# Minimal Assignments for Bounded Model Checking*

Kavita Ravi[1] and Fabio Somenzi[2]

[1] Cadence Design Systems
kravi@cadence.com
[2] University of Colorado at Boulder
Fabio@Colorado.EDU

**Abstract.** A traditional counterexample to a linear-time safety property shows the values of all signals at all times prior to the error. However, some signals may not be critical to causing the failure. A succinct explanation may help human understanding as well as speed up algorithms that have to analyze many such traces. In Bounded Model Checking (BMC), a counterexample is constructed from a satisfying assignment to a Boolean formula, typically in CNF. Modern SAT solvers usually assign values to all variables when the input formula is satisfiable. Deriving minimal satisfying assignments from such complete assignments does not lead to concise explanations of counterexamples because of how CNF formulae are derived from the models. Hence, we formulate the extraction of a succinct counterexample as the problem of finding a minimal assignment that, together with the Boolean formula describing the model, implies an *objective*. We present a two-stage algorithm for this problem, such that the result of each stage contributes to identify the "interesting" events that cause the failure. We demonstrate the effectiveness of our approach with an example and with experimental results.

## 1 Introduction

The success of model checking as a verification technique for both hardware and software depends to a large extent on the counterexamples that are produced for failing properties. However, the time consuming process of devising appropriate stimuli to test a model is often replaced by the laborious interpretation of the error trace returned by the model checker. For circuits with many inputs and outputs, analyzing an execution trace—be it a counterexample or a witness—may require examination of thousands of events. Not all these events are important, as in Fig. 1, where the "full" trace produced by the model checker is shown alongside a "reduced" counterexample from which irrelevant events have been removed.

The traces of Fig. 1 describe the solution to a puzzle obtained by model checking a property claiming that no solution exists. Four children (A, B, C, and D) must cross a river over a bridge that can be passed by only two children at the time. They have only one flashlight, which is needed to pass the bridge. When two children cross the river together, they walk at the speed of the slower one. Given the speeds of the children (1, 2, 4, and 5 minutes for A, B, C, and D, respectively) one seeks the quickest way for all four to cross the river. The model of the puzzle keeps track of the location of the children

* This work was supported in part by SRC contract 2003-TJ-920.

| | full | | | | reduced | | | |
|---|---|---|---|---|---|---|---|---|
| time | left / right | m-t-dest | sel1 | sel2 | left / right | m-t-dest | sel1 | sel2 |
| 0 | ABCDL/ | 0 | A | B | ABCDL/ | 0 | A | B |
| 1 | CD/ABL | 1 | B | A | CD/ABL | 1 | | |
| 2 | CD/ABL | 0 | B | A | CD/ABL | 0 | B | |
| 3 | BCDL/A | 1 | B | A | BCDL/A | 1 | | |
| 4 | BCDL/A | 0 | C | D | BCDL/A | 0 | C | D |
| 5 | B/ACDL | 4 | A | C | B/ACDL | 4 | | |
| 6 | B/ACDL | 3 | A | C | B/ACDL | 3 | | |
| 7 | B/ACDL | 2 | B | A | B/ACDL | 2 | | |
| 8 | B/ACDL | 1 | A | A | B/ACDL | 1 | | |
| 9 | B/ACDL | 0 | A | A | B/ACDL | 0 | A | A or B |
| 10 | ABL/CD | 0 | A | B | ABL/CD | 0 | A | B |
| 11 | /ABCDL | 1 | B | A | /ABCDL | 1 | | |
| 12 | /ABCDL | 0 | | | /ABCD | 0 | | |

**Fig. 1.** Two ways to present a counterexample. The values of a variable that have not changed since the previous time are shown in gray

and the light. When a child starts crossing the river, her location is immediately changed to reflect her destination, and a counter (m-t-dest) is started to keep track of how many minutes are needed to reach the destination. The model uses two inputs (sel1 and sel2) to choose one or two children who are to cross the river. A child may be chosen only if she is on the same bank as the flashlight. Moreover, the second child must be slower than the first. If, for example, sel1=B and sel2=A, as in the “full” example at time 2, then only B crosses the river.

Examination of the reduced trace shows that the selection inputs are only relevant when m-t-dest is 0. It also shows that at time 2 the value of sel2 is immaterial, given that sel1=B: Since A is faster than B, and C and D are on the opposite bank to the flashlight, B will go alone back to the left bank. At time 4, the slowest children C and D will carry the flashlight to the right bank. At time 9, C and D are slower than A and on the same bank. Hence, for A to go alone, sel2 must be neither C nor D. Finally, the position of the flashlight at time 12 is irrelevant, since the objective has been achieved. By contrast, the “full” output of the model checker shows many needless transitions, for instance, at time 5 for both sel1 and sel2. The elimination of irrelevant events can substantially simplify the designer’s task, and is the subject of this paper.

In SAT-based Bounded Model Checking (BMC), the set of counterexamples to a linear-time property is the set of solutions to a satisfiability (SAT) problem. A solution to a SAT problem is an assignment of values (1 and 0, or true and false) to the variables of a Boolean formula. A partial assignment will sometimes do: for instance, $a = 1$ is sufficient to satisfy $(a \vee b) \wedge (a \vee c)$. However, modern SAT solvers usually return complete assignments. Fig. 1 witnesses that they also pay no attention to minimizing input changes over time. In both cases, the speed of the solver would likely suffer if additional concerns were added to the basic one of finding a solution. In addition, SAT solvers are not designed to produce minimal solutions. Hence, detecting and removing unnecessary values from an assignment is best done as a post-processing.

