Combining Abstraction Refinement and SAT-based Model Checking

Nina Amla and Kenneth L. McMillan

Cadence Design Systems

Abstract. Unbounded model checking methods based on Boolean satisfiability (SAT) solvers are proving to be a viable alternative to BDD-based model checking. These methods include, for example, interpolation based and sequential ATPG-based approaches. In this paper, we explore the implications of using abstraction refinement in conjunction with interpolation-based model checking. Based on experiments using a large industrial benchmark set, we conclude that when using interpolation-based model checking, measures must be taken to prevent the overhead of abstraction refinement from dominating runtime. We present two new approaches to this problem. One is a hybrid approach that decides heuristically when to apply abstraction. The other is a very coarse but inexpensive abstraction method based on ideas from ATPG. This approach can produce order-of-magnitude reductions in memory usage, allowing significantly larger designs to be verified.

1 Introduction

Model checking [9, 26, 7], which is a widely used formal verification technique, is traditionally implemented with Binary Decision Diagrams (BDDs) [6]. Due to recent advances in tools that solve the Boolean satisfiability problem (SAT), formal reasoning based on SAT is proving to be an effective alternative to BDDs. At the core of these algorithms is Bounded Model Checking (BMC) [5], where a system is unfolded k times and encoded as a SAT problem to be solved by a SAT solver. A satisfying assignment returned by the SAT solver corresponds to a counterexample of length k. If the problem is determined to be unsatisfiable, the SAT solver produces a proof of the fact that there are no counterexamples of length k. BMC, while successful in finding errors, is incomplete: there is no efficient way to decide that a property is *true*.

Nonetheless, there are many Unbounded Model Checking (UMC) techniques that make use of SAT-based BMC in some way (see [25] for a comprehensive survey). Two methods, proof-based [23] and interpolation [22], were found to be the most robust in a recent experimental study [1] that compared many SAT-based UMC techniques. The proof-based algorithm is an iterative abstraction refinement method that typically uses a traditional BDD-based model checker to prove properties of the abstract models. It starts with a short BMC run and if the problem is satisfiable then an error has been found. However, if the problem is unsatisfiable, the resulting proof of unsatisfiability is used to guide

the formation of a new conservative abstraction. The algorithm terminates if the BDD-based model checker proves the property on the abstraction; otherwise the length of the counterexample generated by the model checker is used as the next BMC length.

The interpolation-based model checking algorithm is a purely SAT-based unbounded model checking method that does not rely on abstraction refinement, though like abstraction methods, it tends to work well on properties that are localizable, and is fairly insensitive to the addition of irrelevant logic. This method uses BMC to find failures and proves properties by doing a SAT-based approximate reachability analysis. The results in [1] show that the proof-based method does better on problems where BDDs are particularly effective. On the other hand the interpolation method has the advantage on larger problems. That paper also shows that model checking algorithms based on sequential Automatic Test Pattern Generation (ATPG) [15, 16] are competitive with BDD UMC. These findings suggest that combining the strengths of these different techniques may yield more general and robust methods.

This paper explores experimentally the issue of whether abstraction can be fruitfully combined with SAT-based UMC methods. In particular, we consider the question of how best to combine abstraction with interpolation-based UMC. Since the latter is already fairly insensitive to the inclusion of irrelevant logic, a naïve approach to localization abstraction spends more time in the refinement phase than is gained in the UMC phase. We report on two approaches to solve this problem. The first is to judiciously apply proof-based abstraction with BDDs only when it is likely to improve performance. The second is to apply a very coarse but inexpensive refinement method based on ideas from ATPG. The latter approach avoids the concretization phase that applies BMC to the concrete model, and thus results in an order-of-magnitude savings of space, though the abstractions obtained are far from optimal.

There is a fair amount of related work on integrating BDDs and various SAT-based techniques. In [12], conflict clauses that were learned from BDDs are used to improve the performance of SAT BMC. The method proposed in [8] uses BDDs to compute an over-approximation of the reachable states and applies these constraints to the SAT BMC problem. The technique described in [3] uses BDD-based reachability analysis to compute lower bounds on reachable states to accelerate SAT-based induction. Proof-based and counterexample-based abstraction methods have been combined in different phases of an iterative abstraction refinement process in [13]. The hybrid method in [2] use a single abstraction phase that is intermediate between the proof-based and counterexample-based abstraction refinement. Abstraction refinement has also been used with BMC [14] to find failures more effectively. A recent technique [18], that is closest to our work, combines abstraction refinement and interpolation in a manner which is similar to using interpolation as the UMC in a proof-based technique. The differences between this approach and the ones presented in this paper will be discussed in detail in Section 3.

