Automatic Abstraction without Counterexamples

K. L. McMillan
Cadence Berkeley Labs

Nina Amla
Cadence Design Systems

Abstract

A method of automatic abstraction is presented that uses proofs of unsatisfiability derived from SAT-based bounded model checking as a guide to choosing an abstraction for unbounded model checking. Unlike earlier methods, this approach is not based on analysis of abstract counterexamples. The performance of this approach on benchmarks derived from microprocessor verification indicates that SAT solvers are quite effective in eliminating logic that is not relevant to a given property. Moreover, benchmark results suggest that when bounded model checking successfully terminates, and the problem is unsatisfiable, the number of state variables in the proof of unsatisfiability tends to be small. In all cases tested, when bounded model checking succeeded, unbounded model checking of the resulting abstraction also succeeded.

1 Introduction

Abstraction is commonly viewed as the key to applying model checking to large scale systems. Abstraction means, in effect, removing information about a system which is not relevant to a property we wish to verify. In the simplest case, we can view the system as a large collection of constraints, and abstraction as removing constraints that are deemed irrelevant. The goal in this case is not so much to eliminate constraints *per se*, as to eliminate state variables that occur only in the irrelevant constraints, and thereby to reduce the size of the state space. A reduction of the state space in turn increases the efficiency of model checking, which is based on exhaustive state space exploration.

The first attempt to automate this simple kind of abstraction is due to Kurshan [14], and is known as *iterative abstraction refinement*. This method begins with an empty set of constraints (or a seed set provided by the user), and applies model checking to attempt to verify the property. If a counterexample is found, it is analyzed to find a set of constraints whose addition to the system will rule out the counterexample. The process is then repeated until the property is found to be true, or until a concrete counterexample is produced. To produce a concrete counterexample, we must find a valuation for the unconstrained variables, such that all the original constraints are satisfied.

A number of variations on this basic technique have appeared [1, 8, 11, 23, 17]. Some of the recent methods pose the construction of a concrete counterexample as a Boolean satisfiability (SAT) problem (or equivalently, an ATPG problem) and apply modern SAT methods [20, 15] to this problem. A recent approach [7] also applies ILP and machine learning techniques to the problem of choosing which constraints to add to rule out an abstract counterexample in the case when a concrete counterexample is not found.

Another recent and related development is that of *bounded model checking* [4]. In this method, the question of the existence of a counterexample of *no more than k steps*, for fixed k, is posed as a SAT problem. In various studies [9, 5], SAT solvers have been found quite efficient at producing counterexamples for systems that are too large to allow standard model checking. The disadvantage of this approach is that, if a counterexample is not found, there is no guarantee that there do not exist counterexamples of greater than k steps. Thus, the method can falsify, but cannot verify properties (unless an upper bound is known on the depth of the state space, which is not generally the case).

In this paper, a method is presented for automated abstraction which exploits an under-appreciated fact about SAT solvers: in the unsatisfiable case, they can produce a proof of unsatisfiability. In bounded model checking, this corresponds to a proof that there is no counterexample to the property of k steps or fewer. Even though this implies nothing about the truth of the property in general, we can use this proof to tell us which constraints are relevant to the property (at least in the first k steps) and thus provide a guess at an abstraction that may be used to fully verify the property using standard model checking methods. The method differs from the earlier, counterexample-based iterative abstraction approaches, in that counterexamples produced by the standard model checker are ignored. The abstraction is based not on refuting these counterexamples, but rather on proofs provided by the SAT solver. Thus, we will refer to it as *proof-based abstraction*. This approach has the advantage that it rules out all counterexamples up to a given length, rather than the single counterexample that the model checker happened to produce.

1.1 Related work

The notion of proof-based abstraction has already appeared in the context of infinite state verification. Here, a finite state abstraction of an infinite state system is generated using as the abstract states the valuations of a finite set of first order predicates over the concrete state. The key to this method, known as predicate abstraction [19], is to choose the right predicates. An iterative abstraction method proposed by Henzinger *et al.* [21] uses a theorem prover to refute counterexamples generated by predicate abstraction. In the case when the counterexample is proved false, the proof is "mined" for new state predicates to use in predicate abstraction.

The technique presented here is similar to this method in spirit, but differs in some significant aspects. First, it applies only to finite state systems, and uses a SAT solver rather than a first order prover. Second, instead of choosing predicates to define the abstract state space, it merely chooses among the existing constraints to form the abstraction – the encoding of the state remains the same.

Another related technique [17] uses a SAT solver to derive an abstraction sufficient to refute a given abstract counterexample. The abstraction is generated by tracing the execution of the SAT solver in a way that is similar to the method presented here. However, that method, like the earlier methods, still refutes one counterexample at a time, accumulating an abstraction.

The key difference the methods of [21, 17] and the present one is the the present method does not use abstract counterexamples as a basis for refining the abstraction. Rather, it generates an abstraction sufficient to refute all counterexamples within a given length bound. Intuitively, the motivation for refuting all counterexamples at once is that a given abstract counterexample may be invalid for many reasons that are not relevant to the truth of the property being proved. In the present method, the abstraction is directed toward the property itself and not a single execution trace that violates it.

Another important difference is that the generated abstraction is not cumulative. That is, a constraint that is present in abstraction in one iteration of the algorithm may be absent in a later iteration. Thus, strictly speaking, the method cannot be viewed as "iterative abstraction refinement". In the counterexample-based methods, an irrelevant constraint, once added to the abstraction, cannot be removed.

1.2 Outline

We begin in the next section by considering how a Boolean satisfiability solver can be extended to produce proofs of unsatisfiability. Then, in section 3, we introduce the proof-based abstraction method, first for safety, then for liveness properties. In section 4, we introduce a number of optimizations to this basic technique that have been found to be useful in practice. Finally, in section 5, we test the method in practice, applying it to the verification of some properties of a commercial microprocessor design.

Benchmark results provide evidence for two significant conclusions: first, that SAT solvers are quite effective at isolating the parts of a large design that are relevant to a given property, and second, that if a property is true and bounded model checking succeeds, then in most cases unbounded model checking can be applied to an abstraction to prove the property in general.

2 Extracting proofs from SAT Solvers

A DPLL-style SAT solver, such as CHAFF [15], or GRASP [20], is easily instrumented to produce proofs of unsatisfiability using resolution. This is based on the observation that "conflict clause" generation can be viewed as a sequence of resolution steps, following the so-called "implication graph". Readers familiar with SAT methods may find this observation quite trivial, and therefore may wish to skip this section. Otherwise, we will now define what is meant by a "proof of unsatisfiability", and show how one can be extracted from a run of a typical SAT solver.

