Progress on the State Explosion Problem in Model Checking *

Edmund Clarke[1], Orna Grumberg[2], Somesh Jha[3], Yuan Lu[4], and Helmut Veith[5]

[1] School of Computer Science, Carnegie Mellon University, USA
edmund.clarke@cs.cmu.edu
[2] Computer Science Department, Technion, Haifa, Israel
orna@cs.technion.ac.il
[3] Computer Sciences Department, University of Wisconsin, Madison, USA
jha@cs.wisc.edu
[4] Network Switch Department, Broadcom Co, USA
ylu@broadcom.com
[5] Institute of Information Systems, Vienna University of Technology, Austria
veith@dbai.tuwien.ac.at

Abstract. Model checking is an automatic verification technique for finite state concurrent systems. In this approach to verification, temporal logic specifications are checked by an exhaustive search of the state space of the concurrent system. Since the size of the state space grows exponentially with the number of processes, model checking techniques based on explicit state enumeration can only handle relatively small examples. This phenomenon is commonly called the "State Explosion Problem". Over the past ten years considerable progress has been made on this problem by (1) representing the state space symbolically using BDDs and by (2) using abstraction to reduce the size of the state space that must be searched. As a result model checking has been used successfully to find extremely subtle errors in hardware controllers and communication protocols. In spite of these successes, however, additional research is needed to handle large designs of industrial complexity. This aim of this paper is to give a succinct survey of symbolic model checking and to introduce the reader to recent advances in abstraction.

1 Introduction

During the last two decades, temporal logic model checking [12, 13] has become an important application of logic in computer science. Temporal logic model checking is a technique for verifying that a system satisfies its specification by (i) representing the

* This research is sponsored by the Semiconductor Research Corporation (SRC) under Contract No. 97-DJ-294, the National Science Foundation (NSF) under Grant No. CCR-9505472, the Defense Advanced Research Projects Agency (DARPA) under Air Force contract No. F33615-00-C-1701, the Max Kade Foundation and the Austrian Science Fund Project N Z29-INF. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of SRC, NSF, or the United States Government.

system as a Kripke structure, (ii) writing the specification in a suitable temporal logic, and (iii) algorithmically checking that the Kripke structure is a model of the specification formula. Model checking has been successfully applied in hardware verification, and is emerging as an industrial standard tool for hardware design. For extensive overviews of model checking, please refer to [10, 11].

Model checking has several important advantages over mechanical theorem provers or proof checkers for verification of circuits and protocols. The most important is that the procedure is completely automatic. Typically, the user provides a high level representation of the model and the specification to be checked. The model checking algorithm will either terminate with the answer *true*, indicating that the model satisfies the specification, or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex transition systems. The procedure is also quite fast and often produces an answer in a matter of minutes. Since partial specifications can be checked, it is unnecessary to specify the circuit completely before useful information about its correctness can be obtained. Finally, the logics used for specifications can directly express many of the properties that are needed for reasoning about concurrent systems.

The main technical challenge in model checking is the *state explosion* which can occur if the system being verified has many components which make transitions in parallel. A fundamental breakthrough was made in the fall of 1987 by Ken McMillan, who was then a graduate student at Carnegie Mellon. He argued that larger systems could be handled if transition relations were represented implicitly with ordered binary decision diagrams (BDDs) [3]. By using the original model checking algorithm with the new representation for transition relations, he was able to verify some examples that had more than 10^{20} states [6, 25]. He made this observation independently of the work by Coudert et al [16] and Pixley [28–30] on using BDDs to check equivalence of deterministic finite-state machines. Since then, various refinements of the BDD-based techniques by other researchers have pushed the state count up to more than 10^{120} [5]. The widely used symbolic model checker SMV [25] is based on these ideas.

Despite the success of symbolic methods, the state explosion problem remains a major hurdle in applying model checking to large industrial designs. *Abstraction* is among the most important techniques for tackling this problem. In fact, abstraction based methods have been essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, often requiring considerable creativity. In order for model checking to be used more widely in industry, automatic techniques are needed for generating abstractions.

This paper is intended as an overview of a recently developed *automatic abstraction technique* [9] which extends the general framework of *existential abstraction* [14]. Existential abstraction computes an *upper approximation* of the original model. When a specification in the temporal logic ACTL is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction so that the behavior which caused the erroneous counterexample is eliminated. The main contribution of [9] is an efficient automatic refinement technique which

uses information obtained from erroneous counterexamples. The refinement algorithm keeps the size of the abstract state space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code demonstrate the utility of this approach. Although our current implementation is based on NuSMV [8], it is in principle not limited to the input language of SMV and can be applied to other languages.

Organization of the Paper. The paper is organized as follows: Section 2 contains an introduction to model checking, temporal logic and the state explosion problem. In Sections 3 and 4, a succinct overview of symbolic verification, and a more detailed overview of our recent counterexample-guided abstraction methodology [9] are given. Directions for future research are outlined in Section 5.

2 Fundamentals of Model Checking

In this section, we outline important notions which are necessary to understand the subsequent discussion of the state explosion problem. A more rigorous and detailed introduction can be found in [10].

Kripke Structures. In model checking, the system to be verified is formally represented by a finite *Kripke structure*. Essentially, a Kripke structure is a directed graph whose vertices are labeled by sets of atomic propositions. Vertices and edges are called *states* and *transitions* respectively. One or more states are considered to be *initial states*. Consider for example the Kripke structures in Figure 1 which represent traffic lights in the US and Austria, respectively.

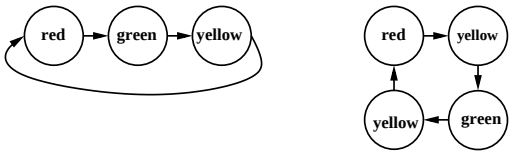

Fig. 1. US and Austrian Traffic Lights as Kripke Structures

Thus, a Kripke structure over a set of atomic propositions A is a tuple $K = (S, R, L, I)$ where S is the set of states, $R \subseteq S^2$ is the set of transitions, $I \subseteq S$ is the non-empty set of initial states, and $L : S \to 2^A$ labels each state by a set of atomic propositions. Note that more complicated definitions of Kripke structures are also used in the literature. In particular, it is common to label the transitions of a Kripke structure by *actions*. As demonstrated by the traffic light example, a Kripke structure can be viewed as a kind of automaton.

