Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis*

Pankaj Chauhan[1] Edmund Clarke[1] James Kukula[3]
Samir Sapra[1] Helmut Veith[2] Dong Wang[1]

[1] Carnegie Mellon University [2] TU Vienna, Austria
[3] Synopsys Inc., Beaverton, OR

Abstract. We introduce a SAT based automatic abstraction refinement framework for model checking systems with several thousand state variables in the cone of influence of the specification. The abstract model is constructed by designating a large number of state variables as *invisible*. In contrast to previous work where invisible variables were treated as free inputs we describe a computationally more advantageous approach in which the abstract transition relation is approximated by *pre-quantifying* invisible variables during image computation. The abstract counterexamples obtained from model-checking the abstract model are symbolically simulated on the concrete system using a state-of-the-art SAT checker. If no concrete counterexample is found, a subset of the invisible variables is reintroduced into the system and the process is repeated. The main contribution of this paper are two new algorithms for identifying the relevant variables to be reintroduced. These algorithms monitor the SAT checking phase in order to analyze the impact of individual variables. Our method is complete for safety properties ($\mathbf{AG}\, p$) in the sense that – performance permitting – a property is either verified or disproved by a concrete counterexample. Experimental results are given to demonstrate the power of our method on real-world designs.

1 Introduction

Symbolic model checking has been successful at automatically verifying temporal specifications on small to medium sized designs. However, the inability of BDD based model checking to handle large state spaces of "real world" designs hinders the wide scale acceptance of these techniques. There have been advances on various fronts to push the limits of automatic verification. On the one hand, improving BDD based algorithms improves the ability to handle large state machines, while on the other hand, various abstraction algorithms reduce the size of the design by focusing only on

* This research is sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the Gigascale Silicon Research Center (GSRC), the National Science Foundation (NSF) under Grant No. CCR-9803774, and the Max Kade Foundation. One of the authors is also supported by 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, GSRC, NSF, or the United States Government.

relevant portions of the design. It is important to make improvements on both fronts for successful verification.

A conservative abstraction is one which preserves all behaviors of a concrete system. Conservative abstractions benefit from a *preservation* theorem which states that the correctness of any universal (e.g. ACTL*) formulae on an abstract system automatically implies the correctness of the formula on the concrete system. However, a counterexample on an abstract system may not correspond to any real path, in which case it is called a *spurious* counterexample. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. It is obviously desirable to automate this procedure.

This paper focuses on automating the abstraction process for handling large designs containing up to a few thousand latches. This means that using any computation on concrete systems based on BDDs will be too expensive. Abstraction refinement [1, 6, 8, 11, 13, 17] is a general strategy for automatic abstraction. Abstraction refinement usually involves the following process.

1. **Generation of Initial Abstraction.** It is desirable to derive the initial abstraction automatically.
2. **Model checking of abstract system**. If this results in a conclusive answer for the abstract system, then the process is terminated. For example, in case of existential abstraction, a "yes" answer for an ACTL* property in this step means that the concrete system also satisfies the property, and we can stop. However, if the property is false on the abstract system, an abstract counterexample is generated.
3. **Checking whether the counterexample holds on the concrete system**. If the counterexample is valid, then we have actually found a bug. Otherwise, the counterexample is *spurious* and the abstraction needs to be refined. Usually, refinement of abstraction is based on the analysis of counterexample(s) generated.

Our abstraction function is based on hiding irrelevant parts of the circuit by make a set of variables *invisible*. This simple abstraction function yields an efficient way to generate minimal abstractions, a source of difficulty in previous approaches. We describe two techniques to produce abstract systems by removing invisible variables. The first is simply to make the invisible variables into input variables. This is shown to be a minimal abstraction. However, this leaves a large number of input variables in the abstract system and, consequently, BDD based model checking even on this abstract system becomes very difficult [19]. We propose an efficient method to pre-quantify these variables on the fly during image computation. The resulting abstract systems are usually small enough to be handled by standard BDD based model checkers. We use an enhanced version [3, 4] of NuSMV [5] for this. If a counterexample is produced for the abstract system, we try to simulate it on the concrete system symbolically using a fast SAT checker (Chaff [16, 21] in our case).

The refinement is done by identifying a small set of invisible variables to be made visible. We call these variables the *refinement variables*. Identification of refinement variables is the main focus of this paper. Our techniques for identifying important variables are based on analysis of effective *boolean constraint propagation (BCP)* and *conflicts* [16] during the SAT checking run of the counterexample simulation. Recently, propositional SAT checkers have demonstrated tremendous success on various classes of SAT formulas. The key to the effectiveness of SAT checkers like Chaff [16], GRASP [18] and SATO [20] is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

SAT checkers have been successfully used for Bounded Model Checking (BMC) [2], where the design under consideration is unrolled and the property is symbolically verified using SAT procedures. BMC is effective for showing the presence of errors. However, BMC is not at all effective for showing that a specification is true unless the diameter of the state space is known. Moreover, BMC performance degrades when searching for deep counterexamples. Our technique can be used to show that a specification is true and is able to search for deeper concrete counterexamples because of the guidance derived from abstract counterexamples.

The efficiency of SAT procedures has made it possible to handle circuits with a few thousand of variables, much larger than any BDD based model checker is able to do at present. Our approach is similar to BMC, except that the propositional formula for simulation is constrained by assignments to visible variables. This formula is *unsatisfiable* for a spurious counterexample. We propose heuristic scores based on backtracking and conflict clause information, similar to VSIDS heuristics in Chaff, and conflict dependency analysis algorithm to extract the reason for unsatisfiability. Our techniques are able to identify those variables that are critical for unsatisfiability of the formula and are, therefore, prime candidates for refinement. The main strength of our approach is that we use the SAT procedure itself for refinement. We do not need to invoke multiple SAT instances or solve separation problems as in [8].