To begin at the beginning, a *clause* is a disjunction of a set of zero or more *literals*, each of which is either a Boolean variable or its negation. We assume that clauses are *non-tautological*, that is, no clause contains both a variable and its negation. A set of clauses is said to be *satisfiable* when there exists a truth assignment to all the Boolean variables that makes every clause in the set true.

Given two clauses of the form $c_1 = v \vee A$ and $c_2 = \neg v \vee B$, we say that the *resolvent* of c_1 and c_2 is the clause $A \vee B$, provided $A \vee B$ is non-tautological. For example, the resolvent of $a \vee b$ and $\neg a \vee \neg c$ is $b \vee \neg c$, while $a \vee b$ and $\neg a \vee \neg b$ have no resolvent, since $b \vee \neg b$ is tautological. It is easy to see that any two clauses have at most one resolvent. The resolvent of c_1 and c_2 (if it exists) is a clause that is implied by $c_1 \wedge c_2$ (in fact, it is exactly $(\exists v)(c_1 \wedge c_2)$).

Definition 1 *A* proof of unsatisfiability *P for a set of clauses C is a directed acyclic graph (V_P, E_P), where V_P is a set of clauses, such that*

- *for every vertex $c \in V_P$, either*
  - *$c \in C$, and c is a root, or*
  - *c has exactly two predecessors, c_1 and c_2, such that c is the resolvent of c_1 and c_2, and*
- *the empty clause is the unique leaf.*

Theorem 1 *If there is a proof of unsatisfiability for clause set C, then C is unsatisfiable.*

Proof. By induction over the depth of the DAG, and transitivity of implication, every clause is implied by the conjunction of C, hence C implies the empty clause (*i.e.*, false), and is thus unsatisfiable. $\square$

Now we consider how a standard SAT solver might be modified to produce proofs of unsatisfiability. While searching for a satisfying assignment, a DPLL solver makes *decisions*, or arbitrary truth assignments to variables, and generates from these an *implication graph*. This is a directed acyclic graph whose vertices are truth assignments to variables, where each node is implied by its predecessors in the graph together with single clause.

As an example, suppose that our clause set is $\{(\neg a \vee b), (\neg b \vee c \vee d)\}$ and we have already decided the literals $\{a, \neg c\}$. A possible implication graph is shown below:

$$\begin{tikzcd} a \arrow[r] & b \arrow[dr] & \\ \neg c \arrow[rr] & & d \end{tikzcd}$$

The literal b is implied by node a and the clause $(\neg a \vee b)$, while d is implied by the nodes b, $\neg c$, and clause $(\neg b \vee c \vee d)$.

A clause is said to be in *conflict* when the negations of all its literals appear in the implication graph. When a conflict occurs, the SAT solver generates a *conflict clause* – a new clause that is implied by the existing clauses in the set. This is usually explained in terms of finding a cut in the implication graph, but from our point of view it is better understood as a process of resolving the "clause in conflict" with clauses in the implication graph to generate a new clause (that is also in conflict). We can also think of each resolution step as applying an implication from the implication graph in the contrapositive.

As an example, suppose that that we add the clause $(\neg b \vee \neg d)$ to the example above. This clause is in conflict, since the implication graph contains both b and d. Note that d was implied by the clause $(\neg b \vee c \vee d)$. Taking the resolvent of this clause with the conflicting clause $(\neg b \vee \neg d)$, we obtain a new implied clause $(\neg b \vee c)$, which is also in conflict. Now, the literal b in the implication graph was implied by the clause $(\neg a \vee b)$. Resolving this with our new clause produces another implied clause $(\neg a \vee c)$, also in conflict. Either of these implied clauses might be taken as the "conflict clause", and added to the clause set.

In order to generate a proof in the unsatisfiable case, we have only to record, for each generated conflict clause, the sequence of clauses that were resolved to produce that clause. The SAT solver produces an "unsatisfiable" answer when it generates the empty clause as a conflict clause (actually, most solvers do not explicitly produce this clause, but can be made to do so). At this point, we can easily produce a proof of unsatisfiability by, for example, a depth-first search starting from the empty clause, recursively deducing each clause in terms of the sequence of clauses that originally produced it. Note that, in general, not all conflict clauses generated during the SAT procedure will actually be needed to derive the empty clause.

3 Proof-based abstraction

Now we will show how such proofs of unsatisfiability can be used to generate abstractions for model checking. What follows does not rely on the fact that we are using a DPLL-style SAT solver. Any solver which can produce a proof of unsatisfiability will suffice.

Bounded model checking [4] is a technique for proving that a transition system admits no counterexample to a given temporal formula of k or fewer transitions, where k is a fixed bound. This can be accomplished by posing the existence of a counterexample of k steps or fewer as a SAT problem. Note that with a proof-generating SAT solver, in the unsatisfiable case we can in effect extract a proof of the non-existence of a counterexample of length k. This proof can in turn be used to generate an abstraction of the transition system in a very straightforward way.

We first observe that a bounded model checking problem consists of a set of constraints – initial constraints, transition constraints, final constraints (in the case of safety properties) and fairness constraints (conditions that must occur on a cycle, in the case of a liveness property). These constraints are translated into conjunctive normal form, and, if appropriate, instantiated for each time step $1 \ldots k$. If no clause deriving from a given constraint is used in the proof of unsatisfiability, then we can remove that constraint from the problem, without invalidating the proof. Thus, the resulting abstract system (with unused constraints removed) is also guaranteed to admit no counterexample of k steps or fewer.

We can now apply ordinary (unbounded) model checking to the abstracted system. This process will have two possible outcomes. The first is that the property is true in the abstracted system. In this case, since removing constraints preserves all properties of our logic (linear temporal logic) we can conclude that the property is true in the original system and we are done. The second possibility is that the unbounded model checker will find a counterexample of greater than k transitions (say, k' transitions). Note that a counterexample of fewer transitions is ruled out, since we have a proof that no such counterexample exists (in both the original or the abstracted system). In this case, we can simply return to bounded model checking using k' as the new length bound.

This procedure, which alternates bounded and unbounded model checking, is guaranteed to terminate for finite models, since k is always increasing. At some point, k must be greater than the depth of the abstract state space (*i.e.*, the depth of a breadth-first search starting from the initial states). At this point, if there is no counterexample of length k, there can be no counterexample of length greater than k, thus the unbounded model checking step must yield "true". In practice, we usually find that when the procedure terminates, k is roughly half the depth of the abstract state space.