The paper is organized as follows: Section 2 gives a brief overview of the algorithms, Section 3 describes the two new interpolation-based techniques, Section 4 briefly describes the experimental framework, and discusses the experimental results and Section 5 summarizes our findings.

2 Overview of the Algorithms

2.1 Preliminaries

A model $M = (S, I, T, L)$ has a set of states S, a set of initial states $I \subseteq S$, a transition relation $T \subseteq S \times S$, and a labeling function $L : S \to 2^A$ where A is a set of atomic propositions. For the purposes of this paper, we consider properties specified in the logic LTL. The construction given in [17] can be used to reduce model checking of safety properties to checking invariant properties. We use the liveness to safety construction in [4] for methods, like interpolation based model checking, which do not support liveness checks. The syntax and semantics of LTL and other temporal logics is not given here but can be found in [10].

Given a finite state model M and a safety property p, the model checking algorithm checks that M satisfies p, written $M \models p$. The forward reachability algorithm starts at the initial states and computes the *image*, which is the set of states reachable in one step. This procedure is continued until either the property is falsified in some state or no new states are encountered (a fixed point). The backward reachability algorithm works similarly but starts from the states where the property is *false* and computes the *preimage*, which is the set of states that can reach the current states in one step. The representation and manipulation of the sets of states can be done explicitly or with BDDs.

2.2 DPLL-style SAT solvers

The Boolean satisfiability problem (SAT) determines if a given Boolean formula has a satisfying assignment. This is generally done by converting the formula into a satisfiability-equivalent formula in Conjunctive Normal Form (CNF), which can be solved by a SAT solver. A key operation used in SAT solvers is *resolution*, where two clauses $(a \vee b)$ and $(\neg a \vee c)$ can be resolved to give a new clause $(b \vee c)$. Modern DPLL-style SAT solvers [21, 24, 11] make assignments to variables, called *decisions*, and generate an implication graph that records the decisions and the effects of Boolean constraint propagation. When all the variables are assigned, the SAT solver terminates with the satisfying assignment. But if there is a *conflict*, which is a clause where the negation of every literal already appears in the implication graph, a conflict clause is generated through resolution. This conflict clause is added to the formula to avoid making those assignments again. The SAT solver then backtracks to undo some of the conflicting assignments. The SAT solver terminates with an *unsatisfiable* answer when it derives the empty clause, ruling out out all possible assignments. The resolution steps used in generating the empty clause can now be used to produce a *proof of unsatisfiability*.

2.3 SAT-based Bounded Model Checking

Bounded Model Checking (BMC) [5] is a restricted form of model checking, where one searches for a counterexample (cex) in executions bounded by some length k. In this approach the model is unfolded k times, conjoined with the negation of the property, and then encoded as a propositional satisfiability formula. Given a model M and an invariant property p, the BMC problem is encoded as follows:

$$BMC(M,p,k) = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

The formula can be converted into CNF and solved by a SAT solver. If the formula is satisfiable, then the property is *false*, and the SAT solver has found a satisfying assignment that corresponds to a counterexample of length k. In the unsatisfiable case, there is no counterexample of length k and a proof of unsatisfiability can be obtained from the SAT solver.

2.4 Proof-based Abstraction Refinement

The proof-based abstraction refinement algorithm in [23] iterates through SAT-based BMC and BDD-based MC. It starts with a short BMC run, and if the problem is satisfiable, an error has been found. If the problem is unsatisfiable, the proof of unsatisfiability is used to guide the formation of a new conservative abstraction on which BDD-based MC is run. In the case that the BDD-based model checker proves the property then the algorithm terminates; otherwise the length of the counterexample generated by the model checker is used as the next BMC length. This procedure, shown in Figure 1, is continued until either a failure is found in the BMC phase or the property is proved in the BDD-based MC phase.

```
procedure PbaBdd(M,p)
1. initialize k
2. while true do
3.     if Bmc(M,p,k) is SAT then return cex
4.     M' = new abstraction derived from proof
5.     if BddMc(M',p) holds then return true
6.     let k = length of abstract cex
7. end while
end
```