Thus the main contributions of our work are, (a) use of SAT for counterexample validation, (b) refinement procedures based on SAT conflict analysis, and, (c) a method to remove invisible variables from the abstract system for computational efficiency.

Outline of the Paper

The rest of the paper is organized as follows. Section 2 briefly reviews how abstraction is used in model checking and introduces notation that is used in the following sections. In Section 3, we describe in detail, our abstraction technique and how we check an abstract counterexample on the concrete model. The most important part of the paper is Section 4, where we discuss our refinement algorithms based on scoring heuristics for variables and conflict dependency analysis. In section 5, we present experimental evidence to show the ability of our approach to handle large state systems. In Section 6, we describe related work in detail. Finally, we conclude in Section 7 with directions for future research.

2 Abstraction in Model Checking

We give a brief summary of the use of abstraction in model checking and introduce notation that we will use in the remainder of the paper (refer to [7] for a full treatment). A transition system is modeled by a tuple $M = (S, I, R, \mathcal{L}, L)$ where S is the set of states, $I \subseteq S$ is the set of initial states, R is the set of transitions, $\mathcal{L}$ is the set of atomic propositions that label each state in S with the labeling function $L : S \rightarrow 2^{L}$. The set I is also used as a predicate $I(s)$, meaning the state s is in I. Similarly, the transition relation R is also used as a predicate $R(s_1, s_2)$, meaning there exists a transition between states s_1 and s_2. Each program variable v_i ranges over its non-empty domain D_{v_i}. The state space of a program with a set of variables $V = \{v_1, v_2, \dots, v_n\}$ is defined by the Cartesian product $D_{v_1} \times D_{v_2} \times \dots \times D_{v_n}$.

In existential abstraction [7] a surjection $h : S \to \hat{S}$ maps a concrete state $s_i \in S$ to an abstract state $\hat{s}_i = h(s_i) \in \hat{S}$. We denote the set of concrete states that map to an abstract state $\hat{s}_i$ by $h^{-1}(\hat{s}_i)$.

Definition 1. *The **minimal existential abstraction** $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{\mathcal{L}}, \hat{L})$ corresponding to a transition system $M = (S, I, R, \mathcal{L}, L)$ and an abstraction function h is defined by:*

1. $\hat{S} = \{\hat{s} | \exists s. s \in S \wedge h(s) = \hat{s}\}$.
2. $\hat{I} = \{\hat{s} | \exists s. I(s) \wedge h(s) = \hat{s}\}$.
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) | \exists s_1. \exists s_2. R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$.
4. $\hat{\mathcal{L}} = \mathcal{L}$.
5. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$.

Condition 3 can be stated equivalently as

$$\exists s_1, s_2 (R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Leftrightarrow \hat{R}(\hat{s}_1, \hat{s}_2) \tag{1}$$

An atomic formula f *respects* h if for all $s \in S$, $h(s) \models f \Rightarrow s \models f$. Labeling $\hat{L}(\hat{s})$ is *consistent*, if for all $s \in h^{-1}(\hat{s})$ it holds that $s \models \bigwedge_{f \in \hat{L}(\hat{s})} f$. The following theorem from [6, 15] is stated without proof.

Theorem 1. *Let h be an abstraction function and ϕ an ACTL* specification where the atomic sub-formulae respect h. Then the following holds: (i) For all $\hat{s} \in \hat{S}$, $\hat{L}(\hat{s})$ is consistent, and (ii) $\hat{M} \models \phi \Rightarrow M \models \phi$.*

This theorem is the core of all abstraction refinement frameworks. However, the converse may not hold, i.e., even if $\hat{M} \not\models \phi$, the concrete model M may still satisfy ϕ. In this case, the counterexample on $\hat{M}$ is said to be spurious, and we need to refine the abstraction function. Note that the theorem holds even if only the right implication holds in Equation 1. In other words, even if we add more transitions to the minimal transition relation $\hat{R}$, the validity of an ACTL* formula on $\hat{M}$ implies its validity on M.

Definition 2. *An abstraction function h' is a **refinement** for the abstraction function h and the transition system $M = (S, I, R, \mathcal{L}, L)$ if for all $s_1, s_2 \in S, h'(s_1) = h'(s_2)$ implies $h(s_1) = h(s_2)$. Moreover, h' is a **proper refinement** of h if there exist $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$ and $h'(s_1) \neq h'(s_2)$.*

In general, ACTL* formulae can have *tree-like* counterexamples [9]. In this paper, we focus only on safety properties, which have finite path counterexamples. It is possible to generalize our approach to full ACTL* as done in [9]. The following iterative abstraction refinement procedure for a system M and a safety formula ϕ follows immediately.

1. Generate an initial abstraction function h.
2. Model check $\hat{M}$. If $\hat{M} \models \phi$, return `TRUE`.

3. If $\hat{M} \not\models \phi$, check the generated counterexample $\widehat{T}$ on M. If the counterexample is real, return `FALSE`.
4. Refine h, and goto step 2.

Since each refinement step partitions at least one abstract state, the above procedure is complete for finite state systems for ACTL* formulae that have path counterexamples. Thus the number of iterations is bounded by the number of concrete states. However, as we will show in the next two sections, the number of refinement steps can be at most equal to the number of program variables.

We would like to emphasize that we model check abstract system in step 2 using BDD based symbolic model checking, while steps 3 and 4 are carried out with the help of SAT checkers.

