Methods for exploiting SAT solvers in unbounded model checking

K. L. McMillan

Abstract— **Modern SAT solvers have proved highly successful in finding counterexamples to temporal properties of systems, using a method known as "bounded model checking". It is natural to ask whether these solvers can also be exploited for proving correctness. In fact, techniques do exist for proving properties using SAT solvers, but for the most part existing methods are either incomplete or have a low capacity relative to bounded model checking. Here we consider two new methods that exploit a SAT solver's ability to generate refutations in order to prove properties in an unbounded sense.**

I. Introduction

A SAT solver [16], [12] is a tool for finding a satisfying assignment to a set of propositional clauses. This ability has been quite effectively exploited in finding counterexamples to temporal properties of finite state systems by a technique called *bounded model checking* [3]. This is simply a search for a counterexample of bounded length using a SAT solver. Unfortunately, this technique cannot prove properties correct unless a bound is known on the diameter of the state space, which is not generally the case. By contrast, the completeness of SAT solvers (*i.e,*, their ability to produce a negative result in the unsatisfiable case) has been exploited in a technique called k-induction [15] that makes it possible to prove properties rather than falsifying them. This technique, however, does not appear to be as robust as bounded model checking.

A third, under-exploited ability of SAT solvers is their ability to produce refutations (*i.e.*, proofs of unsatisfiability). It is reasonable to ask whether this ability might not be used in some way to prove temporal properties. In this paper we will consider two recent methods that use refutations to bounded model checking instances in order to aid in the proof of properties (in the unbounded sense). The first method uses the refutation to construct an abstraction of the model to be used in standard unbounded model checking. The second method uses the refutation and a technique called Craig interpolation to perform an over-approximated image computation.

Experiments on hardware verification benchmarks indicate that these methods are, in a certain sense, as robust as bounded model checking. In the worst case, they require unfolding only to the depth of the state space, whereas the k-induction method requires unfolding to the length of the "longest simple path", which can be exponentially longer. Moreover, both methods have certain characteristics that suggest they might be useful in software model checking.

II. Extracting refutations from SAT Solvers

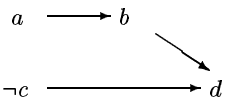
A SAT solver, such as CHAFF [12], or GRASP [16], is easily instrumented to produce proofs of unsatisfiability using resolution. This is based on the observation that "conflict clause" generation can be viewed as a sequence of resolution steps, following the so-called "implication graph". We will now 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 $(\exists v)(c_1 \wedge c_2)$). A *proof of unsatisfiability* of a set of clauses C is a DAG whose vertices are clauses, in which the roots are clauses in C, the unique leaf is the empty clause (FALSE) and each non-root clause is the resolvent of its two parents.

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:



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.

III. Proof-based abstraction

We now consider a simple technique that uses proofs of unsatisfiability to produce abstractions [11]. Given a model and a temporal property, we choose a bound k, and we convert the problem of existence of a counterexample of k steps into a SAT problem (*i.e.*, we perform bounded model checking at depth k). If this problem is satisfiable, we have a counterexample and we are done. Otherwise, we have a proof that there is no counterexample of length k. We use this proof to generate an abstraction of the model and apply an unbounded model checker to the abstraction. If the property is true in this abstraction, it is true in the original, so we are done. Otherwise, we simply increase k and start again. Note this is in contrast to counterexample-based abstraction refinement methods such as [9], [1], [5], [8], [17], [4], [13] that rely on analysis of counterexamples to refine the abstraction. Here, counterexamples from the abstraction are ignored.

A system is modeled as a set of constraints, and abstraction consists simply in throwing away constraints that are deemed irrelevant to the property. This kind of abstraction was introduced in [1], [9]. As an example, Figure 1 shows a simple digital circuit and its model as a set of constraints C. Each gate or register is characterized by a Boolean equation. Note that c' here refers to the value of c at the "next time".

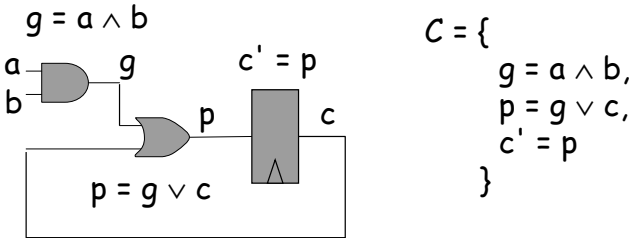


Fig. 1. Model of a simple circuit.

