Interpolation and SAT-based Model Checking

K. L. McMillan

Cadence Berkeley Labs

Abstract. We consider a fully SAT-based method of unbounded symbolic model checking based on computing Craig interpolants. In benchmark studies using a set of large industrial circuit verification instances, this method is greatly more efficient than BDD-based symbolic model checking, and compares favorably to some recent SAT-based model checking methods on positive instances.

1 Introduction

Symbolic model checking [8, 9] is a method of verifying temporal properties of finite (and sometimes infinite) state systems that relies on a symbolic representation of sets, typically as Binary Decision Diagrams [7] (BDD's). By contrast, bounded model checking [4] can falsify temporal properties by posing the existence of a counterexample of k steps or fewer as a Boolean satisfiability (SAT) problem. Using a modern SAT solver, this method is efficient in producing counterexamples [10, 6]. However, it cannot verify properties unless an upper bound is known on the depth of the state space, which is not generally the case.

This paper presents a purely SAT-based method of *unbounded* model checking. It exploits a SAT solver's ability to produce refutations. In bounded model checking, a refutation is a proof that there is no counterexample of k steps or fewer. Such a proof implies nothing about the truth of the property in general, but does contain information about the reachable states of the model. In particular, given a partition of a set of clauses into a pair of subsets (A, B), and a proof by resolution that the clauses are unsatisfiable, we can generate an *interpolant* in linear time [20]. An interpolant [11] for the pair (A, B) is a formula P with the following properties:

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

Using interpolants, we obtain a complete method for finite-state reachability analysis, and hence LTL model checking, based entirely on SAT.

1.1 Related work

SAT solvers have been applied in unbounded model checking in several ways. For example, they have been used in a hybrid method to detect fixed points, while the quantifier elimination required for image computations is performed by other means (*e.g.*, by expansion of the quantifier as $\exists v.f = f\langle 0/v\rangle \vee f\langle 1/v\rangle$, followed by simplification). Such methods include [5, 2, 24]. Because of the expense of quantifier elimination, this approach is limited to models with a small number of inputs (typically zero or one). By contrast, the present approach is based entirely on SAT, does not use quantifier elimination, and is not limited in the number of inputs (examples with thousands of inputs have been verified). SAT algorithms have also been used to generate a disjunctive decompositions for BDD-based image computations [13]. Here, BDD's are not used.

Another approach is based on unfolding the transition relation to the length of the longest simple path between two states [21]. The fact that this length has been reached can be verified using a SAT solver. The longest simple path can, however, be exponentially longer than the diameter of the state space (for example, the longest simple path for an n-bit register is 2^n, while the diameter is 1). The present method does not require unfolding beyond the diameter of the state space, and in practice often succeeds with shorter unfoldings.

Finally, Baumgartner, *et al.* [3], use SAT-based bounded model checking with a structural method for bounding the depth of the state space. This requires the circuit in question to have special structure and does not always give useful bounds. In a suite of benchmarks, we find that the present method successfully resolves almost all of the model checking problems that could not be resolved by the structural method.

1.2 Outline

The next section covers resolution proofs and interpolation. Then in section 4 we give a method for unbounded model checking based on interpolation. Finally, in section 5, we test the method in practice, applying it to the verification of some properties of commercial microprocessor designs.

2 Interpolation algorithm

To begin at the beginning, a *clause* is a disjunction of zero or more *literals*, each being either a Boolean variable or its negation. We assume that clauses are *non-tautological*, that is, no clause contains a variable and its negation. A clause set is *satisfiable* when there is a truth assignment to the Boolean variables that makes all clauses 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 $(\exists v)(c_1 \wedge c_2)$). We will call v the *pivot variable* of c_1 and 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.*

A SAT solver, such as CHAFF [17], or GRASP [22], is a complete decision procedure for clause sets. In the satisfiable case, it produces a satisfying assignment. In the unsatisfiable case, it can produce a proof of unsatisfiability [**?**,**?**]. This, in turn, can be used to generate an interpolant by a very simple procedure [20]. This procedure produces a Boolean circuit whoses gates correspond to the vertices (*i.e.*, resolution steps) in the proof.

Suppose we are given a pair of clause sets (A, B) and a proof of unsatisfiability P of $A \cup B$. With respect to (A, B), say that a variable is *global* if it appears in both A and B, and *local* otherwise. Similarly, a literal is global or local depending on the variable it contains. Given any clause c, we denote by $g(c)$ the disjunction of the global literals in c and by $l(c)$ the disjunction of the local literals.