Fig. 1. Proof-based procedure

2.5 Interpolation-based Model Checking

An *interpolant* $\mathcal{I}$ for an unsatisfiable formula $A \wedge B$ is a formula such that: (1) $A \Rightarrow \mathcal{I}$ (2) $\mathcal{I} \wedge B$ is unsatisfiable and (3) $\mathcal{I}$ refers only to the common variables of A and B. Intuitively, $\mathcal{I}$ is the set of facts that the SAT solver considers relevant in proving the unsatisfiability of $A \wedge B$.

```
procedure INTERPOLATION(M, p, k)
1. while true do
2.     if BMC(M, p, k) is SAT then return cex
3.     if ARC(M, p, k) then return true
4.     increase k
5. end while
end
```

Fig. 2. Interpolation procedure

The interpolation-based algorithm [22] uses interpolants to derive an over-approximation of the reachable states with respect to the property. This is done as follows (Figures 2 and 3). The BMC problem $BMC(M, p, k)$ is solved for an initial depth k. If the problem is satisfiable, a counterexample is returned, and the algorithm terminates. If $BMC(M, p, k)$ is unsatisfiable, the formula representing the problem is partitioned into $Pref(M, p, k) \wedge Suff(M, p, k)$, where $Pref(M, p, k)$ is the conjunction of the initial condition and the first transition, and $Suff(M, p, k)$ is the conjunction of the rest of the transitions and the final condition. The interpolant $\mathcal{I}$ of $Pref(M, p, k)$ and $Suff(M, p, k)$ is computed. Since $Pref(M, p, k) \Rightarrow \mathcal{I}$, it follows that $\mathcal{I}$ is *true* in all states reachable from $I(s_0)$ in one step. This means that $\mathcal{I}$ is an over-approximation of the set of states reachable from $I(s_0)$ in one step. Also, since $\mathcal{I} \wedge Suff(M, p, k)$ is unsatisfiable, it also follows that no state satisfying $\mathcal{I}$ can reach an error in $k-1$ steps. If $\mathcal{I}$ contains no new states, that is, $\mathcal{I} \Rightarrow I(s_0)$, then a fixed point of the reachable set of states has been reached, thus the property holds. If $\mathcal{I}$ has new states then R' represents an over-approximation of the states reached so far. The algorithm then uses R' to replace the initial set I, and iterates the process of solving the BMC problem at depth k and generating the interpolant as the over-approximation of the set of states reachable in the next step. The property is determined to be *true* when the BMC problem with R' as the initial condition is unsatisfiable, and its interpolant leads to a fixed point of reachable states. However, if the BMC problem is satisfiable, the counterexample may be spurious since R' is an over-approximation of the reachable set of states. In this case, the value of k is increased, and the procedure is continued. We use the optimization in [20] that sets the new value of k to be old value of k plus the number of approximate image steps done in ARC.

```
procedure ARC(M, p, k)
1. R = I, steps = 0
2. while true do
3.     M' = (S, R, T, L)
4.     let C = Pref(M', p, k) ∧ Suff(M', p, k)
5.     if C is SAT then return false
6.     compute interpolant I of C
7.     R' = I
8.     if R ⇒ R' then return true
9.     R = R' ∨ R
10.    steps = steps + 1;
11. end while
```

Fig. 3. Approximate Reachable states Computation

3 Combining Interpolation and Abstraction Refinement

3.1 Using Proof-based Abstraction in Interpolation

On certain problems BDDMC can be far more effective than the interpolation-based algorithm. In particular, on problems where one needs to go deep to find proofs or failures, BDDMC can be significantly faster. This is the motivation for using proof-based abstraction (lines 4-5 in Figure 1) in the interpolation method as shown in Figure 4.

