variable reordering was enabled (with method *sift*) for all BDD operations. In Table 4.1, the first column lists the names of the test cases. The second column lists the number of binary state variables in the cone of influence (COI) of the property. The third column shows the length of the counterexample, or of the last ACE encountered by Grab if the property holds (indicated by a T). CPU times are in seconds and are all-inclusive. For each of the abstraction refinement methods compared, **ite** is the number of refinement iterations; **reg** is the number of state variables in the proof or refutation. If an experiment ran out of time, the number of iterations performed up to that point and the number of state variables in the last abstract model are given in parentheses. For Grab we also report **sat**, the time spent in the SAT solver during ACE concretization. Note that in Grab **ite** can be larger than **reg** because of refinement minimization.

Both variants of the Grab algorithm significantly outperform CI, SepSet, and CA in terms of CPU time. BMC has the best times for several failing properties, but is slow for the hardest problems and fails for the passing properties. Note that the last two properties cannot be proved or refuted by any method. Regarding the size of the BDDs used through verification, Grab is much more efficient than CI; SepSet and CA have even fewer BDD nodes, because they use the SAT solver (instead of BDDs) to compute the refinement; unlike Grab, they do not need backward reachability analysis. BMC uses no BDDs.

Table 4.2 compares the final abstractions of Grab and CA. In the table, g denotes the cardinality of the final set of state variables produced by Grab, while c denotes the cardinality of the final set of state variables produced by CA. The first three columns are repeated from Table 4.1. Table 4.2 shows that in general there is very good correlation between the final abstractions produced by CA and Grab. In the 22 experiments that both methods completed, Grab and CA produced the same final abstraction in three cases. In another 10 cases, the abstraction produced by Grab is strictly better than the one of CA. Conversely, in two cases, CA produces an abstraction that is strictly better than the one of Grab. These differences are in part a consequence of applying refinement minimization once every outer iteration in Grab, instead of once for every single counterexample in CA. The other sources of difference are the order in which variables are selected for refinement (this is what happens in D24-p2) and the order in which they are considered by the greedy minimization procedure.

Although we exercised diligence in implementing the algorithms of [CGKS02, CCK$^+$02], there remain differences between the originals and the rewritings. For instance, we used the coarse grain approach when comparing various refinement methods. This is not the case of the original methods of [CCK$^+$02], and will in some cases impede the search for a good abstraction. However, in this set of controlled experiments, the drawback is shared by all methods we implemented, and therefore should not have a major impact on the comparison we present.

Further evidence for the importance of global guidance, i.e., SOR guided vs. single counterexample guided, is provided by an analysis of abstraction efficiency for 80 medium size invariant checking problems from the VIS Verification Benchmarks [VVB]. Each test case in this experiment has a passing property and a non-trivial abstract model (it requires at least one refinement iteration). The abstraction efficiency is 0 (100%) if the final model contains all (no) state variables. Figure 4.9 shows scatter plots of the abstraction efficiency of SepSet, CA, and Grab. Note that each point below the diagonal represents a win for Grab. SepSet+ behaves like SepSet. Scatter plots for the other pairs of methods (not shown) show no clear winner.



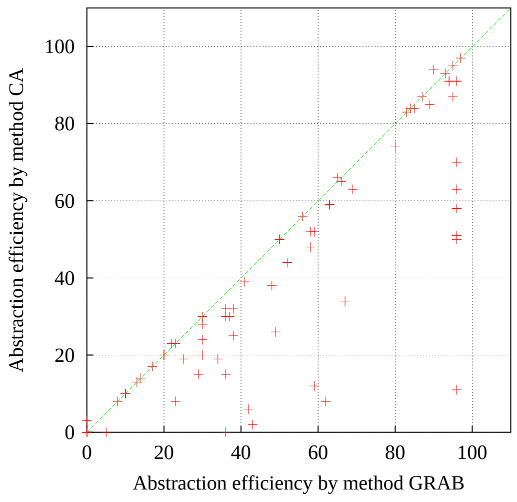


Figure 4.9. Comparing the abstraction efficiency of different refinement algorithms: (1) Grab vs. SepSet; (2) Grab vs. CA.

Refinement minimization, though essential for good performance of CA, does not always improve CPU time when applied to the proposed refinement scheme. The time spent checking the variables for redundancy and the additional iterations are not always offset by the reduction in the size of the abstraction. Nonetheless, as one progresses toward larger models, refinement minimization adds to the robustness of the method.

Experiments with Fine-Grain Abstraction

Experiments were also conducted to test the effectiveness of fine-grain abstraction and the use of sequential Don't Cares. In the experiments, we set the BDD size threshold to 1000 for frontier partitioning. Therefore, every time the BDD size of the transition function went beyond this threshold, a Boolean network variable was inserted in the combinational logic cones.

The first four columns of Table 4.3 repeat the statistics of the test cases: the first column shows the names of the designs; the second and third columns give the numbers of binary state variables and logic gates in the cone of influence, respectively. The forth column indicates whether the properties are true (T) or false (F). If the properties are false, the lengths of the shortest counterexamples are given. The following six columns compare the performance of three different implementations: Grab uses the coarse-grain abstraction, +FineGrain uses the fine-grain abstraction method, and +ARDC uses fine-grain abstraction plus the use of sequential Don't Cares. The underlying algorithm for picking refinement variables is the same game-based strategy for the three methods. For each method, the CPU time in seconds and the number of state variables in the final abstract model are shown.

The fine-grain abstraction approach shows a significant performance improvement over Grab. First, it is able to finish the two largest test cases that cannot be verified by Grab. Careful analysis of *IU-p1* and *IU-p1*, two problems from the instruction unit of the PicoJava microprocessor, shows that some of their registers have extremely large fan-in combinational logic cones. Without fine-grain abstraction, abstract models with less than 10 registers would have been too complex for the model checker to deal with. For the other test cases that both methods managed to finish, +FineGrain is significantly faster than Grab. In fact, the total CPU time required to finish the 24 remaining test cases is 12,207 seconds for Grab, and 7,562 seconds for +FineGrain.

With the use of sequential Don't Cares, the performance of +FineGrain is further improved. +ARDC is significantly faster than both +FineGrain and Grab on more than half of the 26 test cases, and is also comparable for the remaining ones. The total CPU time required to finish all the 26 test cases is 10,724 seconds for +FineGrain and 8,130 seconds for +ARDC; this is an average of 25% speed-up.

Figure 4.10 shows the allocation of CPU time among the different phases in abstraction refinement. The data were extracted with +ARDC; therefore, they correspond to the last column in Table 4.3. The four figures at the left-hand side give in percentage the CPU time spent on reachability analysis, on computing the SORs, on the multi-thread concretization test, and on computing the refinement with Grab, respectively. The four figures at the right-hand side give the corresponding CPU time in seconds. In each figure, the 26 test cases are listed on the x-axis in the same order as they appear in Table 4.3 (1 represents D24-p1, 26 represents IU-p2). Note that other things also consume sometimes non-negligible part of the CPU time, such as incrementally building the BDD partitions, the creation and deletion of abstract FSM, etc.

Figure 4.10 demonstrates that the forward reachability computation and computing refinement with Grab have consumed most of the CPU time. The backward reachability computation for building the SORs, on the other hand, often takes significantly less time than its forward counter-part, even though it collects all the shortest counterexamples. This is due to the application of forward onion rings as care sets in the corresponding pre-image computations. Furthermore, the actual run time of the concretization test is often small (as shown by the "in-seconds" figure), even though it takes a significant amount in percentage from the total CPU time (as shown by the "in-percentage" figure). On this particular set of test cases, multi-thread concretization test is never the performance bottleneck—on the harder problems, test cases 19-26, its overhead becomes negligible.

The performance of the forward reachability computation is limited by the capacity of the state-of-the-art BDD based symbolic techniques. (However, there do exist other examples on which BMC is extremely time-consuming; for them, the SAT multi-thread concretization test may take a significant amount of CUP time.) As the abstract model gets larger, BDD based computations become more and more expensive. The size the abstract model also affects the overhead of the Grab refinement algorithm—the size of the BDDs for representing the SORs become larger as the model gets more complex. In addition, a larger abstract model often has more invisible variables in its local support, which means that more CPU time needs to be spent on scoring them. In general, this trend is true for almost all abstraction refinement methods.

4.6 Further Discussion

The GRAB refinement algorithm differs from [CGJ$^+$00, CGKS02] and other single counterexample guided algorithms [CCK$^+$02, BGG02] in that: (1) it handles all shortest abstract counterexamples rather than a single counterexample; (2) at each abstract counterexample level, a set of abstract states, instead of just one abstract state, is used to constrain the unrolled concrete model at each time step in concretization test; (3) the refinement is based on the systematic analysis of all the spurious counterexamples in the SORs with a game-based approach.

Since the GRAB refinement variable selection method operates solely on the abstract model and its local support variables, it is more scalable than those methods that involve symbolic computations in the concrete model. To our knowledge, the authors of [CGKS02, CGKS02] also had some preliminary experiments with multiple counterexamples and translation of multiple counterexamples to the SAT problem for invalidation, although the work has not been published.

The refinement algorithm in [GKMH$^+$03] also relies on analyzing multiple counterexamples. In their approach, multiple abstract error traces are represented by a data structure called the multi-valued counterexample. However, their multi-valued counterexample do not guarantee to capture all the shortest ones, making it incapable of catching concretizable counterexamples at the earliest possible refinement step. Furthermore, their variable selection algorithm is based on the classification of invisible variables into *strong 0/1 signals* and *conditional 0/1 signals*. We have shown that *strong 0/1 signals* in particular are rare cases in practice. As a result, their refinement is often less accurate than GRAB.

In [MH04], Mang and Ho proposed a refinement algorithm based on controllability and cooperativeness analysis. Their cooperativeness analysis extracts a small subset of candidate input signals by applying a 3-value simulation engine [WHL$^+$01] to simulate the abstract counterexamples and then ranking all the inputs (i.e., invisible state variables and BNVs) according to various criteria. Their controllability analysis is independent of any particular counterexample; it is applied to a subset of input signals by scoring them according to a game-theoretic formula derived from the SORs. These two proposed analysis are then carefully integrated together to better refine the abstract model. Their controllability analysis is an improvement of the GRAB algorithm. Their experimental results showed a significant improvement over both GRAB and the RFN method in [WHL$^+$01].

The proof-based abstraction refinement methods in [MA03, GGW$^+$03b, LWS03, LS04, LWS05, ZPH04, ZPHS05] also handle implicitly all the counterexamples of a finite length. These methods differ from ours in that their refinement variable selection algorithms are all SAT based, i.e., relying on the SAT solver's capability to produce succinct unsatisfiability proofs. In contrast, our core refinement variable selection algorithm is pure BDD based, even though we use SAT as well in predicting refinement direction and in the concretization test. We note that a small unsatisfiability proof, i.e., the one with a small subset of Boolean variables or clauses, does not automatically give a small refinement set [LS04, GGA05].

Both proof-based and counterexample based methods have their own advantages and disadvantages. A detailed experimental comparison of GRAB with a proof-based refinement algorithm can be found in our recent paper [LWS05], showing that these two kinds of methods complement each other on the various test cases. Amla *et al.* [ADK$^+$05] also published results of their experimental evaluation of the various SAT and interpolation based abstraction methods. There is also a trend of combining counterexample based methods and proof-based methods in abstraction refinement [AM04].

Table 4.1. Comparing invariant checking algorithms.

circuit	COI	cex	CI	BMC	SepSet			SepSet+			CA			Grab–			Grab			
	regs	len	time	time	time	ite	reg	time	ite	reg	time	ite	reg	time	ite	reg	time	ite	reg	sat
D1-p1	101	9	45	**1**	48	11	38	74	9	21	98	15	26	9	18	21	9	18	21	1
D23-p1	85	5	7	**1**	8	2	21	17	2	21	11	1	21	29	5	23	20	5	21	1
D24-p1	147	9	>8h	27	**1**	0	4	**1**	0	4	**1**	0	4	**1**	0	4	**1**	0	4	1
D24-p2	147	T(9)	>8h	-	6982	2	8	7087	2	8	2153	34	77	**1**	3	8	3	3	8	1
D1-p2	101	13	1947	**2**	1774	27	45	962	23	38	423	28	44	27	25	28	51	37	23	1
D22-p1	140	10	58	**2**	615	3	133	1005	5	135	728	3	133	537	3	134	720	3	132	1
D1-p3	101	15	1157	**3**	623	22	36	446	19	32	636	25	39	39	23	27	56	34	25	2
D24-p5	147	T(2)	>8h	-	310	4	7	944	3	7	36	4	11	4	4	6	**3**	4	5	1
D12-p1	48	16	**5**	**5**	106	22	32	124	20	35	64	12	28	6	17	24	14	25	23	1
D2-p1	94	14	166	**6**	147	5	48	280	5	48	239	7	50	124	5	53	180	10	48	1
D16-p1	531	8	837	**10**	>8h	(35)	(41)	>8h	(36)	(41)	890	3	16	282	9	14	92	9	14	5
D24-p3	147	T(3)	>8h	-	>8h	(1)	(4)	>8h	(2)	(4)	62	5	11	37	6	8	**20**	6	8	1
D5-p1	319	31	513	58	43	4	13	148	4	13	82	3	13	**26**	9	18	31	9	18	12
D24-p4	147	T(3)	>8h	-	545	4	7	711	4	7	70	5	11	**29**	6	8	43	6	8	1
D21-p1	92	26	**63**	3787	3790	39	88	2402	36	85	1922	28	79	1010	11	76	2817	26	66	3
B-p1	124	T(18)	7453	-	4359	14	27	4360	14	27	284	5	19	**88**	19	24	173	19	18	6
B-p2	124	17	12988	150	110	2	7	115	2	7	108	2	7	220	8	13	**93**	8	7	11
M0-p1	221	T(3)	>8h	-	>8h	(0)	(3)	>8h	(0)	(3)	1182	9	19	219	14	17	**136**	14	16	20
B-p3	124	T(4)	12466	-	>8h	(74)	(80)	>8h	(95)	(101)	167	6	42	**144**	35	52	223	35	43	2
D21-p2	92	28	**152**	10515	4146	36	85	2930	37	86	2962	30	83	2079	19	89	4635	41	70	6
B-p4	124	T(5)	7089	-	9255	49	67	10360	54	68	228	8	43	**157**	36	54	393	47	42	3
B-p0	124	T(17)	7467	-	>8h	(54)	(61)	>8h	(39)	(47)	2644	7	49	**330**	28	29	1256	32	24	10
rcu-p1	2453	T(2)	>8h	-	375	7	11	375	7	11	>8h	5	(9)	197	9	12	**195**	9	10	0
D4-p2	230	T(19)	765	-	>8h	(5)	(16)	>8h	(10)	(22)	>8h	(3)	(171)	**682**	38	69	1103	69	38	6
IU-p1	4494	-	>8h	>8h	>8h	-	-	>8h	-	-	>8h	-	-	>8h	-	-	>8h	-	-	-
IU-p1	4494	-	>8h	>8h	>8h	-	-	>8h	-	-	>8h	-	-	>8h	-	-	>8h	-	-	-

Table 4.2. Correlation between the final proofs of GRAB and CA.

circuit	COI	cex	$\|g\|$	$\|c\|$	$\|g \cup c\|$	$\|g \cap c\|$	$\|g \setminus c\|$	$\|c \setminus g\|$	subset?
D1-p1	101	9	21	26	27	20	1	6	no
D23-p1	85	5	21	21	21	21	0	0	yes
D24-p1	147	9	4	4	4	4	0	0	yes
D24-p2	147	T(9)	8	77	77	8	0	69	strict
D1-p2	101	13	23	44	44	23	0	21	strict
D22-p1	140	10	132	133	133	132	0	1	strict
D1-p3	101	15	25	39	40	24	1	15	no
D24-p5	147	T(2)	5	11	11	5	0	6	strict
D12-p1	48	16	23	28	28	23	0	5	strict
D2-p1	94	14	48	50	50	48	0	2	strict
D16-p1	531	8	14	16	16	14	0	2	strict
D24-p3	147	T(3)	8	11	13	6	2	5	no
D5-p1	319	31	18	13	18	13	5	0	strict
D24-p4	147	T(3)	8	11	13	6	2	5	no
D21-p1	92	26	66	79	81	64	2	15	no
B-p1	124	T(18)	18	19	19	18	0	1	strict
B-p2	124	17	7	7	7	7	0	0	yes
M0-p1	221	T(3)	16	19	21	14	2	5	no
B-p3	124	T(4)	43	42	43	42	1	0	strict
D21-p2	92	28	70	83	85	68	2	15	no
B-p4	124	T(5)	42	43	43	42	0	1	strict
B-p0	124	T(17)	24	49	49	24	0	25	strict
rcu-p1	2453	T(3)	10	(9)	?	?	?	?	strict
D4-p2	230	T(19)	38	(171)	?	?	?	?	?
IU-p1	4494	?	?	?	?	?	?	?	?
IU-p2	4494	?	?	?	?	?	?	?	?

Table 4.3. Comparing Grab, +FineGrain, and +ARDC.

circuit	COI regs	COI gates	cex len	Grab time	Grab regs	+FineGrain time	+FineGrain regs	+ARDC time	+ARDC regs
D24-p1	147	8 k	9	1	4	1	4	1	4
D24-p2	147	8 k	T	3	8	3	8	3	8
D24-p3	147	2 k	T	20	8	4	6	**2**	5
D24-p4	147	8 k	T	43	8	4	6	**2**	5
D24-p5	147	8 k	T	3	5	4	6	**2**	5
D12-p1	48	2 k	16	**14**	23	24	23	19	24
D23-p1	85	3 k	5	20	21	**3**	21	14	21
D5-p1	319	25 k	31	**31**	18	42	13	32	13
D1-p1	101	5 k	9	**9**	21	12	20	14	20
D1-p2	101	5 k	13	51	23	**27**	23	29	23
D1-p3	101	5 k	15	56	25	**32**	23	33	23
D16-p1	531	34 k	8	92	14	25	14	**21**	14
D2-p1	94	18 k	14	180	48	108	49	**59**	48
M0-p1	221	29 k	T	**136**	16	204	13	942	13
rcu-p1	2453	38 k	T	195	10	**188**	10	216	10
B-p0	124	2 k	T	**1256**	24	1507	24	1484	24
B-p1	124	2 k	T	173	18	189	19	**159**	18
B-p2	124	2 k	17	93	7	95	7	**90**	7
B-p3	124	2 k	T	223	43	76	43	**62**	43
B-p4	124	2 k	T	393	42	**101**	43	108	42
D22-p1	140	7 k	10	720	132	242	132	**191**	132
D4-p2	230	8 k	T	1103	38	204	38	**195**	38
D21-p1	92	14 k	26	2817	66	2725	70	**622**	67
D21-p2	92	14 k	28	4635	70	1748	75	**868**	67
IU-p1	4494	154 k	T	>8h	-	**2226**	12	2263	12
IU-p2	4494	154 k	T	>8h	-	930	14	**699**	12

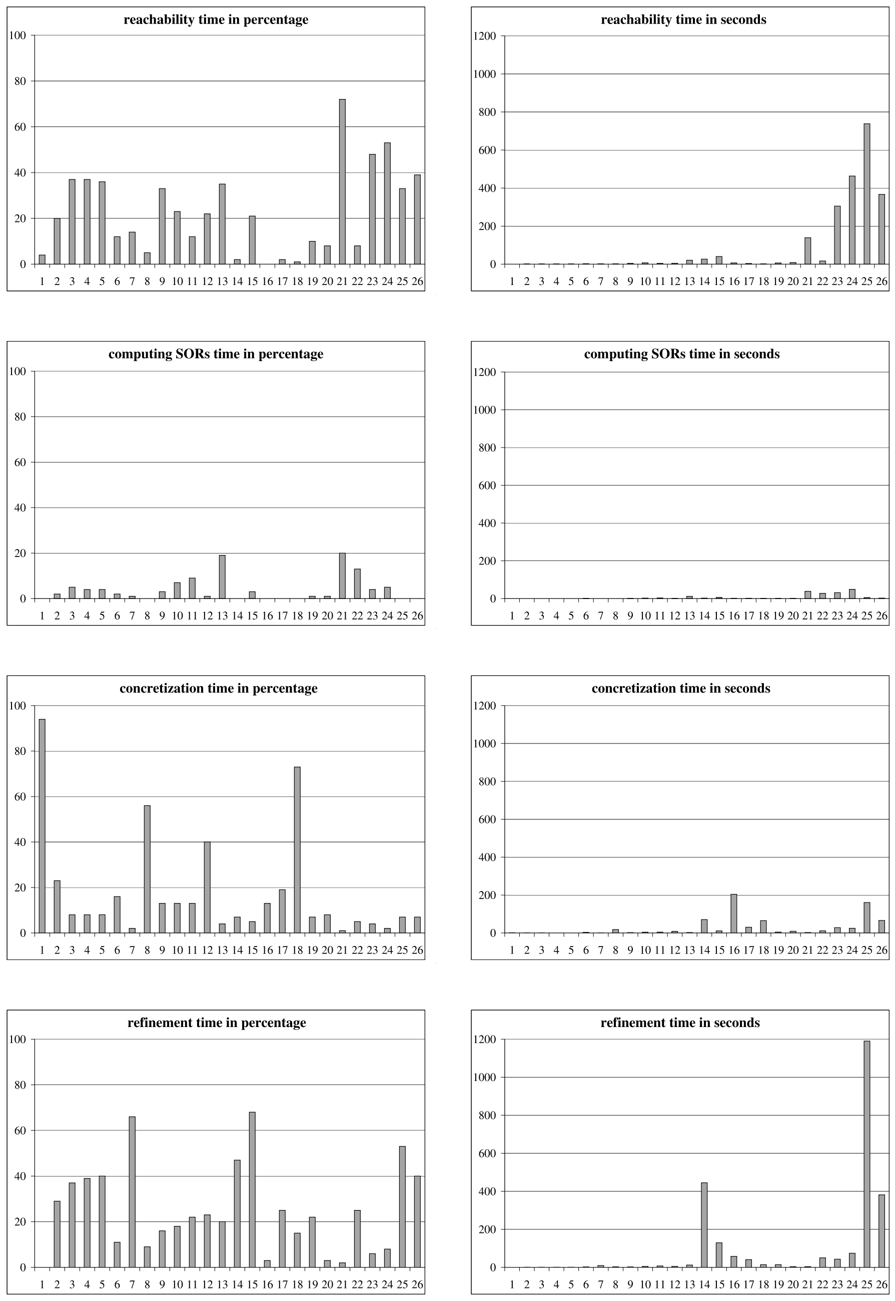


Figure 4.10. The CPU time distribution among the different phases of abstraction refinement: forward reachability analysis, computing SORs, multi-thread concretization test, and computing the refinement set with Grab.

Chapter 5

COMPOSITIONAL SCC ANALYSIS

In abstraction refinement, the given property needs to be checked repeatedly in the abstract model, while the model is gradually refined. Information learned from previous abstraction levels can be carried on to help the verification at the current level. The major problem, however, is to identify the information that needs to be carried on and apply it to improve the computation efficiency.

In this chapter, we propose a compositional SCC analysis framework for LTL and fair-CTL model checking. In this framework, the *SCC partition* of the state space from the previous abstract model is carried on to the next level. When we check the current abstract model, previous SCC partitions can be used as the starting point for computing the new SCC partition.