Now suppose we wish to verify the linear temporal property $p = G(c \Rightarrow Xc)$, that is, at all times, if c is true, then c is true at the next time. This particular property happens to hold independently of the value of signal g. That is, if we remove the constraint $g = a \wedge b$ from C, we obtain an abstract system C' that still satisfies the property. Note that in this abstraction, g is left as a free variable, while a and b are unreferenced. If we succeed in verifying property p on system C' we can conclude that C also satisfies p, since C' is weaker than C (*i.e*, C' admits a superset of the traces of C). The advantage of this kind of abstraction is that the abstract system C' refers to a smaller set of variables than the original system C. Thus it should be easier to verify using standard model checking methods.

The important question is, of course, how to choose the abstraction C'. For the sake of simplicity we will consider here only safety properties. Satisfaction of a safety property reduces to a reachability problem, that is, whether a state satisfying an initial condition I can reach a state satisfying a final condition F by some sequence of steps of the model. If so, the property is false and the state sequence represents a counterexample to the property. To solve this problem for a fixed bound k on the number of steps, we simply build a k-step unfolding of the circuit $U = C_0 \wedge \cdots \wedge C_{k-1}$, where C_t represents C with each occurrence of variable v replaced by v_t and each occurrence of v' replaced by v_{t+1}. A counterexample of k steps exists exactly when the formula $I_0 \wedge U \wedge F_k$ is satisfiable. Similarly, a counterexample of *up to* k steps exists when the formula $I_0 \wedge U \wedge (F_0 \wedge F_k)$ is satisfiable[1]

This bounded model checking formula can be converted into CNF and checked for satisfiability using a standard SAT solver. We avoid an exponential expansion in the CNF form by converting each conjunct in the formula separately. If the bounded model checking formula is satisfiable, then the satisfying assignment represents a counterexample to the property. If it is unsatisfiable, the SAT solver can produce a refutation P, that is, a proof by binary resolution steps that the formula implies FALSE.

Our abstraction C' is simply the set of constraints in C that are actually used in proof P. That is, C' is the set of all constraints $c \in C$, such that some clause in the CNF representation of c_i occurs in proof P, for some i in

[1]Actually, this is true only when C is known to represent a total relation. In the general case, a slightly more complex construction is required.

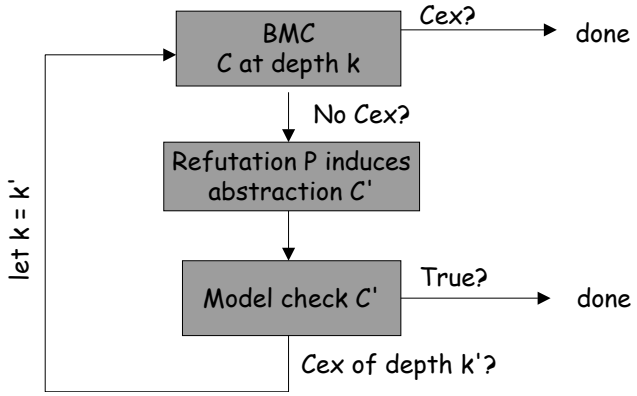


Fig. 2. Proof-based abstraction.

$0 \ldots k-1$.[2]

It is easy to see that this reduced set of constraints also admits no counterexample up to length k, since the bounded model checking formula for C' induces all the clauses used in proof P, hence it is also refuted by P. Intuitively, if k is large enough, we expect C' to be sufficient to rule out counterexamples of any length, a notion we will formalize shortly.

We next apply standard unbounded model checking methods to the abstract system C'. If the property is found to be true for C', then it must also be true for C, since C' is a sound abstraction of C. On the other hand, suppose a counterexample is found, of length k'. From the above argument, we know that $k' > k$. However, the counterexample for C' may not be a counterexample for C because some constraints from C have been omitted. We therefore discard the counterexample and start again with a larger value of k (in practice setting $k = k'$ seems to be effective). The overall procedure for proof-based abstraction is shown in Figure 2. Note that counterexamples are only produced by the bounded model checking phase, and proofs of correctness are only produced by the unbounded model checking phase.