Since many problems can be cast as SAT problems, finding minimal satisfying assignments to CNF formulae may have several applications. Hence, in Sect. 3, we describe that problem in some depth. In Sect. 4, however, we show why the minimization of model checking counterexamples should not be formulated that way, and present a computation of minimal counterexamples in two steps. The one described in Sect. 4.4 provides a stepping stone to the final solution, and is guaranteed to be verifiable by three-value simulation. In Sect. 5, we discuss related work. Section 6 reports experiments, and Sect. 7 concludes with a summary and future directions.

## 2 Satisfiability Solvers

An assignment $A$ for a Boolean formula $F$ over variables $V$ is a function from $V$ to $\{0,1\}$. An assignment $A$ is *total* if the function is total; otherwise it is *partial*. An assignment $A'$ is *compatible* with assignment $A$ if $A' \subseteq A$. A *satisfying* assignment $A$ for a formula $F$ is one that causes $F$ to evaluate to true. A minimal satisfying assignment $A$ for a Boolean formula $F$ is a satisfying assignment such that no $A' \subset A$ satisfies $F$. A minimal satisfying assignment for $F$ corresponds to a *prime implicant* of $F$.

A *least-cost* assignment can be defined by assigning costs to the variables. Given an assignment $A$ satisfying $F$, a least-cost (least-size) assignment compatible with $A$ is a cheapest (smallest) subset of $A$ that still satisfies $F$. Finding a least-size assignment compatible with an assignment $A$ corresponds to finding a largest cube of $F$ that contains $A$ or, equivalently, a minimal *blocking clause* of $F$ [9].

Given a Boolean formula $F$, the SAT problem involves finding whether a satisfying assignment for $F$ exists. This problem is NP-complete when $F$ is in circuit form or is in Conjunctive Normal Form (CNF). However, due to the wide variety of applications of this problem, much research has been spent in developing fast and efficient SAT solvers.

A SAT solver typically computes a total satisfying assignment for $F$, if one exists, otherwise returns an *UNSATISFIABLE* answer. In a SAT solver a Boolean formula $F$ is usually represented in CNF. For instance,

$$f = (a \vee b) \wedge (\neg b \vee d) \ . \tag{1}$$

A CNF formula is a conjunction of *clauses*. A clause is a disjunction of *literals*. A literal is a variable or its negation. In the sequel, it is also convenient to consider the CNF formula as a set of clauses and a clause as a set of literals. The literals in $f$ above are $a$, $b$, $\neg b$, and $d$; $b$ and $\neg b$ are the positive and negative literals (or *phases*) of variable $b$. The clauses of the formula are $(a \vee b)$ and $(\neg b \vee d)$. A satisfying assignment for $f$ is $\{(a,0),(b,1),(d,1)\}$. It is also convenient to use literals to designate variable-value pairs. For example, the above assignment is the set of literals $\{\neg a, b, d\}$.

A literal $l$ is true in assignment $A$ if $l \in A$; it is false in $A$ if $\neg l \in A$. A clause is *satisfied* by an assignment when at least one of its literals is true. A clause is *conflicting* if all of its literals are false. A clause that is neither satisfied nor conflicting is *undecided*. A *unit* clause consists of a single literal. A *pure literal* of a CNF formula $F$ is a variable that has the same phase in all the clauses of $F$ in which it appears.

A new family of SAT solvers [12,14,10,7] have evolved that incorporate engineering advances atop the basic DPLL algorithm [6,5] and can solve many industrial-strength problems in applications such as BMC. The optimizations include non-chronological backtracking, conflict-driven learning, and efficient Boolean constraint propagation using the two-literal watching scheme. (Refer to [15] for a survey.) We discuss our problem and solution in the context of the Zchaff solver.

*Boolean Constraint Propagation:* Implications in these SAT solvers are computed using Boolean Constraint Propagation (BCP). Each clause, except unit clauses, contains two literals that are marked as *watched* literals [14,10]. In computing the satisfying assignment, the SAT solver makes *decisions* or assignments by setting values to variables. As decisions are made, a clause is updated only if one of the two watched literals in the clause is rendered false by the decision. The clauses in which the false literal is watched are examined to find another literal to watch instead. If such a literal is not found, then the other watched literal is implied (*unit clause rule*). The clause then becomes the *antecedent clause* of the implication. This implied value is again propagated using BCP.

*Implication Graph:* Given a satisfying assignment of a CNF formula, and a subset of the assignment marked as decisions, one can construct an *implication (hyper-)graph*, $G_I = (V, E)$. The nodes of this graph $V$ represent the literals of the assignment and the roots represent the decisions. Each *directed hyperedge* $E \subseteq 2^V \times V$ represents an implication, caused by an antecedent clause. A vertex (literal) can be the destination of multiple hyperedges, associated with multiple antecedent clauses. Equation 1 yields the following implication graph if $\neg a$ is marked as decision.

$$\neg a \longrightarrow b \longrightarrow d$$

The literal $\neg a$ is the root of this graph. The first implication is caused by the antecedent clause $(a \vee b)$ and the second implication is caused by $(\neg b \vee d)$.