We also exploit the reduction in automaton strength during abstraction refinement to speed up the verification procedure. The *strength* of a Büchi automaton affects the complexity of checking the emptiness of its language. For *weak* or *terminal* automaton [KV98, BRS99], specialized fair cycle detection algorithms often outperform the general one. We have found that composing two automata together may reduce, but may never increase the automaton strength. Therefore, even if the abstract model or property automaton is initially strong, its strength can be reduced through the successive refinements. In this chapter, we propose a new algorithm that dynamically selects model checking procedures to suit the current strength of the individual SCCs, and therefore takes maximal advantage of their weakness.

5.1 Language Emptiness

Checking language emptiness of a Büchi automaton is a core procedure in LTL [LP85, VW86] and fair-CTL model checking [McM94], and in approaches to verification based on language-containment [Kur94]. The cycle detection algorithms commonly used in symbolic model checkers fall into two categories: one is based on the computation of an *SCC hull* [EL86, HTKB92, TBK95, KPR98, SRB02], and the other is based on SCC enumeration [XB00, BGS00, GPP03]. Although some SCC enumeration algorithms [BGS00, GPP03] have better worst-case complexity bounds than the SCC hull algorithms—$O(\eta \log \eta)$ or $O(\eta)$ versus $O(\eta^2)$, where η is the number of states of the system—the comparative study of [RBS00] shows that the worst-case theoretical advantage seldom translates into shorter CPU times. In many practical cases, applying any of these symbolic algorithms directly to the entire system to check language emptiness remains prohibitively expensive.

In abstraction refinement, language emptiness is checked repeatedly in the abstract model while the model is gradually refined. It is natural to ask whether information learned from previous abstract models can be carried on to the current level to improve the computation efficiency. The major problem is to identify the kind of information that can be carried on, and find ways to apply it to speed up the verification.

Although model checking applied to the abstract models may have conservative result, the *SCC partition* of the state space computed in the abstract model still provides valuable information for language emptiness checking of the concrete system (or a more refined abstract model). Given a model $\mathcal{A}$ and an over-approximation $\widehat{\mathcal{A}}$, every SCC in $\widehat{\mathcal{A}}$ consists of one or more complete SCCs of $\mathcal{A}$. In other words, an SCC in the concrete system must be either included in or excluded completely from an SCC in the abstract model. Let Π be the set of SCCs in $\mathcal{A}$; then Π is a *refinement* of the set of SCCs in $\widehat{\mathcal{A}}$. Similarly, an SCC C in $\mathcal{A}$ is a refinement of another SCC C' in $\mathcal{A}'$ if $C \subseteq C'$. If an SCC in the abstract model does not contain a fair cycle, none of its refinements will. Therefore, it is possible to inspect the fair SCCs in $\widehat{\mathcal{A}}$ first, and then refine them individually to compute the fair SCCs in $\mathcal{A}$.

We will present a compositional SCC analysis framework for language emptiness checking called DnC (for Divide and Compose), which is based on the enumeration and the successive refinement of SCCs in a set of over-approximated abstract models. By combining appropriate cycle-detection algorithms (SCC hull or SCC enumeration) into the general framework, we create a hybrid algorithm that shares the good theoretical characteristics of SCC enumeration algorithms, while outperforming the most popular SCC-hull algorithms, including the one by Emerson and Lei [EL86].

The procedure starts by performing SCC enumeration on the most primitive abstract model, which produces the initial SCC partition of the set of states. The partition is then made more accurate on a refined abstract model—one that is usually the composition of the current abstract model and a previously omitted submodule. SCCs that do not contain fair cycles are not considered in any more refined model. If no fair SCC exists in an abstract model, the language of the concrete model is proved empty. If fair SCCs exist, the abstract counterexamples can be checked against the concrete model in a way similar to the SAT based concretization test; the existence of a real counterexample means the language is not empty. When the concrete model is reached, the procedure terminates with the set of fair SCCs of the concrete model. Since each concrete SCC is contained in an SCC of the abstract model, SCC analysis at previous abstraction levels can drastically limit the potential space that contains a fair cycle.

In language emptiness checking, the model is regarded as a generalized Büchi automaton. The *strength* of a Büchi automaton is an important factor for the complexity of checking the emptiness of its language. As shown in previous work [KV98, BRS99], when an over-approximation of $\mathcal{A}$ is known to be *terminal* or *weak*, specialized algorithms exists for checking the emptiness of the language in $\mathcal{A}$. These specialized algorithms usually outperform the general language emptiness algorithm. However, the previous classification of strong, weak, and terminal was applied to the entire Büchi automaton. This can be inefficient, because a Büchi automaton with a strong SCC and several weak ones would be classified as strong. In this chapter, the definition of strength is extended to each individual SCC so that the appropriate model checking procedure can be deployed at a finer granularity. In addition, it is shown that the strength of an SCC never increases during the composition, but may actually decrease as submodules are composed. After the composition, a strong SCC can break into several weak SCCs, but a weak one cannot generate strong SCCs. DnC analyzes SCCs as they are computed to take maximal advantage of their weakness.

The DnC algorithm achieves favorable worst-case complexity bound, i.e., $O(\eta)$ or $O(\eta \log \eta)$, depending on what underlying SCC enumeration algorithm is used. This is valid even when it adds one submodule at the time until the concrete system is reached. In practice, however, the effort spent on the abstract systems can be justified only if it does not incur too much overhead. As the abstract system becomes more and more concrete through composition, SCC enumeration in the abstract model may become expensive. In such cases, the algorithm jumps directly to the concrete system, with all the useful information gathered from the abstract systems. Based on the SCC quotient graph of the abstract model, it *disjunctively* decomposes the concrete state space into subspaces. Each subspace induces a Büchi automaton that is an underapproximation of the concrete model; therefore, it accepts a subset of the original language. The decomposition is *exact* in the sense that the union of these language subsets is the original language. Therefore, language emptiness can be checked in each of these subautomata in isolation. By focusing on one subspace at a time, we can mitigate the BDD explosion during the most expensive part of the computation—detecting fair cycles in the concrete system.

To further speed up the search for fair cycles, we propose a new guided search algorithm for the traversal of the exact state space. Early termination is promoted by examining first the promising areas where fair cycles may reside, and by stopping the cycle-detection procedure as soon as a fair cycle is found. In this targeted search, the approximate distance from the initial states is used as the guidance.

5.2 SCC Partition Refinement

We start with the definition of over-approximation of a generalized Büchi automaton, followed by theorems that provide the foundation for the SCC analysis algorithm. Automaton $\mathcal{A}'$ is an over-approximation of $\mathcal{A}$, if $S = S'$, $S_0 \subseteq S'_0$, $T \subseteq T'$, $\mathcal{F} \supseteq \mathcal{F}'$, and $\Lambda = \Lambda'$. An over-approximation always simulates the original automaton, which is denoted by $\mathcal{A} \preceq \mathcal{A}'$. Given a set of automata defined in the same state space and with an alphabet originated from the same set of atomic propositions, the simulation relation $\preceq$ induces a partial order.

Theorem 5.1 (Compositional refinement) *Let $\mathcal{A}, \mathcal{A}_1, \ldots, \mathcal{A}_n$ be a set of labeled generalized Büchi automata such that $\mathcal{A} \preceq \mathcal{A}_i$ for $1 \leq i \leq n$. Then, the set of SCCs $\Pi(\mathcal{A})$ is a refinement of*

$$\Theta = \{C_1 \cap \cdots \cap C_n \mid C_i \in \pi(\mathcal{A}_i)\} \setminus \emptyset \ .$$

Proof: *Every state in an SCC $C \in \Pi(\mathcal{A})$ is reachable from all other states in C. An over-approximation $\mathcal{A}_i$ preserves all the transitions of $\mathcal{A}$, which means that in $\mathcal{A}_i$, every state in C remains reachable from the other states in C. Therefore, for $1 \leq i \leq n$, C is contained in an SCC of $\mathcal{A}_i$; hence it is contained in their intersection, which is an element of Θ. Since the union of all SCCs of $\mathcal{A}$ equals S and distinct elements of Θ are disjoint, Θ is a partition of S, and $\Pi(\mathcal{A})$ is a refinement of it.*

In particular, $\Pi(\mathcal{A})$ is a refinement of an SCC partition of any over-approximation of $\mathcal{A}$; thus, an SCC of $\mathcal{A}'$ is an SCC-closed set of $\mathcal{A}$. This theorem allows one to gradually refine the set of SCCs in the abstract models until $\Pi(\mathcal{A})$ is computed. It can often be decided early that an SCC-closed set does not contain an accepting cycle. For language emptiness checking, these non-fair SCC-closed sets are ignored. By working on only "suspect" SCCs, one can trim the state space with cheap computations in the simplified models.

Observation 5.2 *Let C be an SCC-closed set of $\mathcal{A}$. If $C \cap F_i = \emptyset$ for any $F_i \in \mathcal{F}$, then C has no states in common with any accepting cycle.*

Recall that we have defined the concrete model $\mathcal{A}$ as the synchronous (or parallel) composition of a set of submodules. Composing a subset of these submodules gives us an over-approximated abstract model $\mathcal{A}'$. For instance, let $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$, then both $\mathcal{A}_1$ and $\mathcal{A}_2$ can be considered as over-approximations of $\mathcal{A}$. It follows that an SCC in either $\mathcal{A}_1$ or $\mathcal{A}_2$ is an SCC-closed set in $\mathcal{A}$.

Definition 5.3 *The* strength *of a fair SCC C is defined as follows:*

- *C is* weak *if all cycles contained within it are accepting.*
- *C is* terminal *if it is weak and, for every state $s \in C$ and every proposition $a \in A$, the successor s' of s is in a terminal SCC. Terminality implies acceptance of all runs reaching C.*
- *C is* strong *if it is not weak.*

Strength is defined only for fair SCCs. The strength of an SCC-closed set containing at least one accepting SCC is the maximum strength of its fair SCCs. The strength of an automaton is also the maximum strength of its fair SCCs. Our definition of weakness is more relaxed than that of [KV98, BRS99]. Previously, all the states of a weak SCC must belong to all fair sets $F_i \in \mathcal{F}$, and a terminal SCC must be maximal (i.e., no successor SCCs). Our new definition is more relaxed, while still allowing the use of faster symbolic model checking algorithms in the same way.

Lemma 5.4 *Given a labeled generalized Büchi automaton $\mathcal{A}$, if C is a weak (terminal) SCC of an over-approximation $\mathcal{A}'$ of $\mathcal{A}$, then it contains no reachable fair cycle in $\mathcal{A}$ if and only if $\mathsf{EF}\,\mathsf{EG}\,C \cap S_0 \neq \emptyset$ ($\mathsf{EF}\,C \cap S_0 \neq \emptyset$) holds in $\mathcal{A}$.*

$\mathsf{EF}\,C$ is the subset of states in S that can reach the states in C, while $\mathsf{EG}\,C$ is the subset of states in C that lead to a cycle lying in C. Assume that C is a terminal SCC in $\mathcal{A}'$, and a state $s \in C$ is reachable from the initial states in $\mathcal{A}$. Since for every proposition $a \in A$, the successor s' of s in some terminal SCCs, all runs reaching s in $\mathcal{A}$ remain inside terminal SCCs afterward. Due to the finiteness of the automaton, these runs form cycles. Since terminal SCCs are also weak, all these runs are accepting. Therefore, the language is not empty if and only if $\mathsf{EF}\,C \cap S_0 \neq \emptyset$. If C is a weak SCC in $\mathcal{A}'$, and a state $s \in C$ is reachable from S_0 in $\mathcal{A}$ and at the same time $s \in \mathsf{EG}\,C$, there exists a run through s that forms a cycle in C. Since all cycles in the weak SCC C are accepting, the language is not empty if and only if $\mathsf{EF}\,\mathsf{EG}\,C \cap S_0 \neq \emptyset$. Note that for a *strong* SCC, one must resort to the computation of $\mathsf{EG}_{\mathsf{fair}}\,\mathsf{true}$.

Theorem 5.5 (Strength reduction) *Let $\mathcal{A}$ and $\mathcal{A}'$ be Büchi automata such that $\mathcal{A}$ and $\mathcal{A}'$ are complete and $\mathcal{A} \preceq \mathcal{A}'$. If C is a weak (terminal) SCC-closed set of $\mathcal{A}'$, then C is a weak (terminal) SCC-closed set of $\mathcal{A}$.*

Proof: *We prove this by contradiction. Assume that C is a weak set of $\mathcal{A}'$, but is a strong set of $\mathcal{A}$. Then, at least one cycle in C is not* *accepting in $\mathcal{A}$. As an over-approximation, $\mathcal{A}'$ preserves all paths of $\mathcal{A}$, including this non-accepting cycle, which makes C a strong set of $\mathcal{A}'$ too. However, this contradicts the assumption that C is weak in $\mathcal{A}'$. Therefore, C cannot be a strong set of $\mathcal{A}$. A similar argument applies to the terminal case.*

In other words, the strength of an SCC-closed set never increases as a result of composition. In fact, the strength may actually reduce in going from $\mathcal{A}'$ to $\mathcal{A}$. For example, a strong SCC may be refined into one or more SCC, none of which is strong; a weak SCC may be refined into one or more SCCs, none of which is weak. This strength reduction theorem allows us to use special model checking algorithms inside the abstraction refinement loop as soon as a strong SCC-closed set becomes weak or terminal.

Deciding the strength of an SCC-closed set strictly according to its definition is expensive. In the actual implementation, we can make conservative decisions of the strength of an SCC C as follows:

- C is weak if $C \subseteq F_i$ for every $F_i \in \mathcal{F}$;
- C is terminal if C is weak, and either $(\mathsf{EY}\, C) \setminus C = \emptyset$, or $(\mathsf{EY}\, C) \setminus C \subseteq C_t$ where C_t is a terminal SCC;
- C is strong otherwise.

We use the example in Figure 5.1 to show the impact of composition. The three Büchi automata have one acceptance condition ($\mathcal{F} = \{F_1\}$) and are defined on the same state space. State 01 is labeled $\neg p$; all other states are labeled true implicitly. Double circles indicate that the states satisfy the acceptance condition. In this figure, the parallel composition of the two automata at the top produces the automaton at the bottom. Note that only transitions that are allowed by both parent automata appear in the composed system. Both automata at the top are strong, although their SCC partitions are different. The composed system, however, has a weak SCC, a terminal SCC, and two non-fair SCCs. Its SCC partition is a refinement of both previous partitions.

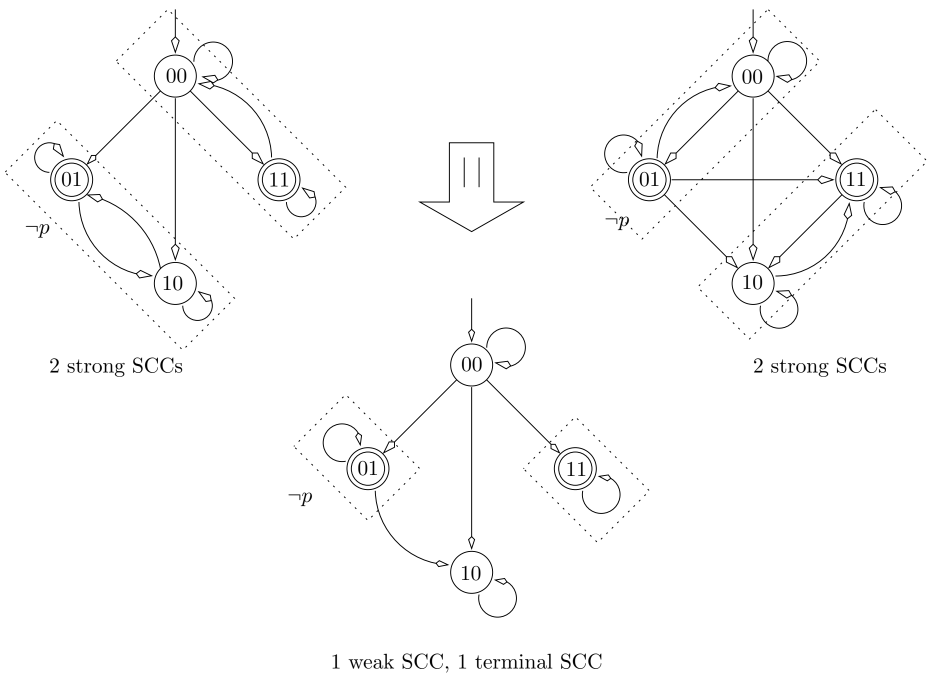


Figure 5.1. Parallel composition of automata and its impact on SCCs.

5.3 The D'n'C Algorithm

Theorems in the previous section motivate the generic SCC analysis algorithm in Figure 5.2. The Divide and Compose (D'n'C) algorithm, whose entry function is GENERIC-REFINEMENT, takes as arguments a Büchi automaton $\mathcal{A}$ and a set L of over-approximated abstract models, which includes $\mathcal{A}$ itself. The relation $\preceq$ among over-approximated models in L is not required to be a total order. The procedure returns true if a fair cycle exists in $\mathcal{A}$, and false otherwise.

The algorithm keeps a set Work of obligations, each consisting of a set of states, the series of abstract models that have been applied to it, and an upper bound on its strength. Initially, the entire state space is in Work, and the algorithm keeps looping until Work is empty or a fair SCC has been found. The loop starts by selecting an element (C, L', s) from Work and a new abstract model $\mathcal{A}'$ from L. If $\mathcal{A}' = \mathcal{A}$, the algorithm may decide to run a standard model checking procedure on the SCC at hand. Otherwise, it decomposes C into accepting SCCs and after

```
type Entry = record
    $C$;   // An SCC-closed set of $\mathcal{A}$
    $L'$;  // Set of abstract models that have been considered
    $s$    // Upper bound on the strength of the SCC
end

GENERIC-REFINEMENT($\mathcal{A}, L$){   // Concrete and abstract models

    var Work: set of Entry;
    Work = $\{(S, \emptyset, \text{strong})\}$;

    while (Work $\neq \emptyset$) {

        Pick an entry $E = (C, L', s)$ from Work;
        Choose $\mathcal{A}' \in L$ such that no $\mathcal{A}'' \in L'$ with $\mathcal{A}'' \preceq \mathcal{A}'$;
        if ($\mathcal{A}' = \mathcal{A}$ or ENDGAME($C, s$)) {
            if (MODEL-CHECK($\mathcal{A}, C, s$))
                return true;
        }
        else {
            Over-approx. reachability computation on $\mathcal{A}'$;
            $\mathcal{C}$ := SCC-DECOMPOSE($C, \mathcal{A}'$);

            if ($\mathcal{C} \neq \emptyset$ and $\mathcal{A}' = \mathcal{A}$)
                return $true$;

            for (all $C \in \mathcal{C}$) {
                $s$ := ANALYZE-STRENGTH($C, \mathcal{A}'$);
                insert $(C, L' \cup \{\mathcal{A}'\}, s)$ in Work;
            }
        }

    }

    return false;
}

MODEL-CHECK($\mathcal{A}, C, s$){   // Automaton, SCC-closed set, strength

    case ($s$) {
        strong:   return $Q_0 \cap \mathsf{EG}_{\mathcal{F}}(C) \neq \emptyset$;
        weak:     return $Q_0 \cap \mathsf{EF}\,\mathsf{EG}(C) \neq \emptyset$;
        terminal: return $Q_0 \cap \mathsf{EF}(C) \neq \emptyset$;
    }

}
```

Figure 5.2. The generic SCC analysis algorithm D‘N’C.

analyzing their strengths, adds them as new Work. At any stage, for any entry (C, L', s) of Work, C is guaranteed to be an SCC-closed set of $\mathcal{A}$, and the sets of states in Work are always disjoint. Termination of the procedure is guaranteed by the finiteness of L and of the set of SCCs of $\mathcal{A}$.

The algorithm uses several subroutines. Subroutine SCC-DECOMPOSE, takes an automaton $\mathcal{A}'$ and a set C, intersects the state space of $\mathcal{A}'$ with C to yield a new automaton $\mathcal{A}''$, and returns the set of accepting SCCs of $\mathcal{A}''$. The subroutine avoids working on any non-fair SCCs, as justified by Observation 5.2. Subroutine ANALYZE-STRENGTH returns the strength of the SCC-closed set. Subroutine MODEL-CHECK returns true if and only if a fair cycle is found using the appropriate model-checking technique for the strength of the given SCC.

The way entries and abstract models are picked is not specified, and neither is it stated when ENDGAME returns true. These functions can depend on factors such as the strength of the entry, the abstract models that have been applied to it, and its order of insertion. In later sections, these functions will be made concrete.

When decomposing an SCC-closed set C, the complement set $\overline{C}$ can be used as the Don't Care conditions to constrain and speed up the computation. This is usually a significantly larger Don't Care set than reachability Don't Cares; therefore, the use of $\overline{C}$ as Don't Cares can lead to a significant improvement in the computation efficiency. The time spent on computing $\overline{C}$ is small because it is from an abstract model where image and pre-image computations are cheaper than in the concrete system. I

The reachable states of the current over-approximation are kept around to discard unreachable SCCs. When the system is refined, the set of reachable states is computed anew, but this computation is limited to the previous reachable states, because the new reachable states are always contained in the previous reachable states as long as the new abstract system is a refinement of the previous one. Therefore, previous reachable states can be used as a care set in computing the new ones. Although reachability analysis is performed multiple times (once for every $\mathcal{A}' \in L$), previous work of [MJH^{+}98] has shown that the use of approximate reachability information as a care set may more than compensate for the overhead.

The proposed algorithm can be extended to include the use of under-approximations as well. Whereas over-approximations can be used to discard the possibility of an accepting cycle, under-approximations can be used to assert its existence. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be under-approximations of $\mathcal{A}$. If $\mathcal{A}_1$ contains an accepting cycle, so does $\mathcal{A}$. Furthermore, if an

SCC C_1 of $\mathcal{A}_1$ and an SCC C_2 of $\mathcal{A}_2$ overlap, then $\mathcal{A}$ contains an SCC $C \supseteq C_1 \cup C_2$. Remember that SCC-enumeration algorithms [XB00, BGS00, GPP03] compute each SCC by starting from a seed, which normally is a single state. Since we know that both C_1 and C_2 are subsets of C, we can use $C_1 \cup C_2$ as the seed to compute C. By doing so, part of the work went into computing C_1 or C_2 can be reused.

Under-approximations of an SCC can also be used as *Don't Cares (DCs)* in BDD based symbolic algorithms to restrict the computation to a state subspace. For instance, in SCC enumeration, as soon as we know that $C_1 \cup C_2$ is a subset of an SCC, we can focus our attention on the state subspace $\overline{C_1 \cup C_2}$. We can modify the transition relation of the model to reflect this shift of attention (with the goal of finding smaller BDD representations for the transition relation and sets of states). To better understand the use of DCs, let us review some background information about the implementation of BDD based image computation.

Image and pre-image computations are the most resource-consuming steps in BDD based model checking. Since their runtime complexity depends on the BDD sizes of the operators involved, it is important to minimize the sizes of the representations of both the transition relation and the argument to the (pre-)image computation—the set of states. The size of a BDD is not directly related to the size of the set it represents. If we need not represent a set exactly, but can instead determine an interval in which it may lie, we can use generalized cofactors [CBM89b, CM90] to find a set within this interval with a small BDD representation.