It is easy to show that this procedure must eventually terminate. We know that k increases with each iteration and that the diameter d of the state space of C' is bounded. Thus, eventually we must have $k > d$. We know there is no path from I to F in C' of length up to k. Thus, when $k > d$ there is no path from I to F of any length. It follows that the unbounded model checker must find the property true, and so the procedure must eventually terminate.

In practice, we usually find that the procedure usually terminates when k is approximately one half of d. Moreover, d is the diameter of the abstraction C' which is often much smaller than the diameter of the original system C.

A. Practical experience

To guage the effectiveness of the proof-based abstraction procedure in generating abstractions, it 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. [3] 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 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. All the properties are true. None of the benchmarks could be successfully verified by standard symbolic model checking methods alone, within a limit of 1800 seconds. 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 C'.

Figure 3 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, without manual abstraction (open bars). 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. Thus, it appears that the SAT solver is in fact quite effective at narrowing the problem down to the relevant logic.

Figure 4 shows total run time of the proof-based abstraction procedure for each of the benchmarks, on a log scale. The times are quite reasonable given the scale of the model

[2] In principle we can also abstract the initial and final conditions in a similar way, or we can simply reduce these conditions to single literals by moving the initial and final condition formulas into the constraint system C.

[3] 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`.

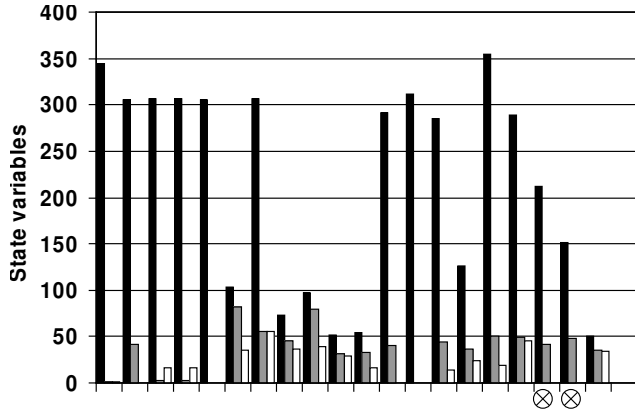


Fig. 3. State variables: (solid) original, (gray) after manual abstraction, (open) after automatic abstraction.

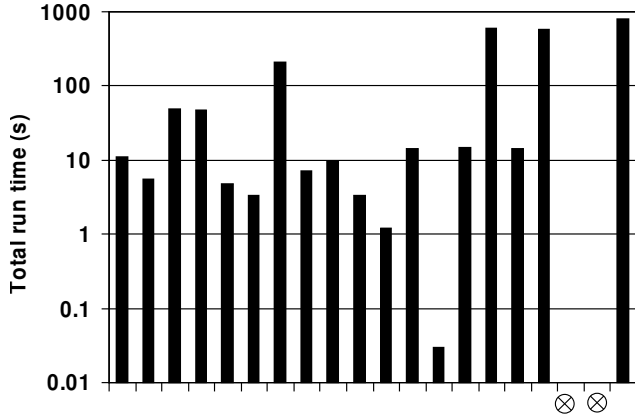


Fig. 4. Total verification time for proof-based abstraction algorithm.

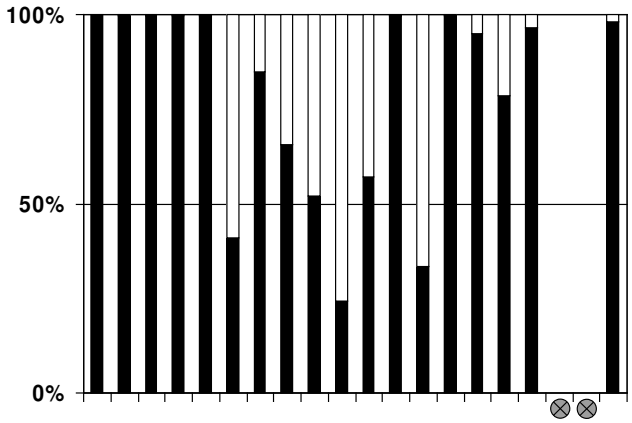


Fig. 5. Fraction of total run time: (solid) bounded model checking (open) unbounded model checking