In this procedure, "false" results (*i.e.*. counterexamples) are only found by bounded model checking – counterexamples produced by the unbounded model checker are discarded, and only their length is taken into account in the next iteration. This is in contrast to counterexample-based methods such as [14, 1, 8, 11, 23] in which the counterexample produced by model checking the abstract system is used as a guide in refining the abstraction.

Also note that the set of constraints in the abstraction is not strictly growing with each iteration, as it is the above cited methods. That is, at each iteration, the old abstraction is discarded and a new one is generated based on the proof extracted from the SAT solver. This new abstraction may not contain all of the constraints present in the previous abstractions, and may even have fewer constraints.

In the remainder of this section, we will endeavor to make the above informal discussion more precise. Our goal is to determine whether a given LTL formula is true in a given finite model. However, this problem will be posed in terms of finding an accepting run of a finite automaton. The translation of LTL model checking into this framework has been extensively studied [16, 22, 12], and will not be described here. We will distinguish two significant instances of the problem – finding finite counterexamples to safety properties, and finding infinite repeating counterexamples to liveness properties.

3.1 Safety properties

We consider the case of safety properties first, because it is simpler. In this case, we wish to determine the existence of a bad finite prefix – a finite sequence which cannot be extended to an infinite sequence satisfying the property. We assume that the problem is given in terms of an automaton on finite words, such that the a bad prefix exists exactly when the automaton has an accepting run. Such a construction can be found, for example, in [13].

As in symbolic model checking, the automaton itself will be represented implicitly by Boolean formulas. The state space of the automaton is defined by an indexed set of Boolean variables $V = \{v_1, \ldots, v_n\}$. A *state* S is a corresponding vector $(s_1, \ldots, s_n)$ of Boolean values. A *state predicate* P is a Boolean formula over V. We will write $P(W)$ to denote $P\langle w_i/v_i\rangle$ (that is, p with w_i substituted for each v_i). We also assume an indexed set of "next state" variables $V' = \{v'_1, \ldots, v'_n\}$, disjoint from V. A *state relation* R is a Boolean formula over V and V'. We will write $R(W, W')$ to denote $R\langle w_i/v_i, w'_i/v'_i\rangle$.

The runs of the automaton are defined by a triple $M = (I, T, F)$, where the initial constraint I and final constraint F are state predicates, and the transition constraint T is a state relation. A *run* of M, of length k, is a sequence of states $s_0 \ldots s_k$ such that $I(s_0)$ is true, and for all $0 \leq i < k$, $T(s_i, s_{i+1})$ is true, and $F(s_k)$ is true. We can translate the existence of a run into a Boolean satisfiability problem by introducing a new indexed set of variables $W_i = \{w_{i1}, \ldots, w_{in}\}$, for $0 \leq i \leq k$. A run of length up to k exists exactly when the following formula is satisfiable:[1]

$$I(W_0) \wedge \left(\bigwedge_{0 \le i < k} T(W_i, W_{i+1}) \right) \wedge \left(\bigvee_{0 \le i \le k} F(W_i) \right)$$

In order to use a standard SAT solver, we must translate this formula into conjunctive normal form. For this purpose, we will assume that I and T are each a conjunction of a collection of terms. That is, $I = \bigwedge_j I_j$ and $T = \bigwedge_j T_j$. This decomposition will allow us to abstract the problem by removing irrelevant terms. Further, to simplify matters, we can assume without loss of generality that the final condition F consists of a single literal. To ensure this we can, for example, create a new state variable corresponding to the formula F and fold the definition of this new variable into T.

We also assume the existence of some function Γ that translates each Boolean formula into a logically equivalent set of clauses. Thus satisfiability of the above formula is equivalent to satisfiability of the following set of clauses:

$$\textsc{Bmc}_k(M) = \left(\bigcup_j \Gamma(I_j(W_0)) \right) \cup \left(\bigcup_{0 \le i < k, j} \Gamma(T_j(W_i, W_{i+1})) \right) \cup \{ \bigvee_{0 \le i \le k} F(W_i) \} \tag{1}$$

Also note that in general the translation of an arbitrary Boolean formula f into CNF is exponential. In practice, the problem can be solved by adding a fresh variable for the value of each subformula of f, as in [18]. This construction does not affect the satisfiability of the result formula, and produces a CNF formula which is linear size in the size of f. The theory that follows, however, does not depend on the manner in which translation to CNF is performed.

At this point, if $\textsc{Bmc}_k(M)$ is found to be satisfiable, then we have a finite counterexample, and we are done. If, on the other hand, a proof of unsatisfiability P is found for $\textsc{Bmc}_k(M)$, then we know only that there is no counterexample of k or fewer transitions. In this case, we build an abstraction M' of M, such that P is also a proof of unsatisfiability of $\textsc{Bmc}_k(M')$. We want $\textsc{Bmc}_k(M')$ to retain all of the clauses used in P. Thus, we let I' be the conjunction of all the components I_j such that some clause in $\Gamma(I_j(W_0))$ occurs in P. Similarly, we let T' be the conjunction of all the components T_j such that for some $0 \le i < k$, some clause in $\Gamma(T_j(W_i))$ occurs in P. We need not abstract F itself, since we assume that F is a single literal. This gives us the following result:

Lemma 1 *Let $M = (I, T, F)$, let P be a proof of unsatisfiability of* $\textsc{Bmc}_k(M)$ *and let $M' = (I', T', F')$ where*

- $I' = \bigwedge \{ I_j \mid \Gamma(I_j(W_0)) \cap V_P \neq \emptyset \}$,
- $T' = \bigwedge \{ T_j \mid (\bigcup_{0 \le i < k} \Gamma(T_j(W_i))) \cap V_P \neq \emptyset \}$, *and*

[1] Actually, this is correct only when the transition relation is total. The generalization to partial transition relations is straightforward.

- $F' = F$.

Then M' has no runs of length k or less, and further, if M' has no runs, then M has no runs.