A *path* is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for $i \geq 0$, $(s_i, s_{i+1}) \in R$. Given a path π, π^i denotes the infinite path $s^i, s^{i+1}, \ldots$ We assume that the transition relation R is *total*, i.e., that all states have positive outdegree. Therefore, each finite path can be extended into an infinite path. Figure 2 indicates how a Kripke structure is unwound into an infinite tree such that the paths in the Kripke structure and the infinite tree coincide.

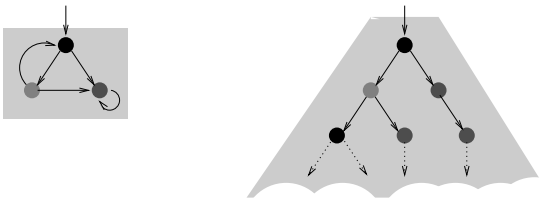

Fig. 2. Unwinding a Kripke Structure. The incoming arrows indicate the initial states.

Computation Tree Logics. CTL* is an extension of propositional logic obtained by adding *path quantifiers* and *temporal operators*.

1. **Path quantifiers**:
   A "for every path"
   E "there exists a path"
2. **Temporal Operators**:
   $\mathbf{X}p$ "p holds next time"
   $\mathbf{F}p$ "p holds sometime in the future"
   $\mathbf{G}p$ "p holds globally in the future"
   $p\mathbf{U}q$ "p holds until q holds"

In the computation tree logic **CTL** each temporal operator must be immediately preceded by a path quantifier. Thus, CTL can be viewed as a temporal logic based on the compound operators $\mathbf{AX}, \mathbf{EX}, \mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}, \mathbf{AU}, \mathbf{EU}$. Let s_0 be a state in K. The formal semantics of $\mathbf{EX}$, $\mathbf{EG}$ and $\mathbf{EU}$ is defined as follows:

$s_0, K \models \mathbf{EX}\varphi$ iff there exists a path $\pi = s_0, s_1, \ldots$ such that $K, s_1 \models \varphi$
$s_0, K \models \mathbf{EG}\varphi$ iff there exists a path $\pi = s_0, s_1, \ldots$ such that for all $i \geq 0$, $K, s_i \models \varphi$
$s_0, K \models \mathbf{E}\varphi\mathbf{U}\psi$ iff there exists a path $\pi = s_0, s_1, \ldots$ and an $i \geq 0$ such that for all $0 \leq j < i$, $K, s_j \models \varphi$, and $K, s_i \models \psi$.

The remaining CTL operators are defined by abbreviations as follows:

$$\mathbf{EF}\varphi \equiv \mathbf{E}(\mathrm{true}\mathbf{U}\varphi) \qquad \mathbf{AG}\varphi \equiv \neg\mathbf{EF}\neg\varphi$$
$$\mathbf{AF}\varphi \equiv \neg\mathbf{EG}\neg\varphi \qquad \mathbf{AX}\varphi \equiv \neg\mathbf{EX}\neg\varphi$$
$$\mathbf{A}\varphi\mathbf{U}\psi \equiv \neg\mathbf{E}(\neg\psi\mathbf{U}(\ \neg\ \varphi \wedge \neg\psi)) \wedge \neg\mathbf{EG}\neg\psi$$

Four important CTL operators are illustrated in Figure 3 by typical computation trees. Each computation tree has s_0 as its root.

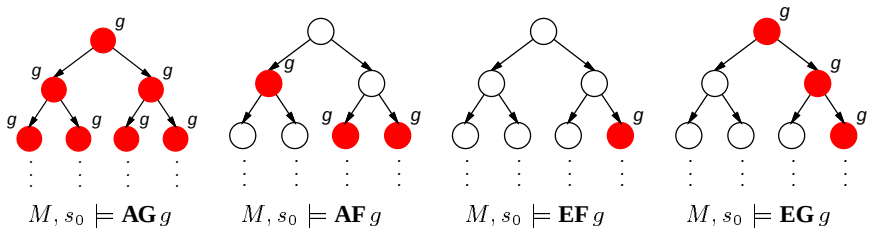

Fig. 3. Example of the most widely used CTL operators. The dark states are states where g holds true.

Example 1. The following list contains some typical CTL formulas:

- **EF**$(Started \wedge \neg Ready)$: it is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG**$(Req \Rightarrow$ **AF**$Ack)$: if a *Request* occurs, then it will be eventually *Acknowledged*.
- **AG**(**AF** $DeviceEnabled$): *DeviceEnabled* holds infinitely often on every computation path.
- **AG**(**EF** $Restart$): from any state it is possible to get to the *Restart* state.

ACTL is the fragment of CTL where only the operators involving **A** are used, and negation is restricted to atomic formulas. An important feature of ACTL is the existence of counterexamples. For example, the CTL specification **AF**p denotes "*On all paths, p holds sometime in the future.*" If the specification $\mathbf{AF}p$ is violated, then there exists an infinite path where p never holds. This path is called a counterexample of $\mathbf{AF}p$. In this paper, we will focus on counterexamples which are finite or infinite paths.

For a formal definition of related temporal logics such as CTL* and LTL, please refer to [10].

Explicit State Model Checking. Given a Kripke structure $K = (S, R, I, L)$ and a specification φ in a temporal logic such as CTL, the *model checking problem* is the problem of finding all states s such that

$$K, s \models \varphi$$

and checking if the initial states are among these. An explicit state model checker is a program which performs model checking directly on a Kripke structure.

Theorem 1. *[13, 15] Explicit state CTL model checking has time complexity* $O(|K||\varphi|)$.

Besides linear time complexity, CTL has a number of other remarkable properties including decidability and the finite model property. Recent research in logic and databases has generalized the favorable properties of CTL and other temporal logics to fragments of first order and fixed point logics [1, 20] and database query languages [18].