Often, we are only interested in the results as far as they lie within a *care set* K (or outside a *don't care set* $\overline{K}$). Since the language emptiness problem is only concerned with the set of reachable states R, we can regard R as a care set, and add or delete edges in the state transition graph that emanate from unreachable states. By doing this, the image of a set that is contained within R remains the same. Likewise, the part of the pre-image of a set S that intersects R remains the same, even if unreachable states are introduced into S by adding edges. This use of the states in $\overline{R}$ as don't cares, which is often called the Reachability Don't Cares (RDCs), depends on the fact that no edges from reachable to unreachable states are added.

SCC-closed sets are care sets that are often much smaller than the set of reachable states, and thus can be used to increase the chance of finding small BDDs. We cannot, however, use the approach outlined for the reachable states directly, since there may be edges from an SCC-closed set to other states, as the one from State 4 to State 6 in Figure 5.3.

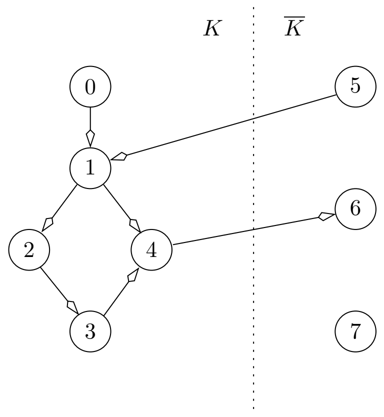


Figure 5.3. An example of using don't cares in the computation of SCCs.

We show here that in order to use arbitrary sets as care sets in image computation, a "safety zone" consisting of the pre-image of the care set needs to be kept; similarly for pre-image computation, a "safe zone" must consist of the image of the care set.

THEOREM 5.6 *Let Q be a set of states and let $T \subseteq Q \times Q$ be a transition relation. Let $K \subseteq Q$ be a care set, $B \subseteq K$ a set of states. Finally, let $T' \subseteq Q \times Q$ be a transition relation and $B' \subseteq Q$ a set of states such that*

$$T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K}), \text{ and}$$
$$B \subseteq B' \subseteq B \cup \overline{\mathsf{EX}_{T'}(K)} \ .$$

Then, $\mathsf{EY}_{T'}(B') \cap K = \mathsf{EY}_T(B) \cap K$.

PROOF: *First, suppose that $q' \in \mathsf{EY}_{T'}(B') \cap K$, and let $q \in B'$ be such that $q' \in \mathsf{EY}_{T'}(\{q\}) \cap K$. Since $q' \in \mathsf{EY}_{T'}(\{q\})$, so $q \in \mathsf{EX}_{T'}(q')$, and because $q' \in K$, we have $q \in \mathsf{EX}_{T'}(K)$. Hence, $q \in B'$ implies $q \in B$, and $q, q' \in K$, which means that $q' \in \mathsf{EY}_T(\{q\}) \cap K$. Finally, $q \in B$ implies $q' \in \mathsf{EY}_T(B) \cap K$.*

Conversely, suppose that $q' \in \mathsf{EY}_T(B) \cap K$, and let $q \in B$ be such that $q' \in \mathsf{EY}_T(\{q\}) \cap K$. Now $q, q' \in K$, and hence $q' \in \mathsf{EY}_{T'}(\{q\}) \cap K$, and since $q \in B'$, $q' \in \mathsf{EY}_{T'}(B') \cap K$.

Hence, we can choose T' and B' within the given intervals so that they have a small representations, and use them instead of T and B. Through symmetry, we can prove the following theorem.

THEOREM 5.7 *Let Q be a set of states and let $T \subseteq Q \times Q$. Let $K \subseteq Q$, $B \subseteq K$, $T' \subseteq Q \times Q$, and $B' \subseteq Q$ be such that*

$$T \cap (K \times K) \subseteq T' \subseteq T \cup (\overline{K} \times Q) \cup (Q \times \overline{K}), \text{ and}$$
$$B \subseteq B' \subseteq B \cup \overline{\mathsf{EY}_{T'}(K)} \ .$$

Then, $\mathsf{EX}_{T'}(B') \cap K = \mathsf{EX}_T(B) \cap K$.

Edges are added to and from states in the set $\overline{K}$ (states outside K), while the safety zone for (pre-)image computation excludes the immediate (successors) predecessors of K. Note that the validity of the aforementioned use of the reachable states as care set follows as a corollary of these two theorems. Figure 5.4 shows a possible choice of T' given the same T and K of Figure 5.3.

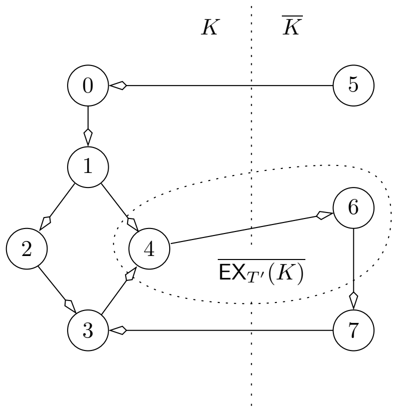


Figure 5.4. Another example of using don't cares in the computation of SCCs.

For that choice of T', it shows, enclosed in the dotted line, the set $\overline{\mathsf{EX}_{T'}(K)}$. If $B = \{1, 2\}$, then $\mathsf{EY}_T(B) \cap K = \{2, 3, 4\}$. Suppose $B' = \{1, 2, 4\}$. Then

$$\mathsf{EY}_{T'}(B') \cap K = \{2, 3, 4, 6\} \cap \{0, 1, 2, 3, 4\} = \{2, 3, 4\} = \mathsf{EY}_T(B) \cap K \ .$$

Note that the addition of the edge from State 7 to State 3 causes the former to be excluded from $\overline{\mathsf{EX}_{T'}(K)}$.

5.4 The Composition Policies

The SCC analysis algorithm described in previous section is generic, since it does not specify:

1 what set of abstract models L is available;

2 the rule to select the next abstract model $\mathcal{A}'$ to be applied to a set S;

3 the priority function used to choose what element to retrieve from the Work set;

4 the criterion used to decide when to switch to the endgame.

These four aspects make up a *policy* and are the subjects of this section.

We assume that $\mathcal{A}$ is the composition of a set of submodules $M = \{M_1, \ldots, M_m\}$, and the set L of over-approximations consists of the compositions of subsets of M:

$$L \subseteq \{M_{j_1} \parallel \cdots \parallel M_{j_p} \mid \{j_1, \ldots, j_p\} \subseteq \{1, \ldots, m\}\} \ .$$

We also assume that states of $\mathcal{A}$ are the valuations of a set of r binary variables V. The set of variables controlled by each module M_i is nonempty and is a subset of V. Furthermore, let $\eta_{\mathcal{A}}$ and $\eta_{\mathcal{A}'}$ be the numbers of states in $\mathcal{A}$ and its over-approximation respectively, then $2\eta_{\mathcal{A}'} \leq \eta_{\mathcal{A}}$.

The set of all over-approximations generated from subsets of M forms a lattice under the relation $\preceq$, as is shown in Figure 5.5 for $m = 4$. In the case illustrated by this figure, the coarsest abstraction, which is the set of no module, is the 1 of the lattice. Note that this abstraction is never used in practice. The concrete system is the composition of all four modules. For sufficiently large m, it is impractical to make use of all 2^m abstract models; consequently, we shall only consider efficient policies in which any given state contained in the SCC-closed set is passed to SCC-DECOMPOSE at most $O(r)$ times.

Specifically, we shall stipulate that there is a constant λ, such that L can be partitioned into subsets $L_1, \ldots, L_r$ satisfying the following conditions:

1 $|L_i| \leq \lambda$;

2 for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$;

3 $\mathcal{A} \in L_r$.

Two such cases are illustrated in Figure 5.5. The first one is called the popcorn-line policy, which corresponds to the solid thick lines at the

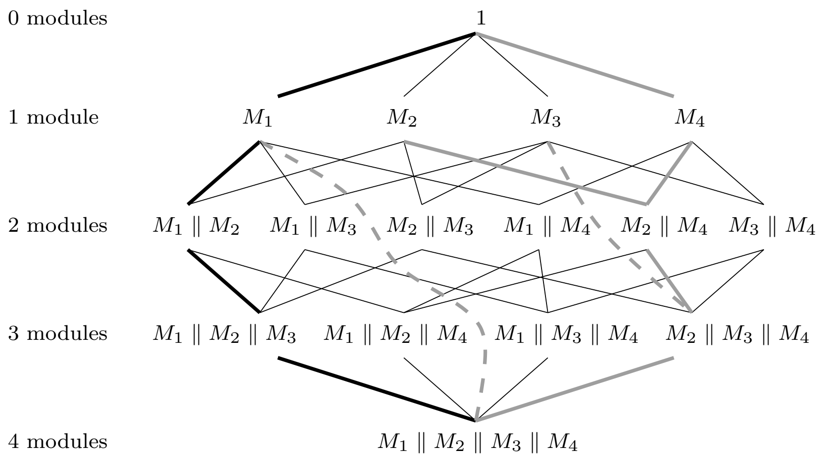

0 modules
1
1 module
M_1
M_2
M_3
M_4
2 modules
$M_1 \parallel M_2$
$M_1 \parallel M_3$
$M_2 \parallel M_3$
$M_1 \parallel M_4$
$M_2 \parallel M_4$
$M_3 \parallel M_4$
3 modules
$M_1 \parallel M_2 \parallel M_3$
$M_1 \parallel M_2 \parallel M_4$
$M_1 \parallel M_3 \parallel M_4$
$M_2 \parallel M_3 \parallel M_4$
4 modules
$M_1 \parallel M_2 \parallel M_3 \parallel M_4$

Figure 5.5. Lattice of approximations.

left of Figure 5.5. Let $(j_1, \ldots, j_m)$ be a permutation of $(1, \ldots, m)$ that identifies a linear order of the modules. The set of approximations with a popcorn-line policy is given as follows,

$$L = \{\mathcal{A}_i = M_{j_1} \parallel \cdots \parallel M_{j_i} \mid 1 \le i \le n\} \ .$$

When an entry $E = (S, L', s)$ is retrieved from Work, the $\mathcal{A}_i$ of lowest index that is not present in L' is chosen as the next approximation $\mathcal{A}'$. With $(j_1, \ldots, j_4) = (1, 2, 3, 4)$ and $\lambda = 1$, the approximations in Figure 5.5 are:

$$\begin{array}{rcl} \mathcal{A}_1 &=& M_1, \\ \mathcal{A}_2 &=& M_1 \parallel M_2, \\ \mathcal{A}_3 &=& M_1 \parallel M_2 \parallel M_3, \\ \mathcal{A}_4 &=& M_1 \parallel M_2 \parallel M_3 \parallel M_4. \end{array}$$

Another policy is called the *lightning-bolt* policy. In Figure 5.5, the lightning-bolt policy is indicated by thick gray lines at the right. Let $(j_1, \ldots, j_m)$ be a permutation of $(1, \ldots, m)$ that identifies a linear order of the modules. The set of approximations with this policy is

$$L = \{\mathcal{A}_{2i-1} = M_{j_1} \parallel \cdots \parallel M_{j_i} \mid 1 \le i \le n\} \cup \{\mathcal{A}_{2i} = M_{j_{i+1}} \mid 1 \le i < n\} \ .$$

When an entry $E = (S, L', s)$ is retrieved from Work, among the two $\mathcal{A}_i$, the one with lower index is chosen first. Let the the order of submodules be $(4, 2, 3, 1)$; the set of approximations in Figure 5.5 is:

$$\begin{aligned} \mathcal{A}_1 &= M_4, & \mathcal{A}_2 &= M_2, \\ \mathcal{A}_3 &= M_4 \parallel M_2, & \mathcal{A}_4 &= M_3, \\ \mathcal{A}_5 &= M_4 \parallel M_2 \parallel M_3, & \mathcal{A}_6 &= M_1, \\ \mathcal{A}_7 &= M_4 \parallel M_2 \parallel M_3 \parallel M_1. \end{aligned}$$

In both cases, the times a state appearing in the set passed to SCC-DECOMPOSE is bounded by the number of approximations in L. Therefore, a popcorn-line policy tends to call SCC-DECOMPOSE fewer times. A lightning-bolt policy may break up the SCC-closed sets with easy approximations ($\{\mathcal{A}_{2i}\}$) before applying harder approximations ($\{\mathcal{A}_{2i-1}\}$) to them, and therefore tends to use less memory.

The popcorn-line approach defines an SCC partition refinement tree, whose roots are the fair SCCs in the most abstract model and whose other nodes correspond to fair SCCs in the more refined abstract models. An example is given in Figure 5.6, which highlights the potential advantages of SCC refinement. The figure corresponds to a model of eight dining philosophers, with a property that states that under the given fairness constraints, if a philosopher is hungry, she eventually eats. The system has nine modules, which are the property automaton and the eight modules for the philosophers. The property passes in the concrete model, i.e., no fair cycles exist in the system.

Only the nodes representing fair SCCs are shown in this tree. The nodes at Level i are the fair SCCs of $\mathcal{A}_i$, together with their numbers of states. ($\mathcal{A}_1$ is the property automaton.) A separate reachability analysis shows that there are about 47k reachable states in the concrete model. Note that only very small sets of states remain after the composition of the first four modules—the property automaton, the philosopher named in the property, and her two neighbors. We are able to prune away a lot of reachable states in the first few levels, and that no work is done on the concrete system.

To define a policy, the order in which elements are retrieved from the Work set also needs to be specified. Two obvious choices are FIFO and LIFO order. As one would expect, the SCC refinement tree can be traversed in breadth-first manner for a FIFO order, and in depth-first manner for a LIFO order. When, as in Figure 5.6, there are no fair cycles in $\mathcal{A}$, the order in which the tree is visited is immaterial. However, in the presence of concrete fair cycles, one strategy may lead to earlier termination than the other may. If one assumes that fair cycles are numerous, then depth-first search is particularly attractive. Breadth-first search, on the other hand, can be implemented with low overhead, because at any time, only one abstract model needs to be constructed and to remain active in the memory.

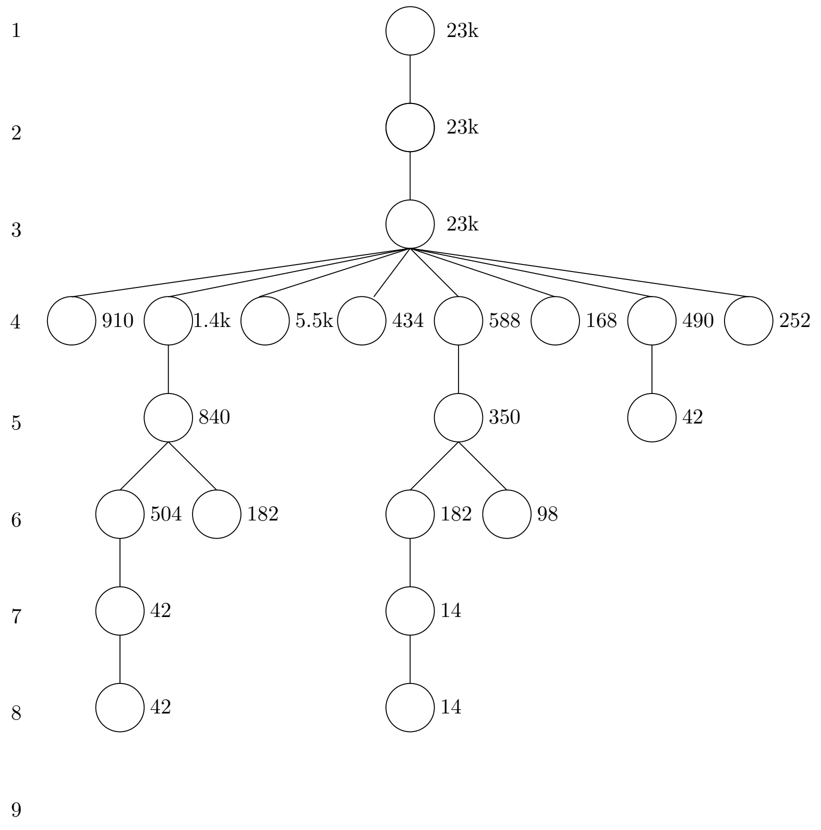


Figure 5.6. An SCC partition refinement tree.

Chapter 6

DISJUNCTIVE DECOMPOSITION

6.1 Adaptive Popcorn-line Policy

It may not be practical to adopt the popcorn-line policy all the way down to the concrete model, because

1 There may be too much overhead in analyzing all the abstract models;

2 if an SCC-closed set becomes weak or terminal, checking it directly in the concrete model may be cheap.

In these two cases, one may decide to switch to the endgame. That is, after spending a reasonable amount of effort on decomposing the SCCs in the abstract models, we jump to the concrete model. When the endgame comes, there are different ways of jumping to the concrete system—all of them can be considered as variants of the popcorn-line policy.

The first variant is to go to $\mathcal{A}$ directly, and search for fair cycles inside each SCC-closed set S in Work. Both SCC hull and SCC enumeration algorithms can be used for the fair-cycle detection. Assume, for instance, that $\mathcal{A}$ is the composition of the set of submodules $\{M_1, ..., M_8\}$, and we decide to jump after composing the first three submodules. The first variant of the popcorn-line policy can be described as follows:

$$\begin{array}{rcl} \mathcal{A}_1 & = & M_1, \\ \mathcal{A}_2 & = & M_1 \parallel M_2, \\ \mathcal{A}_3 & = & M_1 \parallel M_2 \parallel M_3, \\ \hline \mathcal{A}_4 & = & M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8. \end{array}$$

Alternatively, we can further trim the SCC-closed sets before searching for fair cycles in the concrete model. Remaining submodules are applied, one at a time, to further partition these SCC-closed sets. This variant, called the "Cartesian product" approach, is characterized as follows:

$$\begin{array}{rcl}
\mathcal{A}_1 & = & M_1, \\
\mathcal{A}_2 & = & M_1 \parallel M_2, \\
\mathcal{A}_3 & = & M_1 \parallel M_2 \parallel M_3, \\
\hline
\mathcal{A}_4 & = & M_4, \\
 & & \ldots \\
\mathcal{A}_8 & = & M_8, \\
\mathcal{A}_9 & = & M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8.
\end{array}$$

Note that we are using $\mathcal{A}_{4-8}$ to further reducing the SCC closed sets, before going to the concrete model $\mathcal{A}_9$. Given the fact that each submodule M_i is relatively small and we consider them one at a time, the calls to SCC-DECOMPOSE in $\mathcal{A}_{4-8}$ are cheap. In fact, the partition of the state space in $\mathcal{A}_3$ has been based on the assumption that the state variables of other submodules are *free* variables (i.e., they can take arbitrary values at all times); by calling SCC-DECOMPOSE on these remaining modules individually, we can constrain their state variables resulting in further partition of the SCC-closed sets. A direct analogy can be observed between this approach and Machine-by-Machine state space traversal algorithm of [CHM$^+$94] in computing the set of approximate reachable states.

The third variant, called the "one-step further composition" approach, is characterized as follows:

$$\begin{array}{rcl}
\mathcal{A}_1 & = & M_1, \\
\mathcal{A}_2 & = & M_1 \parallel M_2, \\
\mathcal{A}_3 & = & M_1 \parallel M_2 \parallel M_3, \\
\hline
\mathcal{A}_4 & = & \mathcal{A}_3 \parallel M_4, \\
 & & \ldots \\
\mathcal{A}_8 & = & \mathcal{A}_3 \parallel M_8, \\
\mathcal{A}_9 & = & M_1 \parallel M_2 \parallel M_3 \parallel \cdots \parallel M_8.
\end{array}$$

We are using $\mathcal{A}_{4-8}$ to further fracturing the SCC closed sets. Note that the previous Cartesian Product approach does not compose prior to making the full jump; in contrast, this "one-step-further" approach invests more heavily by composing $\mathcal{A}_3$ with each of the remaining submodules. At each step from $\mathcal{A}_4$ to $\mathcal{A}_9$, we start with the refined SCC-closed sets computed in the previous step. For a transition to exist in the composition, it must exist in both of the machines being composed. Whereas the Cartesian Product approach never fractures SCCs by this joint constraint, this third variant does, ultimately leading to the partitioning of these SCCs into smaller SCC-closed sets.

Following this line to the extreme would lead us all the way back to the original popcorn-line policy, which is considered the forth variant. These four variants are only representatives in the general framework of adaptive popcorn-line policy. The first variant represents the least investment in compositional analysis, and therefore suffers the least amount of overhead. However, when it performs the most expensive part of the computation—cycle detection in the exact system—it must search in larger state subspaces. Conversely, the fully iterative approach has the smallest state subspaces to search in the concrete model, but incurs the greatest overhead in analyzing abstract models.

6.2 Disjunctive Decomposition Theorem

After switching to the endgame and further fracturing the SCC-closed sets, we shall search for reachable fair cycles in the concrete system. Since the concrete system often has a large number of states, it is desirable to decompose the entire state space into many subspaces and search them separately. This requires the decomposition of state space to be disjunctive – that is, the language accepted by each subspace is a subset of the original language, and the union of these language subsets is the original language.

Let the SCC quotient graph of a labeled generalized Büchi automaton $\mathcal{A}$ be $\mathcal{G} = \langle \mathcal{C}, \mathcal{C}_0, T_C, F_C \rangle$, where $\mathcal{C}$ is the set of SCCs in $\mathcal{A}$, $\mathcal{C}_0 \subseteq \mathcal{C}$ is the set of initial SCCs, $T_C \subseteq \mathcal{C} \times \mathcal{C}$ is the transition relation, and F_C is the set of fair SCCs.

Let $\mathcal{G}' = \langle \mathcal{C}', \mathcal{C}'_0, T'_C, F'_C \rangle$ be a subgraph of $\mathcal{G}$, where $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{C}'_0 \subseteq \mathcal{C}_0$, $T'_C \subseteq T_C$, and $F'_C \subseteq F_C$. In other words, removing some nodes or edges of an SCC graph, or making some fair nodes non-fair, produces a subgraph.

A subgraph of the SCC graph $\mathcal{G}(\mathcal{A})$ induces a new Büchi automaton.

Definition 6.1 *Given the SCC quotient graph $\mathcal{G}$ of $\mathcal{A}$ and a subgraph $\mathcal{G}'$, the new induced automaton $\mathcal{A} \Downarrow \mathcal{G}' = \langle S', S'_0, T', A, \Lambda, \mathcal{F}' \rangle$ is defined as follows:*

- *$S' \subseteq S$ is the subset of original states that appear in $\mathcal{C}'$,*
- *$S'_0 \subseteq S_0$ is the subset of original initial states in $\mathcal{C}'_0$,*
- *$T' \subseteq T$ is the subset of original transitions among the states that appear in $\mathcal{C}'$.*
- *$F'_i \in \mathcal{F}'$ is the subset of $F_i \in \mathcal{F}$ that intersects the set of states in F'_C.*

Essentially, $\mathcal{A} \Downarrow \mathcal{G}'$ is the original automaton $\mathcal{A}$ restricting its operation only in the set of states S'. It follows that $\mathcal{A} \Downarrow \mathcal{G}(\mathcal{A}) = \mathcal{A}$.

Since accepting runs in the induced automaton $\mathcal{A} \Downarrow \mathcal{G}'$ are always accepting in $\mathcal{A}$, its language is a subset of $\mathcal{L}(\mathcal{A})$. Furthermore, the *pruning* operation on the SCC graph, defined as removing nodes that are not on any path from initial nodes to fair nodes, does not change the language accepted by the corresponding automaton. This claim can be extended to the SCC subgraph of any over-approximation of $\mathcal{A}$.