3 Generating Abstract State Machine

We consider a special type of abstraction for our methodology, wherein, we hide a set of variables that we call *invisible* variables, denoted by $\mathcal{I}$. The set of variables that we retain in our abstract machine are called *visible* variables, denoted by $\mathcal{V}$. The visible variables are considered to be important for the property and hence are retained in the abstraction, while the invisible variables are considered irrelevant for the property. The initial abstraction and the refinement in steps 1 and 4 respectively correspond to different partitions of V. Typically, we would want $|\mathcal{V}| \ll |\mathcal{I}|$. Formally, the value of a variable $v \in V$ in state $s \in S$ is denoted by $s(v)$. Given a set of variables $U = \{u_1, u_2, \ldots, u_p\}, U \subseteq V$, let s^U denote the portion of s that corresponds to the variables in U, i.e., $s^U = (s(u_1)s(u_2)\ldots s(u_p))$. Let $\mathcal{V} = \{v_1, v_2, \ldots, v_k\}$. This partitioning of variables defines our abstraction function $h : S \to \hat{S}$. The set of abstract states is $\hat{S} = D_{v_1} \times D_{v_2} \ldots \times D_{v_k}$ and $h(s) = s^{\mathcal{V}}$.

In our approach, the initial abstraction is to take the set of variables mentioned in the property as visible variables. Another option is to make the variables in the cone of influence (COI) of the property visible. However, the COI of a property may be too large and we may end with a large number of visible variables. The idea is to begin with a small set of visible variables and then let the refinement procedure come up with a small set of invisible variables to make visible.

We also assume that the transition relation is described not as a single predicate, but as a conjunction of bit relations R_j of each individual variable v_j. More formally, we consider a sequential circuit with registers $V = \{v_1, v_2, \ldots, v_m\}$ and inputs $I = \{i_1, i_2, \ldots, i_q\}$. Let $s = (v_1, v_2, \ldots, v_m)$, $i = (i_1, i_2, \ldots, i_q)$ and $s' = (v'_1, v'_2, \ldots, v'_m)$. The primed variables denote the next state versions of unprimed variables as usual. Thus the bit relation for v_j becomes $R_j(s, i, v'_j) = (v'_j \leftrightarrow f_{v_j}(s, i))$.

$$R(s, s') = \exists i \bigwedge_{j=1}^{m} R_j(s, i, v'_j) \tag{2}$$

3.1 Abstraction by Making Invisible Variables as Input Variables

As shown in [8], the minimal transition relation $\hat{R}$ corresponding to R and h described above is obtained by removing the logic defining invisible variables and treating them as free input variables of the circuit. Hence, $\hat{R}$ looks like:

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists i \bigwedge_{v_j \in \mathcal{V}} R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v_j') \tag{3}$$

The quantifications in Equation 3 are performed during each image computation in symbolic model checking of the abstract system. This is done so as not to build a monolithic BDD for $\hat{R}$ and enjoy the benefits of early quantification.

We call this type of abstraction an *input abstraction*. We write s as $s^{\mathcal{V}}, s^{\mathcal{I}}$ to stress the fact that we are leaving invisible variables as input variables in $\hat{R}$. When dealing with systems with a large number of registers, quantifying so many variables for each image computation is expensive (e.g. [19]). An invisible variable can in the support of multiple partitions of the transition relation. In input abstraction, each occurence of an invisible variable has the same value in different partitions of the abstract transition relation. Thus, we say input abstraction preserves *correlations* between different occurrences of an invisible variable. In the next type of abstraction, we pre-quantify most of the invisible variables, to reduce the number of variables during image computation. This means that different occurrences of an invisible variable get decoupled when we push the quantifications inside Equation 3, making the abstraction more approximate.

3.2 Abstraction by Pre-quantifying Invisible Variables

Input abstraction leaves a large number of variables to quantify during the image computation process. We can however, quantify these variables a priori, leaving only visible variables in $\hat{R}$. The transition relation that we get by quantifying invisible variables from $\hat{R}$ in the beginning is denoted by $\tilde{R}$. We can even quantify some of the input variables a priori in this fashion to control the total number of variables appearing in $\tilde{R}$. Let $Q \subseteq \mathcal{I} \cup I$ denote the set of variables to be pre-quantified and let $W = (\mathcal{I} \cup I) \setminus Q$, the set of variable that are not pre-quantified.

Quantification of a large number of invisible variables in Equation 3 is computationally expensive [15]. To alleviate this difficulty, it is customary to approximate this abstraction by pushing the quantification inside conjunctions as follows.

$$\tilde{R}(\hat{s}, \hat{s}') = \exists s^{W} \bigwedge_{v_j \in \mathcal{V}} \exists s^{Q} R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v_j') \tag{4}$$

Since the BDDs for state sets do not contain input variables in the support, this is a safe step to do. This does not violate the soundness of the approximation, i.e., for each concrete transition in R, there will be a corresponding transition in $\hat{R}$, as stated below.

Theorem 2. $\exists s_1, s_2 (R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Rightarrow \tilde{R}(\hat{s}_1, \hat{s}_2)$.

The other direction of this implication does not hold because of the approximations introduced.

Preserving Correlations We can see in Equation 4 that by existentially quantifying each invisible variable separately for each conjunct of the transition relation, we lose the correlation between different occurrences of a variable. For example, consider the trivial bit relations $x_1' = x_3, x_2' = \neg x_3$ and $x_3 = x_1 \oplus x_2$. Suppose x_3 is made an invisible variable. Then quantifying x_3 from the bit relations of x_1 and x_2 will result in the transition relation being always evaluated 1, meaning the state graph is a clique. However, we can see that in any reachable state, x_1 and x_2 are always opposite of each other. To solve this problem partially without having to resort to equation 4, we propose to cluster those bit relations that share many common variables. Since this problem is very similar to the quantification scheduling problem (which occurs during image computations), we propose to use a modification of *VarScore* algorithms [3] for evaluating this quantification. This algorithm can be viewed as producing clusters of bit relations. We use it to produce clusters with controlled approximations. The idea is to delay variable quantifications as much as possible, without letting the conjoined BDDs grow too large. When a BDD grows larger than some threshold, we quantify away a variable. We can of course quantify a variable that no longer appears in the support of other BDDs. Effective quantification scheduling algorithms put closely related occurrences of a variable in the same cluster. Figure 1 shows the VarScore algorithm for approximating existential abstraction.