For example, suppose we have two clauses, $c_1 = (a \vee b \vee \neg c)$ and $c_2 = (b \vee c \vee \neg d)$, and suppose that $A = \{c_1\}$ and $B = \{c_2\}$. Then $g(c_1) = (b \vee \neg c)$, $l(c_1) = (a)$, $g(c_2) = (b \vee c)$ and $l(c_2) = (d)$.

Definition 2. *Let (A, B) be a pair of clause sets and let P be a proof of unsatisfiability of $A \cup B$, with leaf vertex l. For all vertices $c \in V_P$, let p_c be a boolean formula, such that*

- *if c is a root, then*
  - *if $c \in A$ then $p(c) = g(c)$,*
  - *else $p(c)$ is the constant* TRUE.
- *else, let c_1, c_2 be the predecessors of c and let v be their pivot variable:*
  - *if v is local, then $p(c) = p(c_1) \vee p(c_2)$,*
  - *else $p(c) = p(c_1) \wedge p(c_2)$.*

The P-interpolant of (A, B), denoted $\text{ITP}(P, A, B)$ *is p_l.*

Theorem 2. *For all (A, B), a pair of clause sets, and P, a proof of unsatisfiability of $A \cup B$,* $\text{ITP}(P, A, B)$ *is an interpolant for (A, B).*

The formula $\text{ITP}(P, A, B)$ can be computed in time $O(N + L)$, where N is the number of vertices in the proof $|V_P|$ and L is the total number of literals in the proof $\Sigma_{c \in V_P} |c|$. Its circuit size is also $O(N + l)$. Of course, the size of the proof P is exponential in the size of $A \cup B$ in the worst case.

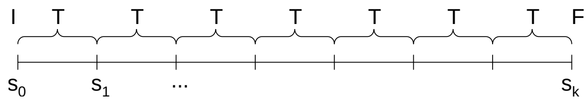

Fig. 1. Bounded model checking.

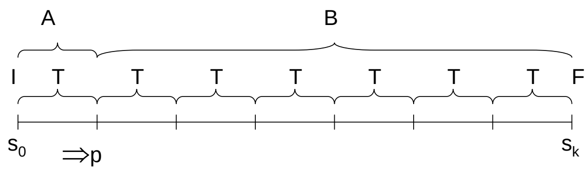

Fig. 2. Computing image by interpolation.

3 Model checking based on interpolation

We show now that bounded model checking and interpolation can be combined to produce an over-approximate image operator that can be used in symbolic model checking.

The intuition behind this is as follows. A bounded model checking problem consists of a set of constraints – initial constraints, transition constraints, final constraints. These constraints are translated to conjunctive normal form, and, as appropriate, instantiated for each time frame $0 \ldots k$, as depicted in Figure 1. In the figure, I represents the initial constraint, T the transition constraint, and F the final constraint. Now suppose that we partition the clauses so that the initial constraint and first instance of the transition constraint are in set A, while the final condition and the remaining instances of the transition constraint are in set B, as depicted in Figure 2. The common variables of A and B are exactly the variables representing state s_1.

Using a SAT solver, we prove the clause set is unsatisfiable (i.e., there are no counterexamples of length k). From the proof we derive an interpolant p for (A, B). Since p is implied by the initial condition and the first transition constraint, it follows that p is true in every state reachable from the initial state in one step. That is, p is an overapproximation of the forward image of I. Further, p and B are unsatisfiable, meaning that no state satisfying p can reach a final state in $k-1$ steps.

This over-approximate image operation can be used to compute an over-approximation of the reachable states. Because of the approximation, we may falsely conclude that F is reachable. However, by increasing k, we must eventually find a true counterexample (a path from I to F) or prove that F is not reachable (the property is true).

3.1 Basic model checking algorithm

The LTL model checking problem can be reduced to finding an accepting run of a finite automaton. The translation into this framework has been extensively studied [18, 23, 14], and will not be described here. Further, we consider only the problem of finding finite counterexamples to safety properties. Liveness properties can then be handled by the method of [1]. We assume that the problem of safety property verification is given in terms of a one-letter automaton on finite words, such that the property is false exactly when the automaton has an accepting run. Such a construction can be found, for example, in [15].

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 $R\langle w_i/v_i, w'_i/v'_i\rangle$.