This method works just like the INTERPOLATION procedure if *condition* is set to *false*. However, when *condition* is *true* then a proof-based abstraction is constructed and BDDMC is done on this abstraction. Thus this hybrid technique makes a choice between two possible UMC methods for proving properties with the aim of using the more efficient UMC method more often than not. The key idea being that if the INTERPOLATION procedure was doing poorly, which typically happens at larger depths, then one would use proof-based abstraction with BDDMC. However, the inability to predict when BDDs will do poorly could cause BDDMC to be the bottleneck on problems that can be proved fairly easily with just interpolation. An optimization that worked well in avoiding wasted effort was setting an effort limit on the BDDMC phase. We found that setting the limit based on the effort taken by the ARC procedure in the previous iteration was adequate. Note that there is no effort limit for the ARC procedure.

Clearly choosing the appropriate condition in Figure 4 is crucial. We use a simple progress measure for BDDMC that is based on the number of image steps completed divided by the effort needed by the BDDMC procedure. A similar measure can be computed for the ARC procedure that uses the number of approximate image steps and effort. The INTERPHYBRID algorithm starts with basic interpolation and the first time that k is greater than some predetermined limit, a proof-based abstraction is created and BDDMC is run on this abstraction with an effort limit. At the end of BDDMC step, if we detect that reachability

analysis did not start within the effort limit then BddMc is never used again. If BddMc does reachability, whether it completes or not, the number of image steps is used to compute the progress measure. If the progress of the Arc procedure becomes much slower than the progress of previous BddMc run, then BddMc is tried again. Thus the heuristic attempts to use the UMC technique which is doing better at that point and in the worst case, when the BDDs are blowing up, we do only one BddMc run with a low effort bound.

```
procedure InterpHybrid(M, p)
1. initialize k
2. while true do
3.     if Bmc(M, p, k) is SAT then return cex
4.     if condition then
5.         derive abstraction M' from proof
6.         if BddMc(M', p) then return true
7.         k = length of abstract cex
8.     else
9.         if Arc(M, p, k) then return true
10.        increase k
11.    end if
12.    update condition
13. end while
end
```

Fig. 4. Hybrid Interpolation and BDD-based PBA procedure

Combining abstraction refinement with interpolation has been explored in [18]. This approach is similar to using interpolation instead of BddMc as the UMC procedure in the proof-based method. However there are a number of differences between this framework [19, 18] and the one in Figure 1. In proof-based abstraction (Figure 1), a new abstraction is derived from the proof of unsatisfiability in each iteration, while the method in [19] starts with an initial abstraction and refines this abstraction in each iteration. Another difference is the way the counterexample is concretized. In proof-based abstraction, BMC at the depth of the abstract counterexample is done on the concrete model to check if the counterexample is real. In contrast, the method in [19] takes an incremental approach that attempts to concretize the counterexample on abstract models and successively refines the abstraction until either the counterexample is eliminated or is determined to be real on the concrete model. In [18], they find that combining abstraction refinement with interpolation results in performance gains over the basic interpolation method, but they also observe the technique was not an effective way to improve the performance of interpolation without optimizations, like refinement prediction and minimization, that are focused on reducing the size of the abstraction.

3.2 Incremental Interpolation

Unlike BDD-based model checking, the interpolation-based method is fairly robust with respect to the addition of irrelevant state variables. For this reason, we can use fairly coarse and inexpensive methods of abstraction refinement. In particular, if the model is large (with greater that a few thousand state variables) it may not be practical to concretize abstract counterexamples using a SAT solver because of the space requirement of the BMC unfolding. Here, we will consider one method that avoids this concretization step, by borrowing some ideas from ATPG methods. Sequential ATPG methods search for input sequences to a circuit that test for the presence of a given fault.

On obtaining an abstract counterexample, we will attempt to produce a minimal justification of the abstract counterexample by assigning Boolean values to a subset of the free variables (that is, the primary inputs and the hidden state variables). A *justification* is a partial assignment that is sufficient to imply that the property is false in the abstraction $\hat{M}$. The set of hidden state variables H that are assigned in this justification at any time frame will be called the *justification frontier*.