Given a set of conjuncts R_V and variables s^Q to pre-quantify
Repeat until all s^Q variables are quantified

1. Quantify away s_Q variables appearing in only one BDD
2. Score the variables by summing up the sizes of BDDs in which a variable occurs
3. Pick two smallest BDDs for the variable with the smallest score
4. If any BDD is larger then the *size threshold*, quantify the variable from BDD(s) and go back to step 2.
5. If the BDDs are smaller than threshold, do *BDDAnd* or *BDDAndExists* depending upon the case

Fig. 1. VarScore algorithm for approximating existential abstraction

A static circuit minimum cut based structural method to reduce the number of invisible variables was proposed in [12] and used in [19]. Our method introduces approximations as needed based on actual image computation, while there method removes the variables statically. Our algorithms achieves a balance between performance and accuracy. This means that the approximations introduced by our algorithm are more accurate as the parts of the circuits statically removed in [12] could be important.

3.3 Checking the Validity of an Abstract Counterexamples

Given an abstract model $\hat{M}$ and a safety formula ϕ, we run the usual BDD based symbolic model checking algorithm to determine if $\hat{M} \models \phi$. Suppose that the model checker produces an abstract path counterexample $\bar{s}_m = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_m \rangle$. To check whether this counterexample holds on the concrete model M or not, we symbolically simulate M beginning with the initial state $I(s_0)$ using a fast SAT checker. At each stage of the symbolic simulation, we constrain the values of visible variables only according to the counterexample produced. The equation for symbolic simulation is:

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge (R(s_{m-1}, s_m) \wedge (h(s_m) = \hat{s}_m)) \quad (5)$$

Each $h(s_i)$ is just a projection of the state s_i onto visible variables. If this propositional formula is satisfiable, then we can successfully simulate the counterexample on the concrete machine to conclude that $M \not\models \phi$. The satisfiable assignments to invisible variables along with assignments to visible variables produced by model checking give a valid counterexample on the concrete machine.

If this formula is not satisfiable, the counterexample is *spurious* and the abstraction needs refinement. Assume that the counterexample can be simulated up to the abstract state $\hat{s}_f$, but not up to $\hat{s}_{f+1}$ ([6, 8]). Thus formula 6 is satisfiable while formula 7 is *not* satisfiable, as shown in Figure 2.

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge (R(s_{f-1}, s_f) \wedge (h(s_f) = \hat{s}_f)) \quad (6)$$

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \\ \wedge (R(s_f, s_{f+1}) \wedge (h(s_{f+1}) = \hat{s}_{f+1})) \quad (7)$$

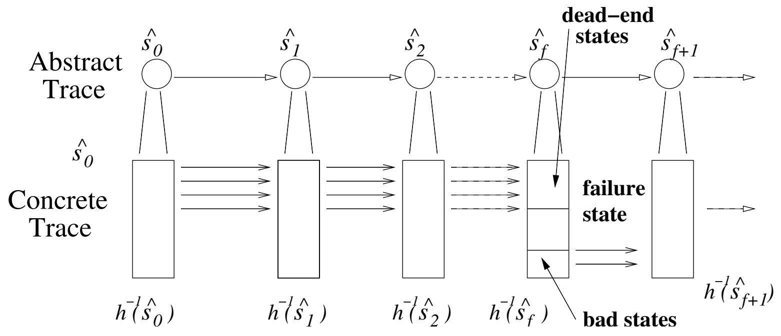

Fig. 2. A spurious counterexample showing failure state [8]. No concrete path can be extended beyond failure state.

Using the terminology introduced in [6], we call the abstract state $\hat{s}_f$ a *failure state*. The abstract state $\hat{s}_f$ contains many concrete states given by all possible combinations of invisible variables, keeping the same values for visible variables as given by $\hat{s}_f$. The concrete states in $\hat{s}_f$ reachable from the initial states following the spurious counterexample are called the *dead-end* states. The concrete states in $\hat{s}_f$ that have a reachable set in $\hat{s}_{f+1}$ are called *bad* states. Because the dead-end states and the bad states are part of the same abstract state, we get the spurious counterexample. The refinement step then is to separate dead-end states and bad states by making a small subset of invisible variables visible. It is easy to see that the set of dead-end states are given by the values of state variables in the f^{th} step for all satisfying solutions to Equation 6. Note that in symbolic simulation formulas, we have a copy of each state variable for each time frame.

We do this symbolic simulation using the SAT checker Chaff [16]. We assume that there are concrete transitions which correspond to each abstract transition from $\hat{s}_i$ to $\hat{s}_{i+1}$, where $0 < i \leq f$. It is fairly straightforward to extend our algorithm to handle spurious abstract transitions. In this case, the set of *bad* states is not empty. Since $\bar{s}_f$ is the shortest prefix that is unsatisfiable, there must be information passed through the invisible registers at time frame f in order for the SAT solver to prove the counterexample is spurious. Specifically, the SAT solver implicitly generates constraints on the invisible registers at time frame f based on either the last abstract transition or the prefix $\bar{s}_f$. Obviously the intersection of these two constraints on those invisible registers is empty. Thus the set of invisible registers that are constrained in time frame f during the SAT process is sufficient to separate *deadend* states and *bad* states after refinement. Therefore, our algorithm limits the refinement candidates to the registers that are constrained in time frame f.