Model checking algorithms are usually fixed point algorithms which exploit the fact that temporal formulas can be expressed by fixed point formulas. For example, the set of states Y where the formula $\mathbf{EF}\varphi$ holds, can be defined inductively as follows:

- If $s \models \varphi$, then $s \in Y$.
- If $s \in Y$ and $R(s', s)$ then $s' \in Y$.
- Nothing else is in Y.

This gives rise to the fixed point characterization

$$\mathbf{EF}\varphi \equiv \mu Y.\varphi \vee \mathbf{EX}\, Y$$

where μ is the least fixed point operator. The fixed point extension of temporal logic is called the μ-calculus, and has been studied extensively. It is easy to see that all CTL and CTL* formulas can be expressed using only least fixed points, propositional logic, and the temporal operator $\mathbf{EX}$. Note that $\mathbf{EX}$ is also known as $\Diamond$ in modal logic. Explicit state model checking for the μ-calculus is known to be in NP $\cap$ coNP, but the existence of a polynomial time algorithm is a famous open problem.

State Explosion. In practice, systems are described by programs in finite state languages such as SMV or VERILOG. These programs are then compiled into equivalent Kripke structures.

Example 2. In the verification system SMV, the state space S of a Kripke structure is given by the possible assignments to the system variables. Thus, a system with 3 variables $x, y, reset$ and variable domains $D_x = D_y = \{0, 1, 2, 3\}$ and $D_{reset} = \{0, 1\}$ has state space $S = D_x \times D_y \times D_{reset}$, and $|S| = 32$.

The binary transition relation R is defined by transition blocks which for each variable define its possible next value in the next time cycle, as in the following example:

init($reset$) := 0;	**init**(x) := 0;	**init**(y) := 1;
next($reset$) := $\{0, 1\}$;	**next**(x) := **case**	**next**(y) := **case**
	$reset = 1 : 0$;	$reset = 1 : 0$;
	$x < y : x + 1$;	$(x = y) \wedge \neg(y = 2) : y + 1$;
	$x = y : 0$;	$(x = y) : 0$;
	else : x;	**else** : y;
	esac;	**esac**;

Here, $\mathbf{next}(reset) := \{0, 1\}$ means that the value of $reset$ is chosen nondeterministically. Such situations occur frequently when $reset$ is controlled by the environment, or

when the model of the system is too abstract to determine the values of $reset$. For details about the SMV input language, we refer the reader to [25]. Typical CTL properties to be verified by the system include the following:

CTL	*Informal Semantics*
$\mathbf{AG}\,\mathbf{EF}\; reset = 1$	*"From all reachable states, it is possible to reset the system in the future."*
$\mathbf{EF}\,\mathbf{AG}\; x = 1$	*"There exists a reachable state, after which $x = 1$ becomes an invariant."*

The main practical problem in model checking is the so-called **state explosion problem** caused by the fact that the Kripke structure represents the *state space* of the system under investigation, and thus it is of size *exponential* in the size of the system description. Therefore, even for systems of relatively modest size, it is often impossible to compute their Kripke structures.

In the rest of this paper, we will focus on two techniques, *symbolic verification* and *abstraction* which alleviate the state explosion problem.

- **Symbolic verification** is a conservative approach where the Kripke structure is represented by succinct data structures (in particular, Binary Decision Diagrams) without losing information.
- **Abstraction techniques** in contrast employ knowledge about the structure and the specification in order to model only relevant features in the Kripke structure.

3 Symbolic Model Checking

In *symbolic verification*, the transition relation of the Kripke structure is not explicitly constructed, but instead a Boolean function is computed which represents the transition relation. Similarly, sets of states are also represented by Boolean functions. Then, the fixed point algorithms mentioned above are applied to the Boolean functions rather than to the Kripke structure. Since in many practical situations the space requirements for Boolean functions are exponentially smaller than for explicit representation, symbolic verification is able to alleviate the state explosion problem in these situations.

In the remainder of this section, we introduce the main ingredients of symbolic model checking, and discuss the theoretical limitations of BDD based methods.

Ordered Binary Decision Diagrams. Let A be a set of propositional variables, and $\prec$ a linear order on A. An *ordered binary decision diagram* (BDD) $\mathcal{O}$ over A is an acyclic graph (V, E) whose non-terminal vertices (*nodes*) are labeled by variables from A, and whose edges and terminal nodes are labeled by 0, 1. Each non-terminal node v has out-degree 2, such that one of its outgoing edges is labeled 0 (the *low edge* or *else-edge*), and the other is labeled 1 (the *high edge* or *then-edge*). If v has label a_i and the successors of v are labeled a_j, a_k, then $a_i \prec a_j$ and $a_i \prec a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $\prec$.

Each BDD node v represents a Boolean function $\mathcal{O}_v$. The terminal nodes of $\mathcal{O}$ represent the constant functions given by their labels. A non-terminal node v with label a_i whose successors at the high and low edges are u and w respectively, defines the function $\mathcal{O}_v := (a_i \wedge \mathcal{O}_u) \vee (\neg a_i \wedge \mathcal{O}_w)$.

As the following example shows, BDDs are related to Boolean decision trees.

Example 3. The Boolean decision tree of Figure 4 represents the Boolean function $x \wedge (y \vee z)$. The BDD in Figure 5 represents the same Boolean function in a more succinct way. Note that the BDD can be obtained from the decision tree by merging isomorphic subtrees, and removing redundant edges. The variable ordering is $x \prec y \prec z$.

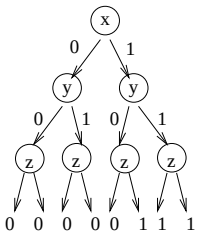

Fig. 4. Decision tree for $x \wedge (y \vee z)$.

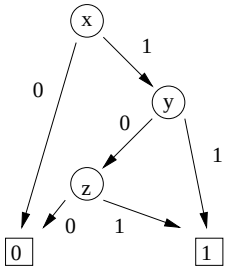

Fig. 5. A BDD for function $x \wedge (y \vee z)$.