Proof. By definition, every clause in P that occurs in $\textsc{Bmc}_k(M)$ also occurs in $\textsc{Bmc}_k(M')$. Thus, since P is a proof of unsatisfiability of $\textsc{Bmc}_k(M)$, it is also a proof of unsatisfiability of $\textsc{Bmc}_k(M')$, so the abstraction M' also has no run of length k or less. Further, since I', T' and F' are weaker than I, T and F, respectively, it follows that every run of M is also a run of M'. $\square$

We can now attempt to perform unbounded symbolic model checking on M', to determine whether it has a run of *any* length. This is preferable to applying model checking directly to M in the case when the number of variables referenced in M' is significantly smaller than the number referenced in M, yielding a reduction in the effective size of the state space. If model checking of M' determines that M' has no runs, then M has no runs, and we are done. On the other hand, if M' does have a run, we know that its length k' is greater than k. In this case, we restart the procedure with k' for k (or in general, any value larger than k' for k). The overall procedure is shown in figure 1.

procedure $\textsc{FiniteRun}(M = (I, T, F))$
    choose $k >= 0$
    while true
        let $C = \textsc{Bmc}_k(M)$
        if C satisfiable
            let A be a satisfying assignment of C
            return the run $s_0, \ldots, s_k$, where $s_{ij} = A(W_{ij})$
        else
            let P be a proof of unsatisfiability of C
            let $M' = \textsc{Abstract}(M, P, k)$
            model check M'
            if M' has a run s of length k'
                let k be some value $\geq k'$
            else return "No Run"
end

procedure $\textsc{Abstract}(M = (I, T, F),\ P = (V_P, E_P),\ k)$
    let $I' = \bigwedge\{I_j \mid \Gamma(I_j(W_0)) \cap V_P \neq \emptyset\}$
    let $T' = \bigwedge\{T_j \mid (\bigcup_{0 \leq i < k} \Gamma(T_j(W_i))) \cap V_P \neq \emptyset\}$
    return (I', T', F)
end

Figure 1: Procedure for existence of a finite run

Theorem 2 *If M has a run, then* FINITERUN*(M) terminates and returns a run of M, else it terminates and returns "No Run".*

Proof. Suppose, toward a contradiction, that the procedure does not terminate. Then, by lemma 1, k increases without bound. Thus, if a run of M does exist, eventually $BMC_k(M)$ will be satisfiable, and the procedure will terminate, returning a run. On the other hand, if M has no run, then eventually k will exceed 2^n, where n is the number of state variables. At this point, since M' has no runs of length up to 2^n (an upper bound on the depth of its state space) it has no runs. Hence the procedure terminates, returning "No Run". $\square$

3.2 Liveness properties

We now consider the question of finding infinite counterexamples to liveness properties. We assume that the problem is posed in terms of existence of a run of a finite Büchi automaton. Specifically, we are given a triple $M = (I, T, F)$, where I and T are as before, and and the *fairness constraint* F is a state predicate which must be true infinitely often. This is easily generalized to a set of fairness constraints. A *run* is a sequence of states $s_0, s_1, \ldots$, such that $I(s_0)$ is true, and for all $i \geq 0$, $T(s_i, s_{i+1})$ is true, and for infinitely many $i \geq 0$, $F(s_i)$ is true. We observe that if a run exists, then there is an infinitely repeating run of the form $\pi\sigma^\omega$, where π and σ are finite sequences of states, and F is true somewhere in σ. We define a *loop* of length k to be a finite sequence of states $s_0, \ldots, s_k$, such that:

- $I(s_0)$ is true, and
- for all $0 \leq i < k$, $T(s_i, s_{i+1})$ is true, and
- for some $0 \leq l < k$, $s_l = s_k$, and
- for some $l \leq j < k$, $F(s_j)$ is true.

A loop corresponds to a run of the form $(s_0 \cdots s_{l-1})(s_l \cdots s_{k-1})^\omega$. We can show that there exists a run of M *iff* there exists a loop. Further, if there is a loop of length k, then there is a loop of length k' for all $k' \geq k$. This loop is just the first $k'+1$ states of the sequence $(s_0 \cdots s_{l-1})(s_l \cdots s_{k-1})^\omega$. Thus, the existence of a loop of length precisely k is equivalent to the existence of a loop of length k or less. This is in turn equivalent to satisfiability of the following Boolean formula:

$$I(W_0) \wedge (\bigwedge_{0 \leq i < k} T(W_i, W_{i+1})) \wedge (\bigvee_{0 \leq l < k} L_l^k(F))$$

where the *loop condition* $L_l^k(F)$ is as follows:

$$L_l^k(F) = (\bigwedge_{j=1\ldots n} (W_{lj} = W_{kj})) \wedge (\bigvee_{l \leq i \leq k} F(W_i))$$

As in the safety case, this formula must be converted into CNF to apply a standard SAT solver. We obtain the following set of clauses:

$$\textsc{Bmc}_k(M) = (\bigcup_j \Gamma(I_j(W_0))) \cup (\bigcup_{0 \leq i < k, j} \Gamma(T_j(W_i, W_{i+1})) \cup \Gamma(L_k(F)) \qquad (2)$$

This definition of $\textsc{Bmc}_k$ can now be used in procedure FINITERUN, substituting the word "loop" for "run", to detect loops rather than finite runs, and thus to model check liveness properties.

4 Optimizations

In this section, we consider some optimizations to the proof-based abstraction procedure that tend to reduce the cost of model checking the abstract automaton.

4.1 Cone of influence

As mentioned above, the removal of constraints in forming the abstract model M' can reduce the number of variables that are referenced, and thus the effective size of the state space. This effect can be enhanced by applying the standard "cone of influence" reduction [3] to M'. That is, there may be constraints which are still present after abstraction, but which can have no bearing on the existence of a run because of the dependency structure of the constraints. As an example, suppose that our final condition is z, and we have the transition constraints $z' = y$, $y' = x$ and $x' = w$. If the constraint $y' = x$ is removed by abstraction, and if x is not referenced elsewhere, then we can remove $x' = w$ as well. In general, it is safe to remove any set of constraints that forms a "conservative extension". A set of constraints C satisfies this criterion if there is some set of variables X, such that variables in X do not occur outside C, and every run with C removed can be made to satisfy C by changing only variables in X. The cone of influence analysis is, in effect, a simple heuristic for finding such a set, and can further reduce the size of the abstraction M'.