Observation 6.2 *Let $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{G}'$ be a subgraph of $\mathcal{G}(\mathcal{A}')$, then $\mathcal{L}(\mathcal{A} \Downarrow \mathcal{G}') \subseteq \mathcal{L}(\mathcal{A})$.*

We define an SCC subgraph $\mathcal{G}^{C_j}(\mathcal{A})$ for every fair node C_j of $\mathcal{G}(\mathcal{A})$, such that it contains all SCCs that are on the paths from the initial

SCCs to C_j (including C_j); furthermore, all the nodes are marked non-fair except for C_j. We can build $\mathcal{G}^{C_j}$ by marking C_j fair and all the other nodes non-fair and then pruning the SCC graph. When the context is clear, we will simply use $\mathcal{G}^{C_j}$ to denote such a subgraph. Each SCC subgraph $\mathcal{G}^{C_j}$ induces a new automaton that accepts a subset of the original language; the union of these subsets of languages is the same as the language of the original automaton.

In addition, $\mathcal{G}^{C_j}$ can be further decomposed into subgraphs. An SCC subgraph of this kind, denoted by $\mathcal{G}_i^{C_j}$, represents the i-th path from an initial SCC to C_j. The languages accepted by the automata $\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}$ also form a disjunctive decomposition of the language accepted by $\mathcal{A} \Downarrow \mathcal{G}^{C_j}$. The claim can be extended to the SCC subgraphs of any over-approximation of $\mathcal{A}$. To summarize, we have the following theorem:

Theorem 6.3 (Disjunctive decomposition) *Let $\mathcal{A} \preceq \mathcal{A}'$ and the SCC graph $\mathcal{G}(\mathcal{A}')$ has a set of SCC subgraphs $\{\mathcal{G}'^{C_j}_i\}$ as defined above. Then, $\mathcal{L}(\mathcal{A}) = \emptyset$ if and only if $\mathcal{L}(\mathcal{A} \Downarrow \mathcal{G}'^{C_j}_i) = \emptyset$ for every subgraph.*

An new automata in the form of $\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}$ is an under-approximation of the exact system. Normally, an under-approximation can be used to certify the existence of fair runs, but not to prove language emptiness. However, Theorem 6.3 shows that the set $\{\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}\}$ produced by disjunctive decomposition forms a complete set of under-approximations. Therefore, they do not produce conservative results for language emptiness checking.

One advantage of applying this disjunctive decomposition theorem is the ability of checking each of these new automata separately. When we can restrict the search to a smaller state space, we increase the effectiveness of applying don't cares in speeding up symbolic image and pre-image computations.

6.3 Guided Search for Fair Cycles

Theorem 6.3 allows us to disjunctively decompose the concrete system into subautomata $\{\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}\}$, where $\mathcal{G}_i^{C_j}$ is an initial-fair path in the SCC quotient graph of $\mathcal{A}'$. Since each of these subgraphs corresponds to a depth-first search path in the SCC graph, and contains a set of abstract counterexamples, it is also called a *hyperline*. Our fair cycle detection algorithm goes through all these hyperlines and checks language emptiness on each of them in isolation.

Computing hyperlines requires not only all fair SCCs of $\mathcal{A}'$, but also the *non-fair* SCCs. These non-fair SCCs can be computed with SCC-DECOMPOSE, and just like the fair ones, they can also be computed incrementally. As one may expect, the number of hyperlines in an SCC graph—a DAG—is exponential in the size of the graph. In order to avoid an excessive partitioning cost on the over-approximations, with the consequent exponential number of hyperlines, we apply the following heuristic control on the size of the SCC graphs:

- Skip SCC-DECOMPOSE on S if S is non-fair in $\mathcal{G}(\mathcal{A}')$ and its size (number of concrete states) is below a certain threshold.
- Switch to the endgame if the number of edges of the SCC graph $\mathcal{G}(\mathcal{A}')$ exceeds a certain threshold.
- Switch to the endgame if the number of fair nodes of the SCC graph $\mathcal{G}(\mathcal{A}')$ exceeds a certain threshold.

With such a heuristic control, the number of hyperlines is bounded by a constant value.

In the endgame, we disjunctively decompose the exact state space into subspaces according to the different hyperlines of the last abstract model. Every hyperline or $\mathcal{G}_i^{C_j}$ induces a subautomaton of the exact system. Although subautomata may share states, we can avoid visiting any state more than once by keeping a global set of visited states. Within $\mathcal{A} \Downarrow \mathcal{G}_i^{C_j}$, we search for cycles that are both reachable and fair. Although reachability analysis shares the same worst-case complexity bound with the best cycle-detection algorithm, in practice it is still much cheaper than fair cycle detection. This motivates us to always make sure a certain state subspace is reachable before deploying the cycle detection procedure, in order to avoid searching unreachable states for a fair cycle.

In particular, fair cycle detection is triggered only after the reachability analysis hits one or more *promising* states—states that are in fair SCC-closed sets and at the same time satisfy some acceptance conditions. Recall that the symbolic SCC enumeration algorithms [BGS00, BGS05, GPP03] used in SCC-DECOMPOSE compute an SCC by first choosing a *seed*. The promising states encountered during forward reachability analysis are good candidates for the seed. We can give higher priority to promising states that are reached earlier in forward reachability computation, in the hope of getting a shorter counterexample. In addition to the order in which they are encountered during the forward search, promising states can also be prioritized according to the number of acceptance conditions they satisfy: if two promising states are hit simultaneously by the forward search, whichever satisfies more acceptance conditions is preferred. By prioritizing the seeds, we heuristically choose the SCC that is expected to be closer to the initial states and more likely to be fair; this may reduce the number of reachable states traversed by forward search and may lead to a shorter counterexample. In prior art, the algorithm in [HTKB92] was also designed to avoid visiting too many reachable states in the search for fair cycles, but their approach was significantly different from ours.

Although disjunctive decomposition have divided the entire state space into smaller pieces, the reachable states of each subautomaton may still be many. The ideal way of finding a fair cycle is to traverse only part of the reachable states of the subautomaton, and go directly to a promising state to start the SCC enumeration. To reach a promising state with the least possible overhead, i.e., by traversing the least number of reachable states, we need some guidance for such a targeted search.

The intermediate results of the reachability analysis of $\mathcal{A}'$ can provide guidance for such a targeted search. Reachability analysis with Breadth-First Search (BFS) gives a set of *reachability onion rings*, denoted by $\{R^0, R^1, \ldots, R^l\}$; each ring is the set of states at a certain distance from the initial states. For example, a state in R^2 can be reached from an initial state in two steps but not less. Suppose that R^3 is the earliest ring that contains a promising state, one wants to spend as little effort as possible in traversing states in R^1 and R^2.

We now present a guided search algorithm for fair cycle detection, called DETECT-FAIR-CYCLE. As shown in Figure 6.1, the procedure is called by MODEL-CHECK for every hyperline $\mathcal{G}'$. There are two global variables *Reach* and *Queue*, representing the set of already reached states and the SCC-closed sets that remain to be inspected, respectively. The reachable onion rings of $\mathcal{A}'_{sub}$ from an abstract model, denoted by *absRings* or $\{R^i\}$, are used to estimate the distance of an SCC closed set to the initial states. We use the distance to rank the relative importance of SCC-closed sets in the priority queue *Queue*. The procedure SCC-DECOMPOSE-WITH-ET searches the SCC sets one by one for fair cycles; the SCC closed set closest to initial states (measured by the distance in the abstract onion rings) is always picked up first. The global reachable state set *Reach* may be updated after each SCC closed set is inspected, if states not yet in *Reach* have been discovered by the forward search of SCC enumeration. The entire procedure terminates when either all reachable states in all $\mathcal{A}_{sub}$ are visited, or a fair cycle is found.

Instead of using the conventional IMAGE computation, we use a heuristic algorithm called *sharp image* computation for the targeted reachability analysis. The pseudo code for a sharp image computation is also given in Figure 6.1. Let D be the *from set* (the set of states for which we want to find the successors), and $\{R^i\}$ be the set of reachable onion rings from an abstract model. The procedure first finds the abstract onion ring that is closest to the target and at the same time intersects D. The intersection of this ring and D has the shortest approximate distance to a promising state. This set is further compacted into $D^{\#}$ by BDD-SUBSETTING [RS95, PH98]. As a generic function, BDD-SUBSETTING can return a minterm, a cube, or an arbitrary subset of $(D \cap R^i)$ with a small BDD representation. Finally, the image of $D^{\#}$ is computed with the conventional IMAGE operation. It is clear that the result is a subset of $\mathsf{EY}(D)$.

Our guided search procedure with sharp image computation is different from the high-density algorithm of [RS95], because our goal in compacting the from set is to get closer to the fair SCCs, not increase the density of its BDD representation. Nevertheless, our approach shares a common problem with high-density search—namely, how to recover from *dead-ends*. Since IMAGE$^{\#}$ computes only a subset of the exact image, it is possible for the frontier set, $Front$, to be empty before the forward search actually reaches the fixpoint. Whenever this happens, we need to backtrack with the standard image computation using the transition relation of $\mathcal{A}_{sub}$ and *Reach*.

Every time a promising state is encountered during the targeted reachability analysis, it is picked as a *seed* for computing the SCC. If the SCC containing this seed intersects all the fair sets $F_i \in \mathcal{F}$, we can terminate the entire procedure immediately. If the SCC is not fair, it is merged into the set of already reached states *Reach* before the targeted reachability analysis is resumed (because the SCC has proved to be reachable). Since every SCC found in this way is guaranteed to be reachable, the SCC enumeration algorithms [BGS00, BGS05, GPP03] can be further enhanced with early termination [SRB02]: they terminate as soon as a fair cycle is found, as opposed to after both the forward and backward search from the seed reach their fixpoints. This requires that after each forward and backward step, we check whether the intersection of the forward and backward results satisfies all the fairness conditions—if it

```
DETECT-FAIR-CYCLE($\mathcal{A}, \mathcal{A}', \mathcal{G}', Reach, Queue$){
                                  // model, abs model, hyperline,
                                  // reached states, and scc-closed sets
    $\mathcal{A}'_{sub} = \mathcal{A}' \Downarrow \mathcal{G}'$;
    $\mathcal{A}_{sub} = \mathcal{A} \Downarrow \mathcal{G}'$;

    $absRings$ = COMPUTE-REACHABLE-ONIONRINGS($\mathcal{A}'_{sub}$);
    $Front = Reach$;

    while (true) {

        while ($Front \neq \emptyset$) and ($Front \cap Queue = \emptyset$) {
            $Front$ = IMAGE$^{\#}$($\mathcal{A}_{sub}, Front, absRings$) $\setminus Reach$;
            if ($Front = \emptyset$)
                $Front$ = IMAGE($\mathcal{A}_{sub}, Reach$) $\setminus Reach$;
            $Reach = Reach \cup Front$;
        }

        if ($Front = \emptyset$)
            return false;
        if (SCC-DECOMPOSE-WITH-ET($\mathcal{A}_{sub}, Queue, absRings$))
            return true;
    }
}

IMAGE$^{\#}$($\mathcal{A}, D, \{R^i\}$) {

    $i = |\{R^i\}|$;
    while ($D \cap R^i = \emptyset$) {
        $i = i - 1$;
    }

    $D^{\#}$ = BDD-SUBSETTING ($D \cap R^i$);

    return IMAGE($\mathcal{A}, D^{\#}$);
}
```

Figure 6.1. Guided search of fair cycles and sharp image.

does, the *union* of the forward and backward search results contains a reachable fair cycle.

When the language is indeed empty, all reachable states of the subautomata must be traversed. Let $\eta_{\mathcal{A}}$ be the number of *reachable* states of the exact system, and let the total number of hyperlines be a constant value; then, the cost of deciding reachability in our guided search procedure is $O(\eta_{\mathcal{A}})$. The total cost of fair cycle detection depends on the underlying symbolic SCC enumeration algorithm, which we analyze in the next section.

6.4 Implementation and Experiments

Complexity Analysis

The refinement algorithm described thus far cannot improve the complexity bound of the language emptiness check. On the other hand, it does not make the theoretical complexity bound worse. In the following, we show that the complexity of our incremental approach is within a constant factor from that of the non-incremental one; this means that it is $O(\eta_{\mathcal{A}})$ when the linear time algorithm of [GPP03] is used in SCC-DECOMPOSE, or when the Lockstep algorithm of $O(\eta_{\mathcal{A}} \log \eta_{\mathcal{A}})$ if [BGS00, BGS05] is used.

In the following theorem, we assume that the linear time algorithm of [GPP03] is used.

THEOREM 6.4 *If the set L of approximations can be partitioned into subsets $L_1, \ldots, L_r$ such that, for some constant λ,*

1 $|L_i| \leq \lambda$*;*

2 for every $\mathcal{A}' \in L_i$, $\eta_{\mathcal{A}'} \leq 2^i$*; and*

3 $\mathcal{A} \in L_r$*,*

then the generic SCC refinement algorithm runs in $O(\eta_{\mathcal{A}})$ steps.

PROOF: *Both reachability computation and SCC enumeration take a linear time, so the total cost of SCC analysis for $\mathcal{A}'$ is bounded by $k\eta_{\mathcal{A}'}$, for some constant k. Let the number of effective states of $\mathcal{A}'$ be denoted by $\eta_{\mathcal{A}'}$, then*

$$\eta_{\mathcal{A}'} \leq \eta_{\mathcal{A}}/2^i \ .$$

Hence, the cost of analyzing all approximations and $\mathcal{A}$ itself is bounded by

$$k\eta_{\mathcal{A}}(\lambda + \lambda/2 + \lambda/4 + \cdots + \lambda/2^r) \ ,$$

which is bounded by $2\lambda k\eta_{\mathcal{A}}$.

While we cannot hope for an improved run time in the worst case, we expect that the refinement-based approach will be beneficial when the state space breaks up into many small SCC-closed sets.

In some spacial cases, we can prove the following linear complexity result—even when the $n \log n$ algorithm of [BGS00, BGS05] is used for SCC enumeration.

THEOREM 6.5 *Under the assumptions for L of Theorem 6.4, if for some constant γ, the pairs $(S, \mathcal{A}')$ passed to* SCC-DECOMPOSE *satisfy $|S| \leq \gamma\eta_{\mathcal{A}}/\eta_{\mathcal{A}'}$, then the refinement algorithm runs in $O(\eta_{\mathcal{A}})$ time.*

PROOF: *The analysis of $\mathcal{A}$ consists of the decomposition of SCC-closed sets of size bounded by γ. Their number is linear in $\eta_{\mathcal{A}}$, and each decomposition takes constant time. Hence, the total time for the analysis of $\mathcal{A}$ is $O(\eta_{\mathcal{A}})$. If $|C|$ is the number of states in SCC C of $\mathcal{A}'$, then $|C|\eta_{\mathcal{A}'}/\eta_{\mathcal{A}}$ is the* effective size *of C. The cost of analyzing $\mathcal{A}'$ is therefore $O(\eta_{\mathcal{A}'})$. With reasoning analogous to the one of Theorem 6.4, one finally shows that the total time is also $O(\eta_{\mathcal{A}})$.*

Experiments on SCC Refinement

First, we describe the details of two implemented policies for the SCC analysis algorithm D'n'C. Both versions implement the basic popcorn-line approach, and correspond to a breadth-first search of the SCC refinement tree. The set of submodules are generated and then ordered according to a static refinement scheduling strategy of [Jan99]. It partitions the entire set of state variables into many smaller clusters. The partitioning is based on the structural information of the model (e.g., latch connectivity). Each cluster is considered as a submodule, and the parallel composition of all these submodules is the concrete system. The submodules are heuristically sorted according to their distances from the state variables appearing in the property automaton.

The two implemented D'n'C policies differ in when to switch to the endgame: the first policy de-emphasizes compositionality in comparison to strength reduction by performing only two levels of composition. At the first level, it computes the SCCs of the property automaton, and at the second level, it composes all the other modules of the system. The second policy tries to exploit the full compositionality implied by Figure 5.5 and 5.6. To avoid too much overhead on analyzing the over-approximations, it heuristically stops the refinement at some point, and then immediately composes all the remaining modules, thus proceeding directly to the exact system. In the implementation, we stop the linear composition after 30% of the state variables have been composed. Once the exact system is reached, the EMERSON-LEI algorithm is applied to its SCC-closed sets. For ease of reference, we refer to the first policy as the *Two-level* method, and to the second as the *Multi-Level* method.

In both policies, weak SCCs are grouped together and are checked for cycles in the concrete system immediately after they are discovered. The underlying assumption for the special handling of weak/terminal SCCs is that model checking these SCCs is cheaper in the concrete model. If D‘n’C finds a concrete fair cycle, it terminates, otherwise it discards these SCCs. At any abstraction level, if no SCCs are present, the algorithm also terminates because there is no cycle in the concrete model either.

The proposed algorithm has been implemented in the symbolic model checker VIS [B$^+$96][VIS]. The results of Table 6.1 were obtained by appropriately calling the standard Language Emptiness command of VIS. SCC analysis was performed with the Lockstep algorithm of [BGS00]. (Separate study showed that the algorithm of [GPP03] had a performance slightly worse than Lockstep [BGS00, BGS05] in practice, because of its additional bookkeeping overhead.) Prior reachability analysis were used as don’t cares where possible.

In Table 6.1, all examples were run with the same fixed BDD order (obtained with previous runs of dynamic variable reordering). For the same set of models and property automata, a second table was also obtained with dynamic variable ordering turned on for each example. Similarly, a third table was obtained using the EL2 variant of the Emerson-Lei algorithm [HTKB92]. The second and third tables were omitted for brevity, since their characters of the results were not significantly different. (The only exception to the statement was the fact that the example nmodem1 took only 209 seconds with EL2, versus 4384 for the original Emerson-Lei algorithm.) The experiments were conducted on an IBM Intellistation running Linux with a 400MHz Pentium II processor with 1GB of SDRAM.

In Table 6.1 has four columns. The three fields of the first column give the name of the example, a symbol indicating whether the formula passes (P: no fair cycles exist) or fails (F: a fair cycle exists), and the number of binary state variables in the system. The three fields of the second column, obtained by directly applying the VIS Emerson-Lei algorithm, give:

1 the time it took to run the experiment (T/O indicates a run time greater than 4 hours);

2 the peak number of live BDD nodes (in millions); and

3 the total number of pre-image (EX) / image (EY) computations needed.

These same field descriptors also apply to the third and fourth columns (for the Two-Level and Multi-Level versions of the D‘n’C algorithm), except that the latter has an additional field that indicates how the verification process terminates: ‘n’ means that the algorithm arrives at some intermediate level of the refinement process in which there no longer exists any fair SCC; ‘w’ means that there is a weak fair SCC found and it contains a fair cycle.

The property automata being used in the experiment are translated from LTL formulae. In order to avoid bias in favor of the new approach, each model is checked against a strong LTL property automaton. Note that the presence of the ’n’ or ’w’ in the last field demonstrates that both pruning of the SCC refinement tree and strength reduction are active in these experiments.

Comparing the D‘n’C algorithm to the one by Emerson and Lei, we find that, with only three exceptions out of 18 examples, there is a significant (more than a factor of 2) performance advantage for the D‘n’C algorithm. Comparing the Two-Level and Multi-Level versions, one sees that with four exceptions (eisenb2, philo2, philo3, and shamp2), the two policies give comparable performance. This is because most of the examples are simple mutual-exclusion and arbitration protocols, in which the properties have little localities. We expect the compositional algorithm to do even better on models with more localities. On the other hand, we have found that the greater compositionality of the Multi-Level version proves its worth, especially on the larger examples.

Experiments on Disjunctive Decomposition

Now we describe the details of another implemented policy for disjunctive decomposition and the targeted search for fair cycles. This policy is a variant of the popcorn-line approach, with breadth-first search of the SCC refinement tree. Before jumping to the exact system, it trims the fair SCC-closed sets further on the remaining submodules by the “Cartesian Product” approach. The enhanced algorithm, called D’n’C$^{\#}$, was compared to D’n’C on the same set of test cases to study the effectiveness of the added feature. The experiments are given in Table 6.2 and 6.3, which were conducted on a 400MHz Pentium II processor with 1GB of SDRAM.

In Table 6.2, prior reachability analysis results were used as don’t cares where possible. Note that this table are results under a somewhat ideal case—it assumes that the exact reachability computation results are available. The table has four columns. The three fields of the first column give the name of the example, a symbol indicating whether the formula passes or fails, and the number of binary state variables in the system. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D’n’C$^{\#}$ algorithm to D’n’C, we find three wins for D’n’C$^{\#}$ and 15 wins for D’n’C. This indicates that the disjunctive decomposition is encumbered by overhead of maintaining and decomposing the SCC graph. However, among the 15 wins of D'n'C, only four are for problems requiring more than 100 seconds to complete—that is, the easy problems. In contrast, on the three wins of D'n'C$^{\#}$, D'n'C took 1337, 1683, and 233 seconds. Therefore, we conclude that in general the additional overhead of disjunctive decomposition is not significant. On the harder problems, D'n'C$^{\#}$ is as competitive as D'n'C when advance reachability analysis is feasible.

In Table 6.3, the same set of test cases were checked with the approximate reachability analysis results as the don't cares where possible—that is, with ARDCs as opposed to RDCs. (Approximate reachability analysis is usually much faster than exact reachability analysis, and in practice, may be the only feasible way of extracting don't cares from reachable states.) The table has four columns. The three fields of the first column repeat the description of the test cases. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D'n'C$^{\#}$ algorithm to D'n'C, we find 12 wins for D'n'C$^{\#}$ and 6 for D'n'C. In addition, all the 6 wins for D'n'C are for problems requiring less than 100 seconds to complete; in contrast, D'n'C$^{\#}$ wins more on the harder ones—on eight out of its 12 wins, D'n'C timed out after 4 hours. The difference here is that both D'n'C and EL depend heavily on full reachability to restrict the search spaces, but the disjunctive decomposition and sharp guided search of D'n'C$^{\#}$ minimize this dependency.

We also conducted experiments on a set of much harder test cases, the Texas-97 benchmark circuits. The property automata being used in the experiments were also translated from LTL formulae. The experiments were run on an IBM Intellistation with a 1700MHz Pentium-IV processor and 2GB of SDRAM. The results are given in Table 6.4.

In Table 6.4, the comparison is with the results of approximate reachability analysis as the don't cares where possible. Note that exact reachability analysis is infeasible for most of these circuits, except for MSI. The table has four columns. The three fields of the first column give the name of the example, a symbol indicating whether the formula passes or fails, and the number of binary state variables in the system. The next three columns compare the run time, the total memory usage, and the peak number of live BDD nodes of the three methods. Comparing the D'n'C$^{\#}$ algorithm to D'n'C, we find five wins for D'n'C$^{\#}$ and two for D'n'C. Again, the two wins for D'n'C are easier problems, and the five wins for D'n'C$^{\#}$ are much harder—among them, two cannot be finished by D'n'C within 8 hours. Therefore, it demonstrates a decisive advantage of the D'n'C$^{\#}$ algorithm over both D'n'C and EL.