Equation 5 is exactly like symbolic simulation with Bounded Model Checking. The only difference is that the values of visible state variables at each step are constrained to the counterexample values. Since the original input variables to the system are unconstrained, we also constrain their values according to the abstract counterexample. This puts many constraints on the SAT formula. Hence, the SAT checker is able to prune the search space significantly. We rely on the ability of Chaff to identify important variables in this SAT check to separate dead-end and bad states, as described in the next section.

4 SAT Based Refinement Heuristics

The basic framework for these SAT procedures is Davis-Putnam-Logeman-Loveland backtracking search, shown in Figure 3. The function `decide_next_branch()` chooses the branching variable at current *decision level*. The function `deduce()` does *Boolean constraint propagation* to deduce further assignments. While doing so, it might infer that the present set of assignments to variables do not lead to any satisfying solution, leading to a conflict. In case of a conflict, new clauses are learned by `analyse_conflict()` that hopefully prevent the same unsuccessful search in the future. The conflict analysis also returns a variable for which another value should be tried. This variable may not be the most recent variable decided, leading to a *non-chronological* backtrack. If all variables have been decided, then we have found a satisfying assignment and the procedure returns. The strength of various SAT checkers lies in their implementation of constraint propagation, decision heuristics, and learning.

Modern SAT checkers work by introducing conflict clauses in the learning phase and by non-chronological backtracking. Implication graphs are used for Boolean constraint propagation. The vertices of this graph are literals, and each edge is labeled with the clause that forces the assignment. When a clause becomes unsatisfiable as a

```
while(1) {
   if (decide_next_branch()) {        // Branching
      while (deduce() == conflict) {  // Propagate implications
         blevel = analyse_conflict(); // Learning
         if (blevel == 0)
            return UNSAT;
         else
            backtrack(blevel);        // Non-chronological
                                      // backtrack
      }
   }
   else                               // no branch means all vars
                                      // have been assigned
      return SAT;
}
```

Fig. 3. Basic DPLL backtracking search (used from [16] for illustration purpose)

result of the current set of assignments (decision assignments or implied assignments), a conflict clause is introduced to record the cause of the conflict, so that the same futile search is never repeated. The conflict clause is learned from the structure of the implication graph. When the search backtracks, it backtracks to the most recent variable in the conflict clause just added, not to the variable that was assigned last. For our purposes, note that Equation 7 is unsatisfiable, and hence there will be much backtracking. Hence, many conflict clauses will be introduced before the SAT checker concludes that the formula is unsatisfiable. A conflict clause records a reason for the formula being unsatisfiable. The variables in a conflict clause are thus important for distinguishing between dead-end and bad states. The decision variable to which the search backtracks is responsible for the current conflict and hence is an important variable. We call the implication graph associated with each conflict a *conflict graph*.The source nodes of this graph are the variable decisions, the sink node of this graph is the conflicting assignment to one of the variables. At least one conflict clause is generated from a conflict graph. We propose the following two algorithms to identify important variables from conflict analysis and backtracking.

4.1 Refinement Based on Scoring Invisible Variables

We score invisible variables based on two factors, first, the number of times a variable gets backtracked to and, second, the number of times a variable appears in a conflict clause. Note that we have adjust the first score by an exponential factor based on the decision level a variable is at, as the variable at the root node can potentially get just two back tracks, while a variable at the decision level dl can get 2^{dl} backtracks globally. Every time the SAT procedure backtracks to an invisible variable at decision level dl, we add the following number to the *backtrack_score*.

$$2^{\frac{|\mathcal{I}|-dl}{c}}$$

We use c as a normalizing constant. For computing the second score, we just keep a global counter *conflict_score* for each variable and increment the counter for each variable appearing in any conflict clause. The method used for identifying conflict clauses from conflict graphs greatly affects SAT performance. As shown in [21], we use the most effective method called the *first unique implication point* (1UIP) for identifying conflict clauses. We then use weighted average of these two scores to derive the final score as follows.

$$w_1 \cdot \mathit{backtrack\_score} + w_2 \cdot \mathit{conflict\_score} \tag{8}$$

Note that the second factor is very similar to the decision heuristic VSIDS used in Chaff. The difference is that Chaff uses these per variable global scores to arrive at local decisions (of the next branching variable), while we use them to derive global information about important variables. Therefore, we do not periodically divide the variable scores as Chaff does.

We also have to be careful to guide Chaff not to decide on the intermediate variables introduced while converting various formulae to CNF form, which is the required input format for SAT checkers. This is done automatically in our method.

4.2 Refinement Based on Conflict Dependency Graph

The choice of which invisible registers to make visible is the key to the success of the refinement algorithm. Ideally, we want this set of registers to be small and still be able to prevent the spurious trace. Obviously, the set of registers appearing in the conflict graphs during the checking of the counterexample could prevent the spurious trace. However, this set can be very large. We will show here that it is unnecessary to consider all conflict graphs.

Dependencies Between Conflict Graphs We call the implication graph associated with a conflict a *conflict graph*. At least one conflict clause is generated from a conflict graph.

Definition 3. *Given two conflict graphs A and B, if at least one of the conflict clauses generated from A labels one of the edges in B, then we say that conflict B* ***directly depends*** *on conflict A.*

For example, consider the conflicts depicted in the conflict graphs of Figure 4. Suppose that at a certain stage of the SAT checking, conflict graph A is generated. This produces the conflict clause $\omega_9 = (\neg x_9 + x_{11} + \neg x_{15})$. We are using the first UIP (1UIP) learning strategy [21] to identify the conflict clause here. This conflict clause can be rewritten as $x_9 \wedge \neg x_{11} \rightarrow \neg x_{15}$. In the other conflict graph B, clause ω_9 labels one of the edges, and forces variable x_{15} to be 0. Hence, we say that conflict graph B directly depends on conflict graph A.

Given the set of conflict graphs generated during satisfiability checking, we construct the *unpruned conflict dependency graph* as follows:

- **Vertices** of the unpruned dependency graph are all conflict graphs created by the SAT algorithm.