4.2 Reduction of variables

To obtain the coarsest possible abstraction, it is best to decompose I and T into the smallest possible components. For example, in circuit verification, we can introduce one component for each gate, and thus perform abstraction at the gate level. In general, the finest decomposition is obtained by introducing a fresh variable for each occurrence of logical operator. For example, the constraint $x = (a \wedge b) \vee (c \wedge d)$ might be decomposed into the constraints $x = w_1 \vee w_2$, $w_1 = a \wedge b$ and $w_2 = c \wedge d$, where w_1 and w_2 are fresh variables. While this makes it possible to obtain very finely tuned abstractions, it can introduce a very large number of variables, which leads to inefficiencies in a BDD based model checker. Here, we will consider a few techniques for reducing the number of these intermediate variables before model checking. For this purpose, it will be convenient to break the transition relation into two parts – a state predicate S which must hold in all states, and a transition predicate R which characterizes the state transition behavior. We let $T(V, V') = S(V) \wedge S(V') \wedge R(V, V')$, to insure that S holds at all states including the final state.

The simplest technique to reduce the number of intermediate variables is to eliminate functionally determined variables by substitution. This in effect undoes the process of decomposing constraints by introducing variables. That is, suppose that S contains the constraint $x = f$ where x is a variable and f is a formula that does not involve x. Since $x = f$ holds in all states, we can eliminate this constraint and the variable x by substituting every occurrence of x in M by f.

Eliminating functionally determined variables in this way removes most of the intermediate variables. However, in practice the removal of constraints results a large number of unconstrained variables which cannot be eliminated in this way. These variables are not state holding, and thus do not, for example, effect the representation of the reached state set in model checking. However, they can result in a substantial combinatorial explosion in BDD based image computations, and substantially complicate the BDD variable ordering problem.

One can imagine a number of approaches for reducing the number of these free variables. One approach that has been found to work well for this purpose exploits additional information that is available in the proof of unsatisfiability. We note that if a variable does not occur in any clause in the proof, then it is safe to replace every occurrence of that variable by a fresh variable. The proof will still apply to the new system. While this might seem to complicate the problem by introducing yet more variables, in fact it can improve matters because of the localization of these new variables. To begin with, since each new variable occurs only once, it is possible that "early quantification" [6] will eliminate these variables quickly. Furthermore, if the model checker evaluates formulas in the ternary $\{0, 1, X\}$ domain, then it is safe under some circumstances to replace uniquely occurring variables with X.

For example, suppose that S contains a constraint $x = f$, where f contains a uniquely occurring variable v. We can safely replace this with the constraint $x = f\langle X/v\rangle$ (assuming that a constraint evaluating to X is taken to be satisfied). The net effect is to project the variable v from the state space, leaving the runs of the automaton otherwise unchanged. Now, suppose as a result we have a constraint $x = g$, where g contains X (but not the variable x). In this case, x is not functionally determined, so we cannot always substitute g for x. We can still safely do this, however if x is uniquely occurring in the remainder of the system. The strategy of replacing uniquely occurring variables with X is thus somewhat at odds with the strategy of substituting determined variables in terms of eliminating variables. However, in practice we find that there is a substantial net benefit.

A good overall approach to the reduction of variables is summarized in figure 2.

```
procedure VariableReduction(M = (I, S, R, F), P = (V_P, E_P), k)
    let Y = {v_i ∈ V | for some 0 ≤ j ≤ k, some clause in V_P contains W_ji}
    for all v_j ∈ V, such that v_i ∉ Y:
        replace every occurrence of v_i in I, S, R, F with a fresh variable
    replace every uniquely occurring variable in I, S, R, F with X
    while S contains x = g, where x ∈ V is uniquely referenced elsewhere
    and g contains X but not x:
        remove x = g from S
        replace the unique occurrence of x in I, S, R, F with g
    while S contains x = g, for some x ∈ V and g not containing X or x:
        remove x = g from S
        replace every occurrence of x in I, S, R, F with g
end
```

Figure 2: Approach to reducing the number of variables for model checking

4.3 Reducing the proof size

In the proof-based abstraction procedure described above, we use bounded model checking to prove that there is no run of k or fewer transitions. However, empirically we observe that for safety properties, smaller proofs (and hence generally better abstractions) are obtained if we consider runs of *exactly* k transitions. Moreover, the abstraction is often found to improve as k increases. This can be explained by the fact that typical properties are of the form $G(p \Rightarrow q)$, where p is a precondition that requires a number of steps to establish, starting from an initial state. Thus, for small values of k, it is possible to prove that there is no counterexample for the spurious reason that the precondition p cannot be established in k steps. For larger values of k, the prover cannot rely on this crutch, and is thus forced to find a more general proof, based on more salient facts. If we attempt to prove at one go that there is no counterexample of k transitions *or fewer*, then the proof will likely include both relevant clauses (to refute large values of k) and irrelevant clauses (to refute short values of k).

Let us consider now what would happen if we checked only for counterexamples of length exactly k. Since counterexamples of length less than k are no longer ruled out, it is possible that model checking the abstraction M' will produce a counterexample of length less than k. Thus, k can decrease at the next iteration, and our procedure may not terminate. We can solve this problem with a compromise approach in which we record the set of values of k for which counterexamples have been found, and always check these lengths in future bounded model checking steps. The bounded model checking problem is now parameterized by a set K of lengths, rather than a single length, and can be written as follows (for safety properties):

$$\textsc{Bmc}_K(M) = (\bigcup_j \Gamma(I_j(W_0))) \cup (\bigcup_{0 \leq i < k, j} \Gamma(T_j(W_i, W_{i+1}))) \cup \{(\bigvee_{i \in K} F(W_i))\} \quad (3)$$

The outline of a modified proof-based abstraction procedure based on this idea is shown in figure 3. This procedure must terminate, since eventually K will contain all lengths up to 2^n.

```
procedure ModFiniteRun(M = (I, T, F))
    choose k >= 0 and let K = ∅
    while true
        let C = Bmc_{K∪{k}}(M)
        if C satisfiable, return counterexample
        else
            model check the abstraction M'
            if M' has a run of length k'
                let K = K ∪ {k'}
                let k be some value ≥ k'
            else return "No Run"
end
```

Figure 3: Modified proof-based abstraction procedure (sketch)

5 Practical experience

The performance of the proof-based abstraction procedure was tested on a set of benchmark model checking problems derived from a sampling of properties used in the compositional verification of a unit of the PicoJava II microprocessor, available in open source from Sun Microsystems, Inc. [2] The unit in question is the ICU, which manages the instruction cache, prefetches instructions, and does some preliminary instruction decoding. Originally, the properties were verified by standard symbolic model checking, using some manual directives to remove parts of the logic not relevant to each property. To make interesting benchmark examples for automatic abstraction, these directives were removed, and a neighboring unit, the instruction folding unit (IFU) was added. The intention of this is to simulate the actions of a naïve user who is unable to localize the verification problem manually (the ultimate naïve user being an automated tool). The function of the IFU is to read instruction bytes from the instruction queue, parse the byte stream into separate instructions and divide the instructions into groups that can be fed into the execution unit in a single

[2] The tools needed to construct the benchmark examples from the Pico-Java II source code can be found at `http://www-cad.eecs.berkeley.edu/~kenmcmil`.

cycle. Inclusion of the IFU increases the number of state variables in the "cone of influence" substantially, largely by introducing dependencies on registers within the ICU itself. It also introduces a large amount of irrelevant combinational logic.