## 3 Minimal Satisfying Assignments for CNF Formulae

When a SATISFIABLE answer is found in a SAT solver, all variables are assigned a value by either a decision or an implication, and at least one watched literal in each clause is true. The two watched-literals scheme is critical for the efficiency of BCP: It drastically reduces the number of clauses examined when a variable is updated. For efficiency, the SAT solver does not keep track of the information required to detect a partial assignment that is sufficient for the satisfaction of the formula. In any case, even the termination criteria that check for the actual satisfaction of the formula (in either CNF or circuit form), rather than checking for complete assignments, do not guarantee a minimal satisfying assignment. (The assignments made after all clauses have been satisfied are certainly redundant, but some assignments made before may also be.) It is therefore interesting to discuss how SAT algorithms may be modified to meet that requirement. It is also interesting to discuss the related issue of generating least-size assignments.

**Candidate Variables:** *Lifting* is the process of removing literals or, equivalently, variables from a satisfying assignment such that for each valuation of the lifted variables the formula is still satisfied. Recall that implied variables result from BCP applied to decisions. It can be easily proved that implied variables cannot be lifted: Only decision variables can be lifted. The literals in the unit clauses are considered to be implications, hence cannot be lifted; the pure literals, on the other hand, are not implications.

**Negligible variables and greedy removal:** The observation that when a satisfying assignment is found, at least one of the two watched literals in a clause must be true can be used to lift some literals. For each literal, the SAT solver maintains the list of clauses in which the literal is watched. The literals in the satisfying assignment that have empty lists and that do not appear in unit clauses can be removed from the satisfying assignment. All these *negligible* variables can be lifted simultaneously. Notice that none of the negligible variables are implied. Further lifting may be achieved by marking one true literal for each clause. All literals in the assignment that remain unmarked after all clauses are scanned can be lifted. Neither the negligible-variable approach nor the greedy approach yields a minimal assignment.

**Brute-Force Lifting:** One can check the possibility of lifting the remaining variables by flipping the value of each candidate variable while retaining the rest of the assignment, and checking satisfiability of the formula. If the formula is satisfiable, the variable can be lifted and the formula updated by dropping the literals associated with this variable (universal quantification of this variable from the formula). The resulting minimal assignment depends on the order in which the variables are considered and is not necessarily of least-cost.

**Covering Approach:** Least-cost assignments can be found by solving covering problems. A covering problem consists of minimizing $\sum_{1 \leq i \leq n} c_i \cdot x_i$, subject to constraint $\Gamma(x_1, \ldots, x_n) = 1$, where $\Gamma$ is a Boolean formula, and the costs $c_i$'s are non-negative integers. We assume that $\Gamma$ is in CNF. If all variables in $\Gamma$ are pure literals, we talk of *unate* covering; otherwise, of *binate* covering. Unate covering is also known as *set covering*. Covering problems can be solved by generic 0-1 integer linear program solvers [11], or by dedicated algorithms based on branch and bound [3].

A unate covering problem may be solved to find a least-cost solution compatible with a starting assignment, while one independent of any particular satisfying assignment from the SAT solver can be obtained by solving a binate covering problem with one variable per literal in the CNF formula.

The above described techniques may be useful toward deriving minimal assignments that are sufficient to show the satisfiability of the given formula. The removal of negligible variables and the greedy scheme are fast and efficient. Brute-force lifting can be implemented in time quadratic in the size of the CNF formula. Finally, the covering approach is expensive, especially when seeking a globally least-cost solution.

## 4 Bounded Model Checking and Minimal Assignments

In this section, we first introduce Bounded Model Checking and then discuss the problem of minimal assignments for concise counterexamples.

### 4.1 Bounded Model Checking

Bounded model checking [1] is a technique to find bounded-length counterexamples to linear time properties. Recently, BMC has been applied with great success by formulating it as a SAT instance and solving it with the improved SAT solvers mentioned in Sect. 2. For simplicity, we only discuss invariant properties here.

We consider finite systems given as circuits composed of memory elements and combinational gates. Nondeterminism in these circuits is only due to primary inputs. Let $T(X, W, X', U)$ be the formula for the transition relation where $X$, $W$, $X'$, and $U$ are sets of variables representing the present states, primary inputs, next states, and gate outputs of the system. The timed version $T_i(X_i, W_i, X_{i+1}, U_i)$ represents the transitions between the $i$-th and the $(i+1)$-th step of the system.

Given a system with a Boolean formula $I$ representing the initial states, a Boolean formula $T_i(X_i, W_i, X_{i+1}, U_i)$ representing the $i$-step transition relation and an invariant with a Boolean formula $P_k$ representing the failure states at step $k$, the length-$k$ BMC problem is posed as a SAT instance in the following manner:

$$F = I \wedge P_k \wedge \bigwedge_{0 \le i < k} T_i(X_i, W_i, X_{i+1}, U_i) \ . \tag{2}$$

We assume that a Boolean formula is translated into CNF by introducing intermediate dependent variables in the standard fashion. If the property may fail in $k$ transitions, the SAT solver computes a satisfying assignment that describes a counterexample to the property. If the SAT solver produces a total assignment, all variables of the system appear in the counterexample.

## 4.2 Motivation and Definitions

Applying the techniques described in Sect. 3 to minimize the set of variables in the counterexample to the BMC problem does not give much reduction in the number of variables. The experiments we performed using these methods yielded $2\%$ to $7\%$ reduction. We explain the poor performance with the following example.