Refinement consists of choosing some subset of the justification frontier and adding these state variables to the abstraction. Heuristically, these variables are more likely to be useful in eliminating the abstract counterexample, since those not occurring in the justification frontier are not relevant to the falsehood of the property in the cex. There is no guarantee, however, that the set of variables we choose is sufficient to eliminate the counterexample. Moreover, we may add many irrelevant variables in this way. We rely on the fact that adding irrelevant variables does not greatly effect the performance of the interpolation-based model checking method, so long as there is sufficient space to build the BMC unfolding.

procedure $\textsc{InterpInc}(M, p, k)$
1. choose initial abstraction $\hat{M}$
2. while *true* do
3. if $\textsc{Arc}(\hat{M}, p, k)$ return *true*
4. let C be the abstract *cex*
5. let $J = \textsc{JustifyCex}(C, \hat{M}, p, k)$
6. let $H = \{r \mid r \text{ is hidden in } \hat{M}\}$
7. let $JF = \{r \in H \mid r_i \in \mathrm{dom}(J), \text{for some } i \in 0..k\}$
8. if $JF = \emptyset$ then return *cex* C
9. choose a non-empty subset of JF
10. add these variables to $\hat{M}$
11. end while
end

Fig. 5. Incremental Interpolation procedure

Thus, the main intent of this approach is to prevent failure due to lack of space. If, for example, for a model with 100,000 state variables, we select 5000 state variables, out of which only 100 are actually necessary to prove the property, then we may succeed in preventing memory exhaustion and successfully prove the property, even though concretization is not feasible in such a large model.

Moreover, if at some point the justification frontier contains no hidden variables, then we have obtained a concrete counterexample, since the abstract counterexample fully justifies the falsehood of the property in the concrete model. This makes it possible to find concrete counterexamples without unfolding the concrete model, and in fact we will see cases where concrete counterexamples are obtained, but a BMC unfolding is infeasible due to lack of space.

The overall refinement procedure is outlined in Figure 5. We begin with an empty abstraction, and check the abstract model. If an abstract counterexample is obtained, we produce a justification. This is a subset of the assignments in the abstract counterexample that is sufficient to imply falsehood of the property. That is, if C is the abstract counterexample, then a justification $J \subseteq C$ is an assignment to the free variables over time, such that

$$J \wedge \textsc{Bmc}(\hat{M}, p, k) \Rightarrow \bigvee_{i \in 0 \ldots k} \neg p(s_i)$$

Procedure JustifyCex computes a justification by a simple greedy approach that traverses the circuit unfolding depth-first from the property, justifying the output of each gate by choosing a sufficient subset of its inputs. We first apply all assignments to the primary inputs, and propagate the implications of these assignments in the unfolding. Then we complete the justification by traversing the unfolding in a depth-first manner from its output (the property node), justifying the output of each gate by choosing a sufficient subset of its inputs. We have also modified the SAT solver to terminate with a partial assignment when the truth of the formula is justified. More sophisticated methods of finding a small justification are possible [27], though this may or may not provide a performance benefit.

The procedure then computes the justification frontier, and picks a subset of the state variables on the frontier to add to the abstraction. We choose a fixed number N of variables that have the highest aggregate VSIDS score in the SAT solver [24] for the BMC run that produced the abstract counterexample. Again, more sophisticated heuristics may be possible here. If the justification frontier is empty, we have a concrete counterexample, and we terminate. Note this procedure is terminating, since it adds at least one variable at each iteration.

The InterpInc procedure can be more expensive than the Interpolation procedure on the smaller problems hence we chose to be conservative in applying this method. This can be done within the Interpolation procedure by checking the size of the unfolding each time k is increased and switching to InterpInc only when a predetermined threshold limit is reached. By setting a very large threshold limit we invoke this procedure only if the size of the model or the value of k is large.

4 Experimental Analysis

In order to measure the relative performance of the algorithms described in the previous sections, we used the same BDD-based model checker and SAT solver. The SAT solver is incremental [28], in the sense that it is possible to add/delete clauses and restart the solver, while maintaining all previously inferred conflict clauses that were not derived from deleted clauses.

The benchmark set used has 1205 problems that were derived from 85 hardware designs which ranged in size from a few hundred to more than 100,000 lines of HDL code. Each design in our benchmark set contained from one up to a few hundred properties to check. The set contained some liveness properties but about 98% of properties were safety checks. There were 799 passing properties, 312 failures and 94 problems with unknown results. We partitioned the problems into three sets (BM1-BM3) based on size and difficulty of the problem. The set MOUT contains problems from BM1-BM3 where some algorithm ran out of memory. Table 1 characterizes the types of problems in each set. The Table shows the number of problems and the time limit used in each set. The average size of the problems in terms of the number of state variables, combinational variables and inputs is also given. For our experiments we used identical Redhat Enterprise Linux machines, each with an AMD Opteron CPU at 2GHZ and 2.6GB of available memory.