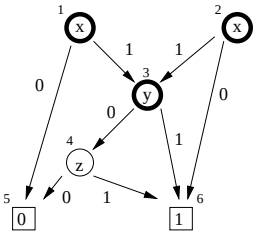

Fig. 6. A shared BDD.

The size of a BDD is the number of nodes of the BDD. The size of a BDD in general depends on the variable order $\prec$, and may be exponential in $|A|$. However, it is well-known [3, 4] that for every variable order $\prec$ and Boolean function f there exists a *unique minimal* BDD $\mathcal{O}$ over A which represents the Boolean function f. Given any BDD for f which respects $\prec$, $\mathcal{O}$ can be computed in polynomial time. Note that $\mathcal{O}$ contains at most two non-terminal nodes, and no two nodes of $\mathcal{O}$ describe the same Boolean function.

In practice, *shared BDDs* are used to represent several Boolean functions at once. For example, in the BDD of Figure 6, the nodes $1, 2$ and 3 represent the Boolean functions $x \wedge (y \vee z)$, $\neg x \wedge (y \vee z)$ and $y \vee z$ respectively.

Effective algorithms for handling BDDs have been described in the literature [3] and highly effective BDD libraries such as CUDD [31] have been developed.

Symbolic Verification Algorithms. A symbolic verification algorithm is an algorithm whose variables denote not single states, but *sets of states* which are represented by Boolean functions (usually as BDDs). Therefore, symbolic algorithms use only such operations on sets which can be translated into BDD operations. For example, union and intersection of sets correspond to disjunction and conjunction respectively. Binary Decision Diagrams have been a particularly useful data structure for representing Boolean

functions; despite their relative succinctness they provide canonical representations of Boolean functions, and therefore expressions of the form $S_1 = S_2$, which are important in fixed point computations, can be evaluated very efficiently.

Image computation is the task to compute for a given set Q of states the set of states

$$\mathbf{EX}(Q) := \{s : \exists s'.R(s,s') \wedge s' \in Q\}.$$

Recall that CTL can be expressed in fixed point logic with a temporal operator $\mathbf{EX}$. Therefore, image computation is a central task in symbolic verification.

Image computation is one of the major bottlenecks in verification. Part of the reason for this, ironically, is the fact that it is in general not feasible to construct a single BDD for R. Instead, R is represented as the conjunction of several BDDs. The problem then arises how to compute $\mathbf{EX}(Q)$ without actually computing R. In a recent series of papers [7, 26, 27], improved algorithms for image computation have been investigated.

Theoretical Limitations of Symbolic Model Checking. Potentially, the BDD representation of a Kripke structure may be exponentially more succinct than the explicit representation. Practical experience with symbolic verification demonstrates that in many cases BDDs indeed yield a significant space improvement.

A classic information theoretic argument shows that only a small fraction of all finite Kripke structures can be exponentially compressed [23]. Of course, the general limitation applies to compression by BDDs as well. On the other hand, we know that the large Kripke structures encountered in model checking usually have small descriptions in terms of hardware description languages. This indicates that BDDs or more advanced data structures can in principle be used to obtain exponentially succinct representations of Kripke structures, at least for those Kripke structures in which we are interested.

Practical experiments show that the performance of symbolic methods is highly unpredictable. This phenomenon can be partially explained by complexity theoretic results which state that BDD representation does not improve worst case complexity. In fact, it has been shown [17, 33] that representing a decision problem in terms of exponentially smaller BDDs usually increases its worst case complexity exponentially. For example, the problem of deciding $\mathbf{EF}p$ (reachability) is complete for nondeterministic logspace NL, while in BDD representation it becomes complete for PSPACE. Similar results can be shown for other Boolean formalisms and are closely tied to principal questions in structural complexity theory [19, 32, 34]. We conclude that symbolic verification is a very powerful method but needs to be complemented by more aggressive techniques. The following section deals with abstraction, one such technique.

4 Abstraction

Existential Abstraction. Intuitively speaking, existential abstraction amounts to partitioning the states of a Kripke structure into clusters, and treating the clusters as new abstract states, cf. Figure 7.

Formally, an abstraction function h is described by a surjection $h : S \to \widehat{S}$ where $\widehat{S}$ is the set of *abstract states*. The surjection h induces an equivalence relation $\equiv$ on the

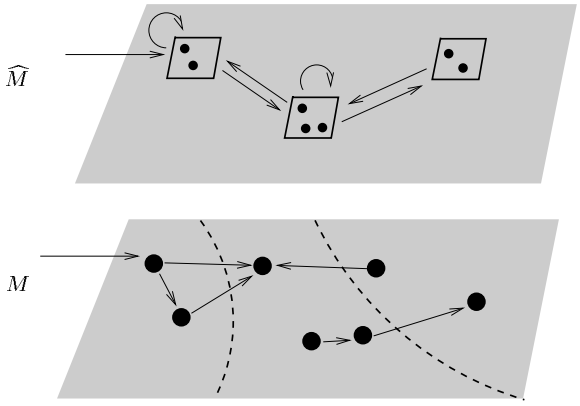

Fig. 7. Existential Abstraction. M is the original Kripke structure, and $\widehat{M}$ the abstracted one. The dotted lines in M indicate how the states of M are clustered into abstract states.

domain S in the following manner: let d, e be states in S, then

$$d \equiv e \;\text{ iff }\; h(d) = h(e).$$

Since an abstraction can be represented either by a surjection h or by an equivalence relation $\equiv$, we sometimes switch between these representations.

The *abstract Kripke structure* $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L})$ corresponding to the abstraction function h is defined as follows:

1. $\widehat{I}(\widehat{d})$ iff $\exists d(h(d) = \widehat{d} \wedge I(d))$.
2. $\widehat{R}(\widehat{d_1}, \widehat{d_2})$ iff $\exists d_1 \exists d_2 (h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2))$.
3. $\widehat{L}(\widehat{d}) = \bigcup_{h(d)=\widehat{d}} L(d)$.