6.5 Further Discussion

We have shown that over-approximations of the concrete system can be used to gradually refine the SCC-closed sets to SCCs. The D‘n’C algorithm has the advantages of being compositional, considering only parts of the complete state space, and taking into account the strength of an SCC to deplore the proper model checking algorithm. We have discussed the different policies in traversing the lattice of over-approximated systems. In comparison to the original Emerson-Lei algorithm, the new algorithm has demonstrate significant and almost consistent performance improvement. This indicates the importance of the three improvement factors built into the proposed algorithm: SCC refinement, compositionality, and strength reduction.

We have also shown that the analysis of SCC quotient graph of an over-approximated system can be used to decompose the concrete search state space. Based on disjunctive decomposition, our guided search algorithm for fair cycle detection demonstrates further performance improvement. Our experiments show that for large systems or otherwise difficult problems, heavy investment in these heuristics is well justified.

The simplicity of the implemented policies in comparison to the generality of our framework suggests that there can be many promising extensions and variations. The joint application of over- and under-approximations of the concrete system, for instance, can be an interesting future work.

The generic framework can be highly parallelized by assigning different entries from the Work list, as well as the disjunctive state subspaces, to different processors. Processors that deal with disjoint sets of states have minimal communication and synchronization requirements. Although the algorithm is geared towards BDD based symbolic model checking, SCC refinement can also be combined with explicit state enumeration and SAT based approaches.

Table 6.1. Comparing Emerson-Lei and D'n'C. With RDC's.

			Emerson-Lei (VIS LE)			D'n'C Two-Level			D'n'C Multi-Level			
Circuit and LTL	P/ F	latch num	Time (s)	BDD (M)	EX/EY	Time (s)	BDD (M)	EX/EY	Time (s)	BDD (M)	EX/EY	
bakery1	F	56	212	5.1	5337/0	31	1.3	354/4	27	1.3	484/328	
bakery2	P	49	69	3.4	526/0	20	1.3	10/4	20	1.3	62/73	n
bakery3	P	50	421	14	1593/0	46	2.5	90/4	43	1.8	537/428	
bakery4	F	58	T/O	-	-/-	1950	3.4	1088/5	1337	4.7	947/96	
bakery5	F	59	T/O	-	-/-	1009	6.1	127/5	623	6.1	216/243	
eisenb1	F	35	23	1.0	416/0	16	0.9	21/4	16	0.9	21/4	
eisenb2	F	35	T/O	-	-/-	4800	8.2	162/5	1683	7.7	105/93	w
elevator1	F	37	210	14	163/0	49	2.8	132/9	41	2.2	155/31	
nmodem1	P	56	4384	11	5427/0	192	1.1	992/4	233	0.6	5007/71	
peterson1	F	70	17	1.1	24/0	20	1.3	19/4	21	1.2	157/173	
philo1	F	133	371	12	258/0	7	0.2	8/12	7	0.2	8/12	w
philo2	F	133	73	2.8	557/0	30	1.3	258/5	12	0.5	25/44	w
philo3	P	133	T/O	-	-/-	T/O	-	-/-	115	1.2	993/224	
shamp1	F	143	44	2.1	8/0	103	5.6	9/6	87	2.2	266/280	
shamp2	F	144	T/O	-	-/-	1892	16.	74/6	101	2.9	345/349	
shamp3	F	145	T/O	-	-/-	337	4.4	19/17	335	4.4	19/17	w
twoq1	P	69	12	0.4	25/0	4	0.1	7/9	4	0.1	7/9	n
twoq2	P	69	241	8.9	175/0	27	0.8	91/5	30	0.9	181/95	

Table 6.2. Comparing EL, D'n'C, and D'n'C$^{\#}$. With RDC's.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/ F	latch num	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'c	D'n'C$^{\#}$
bakery1	F	56	212	27	159	262	75	125	5.1	1.3	1.5
bakery2	P	49	69	20	28	152	73	74	3.4	1.2	1.2
bakery3	P	50	421	43	1514	550	111	125	14	1.8	1.5
bakery4	F	58	T/O	1337	655	-	411	476	-	4.7	4.8
bakery5	F	59	T/O	623	737	-	555	554	-	6.1	9.9
eisen1	F	35	23	16	128	69	50	64	1.0	0.9	0.6
eisen2	F	35	T/O	1683	944	-	564	340	-	7.7	1.7
elevator1	F	37	210	41	192	489	132	369	14	2.2	10.3
nmodem1	P	56	4384	233	227	569	63	169	11	0.6	2.2
peterson1	F	70	17	21	41	73	83	78	1.1	1.2	1.2
philo1	F	133	371	7	56	401	26	37	12	0.2	0.1
philo2	F	133	73	12	58	145	44	42	2.8	0.5	0.3
philo3	P	133	T/O	115	207	-	119	329	-	1.2	7.0
shamp1	F	143	44	87	303	96	113	401	2.1	2.2	9.2
shamp2	F	144	T/O	101	239	-	187	268	-	2.9	3.5
shamp3	F	145	T/O	335	1383	-	478	500	-	4.4	5.8
twoq1	P	69	12	4	14	36	23	24	0.4	0.1	0.0
twoq2	P	69	241	30	289	333	47	509	8.9	0.9	7.9

Table 6.3. Comparing EL, D'n'C, and D'n'C$^{\#}$. With ARDC's.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/ F	latch num	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'c	D'n'C$^{\#}$
bakery1	F	56	T/O	7565	5367	-	609	447	-	17.6	8.0
bakery2	P	49	183	5	2	241	25	15	4.1	0.1	0.0
bakery3	P	50	2794	48	174	609	128	133	18.8	2.1	1.5
bakery4	F	58	T/O	T/O	1964	-	-	477	-	-	4.0
bakery5	F	59	T/O	T/O	1294	-	-	416	-	-	4.9
eisen1	F	35	23	6	107	36	26	73	0.3	0.3	0.5
eisen2	F	35	T/O	T/O	1150	-	-	365	-	-	3.0
ele	F	37	3504	2156	585	663	612	657	24.4	21.1	23.6
nullmodem	P	56	T/O	T/O	3375	-	-	306	-	-	2.6
peterson	F	70	4	8	176	21	42	121	0.0	0.3	1.4
philo1	F	133	T/O	T/O	385	-	-	64	-	-	0.9
philo2	F	133	T/O	T/O	267	-	-	144	-	-	2.1
philo3	P	133	T/O	1139	241	-	609	119	-	21.4	1.4
shampoo1	F	143	12	T/O	168	21	-	127	0.0	-	2.0
shampoo2	F	144	T/O	T/O	189	-	-	153	-	-	3.0
shampoo3	F	145	T/O	53	735	-	51	331	-	0.3	5.0
twoq1	P	69	12	4	23	37	14	24	0.4	0.1	0.0
twoq2	P	69	172	30	665	322	15	496	7.7	0.9	8.2

Table 6.4. Comparing EL, D'n'C, and D'n'C$^{\#}$. With ARDC's.

			Time (s)			Memory (MB)			BDD (M)		
Circuit and LTL	T/F	latch num	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'C	D'n'C$^{\#}$	EL	D'n'C	D'n'C$^{\#}$
Blackjack1	F	176	7296	2566	237	618	610	551	26.8	24.2	18.1
MSI1	P	65	T/O	T/O	51	-	-	83	-	-	2.0
MSI2	F	65	T/O	T/O	165	-	-	342	-	-	6.7
PIbus1	P	387	T/O	73	1700	-	243	539	-	3.5	13.4
PIbus2	F	385	501	292	1302	467	477	609	17.0	15.4	22.6
PPC60X1	F	67	1109	1690	651	609	611	445	20.1	22.4	10.6
PPC60X2	P	69	13459	2811	531	745	625	327	17.8	18.9	6.9

Chapter 7

FAR SIDE IMAGE COMPUTATION

In the next two chapters, we will apply the idea of abstraction followed by successive refinements to two basic decision procedures in formal verification, BDD based image computation and Boolean satisfiability check. Image computation accounts for most of the CPU time in symbolic model checking, while a Boolean SAT solver is the basic work engine in bounded model checking.

In image computation, the peak BDD size is the controlling factor for the overall performance of the algorithm. Don't Care conditions have been routinely used to the present-state variables to minimize the transition relation. However, the use of Don't Cares to the far side, or next-state variables is often ineffective. In this chapter, we present a new algorithm which computes a set of over-approximated images and apply them as the care sets to the far side of the transition relations. The minimized transition relation is then used to compute the exact image.

7.1 Symbolic Image Computation

Image computation is the most fundamental operation in BDD based symbolic fixpoint computation. It has been extensively used in sequential system optimization and formal verification. Given a state transition system, image computation is used to find all the successors of a given set of states according to a set of transitions. Existing algorithms for computing images fall into two categories: one is based on the transition function [CBM89a], and the other is based on the transition relation [GB94, RAB$^+$95, MHS00, CCJ$^+$01b, JKS02]. In this chapter, we focus on the transition relation based methods.

Except for small systems, the transition relation (TR) can not be represented by a monolithic BDD. Instead, it is usually represented by a collection of BDDs (called clusters) whose conjunction is the entire transition relation. This representation is called the *partitioned transition relation.* When the partitioned transition relation is used, the image is computed by conjoining the given set with all the transition relation clusters and then existentially quantifying the present-state variables and inputs.

Given a partitioned transition relation $T = \{T^i\}$ and a set of states D, the image is computed as follows:

$$\textsc{Img}(T, D) = \exists x, w \,.\, \bigwedge_{1 \le i \le k} T^i(x, w, y^i) \wedge D(x) \ . \tag{7.1}$$

The performance of this computation depends heavily on the size of the BDDs that represent the set of states, the transition relation, and the intermediate products during the evaluation of this quantified Boolean formula.

In the *conjoin-quantify* operation (Equ. 7.1), the way in which transition bit-relations are grouped into clusters, and the order in which variables are quantified are all important. The problem of clustering and ordering to minimize the peak size of the intermediate products is called the *quantification scheduling problem.* A technique *early quantification* if often used to exploit the fact that each T^i usually depends on a subset of the present-state variables and inputs, and some of these variables can by quantified out before all the clusters are conjoined.

Let $Q_1, ..., Q_k$ be a partition of the set $(x \cup w)$ of present-state variables and inputs, then *conjoin* and *quantify* can be interleaved as follows:

$$\begin{aligned}
\textsc{Img}(T, D(x)) = \quad & \exists Q_k \,.\{T^k(x, w, y^k) \wedge \{ \\
& \exists Q_{k-1} \,.\{T^{k-1}(x, w, y^{k-1}) \wedge \{ \\
& \ldots \\
& \exists Q_1 \,.\{T^1(x, w, y^1) \wedge D(x)\}\}\}\} \ .
\end{aligned}$$

Temporary results produced in the middle of the computation, such as $\exists Q_1 \, . \, \{T^1(x, w, y^1) \wedge D(x)\}$, are called *intermediate products*. The peak sizes of the intermediate products are often essential in determining whether a given symbolic image computation can be completed on a given computer.

Studies on the effect of early quantification can be traced back to the early work of [TSL+90, Bur91, GB94]. The quantification scheduling problem was proved to be NP-complete in [HKB96]. A practically successful heuristic algorithm, known as IWLS95, was proposed in [RAB+95]. The algorithm goes as follows: first, a heuristic score is used to order the transition bit-relations; second, these bit-relations are linearly clustered together until the BDD size exceeds a certain threshold; finally, the clusters are ordered according to the same heuristic score. The IWLS95 algorithm is a representative of a class of linear quantification schedules. Recent progress along this line of research includes the algorithm based on the Minimum Lifetime Permutation (MLP) [MHS00, CCJ+01b] and the Fine-Grain image algorithm [JKS02]. Alternatively, image computation can be regarded as a problem of constructing an optimal parse tree for the image set. This results in more general quantification schedules [HKB96, GYAG00, CCJ+01a].

The new method we introduce in this chapter is not another heuristic for the quantification scheduling problem. Instead, it provides a higher-level framework that can be implemented on top of any of these heuristics.

Exact or approximate reachable states have been commonly used as don't cares in symbolic model checking to help the pre-image computation. Transitions from unreachable states can be added or removed in order to reduce the BDD size of the transition relation without changing the results of fixpoint computations restricted to reachable states. The *constrain* and *restrict* operators [CBM89b, CM90] are often used in this context to accomplish the BDD minimization. Both of these two operators are specific instances of a more general operation called the *generalized cofactor* [SHSVB94, HBLS98]. A generalized cofactor of a function T with respect to a set R, denoted by $T' = T \Downarrow R$, can be any characteristic function in the interval

$$(T \wedge R) \leq T' \leq (T \vee \neg R) \ .$$

An important property of the generalized cofactor is

$$T' \wedge R = T \wedge R \ .$$

Generalized cofactors heuristically make the choice so that the BDD of T' is minimized in some sense. Therefore, the operation indicated by $(T \Downarrow R)$ is called BDD minimization. In practice, BDD minimization must be applied very carefully for it to be effective. When R has a large BDD or when R contains many variables that do not appear in T, BDD minimization using either *restrict* or *constrain* is ineffective. This is precisely the case when one tries to minimize a sub-relation $T^i(x, w, y^i)$ with respect to the set R of (approximate) reachable states in next-state variables, because T^i often contains few next-state variables, but the set of R contains most of the next-state variables. Previously, simplification of the transition relation by applying reachability don't cares to the *far side* has not been in common use.

7.2 The Far Side Image Algorithm

In this section, we present a new image computation algorithm which applies approximate reachability don't cares to the next-state variables of the transition relation instead of the customary present-state variables. Two problems may arise when one tries to modify the *far side* of the transition relation: First, if a transition is added from a reachable state into a non-reachable state, the result of image computation is changed. Second, minimizing the BDD representation of the transition relation on the far side with the entire approximate reachable states is not effective due to the reason given above. Both problems are solved by the proposed algorithm. For the first problem, we show how the error states introduced by spurious transitions can be eliminated from the result of each image computation. To solve the second problem, we use local approximations of the reachable states, which are practically effective at simplifying the transition relation representations.

The new algorithm, called FarSideImg, is presented in Figure 7.1. The algorithm takes as arguments the partitioned transition relation $\{T^i\}$ and the set D of states, and returns the exact image set of the given states. In the pseudo code, the procedure Image represents a generic image computation procedure, which computes the image using Equ. 7.1. Given the appropriate quantification scheduling, the procedure Image can represent any of the transition relation based image computation methods described in [RAB$^+$95, MHS00, CCJ$^+$01a, JKS02]. In this sense, our FarSideImg algorithm can be built on top of any transition relation based quantification scheduling.

Since the transition relation T is a conjunction of the individual transition relation clusters, each cluster T^i is considered an over-approximation of T. Based on this observation, we first compute a series of upper-bound images, one for each transition relation cluster, as follow,

$$R_i^+(y^i) = \exists x, w \,.\, T^i(x, w, y^i) \wedge D(x) \ .$$

Since D does not contain any input variable, w can be existentially quantified out of T^i before it is conjoined with D. A similar argument is applied to the present-state variables that appear only in D (represented by Q_A) and those that appear only in T^i (represented by Q_B). The often small BDD size of T^i and the early quantifications of w, Q_A, and Q_B make R_i^+ much easier to compute than the exact image R.

Next, the BDD of each T^i is minimized with respect to the corresponding approximated image. It is important to notice that we could have used the set $\bigwedge_i R_i^+$ instead of R_i^+ to minimize T^i. In theory, a smaller care set (or a larger don't care set as in this case) provides more degree of freedom for minimization. However, the subsets of next-state variables

```
FarSideImg($\{T^i\}, D$) {

1   for each  $i \in \{1, ..., k\}$ do
2       $x_T \leftarrow$ present-state variables in the support of $T^i$
3       $x_D \leftarrow$ present-state variables in the support of $D$
4       $Q_A \leftarrow \{x_D\} \setminus \{x_T\}$
5       $Q_B \leftarrow \{x_T\} \setminus \{x_D\}$
6       $Q_C \leftarrow \{x_T\} \cap \{x_D\}$

7       $R_i^+ = \exists Q_C . (\exists w, Q_B . T^i) \wedge (\exists Q_A . D)$

8       $\widehat{T^i} = T^i \Downarrow R_i^+$
9   od

10  $\widehat{R} =$ Image $(\{\widehat{T^i}\}, D)$
11  $R = \widehat{R} \wedge \bigwedge R_i^+$                    // clipping

12  return $R$

}
```

Figure 7.1. The Far Side image computation algorithm.

in different $R_i^+(y^i)$ are disjoint. No next-state variable of $R_j^+(y^j)$, where $j \neq i$, appears in $T^i(x, w, y^i)$. This makes the minimization of T^i with respect to the set $\bigwedge R_i^+$ ineffective. On the other hand, the local approximation $R_i^+(y^i)$ contains only the next-state variables of $T^i(x, w, y^i)$, and both of them typically depend only on a small subset y^i of the next-state variables. Heuristic algorithms like *constrain* and *restrict* perform much better in practice when minimization is with respect to $R_i^+(y^i)$. Among the two operators, *restrict* is more robust because it prevents unwanted BDD variables from appearing in the result. Therefore, we will use *restrict* in our implementation and experimental investigation.

The minimized transition relation cluster $\widehat{T^i}$ is a characteristic function within the interval

$$(T^i \wedge R_i^+) \leq \quad \widehat{T^i} \quad \leq (T^i \vee \neg R_i^+) \ .$$

Minimization can be regarded as adding or removing transitions pointing to $\neg R_i^+$, as is illustrated in Figure 7.2. It may add, for instance, transitions that are pointing to $\neg R_i^+$, as represented by the dotted lines.

Likewise, it may remove from T^i transitions that are pointing to $\neg R_i^+$, as described by the solid line marked by a cross.

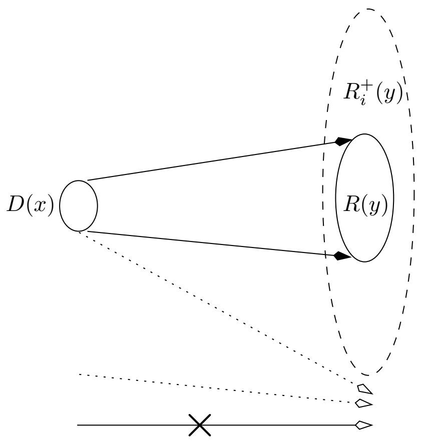


Figure 7.2. Minimizing the transition relation.

Finally, we compute another over-approximation of the overall image, $\widehat{R}$, by applying the generic image computation to the minimized transition relation. $\widehat{R}$ contains all the states of the exact image (represented by R) and possibly some states in $\neg R_i^+$ due to the added transitions. We use the clipping operation of Line 11 to get rid of those error states by conjoining $\widehat{R}$ with all the other over-approximations.

The following theorem establish the correctness of the new algorithm.

THEOREM 7.1 FARSIDEIMG *computes the same image set as* IMAGE *does. That is, given a partitioned transition relation* $\{T^i\}$ *and a set* D *of states,*

$$\textsc{FarSideImg}(\{T^i\}, D) = \textsc{Image}(\{T^i\}, D) \ .$$

PROOF: *Let the result of* IMAGE$(\{T^i\}, D)$*, as described by Equ. 7.1, be denoted by* $R(y)$*. Since the images computed in Lines 2-7 are over-approximations, we have*

$$R_i^+(y^i) \wedge R(y) \quad = \quad R(y) \ .$$

Because of the definition of generalized cofactors,

$$\widehat{T^i} \wedge R_i^+ \quad = \quad T^i \wedge R_i^+ \ .$$

By Lines 10-12 in Figure 7.1

$$\begin{aligned}
&\textsc{FarSideImg}(\{T^i(x,w,y^i)\}, D(x))\\
&= \widehat{R}(y) \wedge \bigwedge R_i^+(y^i)\\
&= \textsc{Img}(\bigwedge \widehat{T}_i(x,w,y^i), D(x)) \wedge \bigwedge R_i^+(y^i)\\
&= \{\exists x, w \,.\, \bigwedge \widehat{T}^i(x,w,y^i) \wedge D(x)\} \wedge \bigwedge R_i^+(y^i)
\end{aligned}$$

because $R_i^+(y^i)$ does not depend on x and w

$$= \exists x, w \,.\{\bigwedge \widehat{T}^i(x,w,y^i) \wedge R_i^+(y^i) \wedge D(x)\}$$

by the property of the generalized cofactor

$$\begin{aligned}
&= \exists x, w \,.\{\bigwedge T^i(x,w,y^i) \wedge R_i^+(y^i) \wedge D(x)\}\\
&= \{\exists x, w \,.\, \bigwedge T^i(x,w,y^i) \wedge D(x)\} \wedge \bigwedge R_i^+(y^i)\\
&= \textsc{Img}(\bigwedge T^i(x,w,y^i), D(x)) \wedge \bigwedge R_i^+(y^i)\\
&= R(y) \wedge \bigwedge R_i^+(y^i)\\
&= R(y)\\
&= \textsc{Img}(\bigwedge T^i(x,w,y^i), D(x))
\end{aligned}$$

7.3 Experiments

We have implemented the FarSideImg procedure in the symbolic model checker VIS 2.0 [B$^+$96, VIS], on top of both the MLP image computation algorithm [MHS00] and the Fine-Grain image computation algorithm [JKS02]. The new algorithm was compared with the standard MLP algorithm and Fine-Grain image computation algorithm in the reachability analysis of 35 circuits from the public domain as well as industry. The "S" circuits come from the ISCAS'89 benchmark [ISC], the "D" circuits come from industry, and the others come from the VIS verification benchmark [VVB]. All the experiments were conducted on an IBM IntelliStation with a 1.7 GHz Pentium IV CPU and 2GB of RAM. The data size limit for each process was set to 750MB.

Table 7.1 shows the comparison of the run time and memory usage of the Far Side algorithm and MLP, with dynamic variable reordering method "sift". The image cluster threshold is set to the default value, 5000. Columns 1-3 are the name, the number of binary state variables, and the number of inputs of each circuit. Columns 4-6 compare the CPU time; Columns 7-9 compare the peak number of live BDD nodes during the image computations. Note that none of the two methods can complete the last 5 circuits, for which the run time and peak live BDD nodes were up to the last step reached by both methods (indicated by the number in parentheses in Column 1). The data of *D14*, for instance, were up to 12 steps, as indicated by (12) in Column 1. Within the 8 hours time limit, FarSideImg was able to finish one more steps than MLP (indicated by [13] in Column 5).

The total run time of the 35 examples was 171,876 seconds for the original MLP, and 114,710 seconds for FarSideImg. Overall, this is a 33% improvement for FarSideImg. However, note that MLP ran out of time on *am2901* and *palu*, which means that the 33% win is a lower bound.

Another way to analyze the data is to partition the examples into groups and compare the performance on different groups. Group "easy" consists of circuits whose reachability analysis can be finished within 15 minutes (the first 15 circuits); Group "hard" consists of circuits whose reachability can be finished by at least one method within 8 hours; Group "harder" consists of the rest of the circuits. The average run time and the geometric mean of the peak live BDD nodes of the two algorithms are compared separately for the three groups as follows: These data show that the run time and the peak BDD size for Groups "hard" and "harder" average an order of magnitude larger than those for Group "easy". On the "easy" problems, FarSideImg does not win because of its additional overhead in approximation and refinement. On the "hard"

	Group "easy"			Group "hard"			Group "harder"		
	MLP	FarSide	%	MLP	FarSide	%	MLP	FarSide	%
CPU(s)	281	280	0	6871	4311	+37	14233	9972	+30
BDD(k)	157	161	-2	1181	845	+25	3348	2729	+18

and "harder" problems, controlling BDD size by applying Don't Cares to the far side inside image computation pays off.