Benchmark	# Probs.	Time Limit (seconds)	Avg. Size		
			State	Comb.	Inputs
BM1	394	100	78	989	154
BM2	494	1000	307	2364	198
BM3	317	3600	2210	19363	1067
MOUT	29	10000	19055	151557	8018

Table 1. Characterization of the benchmark sets.

We ran each problem with the specified time limit and measured the number of problems solved by each method. For all the tables and plots in the sequel, the time reported for any unresolved problem is the time limit for that problem even if the method ran out of memory in far less time. The tables present data for the five algorithms: the original proof-based method with BddMc (PbaBdd), the proof-based method in Figure 1 with Arc(M', p, k) instead of BddMc(M', p) (PbaInterp), the interpolation method (Interpolation), the hybrid method that combines interpolation with BDD-based proof-based abstraction (InterpHybrid) and the new incremental interpolation method (InterpInc). Table 2 reports the number of problems resolved (fin) and average time taken per problem (Av time) for benchmark sets BM1-BM3. For the entire set of problems (ALL), we also report the geometric mean (Gmean) of the run time. Table 3 presents results for the entire set partitioned into passes and failures. Table 3

report the average terminal BMC depth, and the number of "wins" with respect to time, where a win is attributed to a particular algorithm if it does better than all others with respect to runtime. In the case of a tie, which we defined to be two runs where the difference was less than 5% of the run time, we award a win for both methods.

Algorithm	ALL			BM1		BM2		BM3	
	# fin	Av time	Gmean	# fin	Av time	# fin	Av time	# fin	Av time
PbaBdd	1010	345.2	29.2	393	14.6	398	220.3	219	950.6
PbaInterp	951	486.1	40.7	360	26.8	389	252.2	202	1421.7
Interpolation	1032	339.5	24.2	389	16.3	399	227.2	244	916.2
InterpHybrid	1068	272.1	21.4	394	14.6	411	203.9	263	698.4
InterpInc	1047	324.7	24.6	389	17.0	401	224.8	257	863.0

Table 2. Results for benchmark sets (BM1-BM3).

Table 2 shows Interpolation by itself is more robust overall than either PbaBdd or PbaInterp. The fact that PbaInterp performs worse than Interpolation can be explained by observing in Table 3 that the terminal BMC depth for the PbaBdd and PbaInterp methods is on average longer than Interpolation. This is consistent with the observation in [18] that just replacing Interpolation as the UMC in a proof-based abstraction framework does not necessarily improve the performance of interpolation. The fact that, within the proof-based abstraction framework, BddMc is more effective overall than Interpolation is somewhat surprising since in general Interpolation dominates BddMc[1]. One possible reason is that BDDs do well on most small models and the abstractions derived in the proof-based method tend to be small. A second reason, which is the argument made in [23], is that SAT solvers do better when the number of relevant variables is small in comparison to the total number of variables. Since the abstractions are derived from proof generated by the SAT solver, the number of relevant variables is likely to be higher than usual, which could cause Interpolation to be less effective.

Algorithm	Passes				Failures			
	# fin	# wins	depth	Av. time	# fin	# wins	depth	Av. time
PbaBdd	720	129	27	213.1	290	121	25	236.0
PbaInterp	671	49	21	386.9	280	70	20	335.6
Interpolation	735	153	13	211.7	297	14	20	217.9
InterpHybrid	766	341	17	132.7	302	85	24	159.7
InterpInc	746	112	13	192.3	301	13	20	210.3

Table 3. Results partitioned into passes and failures.