*Example 1.* Consider a tautologous circuit with input $a$, output $g_1$, and such that $g_1 = a \vee g_0$ and $g_0 = \neg a$. Suppose that a SAT solver is given the following CNF formula to prove that $g_1$ can be set to 1:

$$F = (a \vee g_0) \wedge (\neg a \vee \neg g_0) \wedge (\neg g_0 \vee g_1) \wedge (\neg a \vee g_1) \wedge (a \vee g_0 \vee \neg g_1) \wedge g_1 \ .$$

Suppose the solver propagates $g_1 = 1$, decides $a = 1$, which in turn implies $g_0 = 0$. This is minimal because the satisfying assignment has to express that the inverter output is consistent with its input. Hence, it must contain both $a$ and $g_0$, though $\exists g_0 \,.\, F = g_1$ does not depend on $a$. The intermediate variable $g_0$ constrains the satisfying assignment beyond what is required to justify the objective $g_1$.

The example shows why the methods of Sect. 3 are not appropriate for the computation of concise counterexamples: A counterexample shows a sequence of transitions from the initial states to the failure states; the most important event in the counterexample is the failure. Hence, "events" implying the failure are primary for the purpose of debugging, whereas the others convey secondary information. We want to distill the primary events out of a counterexample. Further, a user may choose to only see events related to an "interesting" set of variables. Our formulation is aimed at extracting an assignment that implies the failure with a minimal set of "interesting" events, excluding those that have no bearing on the justification of the objective.

To formalize these notions, we assume that we are given a satisfying assignment to (2) and another formula

$$\hat{F} = P_k \wedge \bigwedge_{0 \leq i < k} T_i(X_i, W_i, X_{i+1}, U_i) \ . \tag{3}$$

We want the ability to lift the initial state literals in the satisfying assignment to focus attention on those memory elements that are important to the counterexample. Hence, $I$ is absent from (3). Sometimes, a bounded model checker will add additional constraints to (2) such as the non-existence of counterexamples of length less than $k$ ($\bigwedge_{0 \leq i < k} \neg P_i$) to aid the SAT solver. It is important that these constraints be absent from (3) for maximal freedom in lifting variables in the satisfying assignment. Each gate in the circuit, which corresponds to a subformula, translates to a set of CNF clauses. We assume that the given CNF is such that in each clause not in $P_k$, we can identify the literals that correspond to the inputs and output of the gate that produced it.

We assume that we are given a set of *roots* $R \subseteq V$ that correspond to the inputs to the unrolled circuit in the BMC problem. These roots $R$ form a set of independent variables in the formula, in the sense that $\bigwedge_{0 \leq i < k} T_i$ is satisfiable for every assignment to the variables in $R$. The values of the remaining variables $V \setminus R$ in $\bigwedge_{0 \leq i < k} T_i$ are determined by the valuation of $R$. In the BMC context, the roots are the variables representing the initial states ($X_0$) and the inputs ($W_i$, $i < k$). The user may mark some variables $S \subseteq V \setminus R$ as interesting. Typically, these are the state variables of the system and therefore, of interest in the counterexample.

An *objective* of the formula is defined as a set of clauses whose variables are non-overlapping with the set of roots $R$. For the sake of discussion, we assume that an objective $o$ is represented by a unit clause (a single literal) in the formula[1]. Henceforth, we will refer to $o$ and the literal in $o$ interchangeably. The objective denotes conditions that *must* be implied by the minimal assignment. In the BMC problem, the objective $o$ is a literal stating the failure of the formula at step $k$, $P_k$.

## 4.3 Lifting for a Minimal Assignment

In this section, we describe the lifting of literals from the given satisfying assignment $A$ relative to the objective $o$. We adapt the brute-force lifting method described in Sect. 3. This is a powerful method to eliminate literals that yields a minimal assignment, albeit order-dependent. Instead of lifting a literal if the resulting assignment still satisfies the given formula, here we want to lift a literal if the resulting assignment is still sufficient to imply the objective.

The procedure is described in Fig. 2. The arguments to this procedure are the formula of (3), $\hat{F}$, the objective $o$, the root literals in the original assignment, $A_R$, and the

[1] There is no loss of generality in this assumption as we can augment the formula with the definition of the literal $o$ representing the conjunction of the clauses in the objective and adding the literal itself to the formula.

interesting literals in the original assignment, $A_S$. The lifting of roots is tried first. For each root literal $l$, the negation of the objective $o$ is asserted in place of $o$ in $\hat{F}$. Let $\bigwedge(A_R \setminus \{l\})$ denote the conjunction of literals in $A_R$ except $l$. A satisfiability check (SAT_Solve($F'$)) whether the literals in $A_R$ except $l$ together with the formula $\hat{F}$ fail to imply the objective $o$. If the check results in a satisfiable assignment, then $l$ cannot be lifted; otherwise, both $l$ and $\neg l$ imply the objective and can therefore be lifted from the existing assignment $A_R$. $A_R$ is then updated by removing the lifted literal.

Once the lifting of the roots is determined, the remaining variables in $A_S$ are checked for an implied value. If a variable in $A_S$ can take both values due to the reduced set of roots $A_R$, then it is not added to the assignment $A'$, else it is. The value of this variable in the original complete assignment is known to be possible. An additional SAT check determines whether the opposite value is consistent with the current partial assignment. Since $R$ is an independent set, $\hat{F}$ is satisfiable for any value of the lifted roots. By contrast, since $S$ is dependent on $R$, all we can say about the absent variables of $S$ in $A'$ is that they take different values as the values of the lifted roots are changed. Unlike the literals in $A_R$, those in $A_S \setminus A'$ are not really lifted according to our definition: They are simply not implied by the partial root assignment.

```
Brute_Force_Lifting($\hat{F}$, $o$, $A_R$, $A_S$)
    $F''$ = substitute $o$ with $\neg o$ in $\hat{F}$;
    for each literal $l$ in $A_R$
        $F' = F'' \wedge \bigwedge(A_R \setminus l)$;
        if (SAT_Solve($F'$) $\neq$ SATISFIABLE) $A_R = A_R \setminus l$ ;
    $A' = A_R$;
    for each literal $l$ in $A_S$
        $F' = \hat{F} \wedge \neg l \wedge \bigwedge A'$;
        if (SAT_Solve($F'$) $\neq$ SATISFIABLE) $A' = A' \cup l$;
    return $A'$;
```