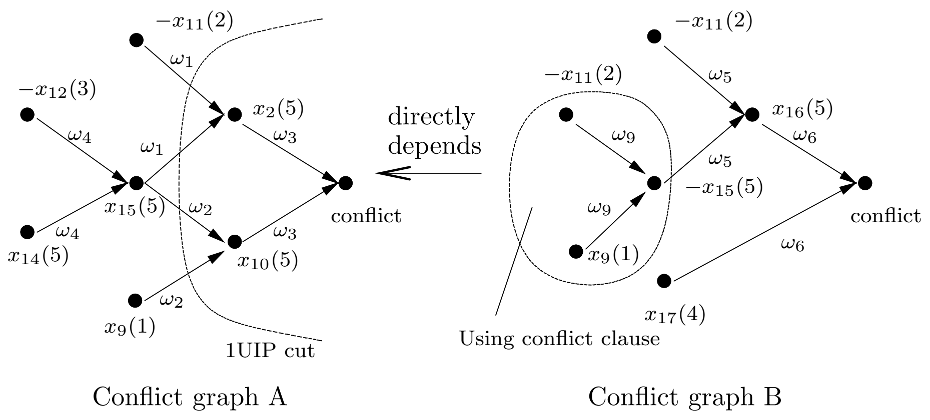

Fig. 4. Two dependent conflict graphs. Conflict B depends on conflict A, as the conflict clause ω_9 derived from the conflict graph A produces conflict B.

– **Edges** of the unpruned dependency graph are direct dependencies.

Figure 5 shows an unpruned conflict dependency graph with five conflict graphs. A conflict graph B depends on another conflict graph A, if vertex A is reachable from vertex B in the unpruned dependency graph. In Figure 5, conflict graph E depends on conflict graph A. When the SAT algorithm detects unsatisfiability, it terminates with the last conflict graph corresponding to the last conflict. The subgraph of the unpruned conflict dependency graph on which the last conflict graph depends is called the *conflict dependency graph*. Formally,

Definition 4. *The* ***conflict dependency graph*** *is a subgraph of the unpruned dependency graph. It includes the last conflict graph and all the conflict graphs on which the last one depends.*

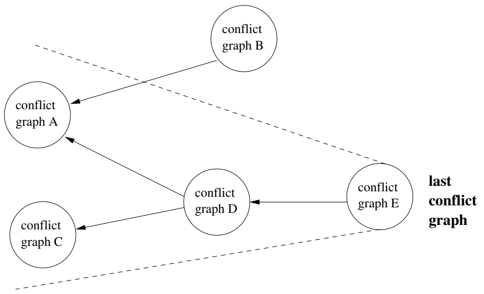

Fig. 5. The unpruned dependency graph and the dependency graph (within dotted lines)

In Figure 5, conflict graph E is the last conflict graph, hence the conflict dependency graph includes conflict graphs A, C, D, E. Thus, the conflict dependency graph can be constructed from the unpruned dependency graph by any directed graph traversal algorithm for reachability. Typically, many conflict graphs can be pruned away in this traversal, so that the dependency graph becomes much smaller than the unpruned dependency graph. Intuitively, all SAT decision strategies are based on heuristics. For a given SAT problem, the initial set of decisions/conflicts a SAT solver comes up with may not be related to the final unsatisfiability result. Our dependency analysis helps to remove that irrelevant reasoning.

Generating Conflict Dependency Graph Based on Zchaff We have implemented the conflict dependency analysis algorithm on top of zchaff [21], which has a powerful learning strategy called first UIP (1UIP). Experimental results from [21] show that 1UIP is the best known learning strategy. In 1UIP, only one conflict clause is generated from each conflict graph, and it only includes those implications that are closer to the conflict. Refer to [21] for the details. We have built our algorithms on top of 1UIP, and we restrict the following discussions to the case that only one conflict clause is generated from a conflict graph. Note here that the algorithms can be easily adapted to other learning strategies.

After SAT terminates with unsatisfiability, our pruning algorithm starts from the last conflict graph. Based on the clauses contained in this conflict graph, the algorithm traverses other conflict graphs that this one depends on. The result of this traversal is the pruned dependency graph.

Identifying Important Variables The dependency graph records the reasons for unsatisfiability. Therefore, only the variables appearing in the dependency graph are important. Instead of collecting all the variables appearing in any conflict graph, those in the dependency graph are sufficient to disable the spurious counterexample.

Suppose $\bar{s}_{f+1} = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{f+1} \rangle$ is the shortest prefix of a spurious counterexample that can not be simulated on the concrete machine. Recall that $\hat{s}_f$ is the failure state. During the satisfiability checking of $\bar{s}_{f+1}$, we generate an unpruned conflict dependency graph. When Chaff terminates with unsatisfiability, we collect the clauses from the pruned conflict dependency graph. Some of the literals in these clauses correspond to invisible registers at time frame f. Only those portions of the circuit that correspond to the clauses contained in the pruned conflict dependency graph are necessary for the unsatisfiability. Therefore, the candidates for refinement are the invisible registers that appear at time frame f in the conflict dependency graph.

Refinement Minimization The set of refinement candidates identified from conflict analysis is usually not minimal, i.e., not all registers in this set are required to invalidate the current spurious abstract counterexample. To remove those that are unnecessary, we have adapted the greedy refinement minimization algorithm in [19]. The algorithm in [19] has two phases. The first phase is the addition phase, where a set of invisible registers that it suffices to disable the spurious abstract counterexample is identified. In the second phase, a minimal subset of registers that is necessary to disable the counterexample is identifed. Their algorithm tries to see whether removing a newly added register from the abstract model still disables the abstract counterexample. If that is the case, this register is unnecessary and is no longer considered for refinement. In our case, we only need the second phase of the algorithm. The set of refinement candidates provided by our conflict dependency analysis algorithm already suffices to disable the current spurious abstract counterexample. Since the first phase of their algorithm takes at least as long as the second phase, this should speed up our minimization algorithm considerably.