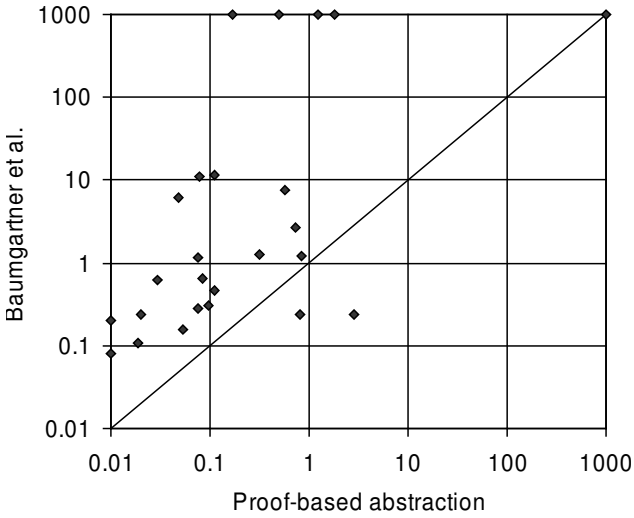


Fig. 6. Fraction of total run time: (solid) bounded model checking (open) unbounded model checking

checking problems, showing that the SAT solver is not unduly slowed by the large amount of irrelevant logic.

Figure 5 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 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.

As an additional point of comparison, Figure 6 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 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 provides a better k bound than the structural method.

[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 III used here. The results presented here have not been adjusted to reflect CPU speed.

B. A (fuzzy) hypothesis

The fact that the unbounded model checker is able to check the abstraction in most cases 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.

We will consider first a simple model of a cache coherence protocol due to Steven German [7]. 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 $i + 1$ mod 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 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. Figure 7 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

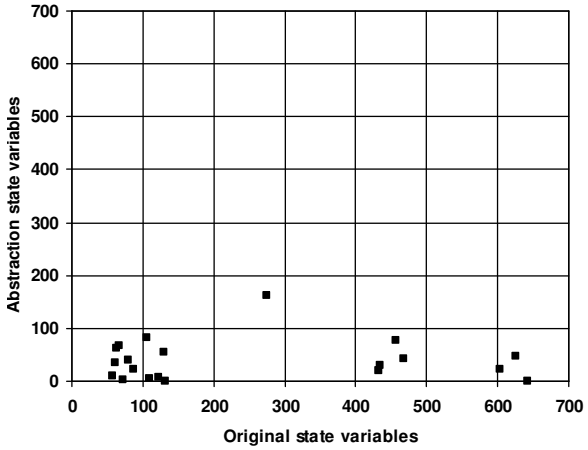


Fig. 7. Results of proof-based abstraction on hardware benchmarks.

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. This means that, most of the time, if bounded model checking succeeds unbounded model checking on the resulting abstraction will also succeed, so there is need to settle for a bounded verification. On the other hand, it may also mean that bounded model checking is only applicable in practice to localizable properties.

IV. Model checking based on Craig interpolation

We now consider an alternative approach that exploits refutations of bounded model checking instances to generate an over-approximated image operator [10]. This method does not rely on a standard model checker. Rather it uses entirely SAT-based methods to perform symbolic model checking.

The image approximation is based on the notion of a Craig interpolant [6]. Given a pair of propositional formulas (A, B), and a proof by resolution that A and B are inconsistent, we can derive a formula called an *interpolant* for (A, B) in linear time. An interpolant for (A, B) is a formula A' with the following properties:

- A implies A',
- $A' \wedge B$ is unsatisfiable, and
- A' refers only to the common variables of A and B.

A simple and efficient construction for the interpolant can be found in [10].

A. Approximate image based on interpolation

Given a set of constraints C characterizing the transitions of a system, and predicate P characterizing a set of states, the forward image of P, denoted $\textsc{Img}(P, C)$, is the set of states that can be reached from a state in P via a transition in C. We can write this as

$$\textsc{Img}(P, C) = \lambda V'.\ \exists V.\ (P \wedge C)$$

A symbolic model checker evaluates this predicate symbolically to compute a representation of the image. This image can then be iterated to a fixed point to compute a representation of the set of states R reachable from an initial state, according to the following recurrence:

$$\begin{aligned} R_0 &= I \\ R_{i+1} &= R_i \vee \textsc{Img}(R_i, C) \\ R &= \cup_i R_i \end{aligned}$$

There is a path from the initial condition I to the final condition F exactly when $R \wedge F$ is satisfiable.