For our purposes, an *automaton* is 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. In bounded model checking, we would translate the existence of a run of length $j \leq i \leq k$ 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 in the range $j \ldots k$ exists exactly when the following formula is satisfiable:[1]

$$\textsc{Bmc}_j^k = I(W_0) \wedge \left(\bigwedge_{0 \leq i < k} T(W_i, W_{i+1}) \right) \wedge \left(\bigvee_{j \leq i \leq k} F(W_i) \right)$$

We will divide this formula into two parts: one formula representing the possible prefixes of a run, and another representing the possible suffixes. The possible prefixes of length l are characterized by the following formula:

$$\textsc{Pref}_l(M) = I(W_{-l}) \wedge \left(\bigwedge_{-l \leq i < 0} T(W_i, W_{i+1}) \right)$$

That is, a prefix begins in an initial state W_{-l} and ends in any state W_0. The possible suffixes of length $j \ldots k$ are characterized by the following formula:

$$\textsc{Suff}_j^k(M) = \left(\bigwedge_{0 \leq i < k} T(W_i, W_{i+1}) \right) \wedge \left(\bigvee_{j \leq i \leq k} F(W_i) \right)$$

[1] Actually, this characterization is correct only if transition relation is total. In this paper we will assume that transition relations are total by construction. The generalization to partial transition relations is not difficult, however.

A suffix begins in any state W_0 and ends in some final state W_i, where $j \leq i \leq k$.

To apply SAT solver, we must translate formulas into conjunctive normal form. Here, we simply assume the existence of some function Γ that translates a Boolean formula f into a set of clauses $\Gamma(f, U)$, where U is a set of "fresh" variables, not occurring in f. The translation function Γ must have the property that $(\exists U.\, \Gamma(f, U)) \equiv f$. That is, the satisfying assignments of $\Gamma(f, U)$ are exactly those of f, if we remove the fresh variables. A suitable translation that is linear in the formula size can be found in [19]. What follows, however, does not depend on the precise translation function.

Our basic procedure to check the existence of a finite run of M is shown in Figure 3. In the figure, U_1 and U_2 are assumed to be sets of fresh variables, disjoint from each other and all the W_i's. The procedure is parameterized by a fixed value $k \geq 0$. We will show that the procedure must terminate for sufficiently large values of k, though for small values it may abort, without deciding the existence of a run. The procedure runs roughly as follows. First, we check that there is no run of length zero. Assuming there is not, we set our initial approximation R of the reachable states to be I, the initial states. We then, in effect, compute an over-approximation of the forward image of R. This is done by treating R as the initial condition and checking the satisfiability of the formula $\textsc{Pref}_1(M) \wedge \textsc{Suff}_0^k(M)$. If this is satisfiable there is a run of length $1 \ldots k+1$, starting at R and ending at F. In the initial iteration, when $R = I$, this means we have found a run of the automaton, and we terminate. If $R \neq I$, we abort, without deciding the existence of a run.

On the other hand, suppose that $\textsc{Pref}_1(M) \wedge \textsc{Suff}_0^k(M)$ is unsatisfiable. In this case, we partition the clauses into to disjoint sets, C_1 being the clause form of $\textsc{Pref}_1(M)$ and C_2 the clause form of $\textsc{Suff}_0^k(M)$. We then compute a Π-refutation from the proof of unsatisfiability, obtaining two formulas, p_1 and p_2 over W_0 (the global variables for this partition). In particular, p_1 is a formula that is implied by $R(W_{-1})$ and $T(W_{-1}, W_0)$. Thus, p_1 is true in all states W_0 reachable from R in one step (or put another way, it is an *over-approximation* of the states reachable in one step). Further, p_2 is true of all states W_0 that can reach F in $0 \ldots k$ steps. Since p_1 and p_2 are mutually unsatisfiable, no state satisfying p_1 can reach F in up to k steps (that is, p_1 is an *under-approximation* of the states that are backward reachable from F in up to k steps). Thus, we obtain a new approximation $R \vee p_1\langle V/W_0\rangle$ of the reachable states. When a fixed point is reached, R is an inductive invariant. Since no state in R satisfies F (nor can reach F in up to k steps), we terminate, indicating that no run exists. Otherwise, we continue the procedure with the new value of R.