**Fig. 2.** Brute-force lifting algorithm

This method of lifting roots is effective in deriving a minimal satisfying assignment and isolating the primary events responsible for the failure of a property. Such a counterexample may help in more effective debugging as in the example of Fig. 1. Sometimes, however, the resulting counterexample may have discontinuities due to the missing values of $S$ variables, thereby requiring some reconstruction by the user. Consider, for instance, a gate $g_2 = (g_0 \equiv g_1)$ such that, under the minimal assignment $A'$, both $\{g_0, g_1\}$ and $\{\neg g_0, \neg g_1\}$ are possible for some values of the lifted roots, but $\{g_0, \neg g_1\}$ and $\{\neg g_0, g_1\}$ are not possible. Then $A'$ would contain $g_2$, but no literal for either $g_0$ or $g_1$. In cases like these, it is useful to have an intermediate counterexample where the trace has fewer holes yet lifts to the counterexample generated by Fig. 2. Ideally, the intermediate counterexample should be three-valued simulatable (see Sect. 4.4) for easy interpretation and validation.

To prove that lifting a root does preserve the implication of the objective, the brute-force lifting algorithm goes through a sequence of unsatisfiability checks. This prevents the application of universal quantification after a successful lifting, which is what keeps the satisfiability checks trivial and ultimately gives quadratic complexity to the lifting algorithm of Sect. 3. The lifting procedure of this section requires solving one SAT instance (one SAT_Solve() call) per candidate variable. This may be expensive but it is possible to solve the multiple SAT_Solve() calls incrementally. Also, the variables of $A_S$ implied by BCP of the reduced set of roots need not be considered for lifting. On the other hand, an intermediate assignment, if cheap to compute, can reduce the number of calls to SAT_Solve() by providing a smaller assignment to start with. This is indeed true of the intermediate assignment proposed in the following section.

## 4.4 Implication Graph and a Covering Solution

In three-valued logic simulation of a circuit, a signal has one of three values: 0 (false), 1 (true), and $X$ (unknown). Boolean operators are extended to the three-valued domain in a conservative manner. For instance, $a \vee \neg a = X$ if $a = X$. As a consequence, if three-valued simulation assigns a signal in a combinational circuit value 0 or 1 for a certain input assignment, then that signal is guaranteed to have the same value under all possible replacements of $X$'s by 0's and 1's. One easily sees that BCP in a SAT solver is closely related to three-valued simulation when "undecided" is interpreted as "unknown." In particular, three-valued simulation of a partial assignment to a CNF formula will imply the value of the objective if and only if BCP does.

An assignment that can justify the objective by pure BCP has two important advantages: On the one hand, it can be understood by a designer by a sequence of easy, local steps. On the other hand, it can be verified in linear time by widely available independent tools. The drawback of a three-valued simulatable assignment is, of course, that it may not be minimal.

We obtain a three-value simulatable partial assignment from the original assignment $A$ produced by the SAT solver using an implication graph. Consider the implication graph where the roots are $A_R$, the root literals in the original assignment. We view these as decisions since they are literals of the independent set $R$. Note that this implication graph must contain the objective $o$ since $o$ is in $A$ and is part of the dependent set. Therefore, we can use this implication graph to obtain a subset of $A_R$ and the interesting variables $A_S$ that transitively imply $o$. This is closely related to the work of [9] where a similar technique is used to enumerate a cover of the formula in terms of a defined set of roots (primary inputs and state variables). Our work is applied in a different context and is extended as explained below.

Our method is described in Fig. 3. The arguments to this procedure are the formula $\hat{F}$ from (3), $A$, $A_R$, $o$, and a cost function described below. First, the literals in $A_R$ are added to the formula $\hat{F}$. The next step involves deriving the input-to-output implications. Implications where the implied literal is the output of a gate are valid input-to-output implications. (Note that the constructed CNF is such that there is only one output literal per clause.) The part of the implication graph that leads to $o$ is traced with a backward search from $o$ to the roots. (The input-to-output implication derivation can be combined with this backward step.) The resulting hypergraph $G$ is a connected sub-graph of $G_I$

since $F'$ is constructed from a Boolean formula representing a circuit and $o$ is a net in the circuit.[2] Its roots are a subset of $A_R$ and its nodes are a subset of $V$. The nodes of $G$ form a lifting of the original assignment $A$.

```
Implication_Graph_Based_Lifting($\hat{F}$, $A$, $A_R$, $o$, $weights$)
    $F' = \hat{F} \wedge \bigwedge A_R$;
    $G_I$ = derive input-to-output implications ($F'$);
    $G$ = trace graph backwards($G_I$, $o$);
    $B$ = formulate binate covering problem($G$);
    $B'$ = Binate Covering Solver($B$, $weights$);
    $A'$ = map $B'$ onto literals of $A$;
    return $A'$;
```

**Fig. 3.** Binate covering-based lifting algorithm

If $G$ is a hyper-graph with at least one set of multiple implications to a literal, it is possible to further lift literals from the assignment. For this, we use a binate covering solver with a user-defined cost function $weights$ that is $\sum_{1 \leq i \leq n} c_i \cdot x_i, x_i \in A_R \cup A_S$. The cost function describes the relative importance of the variables to be lifted. For example, a large $c_i$ associated with a root variable will result in it being lifted rather than a variable with a small $c_i$. Our goal is to lift as many roots as possible in order to identify those roots that are critical in implying the objective.