As we can see in Table 2 the InterpHybrid is the most robust method on all three benchmarks and more so on the larger examples. On the 138 problems that could not be resolved by the Interpolation method, the InterpHybrid procedure resolved 26% of these problems and the InterpInc procedure resolved 12% of them. On the 766 problems that were verified by InterpHybrid, approximately 20% were resolved using BddMC while the rest were resolved with Arc procedure. Table 4 presents the same data as Table 3 but partitions the problems into two sets: one with problems that were resolved by all five methods (All resolved) and the other with problems that were unresolved by some method (Some unresolved). As shown in Table 4, the InterpHybrid method is faster than the Interpolation on both sets. It appears that the simple heuristic in InterpHybrid is fairly effective in choosing the appropriate UMC, and we find that the overhead of using BddMc is minimal. This leads us to conclude that since BddMc works better as the UMC for Pba, it is better to add PbaBdd to Interpolation rather than use Interpolation within the proof-based abstraction refinement method.

Algorithm	All Resolved				Some Unresolved			
	# fin	# wins	depth	Av. time	# fin	# wins	depth	Av. time
PbaBdd	887	138	14	28.5	123	61	96	266.8
PbaInterp	887	57	14	52.5	64	6	56	386.8
Interpolation	887	132	8	27.5	145	35	47	236.6
InterpHybrid	887	376	8	21.3	181	60	95	218.4
InterpInc	887	87	8	28.7	160	38	75	352.9

Table 4. Summary table for resolved problems.

The InterpInc procedure has the same performance on BM1 and BM2 but is slightly more robust on the harder problems in BM3. The incremental interpolation method, however, was intended to be efficient with respect to space. Therefore we consider the results for the 29 problems where some algorithm ran out of memory in Table 5. The Table presents the number of problems that passed, failed, exceeded the time limit (TO), ran out of memory (MO), the average time and the number of state variables in the last abstraction (Abs. size). We increased the time limit for these problems to 10000 seconds to gauge whether the incremental algorithm could solve additional problems with more time. The data indicates that the incremental interpolation approach is very effective in resolving these problems. The average memory usage of the InterpInc procedure on these problems is 623 Megabytes which indicates that the method is highly efficient with respect to memory usage. Table 5 shows that although the InterpInc method yields larger abstractions, it is still more efficient in terms of performance than PbaInterp. This demonstrates that a very coarse but fast abstraction refinement heuristic can be effective with interpolation. Our ATPG-

style heuristic gives a small overall improvement in robustness of interpolation with a large improvement in space.

Algorithm	# Pass	# Fail	TO	MO	Avg. Time	Abs. size
PbaBdd	0	0	5	24	10000.0	38
PbaInterp	0	0	11	18	10000.0	70
Interpolation	1	1	13	14	9349.4	-
InterpHybrid	1	2	5	21	9013.1	-
InterpInc	8	6	13	2	6527.0	999

Table 5. Results for the 29 Memory Intensive Examples (MOUT).

Figures 6 and 7 contain scatter plots of runtime in seconds. We see in Figure 6 that Interpolation and its two variants, InterpHybrid and InterpInc, are highly correlated but both variants have an advantage on problems that are hard for Interpolation to solve. Figure 7 is interesting since it shows that, but for a few cases, PbaInterp does far worse than Interpolation. The right plot in 7) shows the run time of PbaInterp versus a parallel run of Interpolation and Pba (i.e. the best result of both methods). We see that PbaInterp is slower in general but does resolve some problems that the parallel runs could not.

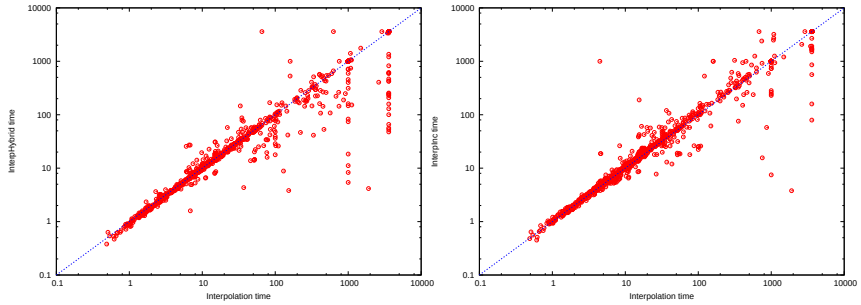

Fig. 6. Plot of time in seconds. Left: X-axis is Interpolation and Y-axis is InterpHybrid. Right: X-axis is Interpolation and Y-axis is InterpInc.