Twenty representative properties were chosen as benchmarks. All of these properties are safety properties, of the form Gp, where p is a formula involving only the current time and the next time (usually only the current time). All the properties are true. Tests were performed on a Linux workstation with a 930MHz Pentium III processor and 512MB of available memory. Unbounded symbolic model checking was performed using the Cadence SMV system. SAT solving was performed using an implementation of the CHAFF algorithm [15], modified to produce proofs of unsatisfiability (verification using a modification of the actual Princeton zChaff implementation produced substantially similar results).

None of the benchmarks could be successfully verified by standard symbolic model checking methods, within a limit of 1800 seconds. [3]

On the other hand, of the 20 benchmarks, all but two were successfully verified by the proof-based abstraction technique. In the two failed cases, the failure was caused by memory exhaustion by the SAT solver during the bounded model checking phase (at k values of 15 and 20 transitions, respectively). Notably, in all cases where the bounded model checking phase completed successfully, the unbounded symbolic model checker was able to successfully check the resulting abstraction M'.

Figure 4 shows, for each benchmark, the original number of state holding variables (solid bars), the number obtained by manual abstraction (gray bars) and the number of state variables remaining in the abstraction at the final iteration of the proof-based abstraction algorithm (open bars). Here, by state variables, we mean any variable v such that v' occurs in R. We will refer to other variables, including inputs and intermediate variables as "combinational variables". A $\otimes$ below the bars indicates that the algorithm did not complete. The number of variables obtained by manual abstraction does not necessarily reflect what could be obtained by concerted effort, but rather reflects only a sufficient effort to make the properties checkable by standard methods. Nonetheless, it is interesting to note that in 11 out of 20 cases a better result is obtained by automatic abstraction.

Figure 5 shows total run time of the proof-based abstraction procedure for each of the benchmarks, on a log scale. Comparison data are not available for standard symbolic model checking, since no problem could be completed within the allotted time. Figure 6 shows the fraction of total run time spent in the two phases of the algorithm. The solid part of the bars represent the total time spent in the bounded model checking phase, while the open part represents the total time spent in the unbounded model checking phase. Note that in most

[3] The primary cause of this failure appears to be inability to construct BDD's for parts of the combinational logic in the IFU. It is possible that some of the benchmarks could be completed by using more advanced transition relation decomposition techniques than are implemented in Cadence SMV.

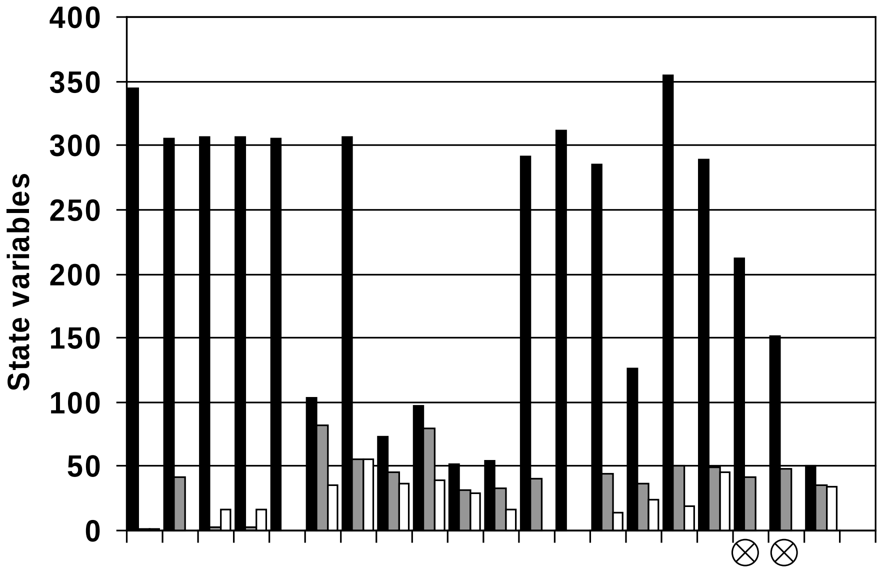

Figure 4: State variables: (solid) original, (Grey) after manual abstraction, (open) after automatic abstraction.

cases, the bottleneck is bounded model checking.

What these data clearly show is that the SAT solver is effective at isolating the part of the logic that is relevant to the given property, at least in the case when this part of the logic is relatively small. We have also found the technique to be very effective at falsification, since the unbounded model checking phase quickly guides the bounded model checker to the appropriate depth. It would be interesting to compare this proof-based abstraction approach to recent automatic abstraction methods based on analysis of counterexamples, such as [23, 7]. Unfortunately, no data is currently available on the performance of those methods on publicly available benchmarks.

As an additional point of comparison, figure 7 compares the performance of the proof-based abstraction approach with results previously obtained by Baumgartner *et al.* [2] on a set of benchmark model checking problems derived from the IBM Gigahertz Processor. Their method involved a combination of SAT-based bounded model checking, structural methods for bounding the depth of the state space, and target enlargement using BDD's. Each point on the graph represents the average verification or falsification time for a collection of properties of the same circuit model. The average time in seconds for proof-based abstraction is represented on the X axis, while the average time in seconds obtained by Baumgartner *et al.* is represented on the Y axis.[4] Thus, a point

[4]The processor speeds for the two sets of experiments are slightly different. Baumgartner *et al.* used an 800MHz Pentium III, as compared to a 930 MHz Pentium II used here. The results presented here have not been adjusted to reflect CPU speed.