The covering problem is formulated thus: Each node of the graph corresponds to a variable. The variable is 1 if the node is in the subgraph that is selected. A subgraph must satisfy the following constraints, which guarantee that the objective is satisfied: (1) The objective variable must be true. (2) If a variable is true, and it is not in $A_R$, then all the variables of at least one hyperedge into it must be true.

As a simple example, if the objective is $a$, and $a$ is implied by $b \wedge c$, or by $c \wedge \neg d$ in the implication hypergraph, the constraints are

$$a \wedge (a \rightarrow c \wedge (b \vee \neg d)) = a \wedge c \wedge (b \vee \neg d) \ .$$

The optimal solution depends on the costs of $b$ and $d$.

The result of the binate covering solver is mapped to a satisfying assignment $A'$ by picking the literals corresponding to the variables in $B'$ from $A$. The result of Fig. 3 is a least-cost assignment that forms a connected subset of nodes in the implication graph $G$ from the roots to the objective.

Such an assignment is three-valued simulatable. It can be presented to the user as a series of implications that lead to the error. The counterexample is easy to understand, yet pruned of the irrelevant implications. There are fewer variables to pay attention to.

Though the restriction to three-valued simulatability means that the implication graph approach is incomplete, it may remove literals from the assignment that the lifting

[2] Here is where we depart from the approach of [9], which is based on a simple graph or, equivalently, a hypergraph in which the in-degree of a node is at most 1.

approach would retain. The latter never discards from $A_S$ a literal $l$ implied by the roots in $A'$. By contrast, the implication graph approach may do so when the value of $l$, although known, is immaterial to the objective. An example is provided by the location of the flashlight at time 12 in Fig. 1.

The implication graph in a SAT solver alone cannot exclude that a certain variable may be lifted. For instance, the implication graph for Example 1 has one root, $a$, and two leaves, $\neg g_0$ and $g_1$. Tracing back from the objective $g_1$ shows that $\{a\}$ is sufficient to imply $g_1$; however, this is not minimal. On the other hand, if we rewrote $F$ as $g_1 \wedge (a \vee g_0) \wedge (\neg a \vee \neg g_0)$ to highlight that $g_1$ is implied by $F$, the implication graph would have no edges, and the minimal empty assignment would be found.

The implication graph derived in the SAT solver is closely linked to the representation of the circuit and to the translation of the design into the CNF formula. A redundant circuit, an inefficient translation or state encoding may mask several implications.

*Example 2.* An 8-state finite state machine is shown in Fig. 4. The machine has one input $i$. Each transition is annotated with the values of $i$ that enable it. The three binary state variables, $c$, $b$, and $a$, form the state vector $state$. If one is interested in the reachability of $G$, the most interesting event, or *objective*, is the state vector taking value $G$; the primary events of interest are those that are critical to causing it. It is easily seen that all paths of length 3 from $A$ lead to $G$. Hence, a minimal witness to the reachability of $G$ in terms of initial state and primary inputs is $\{\neg c_0, \neg b_0, \neg a_0\}$. (Each subscripted variable represents the value of that variable at the time step of the subscript.)

Suppose we start with the following solution obtained by the SAT solver after quantification of the internal variables: $\{\neg c_0, \neg b_0, \neg a_0, i_0, \neg i_1, \neg i_2\}$. This corresponds to the path $A \to C \to E \to G$ in Fig. 4. We analyze the implication graph caused by this assignment in the SAT solver whose leaf is the literal representing $G$ in the third step and whose roots are the initial states and inputs. It shows that $i_2$ and $i_1$ can be lifted. Knowing that the second state of the path is $C$ is enough to infer that $\{c_2, \neg b_2\}$, which in turn is enough to infer that the objective is satisfied. This works because $E$ and $F$ have adjacent encodings. If we look at the internal nodes in the implication graph corresponding to $\{\neg c_0, \neg b_0, \neg a_0, i_0\}$ we find that $a_2$ is not included. That is, we have the path

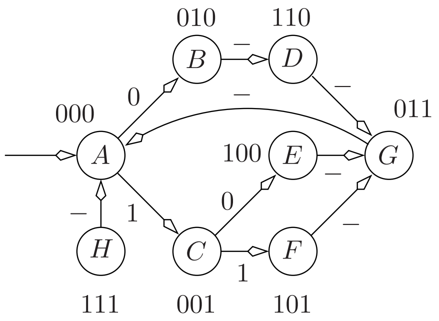

**Fig. 4.** State diagram for minimal error trace example

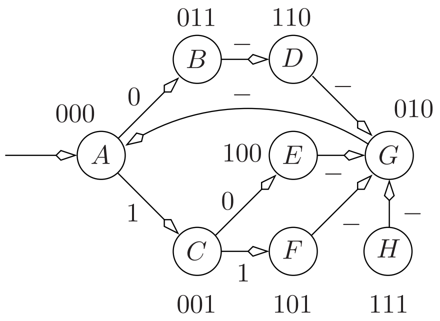

**Fig. 5.** Another state diagram for minimal error trace example

$$A \to C \to \{E, F\} \to G \ .$$

However, we cannot lift $i_0$, for we would not know $b_1$ and $a_1$, and consequently lose track of the fact that we are making progress toward $G$. Now let us change the implementation of Fig. 4 as shown in Fig. 5. From state $= A$ we can imply state $= \{B, C\}$ at the next time, because $B$ and $C$ have adjacent encodings, and from state $= \{B, C\}$ we can imply state $= \{D, E, F, H\}$ at the next time. Finally, from state $= \{D, E, F, H\}$ we can imply state $= G$ at the next time. (All without making assumptions about $i$.) This corresponds to the path

$$A \to \{B, C\} \to \{D, E, F, H\} \to G \ .$$

Note the importance of changing the successor of the unreachable state $H$.