An atomic formula f *respects* an abstraction function h if for all d and d' in the domain S, $(d \equiv d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. Let $\widehat{d}$ be an abstract state. $\widehat{L}(\widehat{d})$ is *consistent*, if all concrete states corresponding to $\widehat{d}$ satisfy all labels in $\widehat{L}(\widehat{d})$, i.e., collapsing a set of concrete states into an abstract state does not lead to contradictory labels.

Spurious Counterexamples. It is easy to see that $\widehat{M}$ contains less information than M. Thus, model checking the structure $\widehat{M}$ potentially leads to wrong results. The following theorem shows that at least for ACTL, specifications which are correct for $\widehat{M}$ are correct for M as well.

Theorem 2. *Let h be an abstraction and φ be an* ACTL *specification where the atomic subformulas respect h. Then the following holds: (i) $\widehat{L}(\widehat{d})$ is consistent for all abstract states $\widehat{d}$ in $\widehat{M}$; (ii) $\widehat{M} \models \varphi \;\Rightarrow\; M \models \varphi$.*

On the other hand, the following example shows that if the abstract model invalidates an ACTL specification, *the actual model may still satisfy the specification*.

Example 4. Assume that for a US traffic light controller (see Figure 8), we want to prove $\psi = \mathbf{AGAF}(state = red)$ using the abstraction function $h(red) = \widehat{red}$ and $h(green) = h(yellow) = \widehat{go}$. It is easy to see that $M \models \psi$ while $\widehat{M} \not\models \psi$. There exists an infinite abstract trace $\langle \widehat{red}, \widehat{go}, \widehat{go}, \ldots \rangle$ that invalidates the specification.

Fig. 8. Abstraction of a US Traffic Light.

If an abstract counterexample does not correspond to some concrete counterexample, we call it *spurious*. For example, $\langle \widehat{red}, \widehat{go}, \widehat{go}, \ldots \rangle$ in the above example is a spurious counterexample.

The Fine Structure of Abstraction Functions. As Example 2 shows, the set of states S of a Kripke structure is typically obtained as the product $D_1 \times \cdots D_n$ of smaller domains. In this situation, an abstraction function h can be described by surjections $h_i : D_i \to \widehat{D_i}$, such that $h(d_1, \ldots, d_n)$ is equal to $(h_1(d_1), \ldots, h_n(d_n))$, and $\widehat{S}$ is equal to $\widehat{D_1} \times \cdots \widehat{D_n}$. The equivalence relations $\equiv_i$ corresponding to the individual surjections h_i induce an equivalence relation $\equiv$ over the entire domain $S = D_1 \times \cdots \times D_n$ in the obvious manner:

$$(d_1, \cdots, d_n) \equiv (e_1, \cdots, e_n) \text{ iff } d_1 \equiv_1 e_1 \wedge \cdots \wedge d_n \equiv_n e_n$$

4.1 Counterexample-guided Abstraction

Recall that for a Kripke structure M, and an ACTL formula φ, our goal is to check whether the Kripke structure M corresponding to P satisfies φ. Our methodology consists of the following steps, cf. Figure 9.

1. *Generate the initial abstraction:* We generate an initial abstraction h by examining the transition blocks corresponding to the variables of the program which describes M, cf. Example 2. A detailed description of the initial abstraction is given in [9].
2. *Model-check the abstract structure:* Let $\widehat{M}$ be the abstract Kripke structure corresponding to the abstraction h. We check whether $\widehat{M} \models \varphi$. If the check is affirmative, then we can conclude that $M \models \varphi$ (see Theorem 2). Suppose the check reveals that there is a counterexample $\widehat{T}$. We ascertain whether $\widehat{T}$ is an actual counterexample, i.e., a counterexample in the unabstracted structure M. If $\widehat{T}$ turns out to be an actual counterexample, we report it to the user, otherwise $\widehat{T}$ is a spurious counterexample, and we proceed to step 3.
3. *Refine the abstraction:* We refine the abstraction function h by partitioning a *single equivalence class* of $\equiv$ so that after the refinement the abstract structure $\widehat{M}$ corresponding to the refined abstraction function no longer admits the spurious

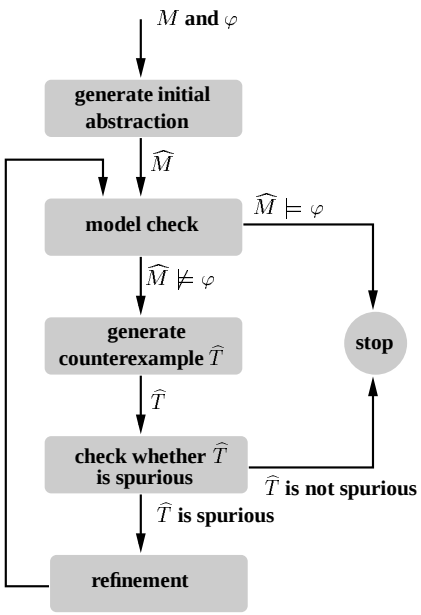

Fig. 9. Counterexample based refinement.

counterexample $\widehat{T}$. We will discuss partitioning algorithms for this purpose in Section 4.3. After refining the abstraction function, we return to step 2.

Using counterexamples to refine abstract models has been investigated by a number of other researchers beginning with the *localization reduction* of Kurshan [21]. He models a concurrent system as a composition of L-processes $L_1, \ldots, L_n$ (L-processes are described in detail in [21]). The localization reduction is an iterative technique that starts with a small subset of relevant L-processes that are topologically close to the specification in the *variable dependency graph*. All other program variables are abstracted away with nondeterministic assignments. If the counterexample is found to be spurious, additional variables are added to eliminate the counterexample. The heuristic for selecting these variables also uses information from the variable dependency graph. A similar approach has been described by Balarin in [2, 22].

4.2 Model Checking the Abstract Model

We use standard symbolic model checking procedures to determine whether $\widehat{M}$ satisfies the specification φ. If it does, then by Theorem 2 we can conclude that the original Kripke structure also satisfies φ. Otherwise, assume that the model checker produces a counterexample $\widehat{T}$ corresponding to the abstract model $\widehat{M}$. In the rest of this section, we will focus on counterexamples which are either *finite paths* or *infinite paths (loops)*.