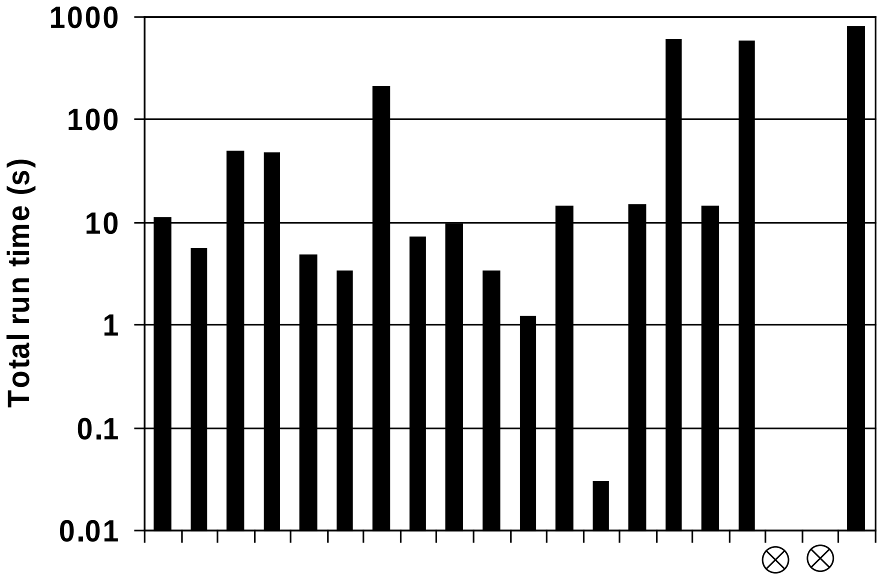

Figure 5: Total verification time for proof-based abstraction algorithm.

above the diagonal line represents a lower average time for proof-based abstraction for one benchmark. Note that in several cases proof-based abstraction has an advantage of two orders of magnitude. A time of 1000 seconds indicates that the truth of one or more properties in the benchmark could not be determined. Of the 28 individual properties that could not be resolved by Baumgartner *et al.*, all but one are successfully resolved by the proof-based abstraction method. Excluding this failed benchmark, the largest average time for the proof-based method is 2.89 seconds. The clear conclusion is that proof-based abstraction is a more effective method of exploiting a SAT solver for model checking than the method of Baumgartner *et al.*

5.1 A hypothesis

The fact that the unbounded model checker is able to check the abstraction in every case when the bounded model checking succeeds suggests an interesting (if somewhat informal) hypothesis: that is, that bounded model checking using SAT solvers tends to succeed when the number of relevant variables is small, and to fail when the number of relevant variables is large. Thus far we have tested only the case when the number of relevant variables is small. To test the other end of the spectrum, one possible approach is to use a set of scalable benchmarks, in which all or most of the state variables are known to be relevant. Such examples tend to occur, for example, in protocol verification. Here, absent any fault tolerance mechanism, a dropped bit anywhere in the system tends to cause the protocol to fail.

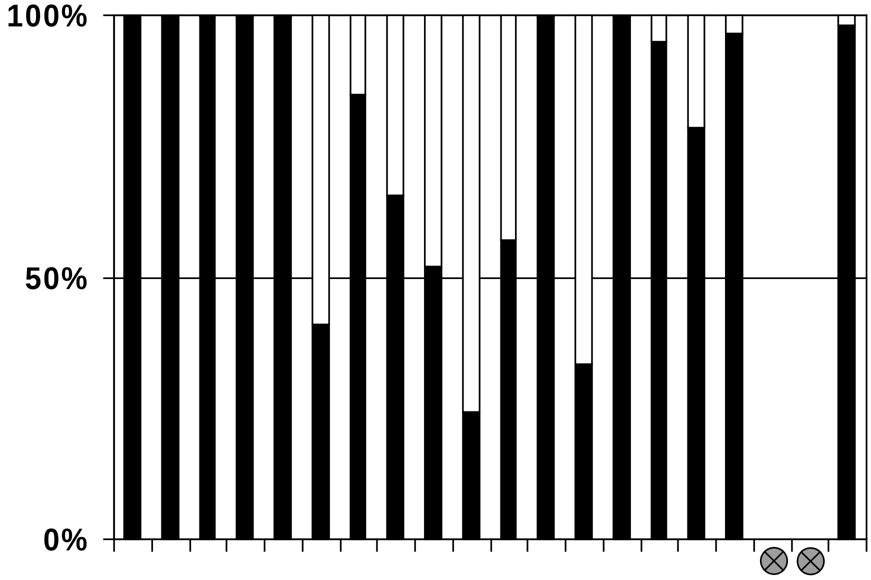

Figure 6: Fraction of total run time: (solid) bounded model checking (open) unbounded model checking

We will consider first a simple model of a cache coherence protocol due to Steven German [10]. This model is parameterized by N, number of processors. The property to be proved is that, if there is an "exclusive" copy of a cache line in the system, then there is no other copy. Empirically, the the depth of the state space of this model is found to be $8N + 2$ transitions. Applying bounded model checking to the model at this depth, we find that the largest instance of this problem we can solve within 1800 seconds is $N = 4$, which has only 42 state variables (of which 37 are found to be relevant). This is quite surprising considering the extreme simplicity of this model relative to the PicoJava II benchmarks. In that case, the SAT solver managed to solve CNF SAT problems with on the order of 1 million variables, while in this case it fails with only about 40,000 variables. On the other hand, the number of relevant state holding variables is roughly similar to what was handled in the PicoJava II benchmarks.

As another test case, let us consider a simple circuit we will call `swap`. This circuit has n j-bit registers. At each clock cycle, it inputs a number i, and swaps the values of register i and its neighbor $i + 1 \bmod j$. We set the number of bits j to $\lceil \log_2 n \rceil$ so that we can initialize all of the registers to different values. The property to prove is that registers 0 and 1 always differ. Clearly, if we unconstrain the value of any one register, the property will be false, since by a series of swaps, we can transfer the value of any register to register 0. Interestingly, we find that the largest instance of `swap` that we can successfully

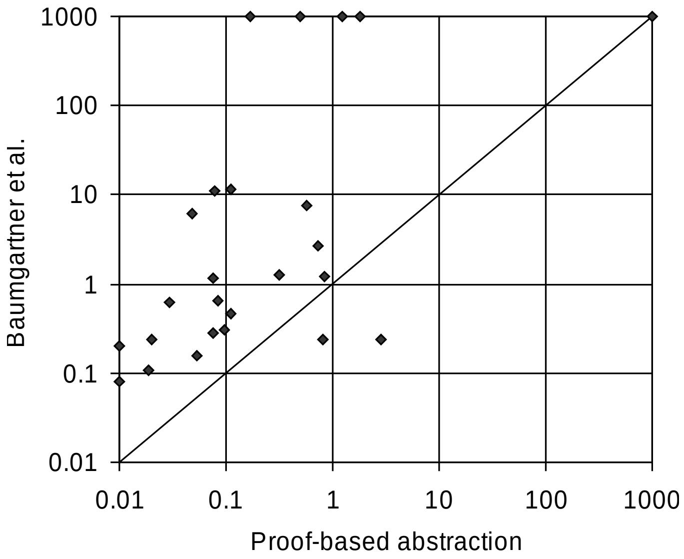

Figure 7: Fraction of total run time: (solid) bounded model checking (open) unbounded model checking

apply bounded model checking to is $n = 7$, corresponding to 21 state bits. At $n = 8$, and $k = 8$, the zChaff solver failed to solve a SAT problem with only 1396 variables in over 40 hours!