Note that in one anomalous "hard" circuit, *prolog*, MLP outperforms FarSideImg by more than a factor of three. We believe that the anomaly is due to the noise introduced by the BDD dynamic variable reordering during reachability analysis. To verify this conjecture, the BDD orders at the end of reachability analysis were stored, and with these fixed variable orders we re-ran all the experiments. (For those "harder" circuits whose reachability analysis can not be finished, the default fixed variable orders generated by VIS's *static_order* command were used.) The results are shown in Table 7.2.

With the fixed orderings, some circuits run much faster (such as *prolog*), some run much slower (*s5378opt* is about 4 times slower), and some run out of time (such as *s3271*). It is important to notice that for the anomalous circuit *prolog*, FarSideImg has a marginal win (94 seconds vs. 104; 7200k BDD nodes vs. 9967k). This confirms that the anomality in Table 7.1 was due to noise introduced by dynamic variable reordering.

With the fixed orders, the average run time (among those completed) was 576 seconds for MLP and 607 seconds for FarSideImg. More importantly, there are now 3 circuits in Group "hard" that can no be completed by either method because they run out of memory (M/O). The peak number of live BDD nodes are often an order of magnitude higher than those in Table 7.1. Given the fact that finding a good fixed order is hard in practice, it is generally accepted that dynamic variable reordering is required when dealing with industrial-strength circuits. Therefore, we claim that the data with dynamic reordering is more significant.

Our experiments with the Fine-Grain image algorithm of [JKS02] demonstrated a similar performance improvement for FarSideImg.

Table 7.1. Comparing FarSideImg and MLP with dynamic variable reordering.

			CPU (s)			Peak BDD nodes (k)		
Design	regs	inputs	MLP	FarSide	%	MLP	FarSide	%
D12	48	16	6	7	-16	204	197	+ 3
abs_fabr	87	21	26	20	+25	43	46	- 5
D23	85	22	11	11	0	24	24	0
nosel	128	65	30	32	- 5	57	53	+ 7
bpb	36	9	33	53	-60	94	108	-14
shampoo	140	21	55	79	-44	91	98	- 7
soap	140	11	73	85	-17	101	97	+ 3
3_proc	62	18	103	106	- 2	213	183	+13
soapLtl3	142	11	360	329	+ 8	341	296	+13
s1512	57	29	364	435	-19	91	89	+ 2
Feistel	293	68	541	567	- 4	159	229	-43
D5	319	24	545	522	+ 4	250	301	-20
D1	101	76	556	452	+18	665	610	+ 8
s4863o	88	35	582	510	+12	402	402	0
cps1364o	134	97	598	672	-12	421	447	- 6
D21	92	6	626	610	+ 2	466	474	- 1
D2	94	6	1024	862	+15	765	745	+ 2
cps1364	231	97	1198	971	+18	363	372	- 2
s4863	104	49	1274	1037	+18	749	602	+19
D4	230	22	1370	1259	+ 8	540	493	+ 8
icctl	62	27	1462	1934	-32	1503	1515	0
s5378opt	121	35	1476	552	+62	508	335	+34
FIFOs	142	7	2129	1907	+10	1098	1021	+ 6
prolog	136	36	2443	7907	-223	1935	2287	-18
s3271	116	26	2627	1788	+31	1085	820	+24
s1269	37	18	3513	2958	+15	3588	3529	+ 1
D22	140	20	9351	12821	-37	3356	2834	+15
s3330	132	40	10733	2382	+77	3110	2922	+49
am2901	68	27	>28800	4827	>+83	-	2849	-
palu	37	10	>28800	19153	>+33	-	8985	-
D14 (12)	96	21	19978	6099 [13]	+69	4833	3066	+63
D15 (31)	106	31	12021	9795 [35]	+19	5855	4435	+24
D16 (16)	531	16	11264	6845 [17]	+40	3142	3142	0
D18 (23)	507	200	11994	11458	+ 4	1621	1385	+14
D20 (7)	562	31	15910	15666	+ 1	2926	2561	+12

Table 7.2. Comparing FarSideImg and MLP with fixed variable ordering.

Design	regs	input	CPU (s) MLP	CPU (s) FarSide	CPU (s) %	Peak BDD nodes (k) MLP	Peak BDD nodes (k) FarSide	Peak BDD nodes (k) %
D12	48	16	1	1	0	67	74	- 9
abs_fabr	87	21	36	35	+ 2	3	2	+11
D23	85	22	1	1	0	58	63	- 7
nosel	128	65	1	2	-66	0.02	0.03	-43
bpb	36	9	66	65	+ 2	0.5	0.5	0
shampoo	140	21	14	21	-53	1	1	0
soap	140	11	16	32	-101	925	1040	-12
3_proc	62	18	10	14	-42	1092	532	+51
soapLtl3	142	11	55	99	-81	4760	4836	- 1
s1512	57	29	837	1120	-33	24085	23469	+ 2
Feistel	293	68	4	5	-29	744	684	+ 8
D5	319	24	94	114	-21	4665	4158	+10
D1	101	76	82	87	-6	8469	8469	0
s4863o	88	35	57	54	+ 5	646	646	0
cps1364o	134	97	19	21	-11	2080	1993	+ 4
D21	92	6	224	305	-35	4649	3670	+21
D2	94	6	248	326	-31	3268	3149	+ 3
cps1364	231	97	25	24	+ 2	2021	1572	+22
s4863	104	49	59	35	+39	822	867	- 5
D4	230	22	79	117	-47	1035	1096	- 5
icctl	62	27	115	134	-15	3084	2268	+26
s5378opt	121	35	6960	6593	+ 5	24160	20221	+16
FIFOs	142	7	444	424	+ 4	4252	4710	-10
prolog	136	36	104	94	+ 9	9967	7200	+27
s3271	116	26	M/O	M/O	-	M/O	M/O	-
s1269	37	18	2668	2603	+ 2	44928	44928	0
D22	140	20	2414	3158	-30	1736	1716	+ 1
s3330	132	40	931	913	+ 1	24735	24736	0
am2901	68	27	M/O	M/O	-	M/O	M/O	-
palu	37	10	M/O	M/O	-	M/O	M/O	-
D14	96	21	M/O	M/O	-	M/O	M/O	-
D15	106	31	M/O	M/O	-	M/O	M/O	-
D16	531	16	M/O	M/O	-	M/O	M/O	-
D18	507	200	M/O	M/O	-	M/O	M/O	-
D20	562	31	M/O	M/O	-	M/O	M/O	-

7.4 Discussion of Hypothesis

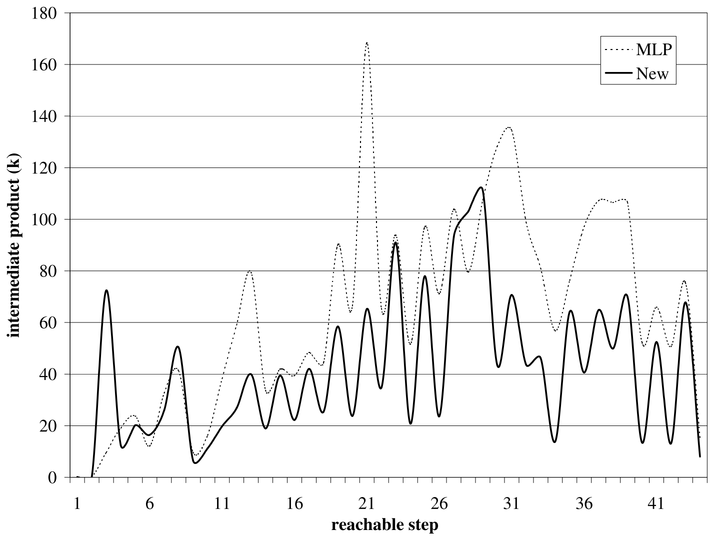
Our hypothesis is that, the performance gain of FarSideImg is due to the reduction of peak BDD size inside the conjoin-quantify operation. However, FarSideImg focuses on minimizing the BDDs used inside the image computation, not the overall BDDs used in the reachable analysis. Therefore, the overall BDD data in the previous two result tables are less informative, since a large part of the BDDs are used for representing the accumulated reachable states. The set of accumulated reachable states can become quite large near the end of the reachability analysis. Therefore, we also performed experiments that attempted to measure data more relevant to the hypothesis.

In Figure 7.3, FarSideImg (solid line) and MLP (dotted line) are compared on two different parameters: the BDD size of the intermediate products and the peak number of live BDD nodes. Note that the latter includes the BDDs representing the accumulated reachable states. The horizontal axis shows the image steps, from 1 to 43, indicating the sequential depth of 43. The upper figure shows that except for a few iteration steps (e.g., Steps 2,3,6 and 8), the minimization is effective at reducing the maximum BDD size of the intermediate products.

Since run times are determined primarily by the maximum BDD size of the intermediate products, the upper part in Figure 7.3 is more instructive in explaining the reason for the speed-up achieved by FarSideImg. In Figure 7.3, the peak occurs at iteration 21, where the MLP size is about 3 times larger than the FarSideImg size. The FarSideImg size peaked near iteration 29, at which the MLP size was about the same. The curve with fixed variable orders is similar.

Figure 7.4 shows the effect of the transition relation minimization by FarSideImg, i.e., the BDD size reduction in percentage at different steps of the reachable analysis. From top down, these data are for circuits *s5378opt*, *prolog*, and *s3271*. (Data for the other 32 circuits are similar.) Each graph has two curves: one for dynamic variable reordering, and the other for fixed ordering. Note that 50% on the curve means that the BDD size of the minimized transition relation is half of the BDD size of the original one. For the first few iterations, the reduction in TR size is substantial. As the iteration count grows, the size reductions saturate at a marginal value (0 to 40%).

In the saturation phase (the right side of the curves), the reductions are greater when a fixed ordering is used. The data for *s5378opt* in Tables 7.1 and 7.2 show that even though the reductions never fell to less than 30%, reachability analysis is 5 times slower for MLP with fixed variable ordering than with dynamic variable reordering. This might appear to be anomalous since we have attributed time reductions with



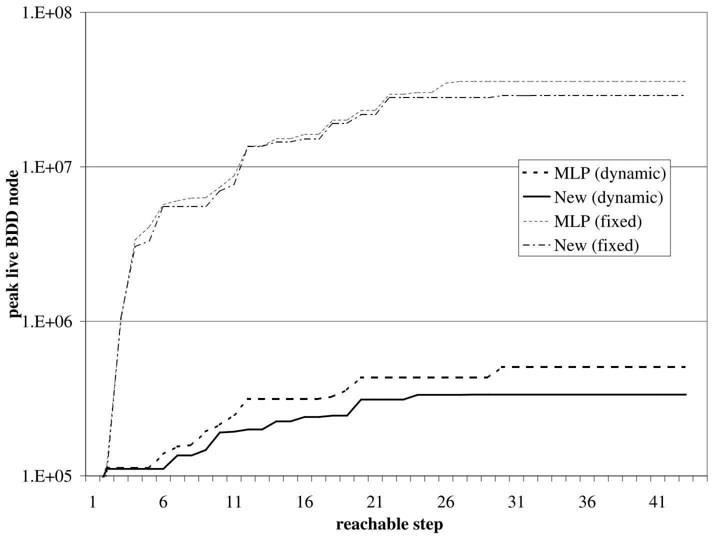
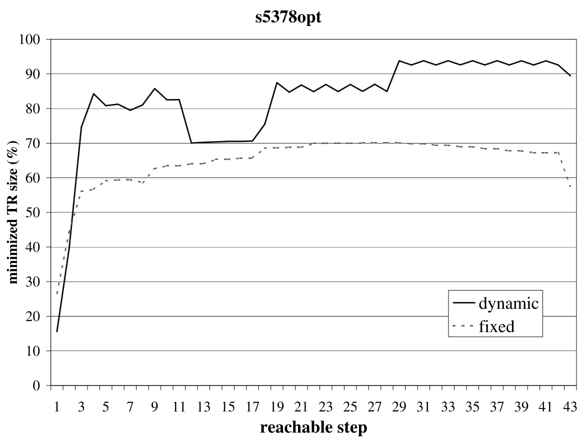
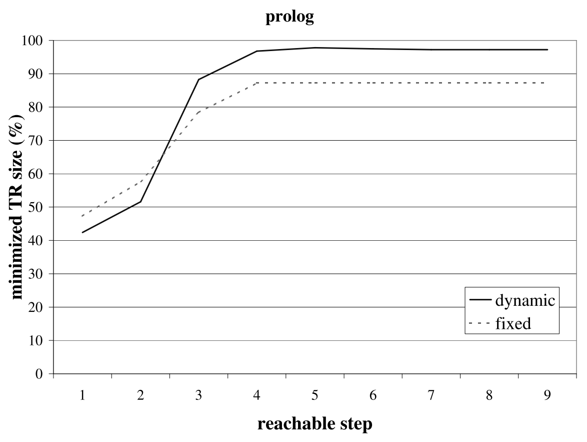


Figure 7.3. s5378opt: The upper part is the BDD size of the intermediate products at different steps during the reachability analysis; the lower part is the total number of live BDD nodes, including BDDs representing the accumulated reachable states.





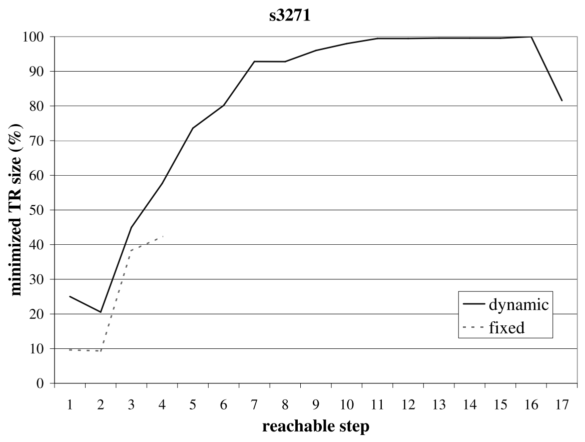


Figure 7.4. The BDD size reduction of the transition relation, in terms of the ratio of the BDD size of minimized transition relation to the original BDD size.

BDD size minimization. However, note that these data are percentages, not absolute values. The size of the minimized transition relation for fixed variable ordering is still much larger than the size of the minimized transition relation for dynamic variable ordering, as indicated by the minimized absolute values in the lower part of Figure 7.3.

The plateaus for *s5378opt* correspond to calls by the BDD manager to the reordering routine (these occurred at iterations 10, 17 and 27). In between these calls, the reductions follow a saturating pattern similar to the curves for fixed BDD ordering. Sometimes there is a final phase of increased reduction (the right side of the curves), due to the fact that image size decreases near the end of the reachability analysis. (A smaller image makes a better constraint for minimization.)

For *prolog*, reachability analysis is more than an order of magnitude faster with fixed ordering. (The size of the transition relation for fixed ordering is about 2 times larger than for dynamic variable reordering.) This is a case where the BDD reordering itself takes a larger proportion of the time.

The bottom part of Figure 7.4 pertains to circuit *s3271*. With the fixed variable ordering, this circuit can complete only 4 iterations before running out of the 750MB memory, whereas with dynamic variable reordering, it can complete all 17 iterations in only about 30 minutes. Thus even though dynamic variable reordering makes it difficult to isolate the effects of algorithmic improvement, it appears to be the only viable option for some hard models.

To summarize, the performance improvement of the FarSideImg algorithm based on compositional BDD minimization is significant on average, and especially significant on difficult circuits. For circuits requiring more than 15 minutes to complete the reachability analysis, FarSideImg, implemented on top of MLP, is significantly faster than the standard MLP in 16 out of the 19 cases. It is reasonable, therefore, to conclude that the new method is more robust in large industrial-strength applications.

Chapter 8

REFINE SAT DECISION ORDERING

In bounded model checking, the series of SAT problems for checking the existence of finite-length counterexamples are highly correlated. This strong correlation can be used to improve the performance of the SAT solver. The performance of modern SAT solvers using the DLL recursive search procedure depends heavily on the variable decision ordering. In this chapter, we propose a new algorithm to predict a good variable decision ordering based on the analysis of unsatisfiability proofs of previous SAT instances and apply it to the current SAT instance. By combining the predicted ordering with the default decision heuristic of the SAT solver, we can achieve a significant performance improvement for SAT based bounded model checking.

8.1 Unsatisfiability Proof as Abstraction

In bounded model checking, the existence of a finite-length counterexample is formulated into a SAT problem. The Boolean formula that encodes a BMC instance can be translated into the CNF format. The satisfiability of a CNF formula can be decided by the Davis-Longeman-Loveland (DLL [DLL62]) recursive search procedure, which has been used by many modern SAT solvers. In the DLL procedure, the basic steps are making decisions (assigning values to free variables) and propagating the implications of these decision on the subformulae.

Like many other search problems, the order in which these Boolean variables are assigned, as well as the values assigned to them, affects a SAT solver's performance significantly. In fact, different variable decision orderings imply different binary search trees, whose sizes and corresponding search overheads can be quite different. Because of the NP-completeness of the SAT problem, finding the optimal decision ordering is unlikely to be easier; modern SAT solvers often use heuristic algorithms to compute decision orderings that are "good enough" for common cases. For instance, the SAT solver Chaff [MMZ^{+}01] uses the Variable State Independent Decaying Sum (VSIDS) heuristic. Some of the pre-Chaff decision heuristics can be found in the survey paper by Silva [Sil99].

SAT solvers are designed to deal with general CNF formulae. Using them to decide the SAT problems encountered in bounded model checking requires the translation of the resulting formulae into CNF. Useful information that is unique to BMC is lost during this translation. In particular, the series of SAT problems that BMC produces for increasing counterexample length is made up of problems that are highly correlated; this means that information learned from previous SAT problems can be used to help solving the current problem. In this chapter, we propose a new algorithm to predict a good variable ordering for the SAT problems in BMC. This linear ordering is computed by analyzing all previous unsatisfiable instances; the ordering is successively refined as the BMC unrolling depth keeps increasing. We also give two different approaches (static and dynamic) to apply this linear ordering. In both cases, the newly created ordering is combined with the default variable decision heuristic of the SAT solver to make the final decisions.

Recall that whenever a Boolean formula is proved to be unsatisfiable by a SAT solver, there exists a final conflict that cannot be resolved by backtracking. Such a final conflict, represented by an empty clause, is the unique root node of a resolution subgraph.

An example of the resolution subgraph is shown in Figure 8.1. The leaves of this graph are the clauses of the original formula (represented by squares on the left-hand side), and the internal nodes are the conflict clauses added during the SAT solving (represented by circles in the middle). By traversing this resolution graph from the unique root backward to the leaves, one can identify a subset of the original clauses that are responsible for this final conflict. This subset of original clauses, called the *unsatisfiable core* [ZM03, GN03], is sufficient to imply unsatisfiability. In Figure 8.1, the conflict clauses that contribute to the final conflict are marked gray; the black squares at the left-hand side form the subset of the original clauses in the unsatisfiable core. The unsatisfiability proof includes both the UNSAT core and the conflict clauses involved in deriving the final empty clause.

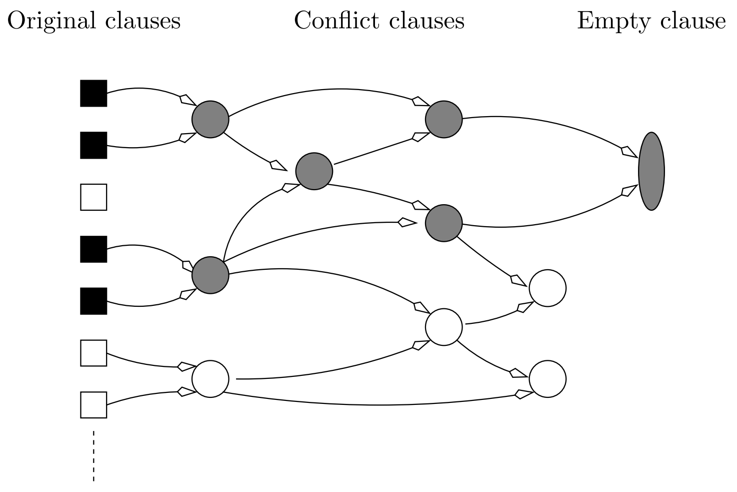


Figure 8.1. Illustration of the resolution graph.

Because of the connection between the CNF formulae and the model (circuit), the unsatisfiable core of an unsatisfiable BMC instance implies an abstraction of the model. Let the abstraction be represented as $\widehat{M} = \langle \widehat{V}, \widehat{W}, \widehat{I}, \widehat{T} \rangle$, where $\widehat{V}$, and $\widehat{W}$ are subsets of V and W, respectively. The set of initial states and the transition relation of the abstract model, denoted by $\widehat{I}$ and $\widehat{T}$, respectively, are existential abstractions of their counter-parts,

$$\widehat{I}(\widehat{V}) = \exists (V \setminus \widehat{V}). I(V) \ ,$$
$$\widehat{T}(\widehat{V}, \widehat{W}, \widehat{V}') = \exists (V \setminus \widehat{V}), (W \setminus \widehat{W}), (V' \setminus \widehat{V}').\ T(V, W, V') \ .$$

In other words, $\widehat{M}$ is constructed from M by removing some state variables, inputs, and logic gates. When a state variable $v \in (V \setminus \widehat{V})$ is quantified out, the logic gates that belong to the fan-in cone of v but not the fan-in cones of other state variables are also removed from $\widehat{T}$. Since all the clauses of the CNF formula come from the model and the LTL property, a subset of these clauses induces a subset of registers, inputs, and logic gates of the model. This subformula implicitly defines an abstraction.

We illustrate this connection using the example in Figure 8.2. The top squares in this figure represent the original clauses of the CNF formula, and the bottom is one copy of the circuit structure. There are k copies of the unrolled circuit structure in a depth-k BMC instance, one for each time frame. Assume that the BMC instance is unsatisfiable. The black squares represent clauses in the unsatisfiable core. Each clause corresponds to some registers or logic gates of the model. A register is considered to be in the abstract model if and only if its present-state or next-state variables are in the UNSAT core. A logic gate is considered to be in the abstract model as long as any of the clauses describing its gate relation appear in the unsatisfiable core.

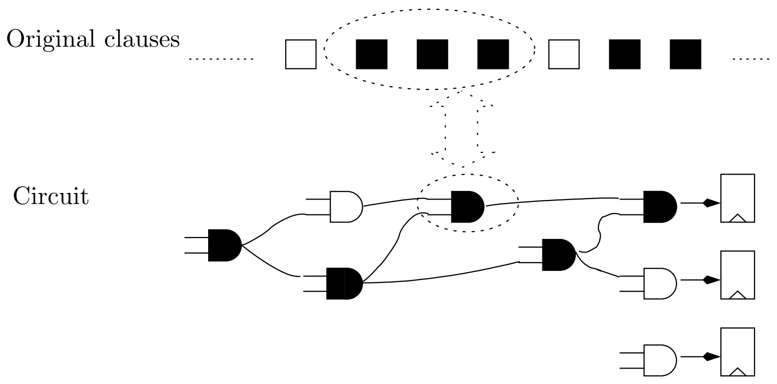


Figure 8.2. From unsatisfiable cores to abstractions.