The reachable state set may, however, contain significantly more information than is needed to show that F is not reachable. By carefully over-approximating the image operator, we may simplify the problem without affecting the reachability of F. An over-approximate image operator is an operator $\textsc{Img}'$, such that, for all predicates P, $\textsc{Img}(P, C)$ implies $\textsc{Img}'(P, C)$. Using $\textsc{Img}'$, we can compute an over-approximation R' of the reachable states. We will say that an over-approximate image operator $\textsc{Img}'$ is *adequate* with respect to the final condition F when, for any P that cannot reach F, $\textsc{Img}'(P, C)$ cannot reach F. In other words, an adequate over-approximation does not add any states to the image that can reach the final condition. Clearly, if $\textsc{Img}'$ is adequate, then F is reachable exactly when it is consistent with R', the over-approximated reachable states, since no states that can reach F have been added to R'. The question, of course, is how to compute an adequate $\textsc{Img}'$.

One answer is to bound our notion of adequacy. Let's say that a k-adequate image operator is an $\textsc{Img}'$ such that, for any P that cannot reach F, $\textsc{Img}'(P, C)$ cannot reach F *within k steps*. We note that if k is greater than the diameter of the state space, then k-adequate is equivalent to adequate, since by definition any state that can be reached can be reached within the diameter.

The idea is that we can use bounded model checking and interpolation to compute a k-adequate image operator. We set up a bounded model checking formula to determine whether our state set P can reach F in $k+1$ steps. However, we break this formula into to parts:

$$\begin{aligned} A &= P_{-1} \wedge C_{-1} \\ B &= C_0 \wedge \cdots \wedge C_{k-1} \wedge (F_0 \vee \cdots \vee F_k) \end{aligned}$$

This formula is depicted in Figure 8. If $A \wedge B$ is satisfiable, then P can reach F, so the image operator can yield any over-approximation, the simplest being the predicate TRUE. On the other hand, if it is unsatisfiable, we can compute an interpolant A' for the pair (A, B). Note that the variables common to A and B are in the set V_0, that is, the variables representing the state at time 0. Thus, A' is a propositional formula over V_0. In fact, the set of states $\textsc{Img}'(P, C) = \lambda V_0.\ A'$ is a k-adequate over-approximation of $\textsc{Img}(P, C)$. This fact derives directly from the defining properties of an interpolant. Since $P_{-1} \wedge C_{-1}$ implies A', it follows that every state in $\textsc{Img}'(P, C)$ is reachable from P in one step,

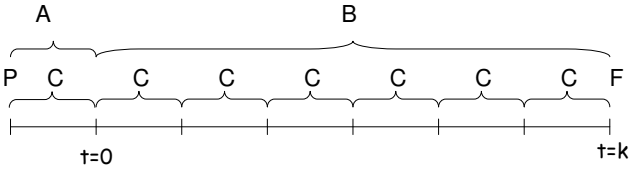


Fig. 8. Forward image based on interpolation..

hence $\textsc{Img}'(P, C)$ is an over-approximation. Further, since A' is inconsistent with B, it follows that no state in $\textsc{Img}(P, C)$ can reach F in k steps. Hence $\textsc{Img}'$ is k-adequate. One way to think about this is that the interpolant is an abstraction of A containing just the information from A that the SAT solver used to prove that P cannot reach F in $k+1$ steps. Thus, it is in a sense an abstraction of the forward image relative to a (bounded time) property.

Now suppose we use this k-adequate image operator to compute an over-approximation of the reachable states R'. If we find that F is unreachable (*i.e.*, $R' \wedge F$ is unsatisfiable), we know that I cannot reach F. If we find that F is reachable, it may be that we have over-approximated too much. In this case, however, we can simply try again with a larger value of k. Note that if the bounded model checking formula $A \wedge B$ turns out to be satisfiable in the first iteration (when $P = I$) then F is in fact reachable and we can terminate with a counterexample.

It is easy to show that if we keep increasing k, this procedure must terminate with either a proof or a counterexample. That is, if we keep increasing k, either we will obtain a counterexample, or k will become greater than the diameter of the state space. In the latter case, our k-adequate image operator is in fact an adequate image operator, so our reachability answer must be correct. In practice, we find that the k values at which we terminate are generally smaller than the diameter. This diameter-based termination bound contrasts with the termination bound for the k-induction methods [15] which is length of the shortest simple path in the state space (also called the recurrence diameter). The shortest simple path can be exponentially longer than the diameter.