Theorem 3. *For $k > 0$, if* $\textsc{FiniteRun}$*(M,k) terminates, it returns* $\textsc{True}$ *iff M has a run.*

Proof. Suppose the procedure returns $\textsc{True}$. Either $I \wedge F$ is satisfiable, in which case M has a run of length 0, or $\textsc{Bmc}_1^k(M)$ is satisfiable, hence M has a run of length $1 \ldots k$. Now, suppose the procedure returns $\textsc{False}$. We can show the following three facts:

```
procedure FiniteRun(M = (I,T,F), k > 0)
    if I ∧ F is satisfiable, return True
    let R = I
    while true
        let M' = (R,T,F)
        let C_1 = Γ(Pref_1(M'), U_1)
        let C_2 = Γ(Suff_0^k(M'), U_2)
        C = C_1 ∪ C_2.
        Run SAT on C. If satisfiable, then
            if R = I return True else abort
        else (if C unsatisfiable)
            let {p_1, p_2} be a {C_1, C_2}-refutation of C
            let R' = p_1⟨W/W_0⟩.
            if R' implies R return False
            let R = R ∨ R'
end
```

Fig. 3. Procedure for existence of a finite run

- I implies R. Trivial.
- R is an invariant of T (in other words, $R(s)$ and $T(s,s')$ imply $R(s')$). Since C_1 implies p_1, it follows that, for all states s, s', $R(s) \wedge T(s,s') \Rightarrow R'(s')$. Thus, when R' implies R, we have $R(s)$ and $T(s,s')$ implies $R(s')$.
- $R \wedge F$ is unsatisfiable. Initially, $R = I$ and $I \wedge F$ is unsatisfiable. At each iteration, we know $p_1 \wedge \textsc{Suff}_0^k(M')$ is unsatisfiable, hence $R' \wedge F$ is unsatisfiable (assuming T is total).

From these three facts it follows by induction that M has no run of any length. □.

We can also show that the procedure must terminate for sufficiently large values of k. Let use define the *reverse depth* of M as the maximum length of the shortest path from any state to a state satisfying F. This can also be viewed as the depth of a breadth-first backward traversal from F. This depth is bounded by $2^{|V|}$ but in most practical cases is much smaller.

Theorem 4. *For every M, there exists a value of k such that* FiniteRun*(M,k) terminates.*

Proof. Let k be the reverse depth of M. In the first iteration, if the SAT problem is satisfiable, the algorithm terminates. Otherwise, we know that R' cannot reach F in k steps. However, since k is the reverse depth, it follows that R' cannot reach F in any number of steps. Thus, at the next iteration R cannot reach F in up to $k+1$ steps, so the SAT problem must be unsatisfiable. Carrying on by induction, we conclude that at every iteration, R cannot reach F in up to $k+1$ steps. Thus R must continue to increase (*i.e.*, become weaker) until it reaches a fixed point, at which time the procedure terminates. □.

As a result of this theorem, when the procedure FiniteRun aborts, we have only to increase the value of k. If we continue to do this, eventually FiniteRun will terminate. The amount by which to increase k has a strong bearing on performance. If we increase it by too little, we will waste time on aborted runs of FiniteRun. On the other hand if we increase it by too much the resulting SAT problems may become intractable. We should also note that in practice, FiniteRun often terminates for values of k substantially smaller than the reverse depth.

3.2 Optimizations

A number of optimizations have been found to improve the performance of this basic algorithm. First, the formulas in the proof partition are typically highly redundant, in the sense that there are many subformulas that are syntactically different but logically equivalent. If redundant subformulas can be identified, they can be eliminated, thus reducing the size of the formula. There is a large literature on identifying logically equivalent formulas. For this paper, a simple method of building BDD's up to a small fixed size was used.

Second, significant efficiencies are obtained by replacing $\textsc{Suff}_0^k$ with $\textsc{Suff}_j^k$, for some $j > 0$ (this corresponds to testing the property for times greater than or equal to j). In most cases, setting $j = k$ seems to produce the best results. This is easily explained, since in this case the SAT solver only needs to refute the final condition at a single step, rather than at all steps. Unfortunately, if $j > 0$, we lose the guarantee of termination, except in the special case where the runs of the automaton are stuttering closed. In practice divergence has been observed for a few hardware models with two-phase clocks. This was correctable by setting $j = k - 1$. Clearly, some automated means is needed to set the value of j, but as yet this has not been developed.