5 Experimental Results

We have implemented our abstraction refinement framework on top of NuSMV model checker [5]. We modified the SAT checker Chaff to compute heuristic scores, to produce conflict dependency graphs and to do incremental SAT. The IU-p1 benchmark was verified by conflict analysis based refinement on a SunFire 280R machine with two 750Mhz UltraSparc III CPUs and 8GB of RAM running Solaris. All other experiments were performed on a dual 1.5GHz Athlon machine with 3GB of RAM running Linux.

The experiments were performed on two sets of benchmarks. The first set of benchmarks in Table 1 are industrial benchmarks obtained from various sources. The benchmarks IU-p1 and IU-p2 refer to the same circuit, IU, but different properties are checked in each case. This circuit is an integer unit of a picoJava microprocessor from Sun. The D series benchmarks are from a processor design. The properties verified were simple **AG** properties. The property for IU-p2 has 7 registers, while IU-p1 and D series circuits have only one register in the property. The circuits in Table 2 are various abstractions of the IU circuit. The property being verified has 17 registers. They are smaller circuits that are easily handled by our methods but they have been shown to be difficult to handle by Cadence SMV [8]. We include these results here to compare our methods with the results reported in [8] for property 2. We do not report the results for property 1 in [8] because it is too trivial (all counterexamples can be found in 1 iteration). It is interesting to note that all benchmarks but IU-p1 and IU-p2 have a valid counterexample.

circuit	# regs	ctrex	CSMV	Heuristic Score			Dependency		
		length	time	time	iters	# regs	time	iters	# regs
D2	105	15	152	105	10	51	79	11	39
D5	350	32	1,192	29	3	16	38.2	8	10
D6	177	20	45,596	784	24	121	833	48	90
D18	745	28	>4 hrs	12,086	69	346	9,995	142	253
D20	562	14	>7 hrs	1,493	56	281	1,947	74	265
D24	270	10	7,850	14	1	6	8	1	4
IU-p1	4855	true	-	9,138	22	107	3,350*	13	19
IU-p2	4855	true	-	2,820	7	36	712	6	13

Table 1. Comparison between Candence SMV (CSMV), heuristic score based refinement and dependency analysis based refinement for larger circuits. The experiment marked with a * was performed on the SunFire machine with more memory because of a length 72 abstract counterexample encountered.

In Table 1, we compare our methods against the BDD based model checker Cadence SMV (CSMV). We enabled cone of influence reduction and dynamic variable reordering in Cadence SMV. The performance of "vanilla" NuSMV was worse than

Cadence SMV, hence we do not report those numbers. We report total running time, number of iterations and the number of registers in the final abstraction. The columns labeled with "Heuristic Score" report the results with our heuristic variable scoring method. We introduce 5 latches at a time in this method. The columns labeled with "Dependency" report the results of our dependency analysis based refinement. This method employs pruning of candidate refinement sets. A "-" in a cell indicates that the model checker ran out of memory.

Table 2 compares our methods against those reported in [8] on IU series benchmarks for verifying property 2.

circuit	# regs	ctrex	[8]	Heuristic Score			Dependency		
		length	time	time	iters	# regs	time	iters	# regs
IU30	30	11	6.5	2.3	2	27	1.9	4	20
IU35	35	20	11	8.9	2	27	10.4	5	21
IU40	40	20	16.1	28.4	3	32	13.3	6	22
IU45	45	20	22.1	32.9	3	32	25	6	22
IU50	50	20	85.1	36	3	32	32.8	6	22
IU55	55	11	-	43	2	27	61.9	4	20
IU60	60	11	-	52.8	2	27	65.5	4	20
IU65	65	11	-	50.3	2	27	67.5	4	20
IU70	70	11	-	55.6	2	27	71.4	4	20
IU75	75	11	130.5	38.5	4	37	15.7	5	21
IU80	80	11	153.4	47.1	4	37	21.1	5	21
IU85	85	11	167.7	44.7	4	37	24.6	5	21
IU90	90	11	167.1	49.9	4	37	24.3	5	21

Table 2. Comparison between [8], heuristic score based refinement and dependency analysis based refinement for smaller circuits.

We can see that our conflict dependency analysis based method outperforms a standard BDD based model checker, the method reported in [8] and the heuristic score based method. We also conclude that the computational overhead of our dependency analysis based method is well justified by the smaller abstractions that it produces. The variable scoring based method does not enjoy the benefits of reduced candidate refinement sets obtained through dependency analysis. Therefore, it results in a coarser abstraction in general. The heuristic based refinement method adds 5 registers at a time, resulting in some uniformity in the final number of registers, especially evident in Table 2. Due to the smaller number of refinement steps it performs, the total time it has to spend in model checking abstract machines may be smaller (as for D5, D6, D20, IU60, IU65, IU70).

6 Related Work

Our work compares most closely to that presented in [6] and more recently [8]. There are three major differences between our work and [6]. First, their initial abstraction is based on predicate abstraction, where new set of program variables are generated representing various predicates. They symbolically generate and manipulate these abstractions with BDDs. Our abstraction is based on hiding certain parts of the circuit.

This yields an easier way to generate abstractions. Secondly, the biggest bottleneck in their method is the use of BDD based image computations *on concrete systems* for validating counterexamples. We use symbolic simulation based on SAT accomplish this task, as in [8]. Finally, their refinement is based on splitting the variable domains. The problem of finding the coarsest refinement is shown to be NP-hard in [6]. Because our abstraction functions are simpler, we can identify refinement variables during the SAT checking phase. We do not need to solve any other problem for refinement.