The abstract model induced by an UNSAT core as described above is an over-approximation of the original model, because the elementary transition relations of logic gates not included in the current abstraction are assumed to be tautologies. The abstract model simulates the concrete model in the sense that, if there is no counterexample of a certain length in the abstract model, there is no counterexample of the same length in the concrete model. Had one known the current abstract model by an *oracle*, one could have applied this information to speed up the solving of the current BMC instance. The idea is to make decisions (variable assignments) only on the variables appearing in the current abstract model, since by definition, variables inside the unsatisfiable core are sufficient to prove the unsatisfiability of the BMC instance. By doing so, we are exploring a much smaller SAT search space since only the logic relations among these variables are inspected, while the other irrelevant variables and clauses are completely ignored. If the size of the abstract model is small (compared to the entire model), this restricted SAT search is expected to be much faster.

Of course, there is no way to know the current unsatisfiable core unless one solves the current SAT problem. In practice, however, the series of SAT problems produced by BMC for the ever increasing counterexample length are often highly correlated, in that their unsatisfiable cores share a large number of clauses. Therefore, abstract models extracted from previous unsatisfiable BMC instance is a good estimation of the abstract model for the current BMC instance. In bounded model checking, the vast majority of the SAT problems are unsatisfiable. For passing properties, all instances are unsatisfiable (i.e., no counterexample); for failing properties, all but the last instance are unsatisfiable. Often, there is a sufficiently large number of previous abstract models for computing an estimation of the current abstraction and refining the "estimation."

The idea of identifying important decision variables from previous unsatisfiable instances and applying them to solve the current instance is illustrated in Figure 8.3. Each rectangle represents a copy of the transition relation of the model for one time frame. The upper part of the figure is a BMC instance for the unrolling depth 3, and the lower part is a BMC instance for the unrolling depth 4. The shaded area represents the unsatisfiable core from the length-3 BMC instance. The dotted line indicates the abstraction of the model derived from an UNSAT core of the length-4 BMC instance. Note that the UNSAT core for $k = 3$ if already a good estimation of the UNSAT core for $k = 4$. Therefore, we can record variables appearing in this first UNSAT core and give them a higher priority inside the SAT solver when solving the length-4 BMC instance.

In the best-case scenario, i.e., the estimation is perfect and the previous abstraction is already sufficient for proving the unsatisfiability of the current SAT instance, no variable other than those in previous unsatisfiable cores needs to be assigned before the SAT solver stops and reports UNSAT. Even if there are some discrepancies between the estimation and the reality, we still expect a significant reduction in the size of the SAT search tree by making decisions on the variables of previ-

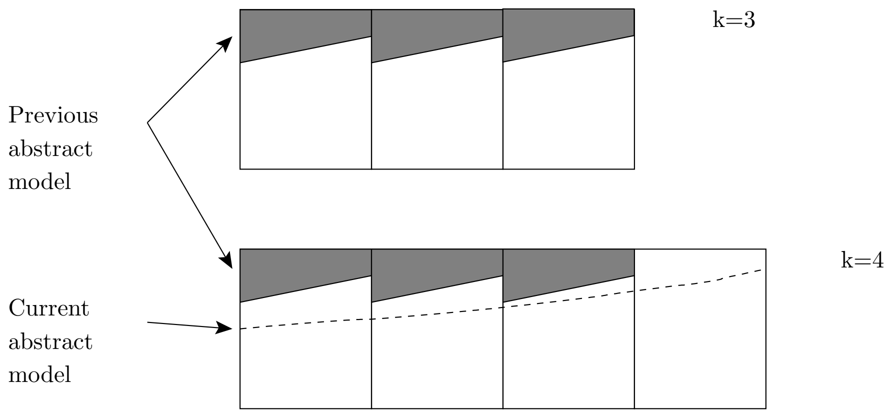


Figure 8.3. Previous abstractions to help solving the current BMC instance.

ous abstract models first. This predicted variable decision ordering can also help when the current SAT instance is indeed satisfiable, since uninteresting part of the search space will be quickly identified and pruned away through the addition of conflict clauses.

8.2 Refine the Decision Ordering

All the previous unsatisfiable SAT instances are used to predict the variable decision ordering for the current instance. We can assign a score to each Boolean variable of the SAT formula. Variables appearing in previous unsatisfiable cores are assigned higher scores. The more frequent a variable appears in previous UNSAT cores, the higher its score is. These scores are combined together in solving the current BMC instance, so that variables with higher scores are given higher priorities in the decision-making.

The augmented bounded model checking algorithm is presented in Figure 8.4. The new procedure, called RefineOrderBmc, accepts two parameters: the model M and the invariant predicate P. List *varRank* is used to store the scores of variables appearing in previous unsatisfiable cores. Integer k is the current unrolling depth. Procedure GenCnfFormula generates the CNF representation of the length-k BMC instance. The satisfiability of F is decided by the SAT procedure SatCheck, which is a Chaff-like SAT solver that also takes the predetermined ordering *varRank* as a parameter. Note that for the formula F, *varRank* is often a partial ordering, since it may not have all the Boolean variables of F.

When F is unsatisfiable, SatCheck computes the UNSAT core and returns all the variables appearing in it. This set of variables, denoted by $unsatVars$, is used to update *varRank*. The heuristic used inside UpdateRanking to update the variable ranking will be explained later. After the unrolling depth k is increased, the updated ordering is applied to SatCheck again. The entire BMC procedure terminates as soon as F becomes satisfiable, in which case the property $\mathsf{G}\,P$ is proved to be false, or the unrolling depth k exceeds a predetermined completeness threshold, in which case the property is declared true.

Inside the procedure UpdateRanking, all the Boolean variables that have ever appeared in any previous unsatisfiable cores are assigned non-zero scores. In this scoring scheme, all previous unsatisfiable cores are used to determine the current variable ordering, but we give a larger weight to the latest UNSAT cores. Let `bmc_score`(x) be the score for the variable x; then we have

$$\texttt{bmc\_score}(x) = \sum_{1 \le j \le k} \textsc{in-unsat}(x, j) \times j \ ,$$

where k is the current unrolling depth, and $\textsc{in-unsat}(x, j)$ returns 1 if variable x appears in the unsatisfiable core of j-th BMC instance, and returns 0 otherwise. The ranking of these variables is based on the `bmc_score`—the one with a higher score gets the higher priority. This

```
REFINEORDERBMC ($M$, $P$) {

    Initialize the list $varRank$;
    for (each $k \in \mathbb{N}$)    {
        $F$ = GENCNFFORMULA ($M, P, k$);
        ($isSat, unsatVars$) = SATCHECK ($F, varRank$);
        if ($isSat$)
            return FALSE;
        else
            UPDATERANKING ($unsatVars, varRank$);
    }
    return TRUE;
}
```

Figure 8.4. Refining the SAT decision order in bounded model checking.

is designed for the following two observations: (1) one wants to give preference to the variables appearing in most recent unsatisfiable cores, which usually have higher correlation to the current one; and (2) one wants to avoid relying exclusively on any particular unsatisfiable core, because it may not always be an accurate estimation of the current one.

In order to generate the unsatisfiable core after the SAT solver reports UNSAT, additional bookkeeping is required during the SAT solving process. In particular, for every conflict clause (new clause learned from a conflict), its complete *conflict graph* must be recorded to memorize all the clauses that are responsible for it. Since some of these clauses may be conflict clauses themselves, at the end, one may have a *Conflict Dependency Graph (CDG)* [CCK$^+$02], in which one conflict graph depends on another. By definition, a CDG is a directed acyclic graph. In the presence of a CDG, the unsatisfiable core can be easily identified by traversing the CDG from the final conflict backward to the original clauses.

To maintain a CDG, we need to record all the conflict clauses added throughout the SAT search. Although some of them may not belong to the UNSAT core, there is no way of identifying them in advance. However, many modern SAT solvers, including Chaff, have a feature of periodically removing conflict clauses that are regarded as irrelevant (or less relevant) to the current search. For example, if a conflict clause has not be used for a considerably long time, it will be deleted from the clause database. This can reduce the total number clauses that need to be inspected during BCP. Disabling this feature may slow down the SAT solver significantly when solving difficult SAT problems. On the other hand, if conflict clauses are allowed to be deleted, the dependency relation in the CDG may be broken, which makes the construction of a complete unsatisfiable core impossible.

In order to generate a complete unsatisfiable core without slowing down the search at the same time, we choose to maintain separately a simplified version of the CDG. Our observation is that the details of the conflict clauses are not needed in the CDG. For the purpose of identifying the unsatisfiable core, which is a subset of the *original* clauses, only the dependency relation of the conflict clauses are required. Therefore, our simplification is mainly on representing the conflict clause—instead of recording both the literals and the depended clauses, we replace the conflict clause by a pseudo clause ID and retain only the dependency relation between the clause IDs. The use of a separate simplified CDG leaves the original clause database intact. Therefore, the periodic removal of irrelevant conflict clauses is not affected. Compared to the number of the literals in the conflict clauses, which is typically 100-200, the overhead of using an integer for the pseudo ID is small.

In practice, the additional overhead of maintaining and finally traversing the simplified CDG is relatively low. In our controlled experiments, we have found that the additional runtime is about 5% of the total run time, and the memory overhead is often negligible.

Applying previously computed variable decision ordering to the successive SAT calls requires a slight modification of the SAT solver. Since SAT solvers are different, the actual modification depends on the individual SAT solvers used in BMC. The following discussion is for the SAT solver Chaff [MMZ$^+$01]. However, the proposed method can be easily adapted to other DLL based SAT solvers. Chaff's default variable decision heuristic is called VSIDS (for Variable State Independent Decaying Sum): every literal l is associated with a score, denoted by `chaff_score(`l`)`. The initial value is the number of clauses of the original formula in which l appears, i.e., the literal counts. Every a certain number of decisions, `chaff_score(`l`)` is updated as follows:

$$\texttt{chaff\_score}(l) = \texttt{chaff\_score}(l)/2 + \texttt{new\_literal\_counts}(l) \enspace ,$$

where `new_literal_counts(`l`)` is the number of new conflict clauses containing literal l. All the variables are sorted periodically by `chaff_score`. When it is the time to make a decision on free variables, the variable l with the highest score will be selected. Depending on whether `chaff_score`(l) is larger than `chaff_score`($\bar{l}$), the variable will be assigned either 1 or 0.

The pre-computed score `bmc_score`, in principle, can either replace or be combined with `chaff_score` to determine the final ordering. However, relying exclusively on `bmc_score` may not be practical in all cases, because the score is available only for a (usually small) subset of variables, and it is for variables instead of the two phases of the same literals—both the positive and negative phase of a variable have the same `bmc_score`. Therefore, we choose to combine it with `chaff_score` in the decision-making, instead of using it as the only criterion.

Two different ways of combining the two types of scores are possible: One is called the *static* configuration, and the other is called the *dynamic* configuration. In both approaches, `chaff_score` is updated as usual and sorting of free variables inside the SAT solver is performed periodically. However, sorting in the static configuration is primarily by `bmc_score`, with `chaff_score` only as a tiebreaker. It is called static because the sorting criteria is fixed throughout the entire SAT solving process.

In the dynamic configuration, the periodic sorting is initially based primarily on `bmc_score` with `chaff_core` as a tiebreaker. However, if the estimation (of the abstract model) is found to be inaccurate, the SAT solver can automatically switch back to the default VSIDS heuristic, which sorts exclusively by `chaff_core`. The rationale behind this approach is that, by starting with the ranking by `bmc_score`, we can quickly learn important clauses and prune away a significant portion of the search space early on. On the other hand, the VSIDS heuristic is designed to favor the most recently added conflict clauses, which may eventually dominate in terms of literal counts for difficult problems. Applying VSIDS heuristic in those cases allows the search process to be driven primarily by recent conflict clauses.

The SAT problem is considered *difficult* when either the estimation of the unsatisfiable core is not accurate, or proving the unsatisfiability indeed needs almost all the variables. In both cases, the number of decisions required to solve the problem is often large; therefore, we can use the number of decisions to predict whether the problem is difficult. In the implementation of the dynamic configuration, we switch back to the VSIDS heuristic as soon as the number of decisions is greater than 1/64 of the number of original literals. (This heuristic threshold was determined by empirical studies.)

8.3 Experimental Analysis

We have implemented the RefineOrderBmc procedure on top of the bounded model checking procedure in VIS-2.0 [B$^+$96, VIS]. The back-end SAT solver is Chaff [MMZ$^+$01]. The BMC command in VIS is based on the basic encoding of BMC as in [BCCZ99] and the basic induction proof as in [SSS00]. Experimental studies were conducted on the set of IBM Formal Verification Benchmark circuits [IBM], each with an invariant property $\mathsf{G}\,P$. The experiments were performed on a 400MHz Pentium II with 1GB of RAM running Linux, with the time out limit set to 2 hours. In our experiments, the only difference between the standard BMC command and RefineOrderBmc is their SAT variable decision orderings. Trivial experiments that can be finished by both methods within 10 seconds were excluded.

Table 8.1 compares the CPU time of the new method (with both static and dynamic configurations) to the standard BMC command in VIS. The first column is the name of the model. The second column indicates whether the given property is true or false. If the experiments cannot be finished within 2 hours, we compare the CPU time taken up to the maximum unrolling depth that all methods can reach; in those cases, the maximum unrolling depth is given in the parenthesis. The next three columns give the CPU time of the standard BMC and the new method with both static and dynamic configurations.

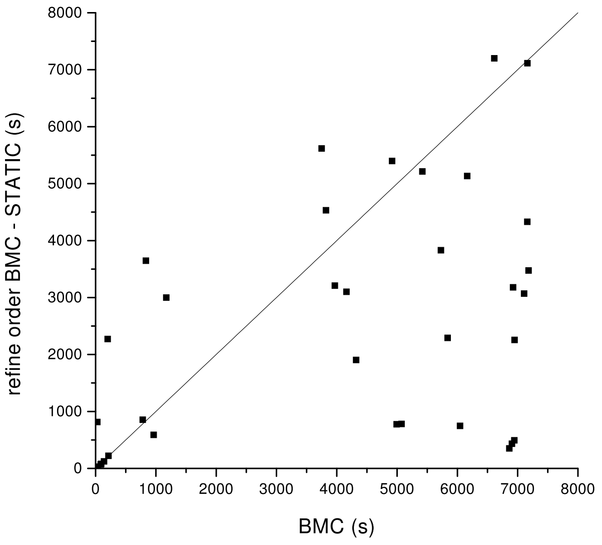
The last two rows of Table 8.1 give the corresponding total CPU time, and the overall speedup of the new methods over the standard BMC. The overall speedup of RefineOrderBcm with the static configuration is 38%; the overall speedup of RefineOrderBmc with the dynamic configuration is 42%. Out of the 37 circuits, the new method has achieved performance gains on 26 (for static) and 32 (for dynamic) circuits. The same results are also given in the scatter plots in Figure 8.5. Note that dots that are under the diagonals represent the wins by the new method.

Figure 8.6 and 8.7 show the detailed information from the SAT solver while it is solving the BMC instances of the example circuit *02_3_batch_2*. In all these figures, the horizontal axis represents the different BMC unrolling steps. The two figures in Figure 8.6 compare plain BMC with RefineOrderBmc (static) on the "maximum decision level" and the "number of decisions." The two figures in Figure 8.7 compare plain BMC with RefineOrderBmc on the "number of conflict clauses" and the "number of implications."

With the help of the predicted variable decision ordering, the size of the SAT search trees have been significantly reduced, as shown by Figure 8.6. At the right-hand side of the figures (when BMC depths are large), the reductions can be up to two orders of magnitude. In addition,

Table 8.1. Comparing BMC with and without refining the SAT decision order.

Model	True/False or (k)	BMC time (s)	refine_order_bmc time static (s)	refine_order_bmc time dynamic (s)
01_batch	F	39	25	24
02_1_batch_1	(41)	6613	7200	5677
02_1_batch_2	(28)	835	3648	894
02_3_batch_2	(65)	6944	494	476
02_3_batch_4	(65)	6906	433	475
02_3_batch_6	(59)	6861	352	368
03_batch	(F)	214	222	238
04_batch	(F	85	70	67
06_batch	F	962	589	596
11_batch_2	(29)	3820	4533	2932
11_batch_3	(28)	4160	3102	3515
14_batch_1	(35)	201	2272	287
14_batch_2	F	35	30	35
15_batch	F	12	13	12
16_1_batch	(83)	6948	2256	4537
17_1_batch_1	(264)	7161	7114	6965
17_1_batch_2	(12)	29	816	44
17_2_batch_1	(167)	7160	4331	4629
17_2_batch_2	(141)	7181	3475	3268
18_batch	(20)	1172	2999	1049
19_batch	F	139	123	108
20_batch	(28)	3748	5617	3992
21_batch	F	93	80	76
22_batch	(41)	6164	5134	3986
23_batch	(25)	3968	3209	3644
24_1_batch_1	(22)	6045	748	1182
24_1_batch_2	(22)	4992	775	1053
24_1_batch_3	(22)	5075	782	1054
25_batch	(90)	7107	3069	2922
27_batch	F	34	27	37
28_batch	F	782	855	683
29_batch	(22)	4917	5397	4270
31_1_batch_1	(21)	5728	3831	4491
31_1_batch_2	(21)	5838	2292	3552
31_1_batch_3	(21)	4321	1904	3748
31_2_batch_1	(20)	5419	5215	2660
31_2_batch_2	(19)	6924	3180	5475
Total		138,632	86,212	79,021
Percentage		100%	62%	57%



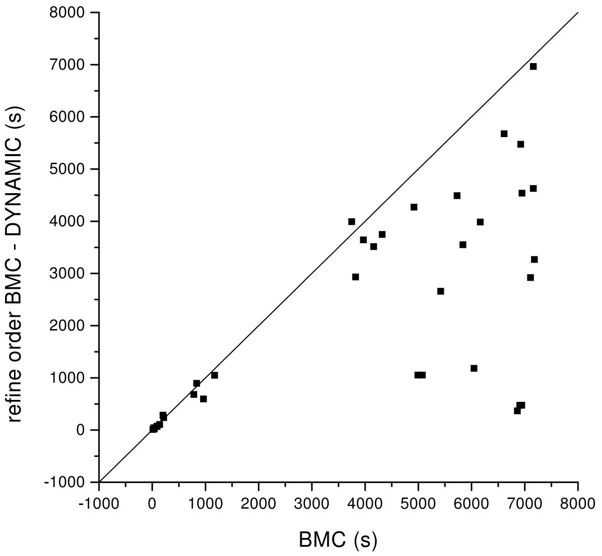
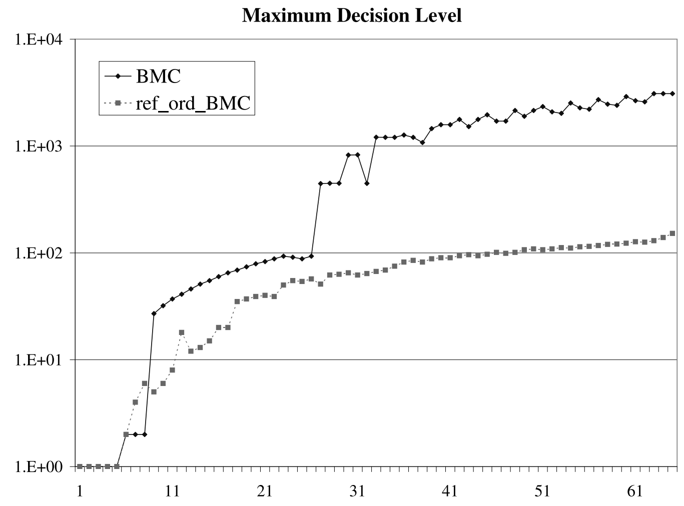


Figure 8.5. Scatter plots: plain BMC vs. BMC with the refined ordering.



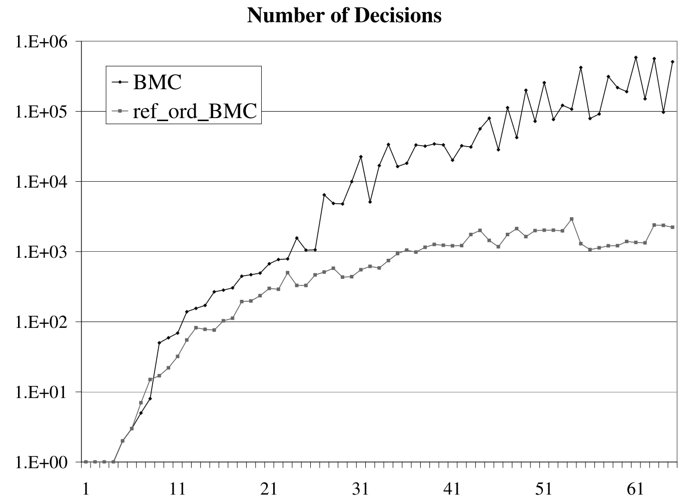
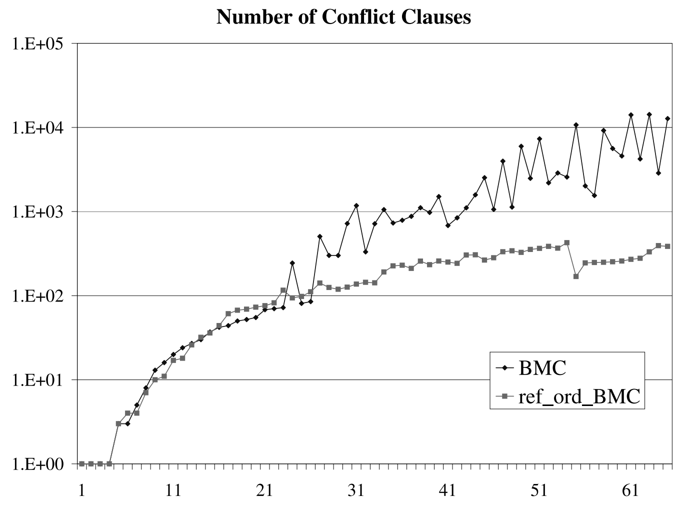


Figure 8.6. Reduction of the size of decision trees on Circuit *02_3_latch_2*: plain BMC vs. BMC with the refined ordering.



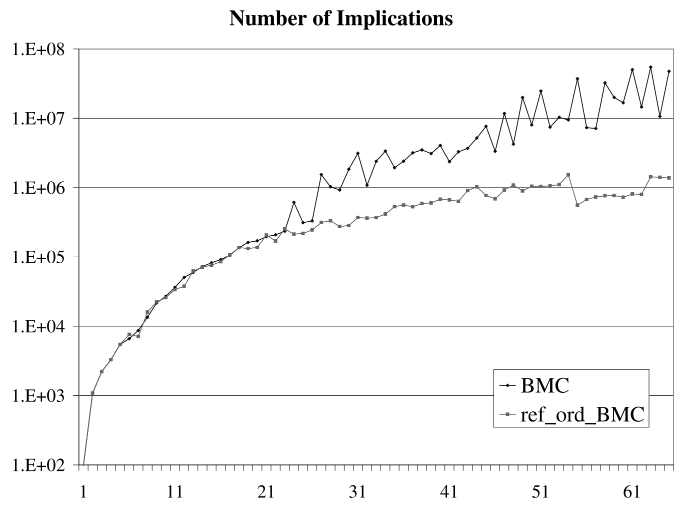


Figure 8.7. Reduction of the number of conflicts and implications on Circuit *02_3_latch_2*: plain BMC vs. BMC with the refined ordering.

the number of conflicts and the number of implications are also reduced significantly, as shown in Figure 8.7. These reductions in turn translate into shorter CPU times.