Identification of Spurious Finite Path Counterexamples First, we will tackle the case when the counterexample $\widehat{T}$ is a finite path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. Given an abstract state $\widehat{s}$, the set of concrete states s such that $h(s) = \widehat{s}$ is denoted by $h^{-1}(\widehat{s})$, i.e., $h^{-1}(\widehat{s}) = \{s | h(s) = \widehat{s}\}$. We extend h^{-1} to sequences in the following way: $h^{-1}(\widehat{T})$ is the set of concrete finite paths given by the following expression

$$\{\langle s_1, \cdots, s_n \rangle | \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \wedge I(s_1) \wedge \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1})\}.$$

We will occasionally write h^{-1}_{path} to emphasize the fact that h^{-1} is applied to a sequence. Next, we give a *symbolic* algorithm to compute $h^{-1}(\widehat{T})$. Let $S_1 = h^{-1}(\widehat{s_1}) \cap I$ and R be the transition relation corresponding to the unabstracted Kripke structure M. For $1 < i \leq n$, we define S_i in the following manner: $S_i := Img(S_{i-1}) \cap h^{-1}(\widehat{s_i})$. Recall that $Img(S_{i-1})$ is the forward image of S_{i-1} with respect to the transition relation R. The sequence of sets S_i is computed symbolically using BDDs and the standard image computation algorithm. The following lemma establishes the correctness of this procedure.

Lemma 1. *The following are equivalent:*

(i) The finite path $\widehat{T}$ corresponds to a concrete counterexample.
(ii) The set of concrete finite paths $h^{-1}(\widehat{T})$ is non-empty.
(iii) For all $1 \leq i \leq n$, $S_i \neq \emptyset$.

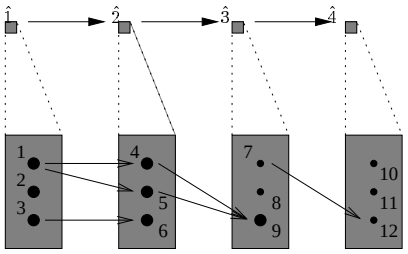

Fig. 10. An abstract counterexample

Algorithm SplitPATH
$S := h^{-1}(\widehat{s_1}) \cap I$
$j := 1$
while $(S \neq \emptyset$ and $j < n)$ {
$j := j + 1$
$S_{\text{prev}} := S$
$S := Img(S) \cap h^{-1}(\widehat{s_j})$ }
if $S \neq \emptyset$ **then** output counterexample
else output j, S_{prev}

Fig. 11. SplitPATH checks spurious finite paths.

Example 5. Consider a program with only one variable with domain $S = \{1, \cdots, 12\}$. Assume that the abstraction function h maps $x \in S$ to $\lfloor (x-1)/3 \rfloor + 1$. There are four abstract states corresponding to the equivalence classes $\{1, 2, 3\}$, $\{4, 5, 6\}$, $\{7, 8, 9\}$, and $\{10, 11, 12\}$. We call these abstract states $\widehat{1}$, $\widehat{2}$, $\widehat{3}$, and $\widehat{4}$. The transitions between states in the concrete model are indicated by the arrows in Figure 10; small dots denote non-reachable states. Suppose that we obtain an abstract counterexample $\widehat{T} = \langle \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4} \rangle$. It is easy to see that $\widehat{T}$ is spurious. Using the terminology of Lemma 1, we have $S_1 = \{1, 2, 3\}$, $S_2 = \{4, 5, 6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. Notice that S_4 and therefore $Img(S_3)$ are both empty.

It follows from Lemma 1 that if $h^{-1}(\widehat{T})$ is empty (i.e., if the counterexample $\widehat{T}$ is spurious), then there exists a minimal i ($2 \leq i \leq n$) such that $S_i = \emptyset$. The symbolic Algorithm **SplitPATH** in Figure 11 computes this number and the set of states in S_{i-1}. In this case, we proceed to the refinement step (see Section 4.3). On the other hand, if the conditions stated in Lemma 1 are true, then **SplitPATH** will report a "real" counterexample and we can stop.

Identification of Spurious Loop Counterexamples. Now we consider the case when the counterexample $\widehat{T}$ includes a loop, which we write as $\langle \widehat{s_1}, \cdots, \widehat{s_i} \rangle \langle \widehat{s_{i+1}}, \cdots, \widehat{s_n} \rangle^\omega$. The loop starts at the abstract state $\widehat{s_{i+1}}$ and ends at $\widehat{s_n}$. Since this case is more complicated than the finite path counterexamples, we first present an example in which some of the typical situations occur.

Example 6. We consider a loop $\langle \widehat{s_1} \rangle \langle \widehat{s_2}, \widehat{s_3} \rangle^\omega$ as shown in Figure 12. In order to find out if the abstract loop corresponds to concrete loops, we unwind the counterexample as demonstrated in the figure. There are two situations where cycles occur. In the figure,

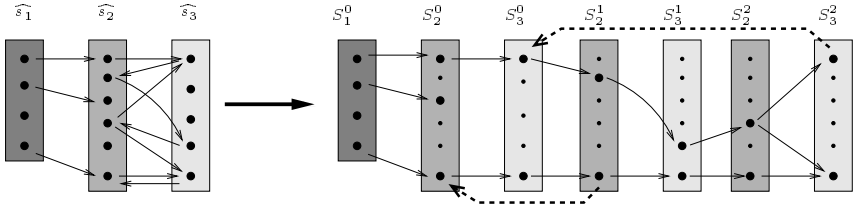

Fig. 12. A loop counterexample, and its unwinding.

for each of these situations, an example cycle (the first one occurring) is indicated by a fat dashed arrow. We make the following important observations: (i) A given abstract loop may correspond to several concrete loops of *different size*. (ii) Each of these loops may start at different stages of the unwinding. (iii) The unwinding eventually becomes periodic (in our case $S_3^0 = S_3^2$), but only after several stages of the unwinding. The size of the period is the least common multiple of the size of the individual loops, and thus, in general *exponential*.