In terms of run time, Implication_Graph_Based_Lifting() involves deriving input-output implications (cost is proportional to the size of the formula), one pass over the implication graph backwards from $o$, which can be combined with the formulation of the Binate Covering Problem, and one call to the Binate Covering Solver. In practice, the combination of these three steps is often cheaper than the Brute_Force_Lifting() algorithm for large CNF formulae. The assignment $A'$ can be provided as an argument $A$ to the Brute_Force_Lifting() algorithm. As discussed in Sect. 4.3, the expense of lifting may be mitigated by the reduction of literals by the algorithm in Implication_Graph_Based_Lifting().

The techniques of Fig. 2 and Fig. 3 complement each other. While the former is powerful, is unaffected by translation inefficiencies and is effective in finding the consequences of the primary events responsible for causing failure, the latter counterexample is three-valued simulatable, eliminates events that are implied but have no bearing on the achievement of the objective, and can serve as a guide in interpreting the result of the former by filling some holes.

## 5 Related Work

As mentioned in Sect. 4.4, the work of McMillan [9] is related to our lifting based on analyzing the implication graph. While his work applies this technique toward enumerating a cover, our work applies it to making traces more understandable. We extend this technique by minimizing the set of implications that justify the objective using a binate covering solver to find a minimal cover of a hypergraph. Brute-force lifting as described in Sect. 4.3 is also feasible since we only need one satisfying assignment.

In general, our problem is related to Quantified Boolean Formulae in terms of existentially quantifying the non-interesting variables but is different in that we have the notion of an objective, weights assigned to different variables, and causality between the inputs (roots) and the output (objective).

Minimization of counterexamples is addressed in [8]. The authors there distinguish between "control" and "data" signals in the counterexample and try to obtain long segments of the counterexample in which the data signals have *don't-care* values. Our work does not need the partition of control and data signals. The advantage of our method is that when segments with data signal don't-cares do not exist, different don't cares (perhaps control signals) will be extracted. On the other hand, the disadvantage is that even if long segments of data don't-cares exist, this algorithm may not find them since it has no control on which don't cares are found.

Minimization of counterexamples is useful in the context of abstraction-refinement [4]. Refinement is often more effective when it is based on the simultaneous elimination of a set of counterexamples rather than on elimination of one counterexample at a time [13]. Therefore, our technique can be applied in this context also.

## 6 Experimental Results

We implemented the algorithms of Sections 4.4 and 4.3 by modifying Zchaff and its interface to the model checker Vis [2]. The lifting based on the implication graph mechanism of Fig. 3 uses the binate covering solver Mincov [3]. The algorithm described in Fig. 2 is implemented in a non-incremental manner.

To enable lifting, Vis generates an auxiliary file that contains the roots $R$, interesting variables $S$, and the objective $o$. The $R$ variables are the inputs of all the time frames and the initial state variables. The $S$ variables contain the state variables over the various time frames. In case a satisfying assignment is found, this file is read by the SAT solver to perform the lifting algorithm described in Sect. 4.4. The cost of each variable in $R$ is 1 and the cost of each variable in $S$ is 0. The result of this algorithm is used as the starting point of the algorithm as described in Sect. 4.3.