Third, as in ordinary symbolic model checking, it is sufficient at each iteration to compute the forward image approximation of the "new" states, rather than the entire reachable set. In fact, any set intermediate between these will do. This is known as "frontier set simplification". Since we are dealing with arbitrary Boolean formulas rather than BDD's, there is no efficient method available for this simplification. In this work, we simply use R' (the previous image result) in place of R (the reachable states).

Finally, we note that the formula $\textsc{Suff}_j^k(M')$ is invariant from one iteration of the next. It constitutes most of the CNF formula that the SAT solver must refute. Clearly it is highly inefficient to rebuild this formula at every iteration. A far better approach would be to keep all the clauses of $\textsc{Suff}_j^k(M')$, and all the clauses inferred from these, from one run of the SAT solver to the next. That was not done here because it would require modification of the SAT solver to allow incremental changes in the SAT instance. However, the potential savings in run time is substantial.

4 Practical experience

The performance of the interpolation-based model checking procedure was tested on two sets of benchmark problems deriving from commercial microprocessor designs. The first is 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 difficult benchmark examples, these directives were removed, and a neighboring unit, the instruction folding unit (IFU) was added. This 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, of the form Gp, where p is a formula involving only the current time and the next time (usually only the current time). The number of state variables in these problems after the cone of influence reduction ranges from around 50 to around 350. All the properties are true. Tests were performed on a Linux workstation with a 930MHz Pentium III processor and 512MB of available memory. Unbounded BDD-based symbolic model checking was performed using the Cadence SMV system. SAT solving was performed using an implementation of the BerkMin algorithm [12], modified to produce proofs of unsatisfiability.

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 one were successfully verified by the interpolation-based method.

As another point of comparison, Figure 4 shows a comparison of the interpolation-based method against another method called proof-based abstraction that uses SAT methods to generate abstract models which are then verified by BDD-based symbolic model checking [16]. This method is significantly more effective than simple BDD-based model checking, successfully verifying 18 of the 20 formulas. In the figure, each point represents one benchmark problem, with the value on

[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`.

[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.

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 present interpolation-based method.[4] A time value of 1000 indicates a time-out after 1000 seconds. Points below the diagonal therefore indicate an advantage for the present 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.

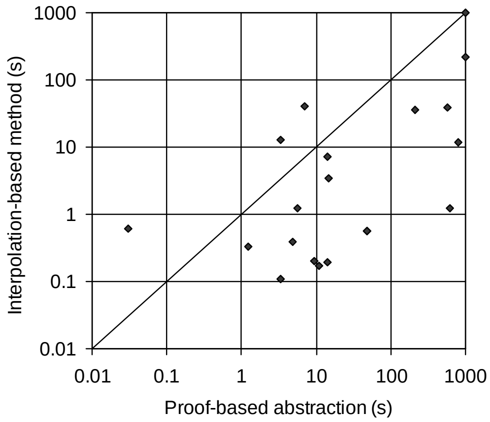

Fig. 4. Run times on PicoJava II benchmarks.

As an additional point of comparison, Figure 5 compares the performance of the proof-based abstraction approach with results previously obtained by Baumgartner *et al.* [3] 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 reported by Baumgartner *et al.* is represented on the X axis, while the average time in seconds for the present method is represented on the Y axis.[5] Thus, a point

[4] 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 Π-refutations was also negligible. A value of $j = k$ was used for these runs.

[5] 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. A value of $j = k - 1$ was used for these runs, since one benchmark was found to diverge for $j = k$.

below the diagonal line represents a lower average time for the interpolation-based method for one benchmark. We note 21 wins for interpolation and 3 for the structural method. In a few 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 (either because of time-out or incompleteness). Of the 28 individual properties that could not be resolved by Baumgartner *et al.*, all but one are successfully resolved by the proof partition method. Excluding this failed benchmark, the largest average time is 12.6 seconds.

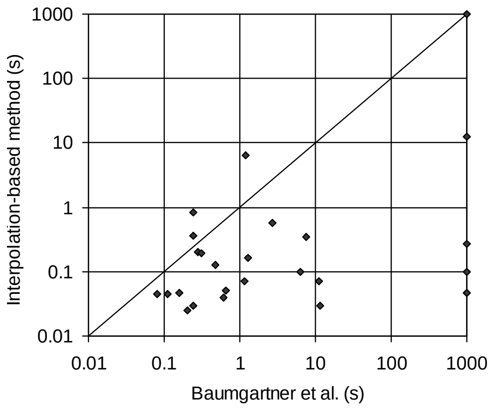

Fig. 5. Run times on IBM Gigahertz Processor benchmarks.