We conclude from the example that a naive algorithm may have exponential time complexity due to an exponential number of loop unwindings. However, it is shown in [9] that a minor modification of the algorithm **SplitPATH** can be used to analyze abstract loop counterexamples effectively. For easy reference we shall refer to this algorithm as **SplitLOOP**.

4.3 Refining the Abstraction

In this section we explain how to refine an abstraction to eliminate the spurious counterexample. Let us first consider the situation outlined in Figure 13. We see that the

abstract path does not have a corresponding concrete path. Whichever concrete path we go, we will end up in state D, from which we cannot go further. Therefore, D is called a *deadend state*. On the other hand, the *bad state* is state B, because it made us believe that there is an outgoing transition. It is easy to see that the algorithm **SplitPath** of the previous section will output the set of deadend states. The question now arises how to partition the abstract state in such a way that the spurious counterexample is eliminated.

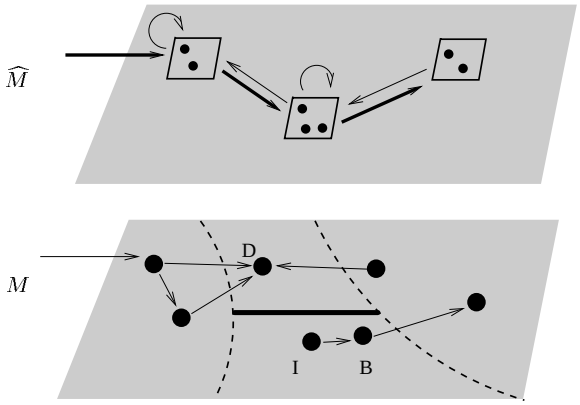

Fig. 13. The abstract path in $\widehat{M}$ (indicated by the thick arrows) is spurious. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line in M.

Since we know from the previous section that loop counterexamples can be treated in a very similar way as finite path counterexamples, we will concentrate on finite path counterexamples. Let us formally consider the case when the counterexample $\widehat{T} = \langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$ is a finite path. Since $\widehat{T}$ does not correspond to a real counterexample, by Lemma 1 (iii) there exists a set $S_i \subseteq h^{-1}(\widehat{s_i})$ with $1 \leq i < n$ such that $Img(S_i) \cap h^{-1}(\widehat{s_{i+1}}) = \emptyset$ and S_i is reachable from initial state set $h^{-1}(\widehat{s_1}) \cap I$. Since there is a transition from $\widehat{s_i}$ to $\widehat{s_{i+1}}$ in the abstract model, there is at least one transition from a state in $h^{-1}(\widehat{s_i})$ to a state in $h^{-1}(\widehat{s_{i+1}})$ even though there is no transition from S_i to $h^{-1}(\widehat{s_{i+1}})$. We partition $h^{-1}(\widehat{s_i})$ into three subsets $S_{i,D}$, $S_{i,B}$, and $S_{i,I}$ as follows (compare Figure 14):

Deadend States	$S_{i,D} = S_i$
Bad States	$S_{i,B} = \{s \in h^{-1}(\widehat{s_i}) \mid \exists s' \in h^{-1}(\widehat{s_{i+1}}).R(s,s')\}$
Irrelevant States	$S_{i,I} = h^{-1}(\widehat{s_i}) \setminus (S_{i,D} \cup S_{i,B})$.

Thus, we have partitioned the abstract state $h^{-1}(\widehat{s_i})$ according to the above discussion. For illustration, consider again the example in Figure 10. Note that $S_1 = \{1,2,3\}$, $S_2 = \{4,5,6\}$, $S_3 = \{9\}$, and $S_4 = \emptyset$. The deadend state is $S_{3,D} = \{9\}$, the bad state is $S_{3,B} = \{7\}$, and the irrelevant state is $S_{3,I} = \{8\}$. Since $S_{i,B}$ is not empty, there is a spurious transition $\widehat{s_i} \rightarrow \widehat{s_{i+1}}$. This causes the spurious counterexample $\widehat{T}$. Hence

in order to refine the abstraction h so that the new model does not allow $\widehat{T}$, we need a refined abstraction function *which separates the two sets* $S_{i,D}$ *and* $S_{i,B}$, i.e., we need an abstraction function, in which no abstract state simultaneously contains states from $S_{i,D}$ and from $S_{i,B}$. In Figure 13, such a refinement of the partition is indicated by a thick line.

It is natural to describe the needed refinement in terms of equivalence relations: Recall from our discussion about the *fine structure of abstraction functions* that $h^{-1}(\widehat{s})$ is an equivalence class of $\equiv$ which has the form $E_1 \times \cdots \times E_n$, where each E_i is an equivalence class of $\equiv_i$. Thus, the refinement $\equiv'$ of $\equiv$ is obtained by partitioning the equivalence classes E_j into subclasses, which amounts to refining the equivalence relations $\equiv_j$. The *size of the refinement* is the number of new equivalence classes. Ideally, we would like to find the coarsest refinement that separates the two sets, i.e., the separating refinement with the smallest size. We can show however that this is computationally intractable.

Theorem 3. *(i) The problem of finding the coarsest refinement is NP-hard; (ii) when* $S_{i,I} = \emptyset$*, the problem can be solved in polynomial time.*

Thus, we conclude that it is the existence of the irrelevant states which makes the problem hard. (Intuitively, the existence of irrelevant states increases the number of possible solutions, and therefore, it is hard to identify the optimal one.)

The polynomial time symbolic algorithm **PolyRefine** corresponding to case (ii) of Theorem 3 is described in Figure 15. The algorithm uses the following notation: Let P_j^+, P_j^- be two projection functions, such that for $s = (d_1, \ldots, d_m)$, $P_j^+(s) = d_j$ and $P_j^-(s) = (d_1, \ldots, d_{j-1}, d_{j+1}, \ldots, d_m)$. Then $proj(S_{i,D}, j, a)$ denotes the *projection* set $\{P_j^-(s) | P_j^+(s) = a, s \in S_{i,D}\}$.