**Table 1.** Experimental results of lifting based on Fig. 3 and Fig. 2

| Example | Vars | CE | Original | | | Mincov Heuristic | | | | | | Optimal | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | mincov | | | lift | | | mincov | | |
| | | | R | S | Time | R | S | Time | R | S | Time | R | S | Time |
| s1269b | 67 | 3 | 12 | 21 | 0.01 | 0.50 | 0.71 | 0.00 | 0.50 | 0.71 | 0.00 | 0.50 | 0.71 | 0.00 |
| avq | 388 | 3 | 30 | 102 | 0.03 | 0.20 | 0.15 | 0.05 | 0.10 | 0.09 | 0.01 | 0.20 | 0.15 | 0.04 |
| min | 536 | 3 | 6 | 15 | 0.03 | 0.67 | 0.87 | 0.01 | 0.50 | 0.53 | 0.03 | 0.67 | 0.87 | 0.01 |
| river | 1952 | 5 | 32 | 67 | 0.12 | 0.66 | 0.69 | 0.04 | 0.47 | 0.42 | 0.88 | 0.66 | 0.69 | 0.05 |
| vending | 4266 | 2 | 35 | 93 | 0.23 | 0.57 | 0.35 | 0.03 | 0.46 | 0.26 | 0.63 | 0.57 | 0.35 | 0.05 |
| flashlight | 6110 | 12 | 56 | 152 | 4.45 | 0.93 | 0.97 | 0.10 | 0.46 | 0.80 | 14.17 | 0.93 | 0.97 | 0.09 |
| b10 | 13466 | 8 | 103 | 223 | 1.33 | 0.59 | 0.58 | 0.20 | 0.17 | 0.33 | 19.25 | 0.57 | 0.52 | 0.19 |
| b13 | 14073 | 10 | 52 | 372 | 0.83 | 0.38 | 0.42 | 0.58 | 0.29 | 0.34 | 25.81 | 0.31 | 0.36 | 8.44 |
| b05 | 20238 | 7 | 33 | 215 | 1.23 | 0.58 | 0.71 | 0.54 | 0.12 | 0.53 | 13.74 | 0.21 | 0.49 | 0.55 |
| s1423 | 30212 | 61 | 1105 | 5253 | 26.66 | 0.49 | 0.35 | 3.94 | 0.28 | 0.29 | 2336.92 | — | — | $>2h$ |
| b09 | 30650 | 20 | 47 | 587 | 1.92 | 0.30 | 0.40 | 0.43 | 0.09 | 0.30 | 44.78 | 0.21 | 0.39 | 0.46 |
| ns1 | 43756 | 8 | 184 | 632 | 14.95 | 0.48 | 0.42 | 0.70 | 0.25 | 0.26 | 562.42 | 0.48 | 0.42 | 0.73 |
| b07 | 56456 | 28 | 57 | 869 | 5.24 | 0.51 | 0.31 | 2.06 | 0.05 | 0.20 | 208.21 | 0.51 | 0.31 | 2.06 |
| usb_phy | 71105 | 36 | 544 | 3280 | 19.87 | 0.22 | 0.31 | 1.82 | 0.14 | 0.26 | 1047.56 | 0.21 | 0.31 | 1.81 |
| blackjack | 117377 | 13 | 168 | 1507 | 54.38 | 0.60 | 0.27 | 1.94 | 0.48 | 0.15 | 861.38 | 0.59 | 0.27 | 1.98 |
| Average | | | | | | 0.51 | 0.50 | | 0.29 | 0.37 | | 0.47 | 0.49 | |

All experiments were run on a Sun Fire 280R machine with 2GB of memory. The results are presented in Table 1. We ran 15 examples of varying sizes. The first column presents the name of the example, the second column contains the number of variables presented to the SAT solver, and the third column shows the length of the counterexample. Each set of three columns subsequently presents results for the original satisfying assignment, results for the three-valued simulatable counterexamples from Fig. 3 and results of further lifting based on Fig. 2, respectively. Both a heuristic and an exact solution were extracted from Mincov. The last three columns show the results of the exact solution. The three columns in each set give the number of root variables $R$ in the resulting assignment, the number of interesting variables $S$, and the time in seconds taken to extract the assignments. The $R$ and the $S$ columns for the last three sets are presented as the fractions of the original numbers of $R$ and $S$ literals in the "full" set.

The results in the table show that applying the algorithm of Sect. 4.4 can provide as much as $80\%$ reduction in the number of root literals and $85\%$ reduction in the number of interesting literals with the average reduction being $49\%$ and $50\%$ of the original literals, respectively. The data for brute-force lifting are even more impressive, the average reduction in root count being $71\%$ of the initial literals. In most cases the Mincov heuristic solutions result in reductions in literals comparable to those of the optimal solutions and can be found quickly for all examples. Further, the run time of the Implication Graph Based Lifting algorithm is often only a fraction of the original run time and the subsequent lifting phase recovers most of the differences between heuristic and exact solutions. We also compared the heuristic solution to a greedy selection of an implication graph. The two methods produce similar literal reductions, but the greedy selection puts a larger burden on the lifting algorithms, and is therefore slower.

The CPU times for the brute-force lifting of the literals are large due to the cost of one call to SAT_Solve() per candidate literal to be lifted. The performance of the brute-force lifting method may improve by the use of an incremental SAT solver. Applying BCP in the brute-force lifting algorithm to the the roots selected by binate covering may also help decrease the run time by reducing the candidate literals.

## 7 Conclusions

We have discussed and presented algorithms for minimal satisfying assignments for CNF formulae, both in general and in the context of Bounded Model Checking. Our experiments show that our technique is effective in terms of reduction of literals. The reduced counterexamples are often more instructive in debugging designs. Although lifting is sometimes expensive, there is room for improvement of its performance. Besides, the Implication Graph Based Lifting is quite effective in the reduction of literals.

In Example 2 of Sect. 4.4, the failing property states that $G$ is never reached. This is a form of vacuous failure, because, in fact, $G$ occurs along all paths from the initial state. Vacuity is related to the inputs' playing no role in the failure of the property. The precise nature of this connection remains the subject of future investigation together with the extension to full LTL model checking.

**Acknowledgment.** We thank Mohammad Awedh, HoonSang Jin, and Bing Li for help with Vis and Zchaff.

## References

[1] 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, Mar. 1999. LNCS 1579.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. of California, Dec. 1995.

[3] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 316–319, Santa Clara, CA, Nov. 1989.

[4] 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.

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

[6] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[7] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[8] 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, Apr. 2002. LNCS 2280.

[9] 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.

[10] 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.

[11] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[12] 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, Nov. 1996.

[13] C. Wang, B. Li, H. Jin, G. D. Hachtel, and F. Somenzi. Improving Ariadne's bundle by following multiple threads in abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 408–415, Nov. 2003.

[14] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[15] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Fourteenth International Conference on Computer Aided Verification, (CAV'02)*, pages 17–36, Copenhagen, Denmark, 2002. LNCS 2404.