We differ from [8] in three aspects. First, we propose to remove invisible variables from abstract systems on the fly by quantification. This reduces the complexity of BDD based model checking of abstract systems. Leaving a large number of input variables in the system makes it very difficult to model check even an abstract system [19]. Secondly, computation overhead for our separation heuristics is minimal. In their approach, refinement is done by separating *dead-end* and *bad* states (sets of concrete states contained in the failure state) with ILP solvers or machine learning. This requires enumerating *all* dead-end and bad states or producing samples of these states and separating them. We avoid this step altogether and cheaply identify refinement variables from the analysis of a single SAT check that is already done. We do not claim any optimality on the number of variables, however, this is a small price to pay for efficiency. We have been able to handle a circuit with about 5000 variables in cone of influence of the specification. Finally, we believe our method can identify a better set of invisible registers for refinement. Although [8] uses optimization algorithms to minimize the number of registers to refine, their algorithm relies on sampling to provide the candidate separation sets. When the size of the problem becomes large, there could be many possible separation sets. Our method is based on SAT conflict analysis. The Boolean constraint propagation (BCP) algorithm in a SAT solver naturally limits the number of candidates that we will need to consider. We use conflict dependency analysis to reduce further the number of candidates for refinement.

The work of [10] focuses on algorithms to refine an approximate abstract transition relation. Given a spurious abstract transition, they combine a theorem prover with a greedy strategy to enumerate the part of the abstract transition that does not have corresponding concrete transitions. The identified bad transition is removed from the current abstract model for refinement. Their enumeration technique is potentially expensive. More importantly, they do not address the problem of how to refine abstract predicates.

Previous work on abstraction by making variables invisible includes the localization reduction of Kurshan [13] and other techniques (e.g. [1, 14]). Localization reduction begins with the set of variables in the property as visible variables. The set of variables adjacent to the present set of visible variables in the variable dependency graph are chosen as the candidates for refinement. Counterexamples are analyzed in order to choose variables among these candidates.

The work presented in [19] combines three different engines (BDD, ATPG and simulation) to handle large circuits using abstraction and refinement. The main difference between our method and that in [19] is the strategy for refinement. In [19], candidates for refinement are based on those invisible registers that get assigned in the abstract counterexample. In our approach, we intentionally throw away invisible registers in the abstract counterexample, and rely on our SAT conflict analysis to select the candidates. We believe there are two advantages to disallowing invisible registers in the abstract counterexample. First of all, generating an abstract counterexample is computationally expensive, when the number of invisible registers is large. In fact, for efficiency reasons, a BDD/ATPG hybrid engine is used in [19] to model check the abstract model. By quantifying the invisible variables early, we avoid this bottleneck. More importantly, in [19], invisible registers are free inputs in the abstract model, their values are totally unconstrained. When checking such an abstract counterexample on the concrete machine, it is more likely to be spurious. In our case, the abstract counterexample only includes assignments to the visible registers and hence a real counterexample can be found more cheaply.

7 Conclusions

We have presented an effective and practical automatic abstraction refinement framework based on our novel SAT based conflict analysis. We have described a simple variable scoring heuristic as well as an elaborate conflict dependency analysis for identifying important variables. Our schemes are able to handle large industrial scale designs. Our work highlights the importance of using SAT based methods for handling large circuits. We believe these techniques complement bounded model checking in that they enable us to handle true specifications effeciently.

An obvious extension of our framework is to handle all ACTL* formulae. We believe this can be done as in [9]. Further experimental evaluation will help us fine tune our procedures. We can also use circuit structure information to accelerate the SAT based simulation of counterexamples, for example, by identifying replicated clauses. We are investigating the use of the techniques described in this paper for software verification. We already have a tool for extracting a Boolean program from an ANSI C program by using predicate abstraction.

8 Acknowledgements

We would like to thank Ofer Strichman for providing us some of the larger benchmark circuits. We would also like to acknowledge the anonymous reviewers for carefully reading the paper and making useful suggestions.

References

[1] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Proceedings of CAV'93*, pages 29–40, 1993.

[2] Armin Biere, Alexandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS, 1999.

[3] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Tom Shiple, Helmut Veith, and Dong Wang. Non-linear quantification scheduling in image computation. In *Proceedings of ICCAD'01*, pages 293–298, November 2001.

[4] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Helmut Veith, and Dong Wang. Using combinatorial optimization methods for quantification scheduling. In Tiziana Margaria and Tom Melham, editors, *Proceedings of CHARME'01*, volume 2144 of *LNCS*, pages 293–309, September 2001.

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499. Springer, July 1999.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of CAV*, volume 1855 of *LNCS*, pages 154–169, July 2000.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[8] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of CAV'02*, 2002. To appear.

[9] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings of the 17^{th} Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, 2002. To appear.

[10] Satyaki Das and David Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16^{th} Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, 2001.

[11] Shankar G. Govindaraju and David L. Dill. Counterexample-guided choice of projections in approximate symbolic model checking. In *Proceedings of ICCAD'00*, San Jose, CA, November 2000.

[12] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *Proceedings of ICCAD'00*, November 2000.

[13] R. Kurshan. *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[14] J. Lind-Nielsen and H. Andersen. Stepwise CTL model checking of state/event systems. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer Aided Verification (CAV'99)*, 1999.

[15] David E. Long. *Model checking, abstraction and compositional verification*. PhD thesis, Carnegie Mellon University, 1993. CMU-CS-93-178.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[17] Abelardo Pardo and Gary D. Hachtel. Incremental CTL model checking using BDD subsetting. In *Proceedings of the Design Automation Conference (DAC'98)*, pages 457–462, June 1998.

[18] J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, April 1996.

[19] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the DAC*, pages 35–40, 2001.

[20] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.

[21] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, November 2001.