B. Practical experience

The performance of the interpolation-based model checking procedure was tested on the PicoJava II and IBM benchmark sets described above. Of the twenty PicoJava benchmarks, all but one were successfully verified by the interpolation-based method. Figure 9 shows a comparison of the interpolation-based method against the proof-based abstraction method. In the figure, each point represents one benchmark problem, with the value on the X axis representing the time in seconds required for the earlier proof-based abstraction method, and the time on the Y axis representing the time in seconds taken

by the interpolation-based method.[5] A time value of 1000 indicates a time-out after 1000 seconds. Points below the diagonal therefore indicate an advantage for the interpolation method. We observe 16 wins for interpolation and 3 for proof-based abstraction, with one problem solved by neither method. In five or six cases, the interpolation method wins by two orders of magnitude. Since both methods are limited by bounded model checking time, what this shows is that the interpolation method tends to terminate at smaller values of k (though both reach the diameter in the worst case).

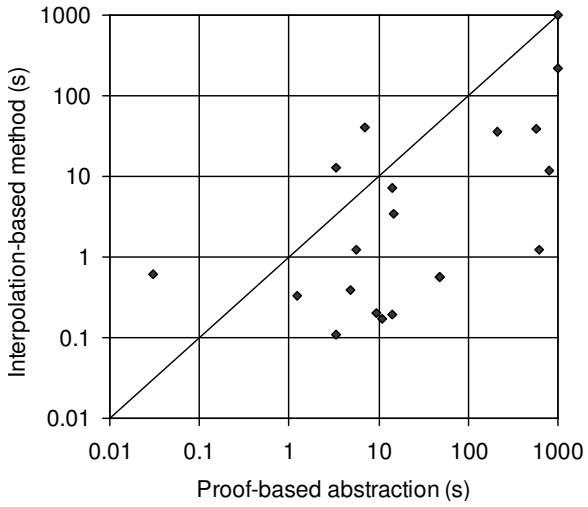


Fig. 9. Run times on PicoJava II benchmarks.

Next we compare the interpolation-based method against proof-based abstraction on the IBM benchmarks. The results are shown in Figure 10. Though the results are mixed, we find that overall the advantage goes to proof-based abstraction (both successfully solve the same set of problems). This appears to be due to the fact that a large number of properties in the benchmark set are false (*i.e.*, have counterexamples). The proof-based abstraction method tends to find counterexamples more quickly because in effect the standard model checker quickly guides the bounded model checker to the right depth, while the proof partition method systematically explores all depths. Figure 11 compares the two methods on only those individual properties that are true, showing an advantage for interpolation. This suggests that a hybrid method might provide the best results overall.

Finally, in Figure 12 we compare the performance of the interpolation method to the k-induction method [15] on the PicoJava benchmarks. The k-induction method terminates within the time limit on 8 of the 20 problems, as opposed to 19 of 20 for the interpolation method. The k-induction method is actually faster on four relatively easy problems. These turn out to be propositional tautologies, and are therefore 0-inductive. On the whole, however, the k-induction is considerably less effective on these benchmarks than interpolation-based model checking.

[5]Times for the present method include only the time actually used by the SAT solver. Overhead in generating the unfoldings is not counted, since this was implemented inefficiently. An efficient implementation would re-use the unfolding from one iteration to the next, thus making the unfolding overhead negligible. Time to generate the interpolants was also negligible. A value of $j = k$ was used for these runs.

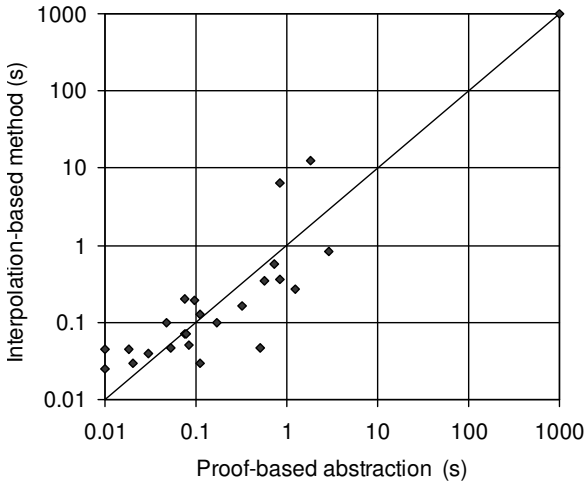


Fig. 10. Run times on IBM Gigahertz Processor benchmarks.