8.4 Further Discussion

We have presented a new algorithm for predicting and successively refining the variable decision ordering for SAT problems encountered in bounded model checking. The algorithm is based on the analysis of the unsatisfiability cores of previous BMC instances. We have described both the static and the dynamic configurations in applying this new ordering to the decision-making inside SAT solvers, by using the SAT solver Chaff as an example. Our experiments conducted on industrial designs have showed that the new method significantly outperforms the standard BMC. Further experimental analysis has indicated that the performance improvement is due to the reduction of the sizes of the SAT search trees.

The proposed algorithm exploits the unique characteristic of the SAT problems in BMC: the different SAT problems are highly correlated. It complements existing decision heuristics of the SAT solvers used for BMC. We believe that the same idea is also applicable to SAT based problems other than bounded model checking, as long as their subproblems have a similar incremental nature.

Tuning the SAT solver for BMC was first studied by Shtrichman in [Sht00], where a predetermined variable ordering was extracted by traversing the Variable Dependency Graph (VDG) in a topological order. The entire BMC instance can be regarded as a large combinational circuit lying on a plane in which the horizontal axis represents different time frames and the vertical axis represents different registers (or state variables). By either forward or backward traversal of the VDG, Shtrichman sorted the SAT variables according to their positions on the "time axis". In contrast, our new method sorts the SAT variables according to their positions on the other axis—the "register axis."

Information from the circuit structure was also used in previous work to help the SAT search. In [GAG+02], Ganai *et al.* proposed a hybrid representation as the underlying data structure of their SAT solver. Both circuits and CNF formulae were included in order to apply fast implication on the circuit structure and at the same time retain the merit of CNF formulae. In [GGW+03b], Gupta *el al.* applied implications learned from the circuit structure (statically and dynamically) to help the SAT search, where the implications were extracted by BDD operations. In [LWCH03], Lu *et al.* proposed to use circuit topological information and signal correlations to enforce a good decision ordering in their circuit SAT solver. The correlated signals of the underlying circuit were identified by random simulation, and were then applied either explicitly or implicitly to the SAT search. In their explicit approach, the original SAT problem was decomposed into a sequence of subproblems for the SAT solver to solve one-by-one. In their implicit approach, correlated signals were dynamically grouped together in such a way that they were most likely to cause conflicts.

The incremental nature of the BMC instances was also exploited by several incremental SAT solvers [WKS01, ES03]. These works focused primarily on how to incrementally create a SAT instance with as little modification as possible to the previous one, and on how to re-use previously learned conflict clauses. However, refining the SAT decision ordering has not been studied in these incremental solvers. Therefore, the method proposed in this chapter can be combined with the incremental SAT techniques to further improve their performance.

Chapter 9

CONCLUSIONS

The purpose of this research is to apply model checking techniques to the verification of large real-world systems. We believe that automatic abstraction is the key to bridge the capacity gap between the model checkers and industrial-scale designs. The main challenge in abstraction refinement is related to the ability of reaching the optimum abstraction, i.e., a succinct abstraction of the concrete model that decides the given property. In this book, we have proposed several fully automatic abstraction refinement techniques to efficiently reach or come near the optimum abstraction efficiency.

9.1 Summary of Results

In Chapter 3, we have proposed a new fine-grain abstraction approach to push the granularity of abstraction refinement beyond the usual state variable level. By keeping the abstraction granularity small, we add at each refinement iteration only the information relevant to verification into the abstract models. Our experience with industrial-scale designs shows that fine-grain abstraction is indispensable in verifying large systems with complex combinational logic.

In Chapter 4, we have proposed a new generational refinement algorithm GRAB, in which we use a game-based analysis of all shortest counterexample in the SORs to select refinement variables. By systematically analyzing all the shortest counterexamples, GRAB identifies important refinement variables from the local support of the current abstract model. The global guidance from all shortest counterexamples and the scalable refinement variable selection computation are critical for GRAB to achieve a higher abstraction efficiency. Compared to previous single counterexample guided refinement methods, GRAB often produces a smaller final abstract model with less run time.

In Chapter 5, we have proposed the DNC compositional SCC analysis algorithm which quickly identifies uninteresting parts of the state space of previous abstract models and prune them away before going to the next abstraction level. We also exploit the fact that the strength of an SCC or a set of SCCs decreases monotonically with refinement, by tailoring the model checking procedure to the strength of the SCC at hand. DNC is able to achieve a speed-up of up to two orders of magnitude over standard symbolic fair cycle detection algorithms, indicating the effectiveness of reusing information learned from previous abstractions to help the verification at the current level, .

In Chapter 6, we have proposed a state space decomposition algorithm in which the SCC quotient graph of an abstract model is used to decompose the concrete state space into disjunctive sets. A nice feature of this composition is that we can perform fair cycle detection in each disjunctive state subset in isolation without introducing inconclusives. We have also proposed a new guided search algorithm for symbolic fair cycle detection which can be used at the end of the adaptive popcorn-line policy in the DNC framework. Our experiments show that for large systems or otherwise difficult problems, heavy investment in disjunctive decomposition and guided search can significantly improve the performance of DNC.

In Chapters 7 and 8, we have proposed two new algorithms to improve the performance of BDD-based symbolic image computation and the Boolean SAT check in the context of bounded model checking. The two decision procedures are basic work engines of most symbolic model checking algorithms; for both of them, we have applied the general idea of abstraction followed by successive refinements. In Chapter 7, we first compute a set of over-approximated images and apply them as care sets to the far side of the transition relations; the minimized transition relation is then used to compute the exact image. In Chapter 8, we first predict a good variable decision ordering based on the UNSAT core analysis of previous BMC instances, and then use the new ordering to improve the SAT check of the current instance. Our experiments on a set of industry benchmark designs show that the proposed techniques can achieve a significant performance improvements over the prior art.

In conclusion, the new algorithms presented in this book have significantly advanced the state of the art for model checking. With automatic abstraction, model checking has been successfully applied to hardware systems with more than 4000 binary state variables. Evidence has shown that the advantages of these new techniques generally increase as the models under verification get larger. Therefore, these techniques will play important roles in verifying industrial-scale systems.

We regret not giving more results on benchmark examples with more than 1000 state variables. At the time this research was conducted, there were not many industrial-scale verification benchmarks in the public domain (and there are not many even now). Most companies in the computer hardware design and EDA industry are reluctant to share their private designs with university investigators. We sincerely hope that this situation will change in the future.

9.2 Future Directions

In this book, the quest for optimum abstraction has been put into a synthesis perspective, where refinement is regarded as a process of synthesizing the smallest deciding abstract model. An interesting open question is related to the theoretical complexity of finding the optimum abstraction. Although finding the optimum abstraction is at least as hard as model checking itself, understanding the theoretical aspect of this problem may shed light on designing more practical algorithms. According to our own experience in design and analysis of VLSI/CAD algorithms, good practical algorithms often come from formulating an optimal algorithm and then making intuitive simplifications to deal with complexity in practice.

The Grab refinement algorithm relies on the analysis of all the shortest counterexamples. An interesting extension is to find, among all the shortest counterexamples, the one counterexample with the maximum likelihood, and apply the same variable selection heuristic to this maximum likelihood path. Note that when a property fails in an abstract model, the abstract counterexamples in the synchronous onion rings may have different concretization probabilities. In some cases, especially when the property is mostly likely to be false in the concrete model, it may be advantageous to focus on those counterexample that are most likely to be concretizable.

In this book, we rely primarily on BDD-based symbolic fixpoint computation to check whether the given property holds in an abstract model. Recent advances in SAT algorithms have demonstrated the possibility of replacing BDDs with CNF formulae in fixpoint computation [ABE00, GYAG00, WBCG00, McM02]. The analysis of abstract models can also be performed by using BMC induction proof techniques; in [LWS03, LWS05], Li *et al.* have shown that SAT-based decision procedures often complement BDD based symbolic fixpoint computation in the analysis of the abstract models. Therefore, the integration of BDD-based algorithms with SAT-based algorithms for the analysis of the abstract models should lead to a more robust and powerful approach to abstraction refinement.

In the more general area of symbolic model checking, research in attacking the capacity problem can be classified into two categories. One is at the "lower level," which includes the improvement of both runtime and memory performance of basic decision procedures. The other is at the "higher level," which includes methods like abstraction refinement, compositional reasoning [AL91, Var95, McM97, McM98, HQR98], symmetry reduction [ES93, ID93], etc. There will be improvements on the lower level algorithms in the years to come; however, we believe that to bridge the existing verification capacity gap, the bulk of the improvement has to come from advances in the higher level techniques. An interesting future research direction is to apply techniques developed in the abstraction refinement framework to compositional reasoning and symmetry reduction. These methods share a common idea—simplifying the model before applying model checking to it, even though the methods of simplification are significantly different.

References

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for the Construction of Systems (TACAS)*, pages 411–425, 2000. LNCS 1785.

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ADK$^+$05] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In *International Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, page to appear, October 2005.

[AH96] R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 207–218, 1996.

[AHH96] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[AM04] N. Amla and K. L. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *Formal Methods in Computer-Aided Design (FMCAD 2004)*, pages 260–274, Austin, TX, November 2004. Springer-Verlag. LNCS 3312.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[B$^+$96] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Com-*

puter Aided Verification (CAV'96), pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Fourth Conference on Computer Aided Verification (CAV'92)*, pages 251–263, 1992.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[BCM$^+$90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[BFG$^+$93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 188–191, Santa Clara, CA, November 1993.

[BGG02] S. Barner, D. Geist, and A. Gringauze. Symbolic localization reduction with reconstruction layering and backtracking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 65–77. Springer-Verlag, July 2002. LNCS 2404.

[BGS00] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, November 2000. LNCS 1954.

[BGS05] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. *Formal Methods in System Design*, 27(2), September 2005. To appear.

[BK04] J. Baumgartner and A. Kuehlmann. Enhanced diameter bounding via structural. In *Design, Automation and Test in Europe (DATE'04)*, pages 36–41, 2004.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[BRS99] R. Bloem, K. Ravi, and F. Somenzi. Efficient decision procedures for model checking of linear time logic properties. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 222–235. Springer-Verlag, Berlin, 1999. LNCS 1633.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[BSV93] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV '93)*. Springer-Verlag, Berlin, 1993. LNCS 697.

[Bur91] J. Burch. Using BDD's to verify multipliers. In *1991 International Workshop on Formal Methods in VLSI Design*, Miami, FL, January 1991.

[CBM89a] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 365–373. Springer-Verlag, 1989. LNCS 407.

[CBM89b] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In L. Claesen, editor, *Proceedings IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by constructions or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 238–250, 1977.

[CCJ$^+$01a] P. Chauhan, E. Clarke, S. Jha, J. Kukula, H. Veith, and D. Wang. Using combinatorial optimization methods for quantification scheduling. In T. Margaria and T. F. Melham, editors, *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, pages 293–309. Springer-Verlag, Berlin, September 2001. LNCS 2144.

[CCJ$^+$01b] P. P. Chauhan, E. M. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith, and D. Wang. Non-linear quantification scheduling in image computation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 293–298, San Jose, CA, November 2001.

[CCK$^+$02] P. Chauhan, E. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 33–51. Springer-Verlag, November 2002. LNCS 2517.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transaction on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGJ$^+$00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 154–169. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[CGKS02] E. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV 2002)*, pages 265–279. Springer-Verlag, July 2002. LNCS 2404.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[CHM$^+$94] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A state space decomposition algorithm for approximate FSM traversal. In *Proceedings of the European Conference on Design Automation*, pages 137–141, Paris, France, February 1994.

[CHM$^+$96a] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for approximate FSM traversal based on state space decomposition. *IEEE Transactions on Computer-Aided Design*, 15(12):1465–1478, December 1996.

[CHM$^+$96b] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. *IEEE Transactions on Computer-Aided Design*, 15(12):1451–1464, December 1996.

[CM90] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 126–129, November 1990.

[CNQ03] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of BDD-based approximate traversal. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 898–905, Munich, Germany, March 2003.

[DHWT91] D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for language inclusion using simulation relations. In K. G. Larsen and A. Skou, editors, *Third Workshop on Computer Aided Verification (CAV'91)*, pages 255–265. Springer, Berlin, July 1991. LNCS 575.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[EH83] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. In *Proc. 10th ACM Symposium on Principles of Programming Languages*, 1983.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, October 1991.

[EK03] E.A. Emerson and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Tols and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 144–159, Warsaw, Poland, April 2003.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[EL87] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Fifth Conference on Computer Aided Verification (CAV '93)*, pages 463–478. Springer-Verlag, Berlin, 1993. LNCS 697.

[ES03] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[FFK$^+$01] K. Fisler, R. Fraer, G. Kamhi, M. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 420–434. Springer-Verlag, April 2001. LNCS 2031.

[FV99] K. Fisler and M. Y. Vardi. Bisimulation and model checking. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 338–341, Berlin, September 1999. Springer-Verlag. LNCS 1703.

[GAG$^+$02] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proceedings of the Design Automation Conference*, pages 747–750, New Orleans, LA, June 2002.

[GB94] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV'94)*, pages 299–310, Berlin, 1994. Springer-Verlag. LNCS 818.

[GGA05] A. Gupta, M. K. Ganai, and P. Ashar. Lazy constraints and SAT heuristics for proof-based abstraction. In *Proceedings of International Conference on VLSI Design*, pages 183–188, January 2005.

[GGW$^+$03a] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Abstraction and BDDs complement SAT-based BMC in DiVer. In W. A. Hunt, Jr. and F. Somenzi, editors, *International Conference on Computer Aided Verification (CAV'03)*, pages 206–209. Springer-Verlag, July 2003. LNCS 2725.

[GGW$^+$03b] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Learning from BDDs in SAT-based bounded model checking. In *Proceedings of the Design Automation Conference*, pages 824–829, June 2003.

[GGYA03] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 416–423, November 2003.

[GKMH$^+$03] M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 176–191, Warsaw, Poland, April 2003. LNCS 2619.

[GN03] E. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe (DATE'03)*, pages 886–891, Munich, Germany, March 2003.

[GPP03] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected componenets in a linear number of symbolic steps. In *Symposium on Discrete Algorithms*, Baltimore, MD, January 2003.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.

[GYAG00] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 354–271. Springer-Verlag, November 2000. LNCS 1954.

[HBLS98] Y. Hong, P. A. Beerel, L. Lavagno, and E. M. Sentovich. Don't care-based BDD minimization for embedded software. In *Proceedings of the Design Automation Conference*, pages 506–509, San Francisco, CA, June 1998.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 423–427. Springer-Verlag, Berlin, 1996. LNCS 1102.

[HKB96] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early quantification and partitioned transition relations. In *Proceedings of the International Conference on Computer Design*, pages 12–19, Austin, TX, October 1996.

[Hol97] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[HQR98] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 440–451. Springer-Verlag, Berlin, 1998. LNCS 1427.

[HTKB92] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.

[IBM] IBM Formal Verification Benchmarks. URL: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/ benchmarks.html.

[ID93] C. N. Ip and D. L. Dill. Better verification through symmetry. In *Proc. 11th International Symposium on Computer Hardware Description Languages and Their Application*, April 1993.

[ISC] Iscas benchmarks. URL: http://www.cbl.ncsu.edu/CBL-_Docs/iscas89.html.

[ITR03] International technology roadmap for semiconductors. URL: http://public.itrs.net, 2003.

[Jan99] J.-Y. Jang. *Iterative Abstraction-based CTL Model Checking*. PhD thesis, University of Colorado, Department of Electrical and Computer Engineering, 1999.

[JKS02] H. Jin, A. Kuehlmann, and F. Somenzi. Fine-grain conjunction scheduling for symbolic reachability analysis. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 312–326, Grenoble, France, April 2002. LNCS 2280.

[JMH00] J.-Y. Jang, I.-H. Moon, and G. D. Hachtel. Iterative abstraction-based CTL model checking. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE00)*, pages 502–507, Paris, France, March 2000.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, pages 445–459, Grenoble, France, April 2002. LNCS 2280.

[KPR98] Y. Kesten, A. Pnueli, and L.-o. Raviv. Algorithmic verification of linear temporal logic specifications. In *International Colloquium on Automata, Languages, and Programming (ICALP-98)*, pages 1–16, Berlin, 1998. Springer. LNCS 1443.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[KV98] O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[LNA99] J. Lind-Nielsen and H. R. Andersen. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer Aided Verification (CAV'99)*, pages 316–327. Springer-Verlag, Berlin, 1999. LNCS 1633.

[Lon93] D. E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie-Mellon University, July 1993.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPJ$^+$96] W. Lee, A. Pardo, J. Jang, G. Hachtel, and F. Somenzi. Tearing based abstraction for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 76–81, San Jose, CA, November 1996.

[LS04] B. Li and F. Somenzi. Efficient computation of small abstraction refinements. In *Proceedings of the International Conference on Computer-Aided Design*, pages 518–525, San Jose, CA, November 2004.

[LWCH03] F. Lu, L. Wang, K. Cheng, and R. Huang. A circuit SAT solver with signal correlation guided learning. In *Proceedings of the Design Automation Conference*, pages 10892–10897, June 2003.

[LWS03] B. Li, C. Wang, and F. Somenzi. A satisfiability-based approach to abstraction refinement in model checking. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003. First International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/volume89.html.

[LWS05] B. Li, C. Wang, and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *Software Tools for Technology Transfer*, 2(7):143–155, 2005.

[MA03] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, pages 2–17, Warsaw, Poland, April 2003. LNCS 2619.

[McM94] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1994.

[McM97] K. L. McMillan. A compositional rule for hardware design refinement. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 24–35. Springer-Verlag, Berlin, 1997. LNCS 1254.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In A. J. Hu and M. Y. Vardi, editors, *Tenth Conference on Computer Aided Verification (CAV'98)*, pages 110–121. Springer-Verlag, Berlin, 1998. LNCS 1427.

[McM02] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 250–264. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[MH04] F. Y. C. Mang and P.-H. Ho. Abstraction refinement by controllability and cooperativeness analysis. In *Proceedings of the Design Automation Conference*, pages 224–229, San Diego, CA, June 2004.

[MHS00] I-H. Moon, G. D. Hachtel, and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 73–90. Springer-Verlag, November 2000. LNCS 1954.

[Mil71] R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Int. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[Min93] S.-I. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the Design Automation Conference*, pages 272–277, Dallas, TX, June 1993.

[MJH$^+$98] I.-H. Moon, J.-Y. Jang, G. D. Hachtel, F. Somenzi, C. Pixley, and J. Yuan. Approximate reachability don't cares for CTL model checking. In *Proceedings of the International Conference on Computer-Aided Design*, pages 351–358, San Jose, CA, November 1998.

[MKSS99] I.-H. Moon, J. Kukula, T. Shiple, and F. Somenzi. Least fixpoint MBM: Improved technique for approximate reachability. Presented at IWLS99, Lake Tahoe, CA, June 1999.

[MMZ$^+$01] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[MPC$^+$02] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Operating System Design and Implementation (OSDI'02)*, Boston, MA, December 2002.

[PH98] A. Pardo and G. D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference*, pages 457–462, San Francisco, CA, June 1998.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[QS81] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth Annual Symposium on Programming*, 1981.

[RAB$^+$95] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. Presented at IWLS95, Lake Tahoe, CA, May 1995.

[RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.

[RS95] K. Ravi and F. Somenzi. High-density reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 154–158, San Jose, CA, November 1995.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[SHSVB94] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic minimization of BDDs using don't cares. In *Proceedings of the Design Automation Conference*, pages 225–231, San Diego, CA, June 1994.

[Sht00] Ofer Shtrichman. Tuning SAT checkers for bounded model checking. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 480–494. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[Sil99] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.

[Som] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/.

[SRB02] F. Somenzi, K. Ravi, and R. Bloem. Analysis of symbolic SCC hull algorithms. In M. D. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer Aided Design*, pages 88–105. Springer-Verlag, November 2002. LNCS 2517.

[SS96] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, November 1996.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 108–125. Springer-Verlag, November 2000. LNCS 1954.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[TBK95] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.

[TSL$^+$90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDD's. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990.

[Var95] M. Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 101–111, June 1995.

[VB00] W. Visser and H. Barringer. Practical CTL* model checking - should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[VIS] URL: http://vlsi.colorado.edu/∼vis.

[VVB] Vis verification benchmarks. http://vlsi.colorado.edu/∼vis.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[WBCG00] P. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 124–138. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[WHL$^+$01] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, T. Ma, and R. Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the Design Automation Conference*, pages 35–40, Las Vegas, NV, June 2001.

[WKS01] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference*, pages 542–545, Las Vegas, NV, June 2001.

[XB99] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components. In *Proceedings of the International Conference on Computer-Aided Design*, pages 37–40, San Jose, CA, November 1999.

[XB00] A. Xie and P. A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design*, 19(10):1225–1230, October 2000.

[ZM03] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Munich, Germany, March 2003.

[ZPH04] L. Zhang, M. R. Prasad, and M. S. Hsiao. Incremental deductive and inductive reasoning for SAT-based bounded model checking. In *Proceedings of International Conference on Computer Aided Design*, pages 502–509, San Jose, CA, November 2004.

[ZPHS05] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle. Dynamic abstraction using SAT-based BMC. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 754–757, San Jose, CA, June 2005.

About the Authors

Chao Wang is currently a research staff member at NEC Laboratories America in Princeton, New Jersey. His research is in the general area of electronic design automation. In the past six years, he has been working on formal specification and verification of concurrent systems, including hardware, software, and embedded systems. Dr. Wang received his Ph.D. degree from the University of Colorado at Boulder in 2004 and his B.S. degree from Peking University, China in 1996. He was the recipient of the 2003-2004 ACM Outstanding Ph.D. Dissertation Award in Electronic Design Automation.

About the Authors

Gary D. Hachtel is a Professor Emeritus who has been working for the last 20 years in the University of Colorado at Boulder in fields of logic synthesis and formal verification. He received his Ph.D. degree from the University of California, Berkeley in 1964 and his B.S. degree from California Institute of Technology in 1959. For the 17 years prior to his stint at the University of Colorado, he was a research staff member in the department of Math Sciences at IBM research in Yorktown Heights, NY. Professor Hachtel is an IEEE Fellow (since 1979). He received the IEEE CASS Mac Van Valkenburg Award in 2004 for his distinguished career of fundamental innovations across the broad spectrum of semiconductor electronic design automation.

About the Authors

Fabio Somenzi is a Professor in the ECE department of the University of Colorado at Boulder. He has published one book and over 140 papers on the synthesis, optimization, verification, simulation, and testing of digital systems. He received his Dr. Eng. Degree in Electronic Engineering from Politecnico di Torino, Italy in 1980. Prior to joining University of Colorado in 1989, he was with SGS-Thomson Microelectronics, Italy managing a team for computer aids for digital design. Professor Somenzi has served as associate editor for IEEE Transactions on Computer-Aided Design and the Springer journal on Formal Methods in Systems Design, and on the program committees of premier EDA conferences including ICCAD, DAC, ICCD, EDAC/DATE, IWLS, and ISLPED. He was the conference co-chair of Computer Aided Verification (CAV) in 2003.

Index