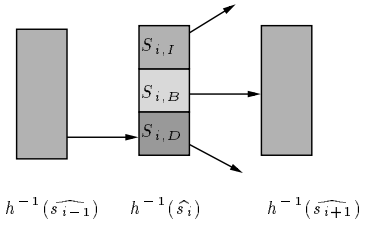

Fig. 14. Three sets $S_{i,D}$, $S_{i,B}$, and $S_{i,I}$

Algorithm PolyRefine
for j := 1 **to** m {
  $\equiv'_j := \equiv_j$
  for every $a, b \in E_j$ {
    if $proj(S_{i,D}, j, a) \neq proj(S_{i,D}, j, b)$
      then $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$ }}

Fig. 15. The algorithm **PolyRefine**

In the implementation [9], we use the following heuristics: We merge the irrelevant states in $S_{i,I}$ into $S_{i,B}$, and use the algorithm **Polyrefine** to find the coarsest refinement that separates the sets $S_{i,D}$ and $S_{i,B} \cup S_{i,I}$. The equivalence relation computed by **PolyRefine** in this manner is not optimal, but it is a correct refinement which separates $S_{i,D}$ and $S_{i,B}$, and eliminates the spurious counterexample. This heuristic has given good results in our practical experiments.

Our procedure continues to refine the abstraction function by partitioning equivalence classes until a real counterexample is found, or the ACTL property is verified.

The partitioning procedure is guaranteed to terminate since each equivalence class must contain at least one element. Thus, our method is complete.

Theorem 4. *Given a model M and an* ACTL *specification φ whose counterexample is either a finite path or a loop, our algorithm will find a model $\widehat{M}$ such that $\widehat{M} \models \varphi \Leftrightarrow M \models \varphi$.*

5 Directions for Future Research

Despite the progress in model checking made during the last twenty years, additional research is needed to realize the full potential of the method. Some particularly fertile research directions are listed below:

- Investigate the use of *abstraction*, *compositional reasoning*, and *symmetry* to reduce the state explosion problem.
- Develop methods for verifying *parametrized systems*, i.e., systems with arbitrarily many identical components.
- Develop practical tools for *real-time* and *hybrid* systems. Such systems involve both discrete variables and variables that change continuously with time.
- Investigate alternatives to BDDs for symbolic model checking, such as the use of efficient SAT procedures like GRASP [24].
- Combine model checking with *deductive verification*, i.e., automated theorem proving.
- Extend current model checking techniques to *software*, in particular safety-critical embedded systems that involve both hardware and software.
- Develop *tool interfaces* suitable for system designers. Temporal logic may not be the most perspicuous specification language for engineers.

Many of these topics, along with appropriate references, are discussed in [10].

References

1. H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27:217–274, 1998.
2. F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer-Aided Verification*, volume 697 of *LNCS*, pages 29–40, 1993.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, pages 35(8):677–691, 1986.
4. R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transaction on Computers*, pages 40:205–213, 1991.
5. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the 1991 International Conference on Very Large Scale Integration*, Aug. 1991. Winner of the Sidney Michaelson Best Paper Award.
6. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

7. P. Chauhan, E. Clarke, S. Jha, and H. Veith. Efficient image computation. Manuscript, 2000.
8. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *Software Tools for Technology Transfer*, 1998.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer-Aided Verification (CAV) 2000*, volume 1855 of *LNCS*. Springer, 2000. Full version available as Technical Report CMU-CS-00-103, Carnegie Mellon University.
10. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Publishers, 1999.
11. E. Clarke and H. Schlingloff. Model checking. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2000. to appear.
12. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, LNCS, 1981.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent system using temporal logic. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, January 1983.
14. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and System (TOPLAS)*, 16(5):1512–1542, September 1994.
15. E. M. Clarke Jr., E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, Apr. 1986.
16. O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
17. J. Feigenbaum, S. Kannan, M. Y. Vardi, and M. Viswanathan. Complexity of problems on graphs represented as OBDDs. *Chicago Journal of Theoretical Computer Science*, 1999.
18. G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: a deductive query language with linear time model checking. *ACM Transactions on Computational Logic (TOCL)*, 2001. Accepted for publication.
19. G. Gottlob, N. Leone, and H. Veith. Succinctness as a source of complexity in logical formalisms. *Annals of Pure and Applied Logic*, 97(1–3):231–260, 1999.
20. E. Grädel and I. Walukiewicz. Guarded fixed point logic. In G. Longo, editor, *Proc. 14th IEEE Symp. on Logic in Computer Science*, pages 45–54, 1999.
21. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
22. Y. Lakhnech. personal communication. 2000.
23. M. Li and P. Vitányi. *An introduction to Kolmogorov Complexity and its applications*. Spinger Verlag, New York, 1993.
24. J. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
25. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
26. I. Moon, J. H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, pages 26–28, Los Angeles, June 2000.
27. I. Moon and F. Somenzi. Border-block triangular form and conjunction schedule in image computation. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD'00)*, November 2000. To appear.
28. C. Pixley. A computational theory and implementation of sequential hardware equivalence. In R. Kurshan and E. Clarke, editors, *Proc. CAV Workshop (also DIMACS Tech. Report 90-31)*, Rutgers University, NJ, June 1990.

29. C. Pixley, G. Beihl, and E. Pacas-Skewes. Automatic derivation of FSM specification to implementation encoding. In *Proceedings of the International Conference on Computer Desgin*, pages 245–249, Cambridge, MA, Oct. 1991.
30. C. Pixley, S.-W. Jeong, and G. D. Hachtel. Exact calculation of synchronization sequences based on binary decision diagrams. In *Proceedings of the 29th Design Automation Conference*, pages 620–623, June 1992.
31. F. Somenzi. CUDD: CU decision diagram package. http://vlsi.colorado.edu/ fabio/.
32. H. Veith. Languages represented by boolean formulas. *Information Processing Letters*, 63:251–256, 1997.
33. H. Veith. How to encode a logical structure as an OBDD. In *Proc. 13th Annual IEEE Conference on Computational Complexity (CCC)*, pages 122–131. IEEE Computer Society, 1998.
34. H. Veith. Succinct representation, leaf languages and projection reductions. *Information and Computation*, 142(2):207–236, 1998.