Testing SAT solvers on other hardware designs tends to confirm the following trend: when proofs are successfully produced by the SAT solver, they tend to involve only a small number of variables in an absolute sense.

On the other hand, a very large example provided by IBM produced a proof-based abstraction with over 1000 state bits. We conjecture that this was possible because a large number of these registers do not leave their initial states within the first k steps, and that the SAT solver did not in fact reach the state space depth. This has not been confirmed, however. Figure 8 shows results on the set of problems in a collection of hardware verification benchmarks used at Cadence Design Systems. Each point represents a single benchmark problem, with the X axis giving the original number of state variables, and the Y axis the number of state values in the abstraction resulting from the longest successful bounded model checking run in the proof-based abstraction procedure. The trend is clear: successful bounded model checking runs tend to produce proofs of unsatisfiability using a small number of state variables, independent of the number of original state variables. In 18 out of 20 cases, the BDD-based model checker is able to check the abstraction. On the whole, while the case studies presented here are certainly too small to draw general conclusions about the performance of bounded model checking, they are consistent with the hypothesis that successful bounded model checking (defined as checking up to the state

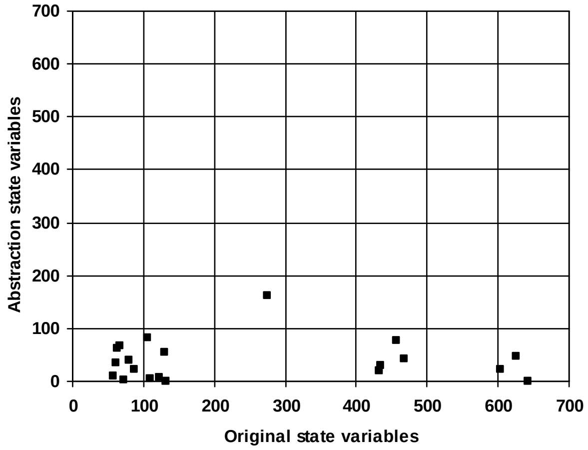

Figure 8: Hardware benchmarks with successful unsatisfiability proofs: (solid) original (open) after abstraction

space depth) depends on having small number of relevant state variables. This suggests that a larger scale study of the question might be in order.

6 Conclusion

We have observed that information generated by bounded model checking can be used to improve the efficiency of unbounded model checking, by suggesting abstractions. Perhaps more interestingly, we have seen some empirical evidence for the hypothesis that bounded model checking succeeds when the number of relevant state variables is small, which implies that *unbounded* model checking is also likely to succeed when applied to the relevant parts of the system. If this hypothesis holds true generally, then it may prove unnecessary in practice to find techniques for bounding the search depth in bounded model checking – if bounded model checking succeeds at roughly the state space depth, then the result can likely be confirmed by standard model checking.

Of course, the negative side of this observation is that it seems unlikely that bounded model checking (and hence proof-based abstraction) can be applied to global properties of systems (those that depend on most or all of the state variables). Techniques are still required to reduce the verification problem to "local" properties that can be proved using only a small set of state variables. However, SAT solvers appear to have significant potential for identifying that set of variables once a suitable property is given.

For future work, it is interesting to consider what other information can be extracted from proofs of unsatisfiability that might be useful in model checking. In addition, since SAT solvers seem to be so effective at isolating relevant facts, it might be that a similar technique could also be applied to infinite state methods such as predicate abstraction, by means of various translations from first order to Boolean satisfiability problems. A final interesting avenue of research might be to consider how the basic SAT algorithms might be modified to improve their performance in terms of producing compact proofs, which would lead in turn to better abstractions.

References

[1] F. Balarin and A. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification (CAV'93)*, pages 29–40, 1993.

[2] J. Baumgratner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification (CAV 2002)*, pages 151–165, 2002.

[3] Sergey Berezin, Sergio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *COMPOS'97*, volume 1536 of *LNCS*. Springer-Verlag, September 1997.

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *LNCS*, pages 193–207, 1999.

[5] P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification (CAV 2001)*, 2001.

[6] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *VLSI '91*, Edinburgh, Scotland, August 1991.

[7] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Computer-Aided Verification (CAV 2002)*, pages 265–279, 2002.

[8] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[9] F. Copty, L. Fix, Fraer R, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking in an industrial setting. In *Computer Aided Verification (CAV 2001)*, pages 436–453, 2001.

[10] S. German. Personal communication.

[11] Shankar G. Govindaraju and David L. Dill. Counterexample-Guided choice of projections in approximate symbolic model checking. In *IEEE International Conference on Computer Aided Design (ICCAD 2000)*, pages 115–119, 2000.

[12] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[13] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

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

[15] M. W. Moskewicz, C. F. Madigan, Y. Z., L. Z., and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference*, pages 530–535, 2001.

[16] A. Pnueli O. Lichtenstein. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85*, pages 97–107, 1985.

[17] James Kukula Samir Sapra Helmut Veith Pankaj Chauhan, Ed Clarke and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD'02)*, November 2002.

[18] D. Plaisted and S. Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[19] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254, pages 72–83, Haifa, Israel, 1997. Springer-Verlag.

[20] 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, November 1996*, 1996.

[21] R. Majumdar T. A. Henzinger, R. Jhala and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL 2002)*, 2002.

[22] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science (LICS '86)*, pages 322–331, 1986.

[23] Dong Wang, Pei-Hsin Ho, Jiang Long, James H. Kukula, Yunshan Zhu, Hi-Keung Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Design Automation Conference*, pages 35–40, 2001.