Finally, we compare the proof partition method against proof-based abstraction on the IBM benchmarks. The results are shown in Figure 6. 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 BDD-based model checker quickly guides the bounded model checker to the right depth, while the proof partition method systematically explores all depths. Figure 7 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.

5 Conclusion

We have observed that refutation proofs generated by bounded model checking can be exploited to allow unbounded model checking. This was observed in two microprocessor verification benchmark studies to be quite efficient relative to

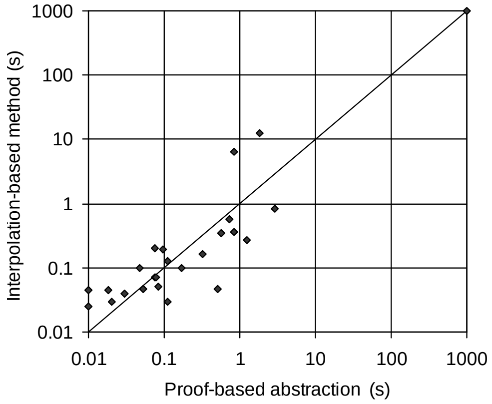

Fig. 6. Run times on IBM Gigahertz Processor benchmarks.

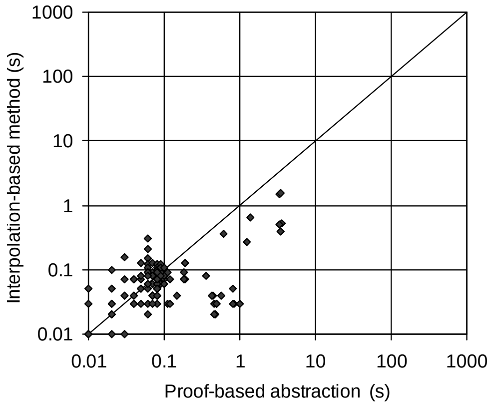

Fig. 7. Run times on IBM Gigahertz Processor true properties.

BDD-based model checking and some recently developed SAT-based methods, for true properties. We should also note that there is room to improve the performance of the implementation by re-using unfoldings from one iteration to the next.

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. For example, it is possible to extract an abstraction of the transition relation from a bounded model checking refutation, using Π-refutations. Initial studies have shown that this abstraction is difficult to handle with current BDD-based model checking methods, which rely on a certain structure in the transition relation formula. If this difficulty can be overcome, however, it might lead to an improvement in the proof-based abstraction method. Also, it is conceivable that information learned by interpolation might be exploited in the SAT algorithm itself, to improve the efficiency of bounded model checking.

Acknowledgment The author would like to thank Jason Baumgartner of IBM for providing the Gigahertz Processor benchmark problems.

References

1. C. Artho A. Biere and V. Schuppan. Liveness checking as safety checking. In *Formal Methods for Industrial Critical Systems (FMICS'02)*, July 2002.
2. P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *TACAS 2000*, volume 1785 of *LNCS*. Springer-Verlag, 2000.
3. J. Baumgartner, A. Kuehlmann, and J. Abraham. Property checking via structural analysis. In *Computer-Aided Verification (CAV 2002)*, pages 151–165, 2002.
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. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-100, Department of Computer Science, Chalmers technical university, March 1999.
6. P. Bjesse, T. Leonard, and A. Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Computer Aided Verification (CAV 2001)*, 2001.
7. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
8. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, June 1990.
9. O. C., C. Berthet, and J.-C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1989.
10. 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.
11. W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250–268, 1957.
12. E.Goldberg and Y.Novikov. BerkMin: a fast and robust SAT-solver. In *DATE 2002*, pages 142–149, 2002.
13. A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *FMCAD 2000*, pages 354–371, 2000.
14. 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.
15. O. Kupferman and M. Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
16. K. L. McMillan and Nina Amla. Automatic abstraction without counterexamples. To appear, TACAS'03.
17. 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.

18. A. Pnueli O. Lichtenstein. Checking that finite state concurrent programs satisfy their linear specification. In *Principles of Programming Languages (POPL '85)*, pages 97–107, 1985.
19. D. Plaisted and S. Greenbaum. A structure preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
20. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(2):981–998, June 1997.
21. M. Sheeran, S. Singh, and G. Stalmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer Aided Design*, 2000.
22. 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.
23. 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.
24. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification*, pages 124–138, 2000.