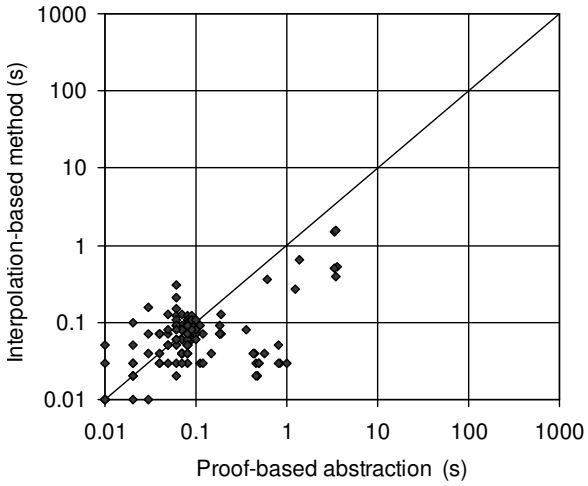


Fig. 11. Run times on IBM Gigahertz Processor true properties.

V. Conclusion

We have considered two methods of model checking that exploit a SAT solver's ability to produce refutations to bounded model checking problems. Both methods are in effect methods of abstraction (one produces a structural abstraction while the other produces an abstract image operator). Both exploit the SAT solvers ability to concentrate on relevant facts and ignore irrelevant ones, and tend to be effective on problems containing a large set of facts of which only a small number are relevant to the given property.

This suggests that the methods might be particularly useful in software model checking, for several reasons. Software model checking is typically based on predicate abstraction [14]. In this method, the infinite state space is reduced to a truth valuation of a finite set of predicates. The success of this

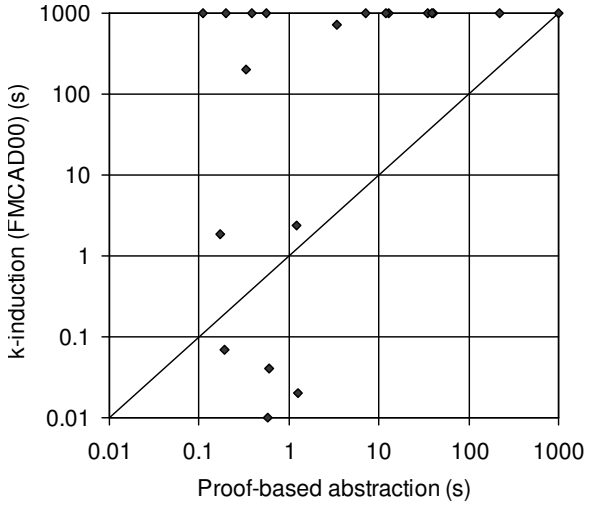


Fig. 12. Run times on PicoJava II benchmarks.

approach requires a judicious choice of predicates. However, if the model checker is able to effectively ignore predicates that are not relevant to the property, then it might be that a significantly larger set of predicates could be used without paying a large penalty in model checking cost.

In addition, software model checkers typically use the "Cartesian abstraction" to reduce the explosion in predicate states. In this abstraction, states are essentially Boolean cubes, and information about the correlation of predicate values can be lost, leading to false counterexamples. Using the interpolation method, however, only those correlations that are actually used in the SAT solver's proof will be present in the interpolant. There is no need to apply the Cartesian abstraction, hence false counterexamples might be avoided. It should also be noted that Craig interpolation is possible in first order theories as well as propositional theories. This means that an interpolation-based image operator might be used directly to compute invariants expressed in first order logic. Because there is no finite diameter bound for infinite state systems, the interpolation method might not terminate. However, it might be effectively used in conjunction with predicates to strengthen the result and avoid false negatives.

Acknowledgment The author would like to thank Jason Baumgartner of IBM for providing the Gigahertz Processor benchmark problems, and Nina Amla of Cadence Design Systems for providing benchmark data.

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] 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.

[4] 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.

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

[6] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250–268, 1957.

[7] S. German. Personal communication.

[8] 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.

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

[10] K. L. McMillan. Interpolation and sat-based model checking. In *Computer-Aided Verification (CAV 2003)*, 2003. to appear.

[11] K. L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 2–17, 2003.

[12] 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.

[13] 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.

[14] 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.

[15] M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Formal Methods in Computer Aided Design*, 2000.

[16] 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.

[17] 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.