5 Conclusions

This paper focused on combining abstraction refinement with interpolation-based UMC, with the goal of making this method more general and robust. First, we added a proof-based abstraction step to interpolation in order to use

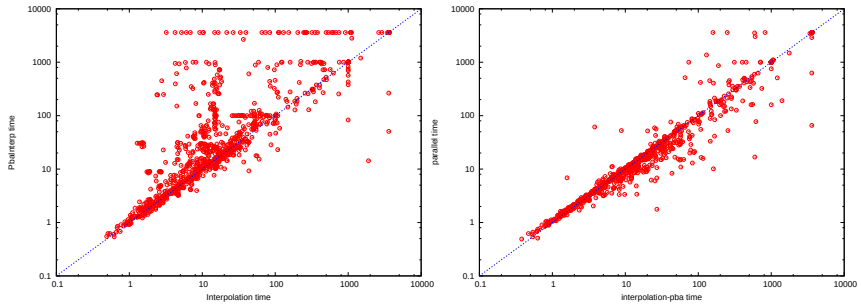

Fig. 7. Plot of time in seconds. Left: X-axis is Interpolation and Y-axis is PbaInterp. Right: X-axis is InterpHybrid and Y-axis is a parallel run of Interpolation and Proof-based abstraction.

BDDs when they prove to be effective. This method was found to be very efficient on the problems in our benchmark set. Next, we describe a new incremental interpolation method that is designed to be memory efficient. This technique uses ATPG style justification in the concretization step which is generally the bottleneck with respect to space. A conservative application of this method was very effective on memory intensive problems and competitive with the interpolation method in general. Our findings can be summarized as follows.

1. The basic interpolation method is more robust overall than proof-based abstraction, with either interpolation or BDDs as the UMC.
2. Simple proof-based abstraction is not an effective way to improve the performance of interpolation as observed in [18]. We found that the terminal BMC depth for PbaInterp is on average longer than interpolation which in part explains the performance differences.
3. Since the data shows that BddMc is more effective as the UMC in Pba, one can conclude that adding PbaBdd to the interpolation method is better than using interpolation as the UMC in Pba.
4. A very coarse but fast abstraction refinement heuristic can be effective with interpolation. Our ATPG-style heuristic gives a small overall improvement in robustness of interpolation with a large improvement in space.

References

1. N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *CHARME*, 2005.
2. N. Amla and K. McMillan. A hybrid of counterexample-based and proof-based abstraction. In *FMCAD*, 2004.
3. M. Awedh and F. Somenzi. Increasing the robustness of bounded model checking by computing lower bounds on the reachable states. In *FMCAD*, 2004.

4. A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002.
5. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
6. R. E. Bryant. Graph-based algorithms for boolean function manipulations. *IEEE Transactions on Computers*, 1986.
7. J. R. Burch, E. M. Clarke, K. L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.
8. G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based bounded model checking by means of bdd-based approximate traversals. In *DATE*, 2003.
9. E.M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, 1981.
10. E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, 1990.
11. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *DATE*, 2002.
12. A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Learning from BDDs in SAT-based bounded model checking. In *DAC*, 2003.
13. A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative abstraction using SAT-based BMC with proof analysis. In *ICCAD*, 2003.
14. A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *CAV*, 2005.
15. C.Y. Huang, B. Yang, H.C. Tsai, and K.T. Cheng. Static property checking using atpg versus bdd techniques. In *ITC*, 2000.
16. M. Iyer, G. Parthasarathy, and K.T. Cheng. SATORI- an efficient sequential SAT solver for circuits. In *ICCAD*, 2003.
17. O. Kupferman and M. Vardi. Model checking of safety properties. In *Formal Methods in System Design*, 2001.
18. B. Li and F. Somenzi. Efficient abstraction refinement in interpolation-based unbounded model checking. In *TACAS*, 2006.
19. B. Li, C. Wang, and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. In *STTT*, 2005.
20. J. Marques-Silva. Improvements to the implementation of interpolant-based model checking. In *CHARME*, 2005.
21. J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEETC: IEEE Transactions on Computers*, 48, 1999.
22. K. McMillan. Interpolation and SAT-based model checking. In *CAV*, 2003.
23. K. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, 2003.
24. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.
25. M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. In *STTT*, 2005.
26. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. of the 5th International Symposium on Programming*, 1982.
27. K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *TACAS*, 2004.
28. J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *DAC*, 2001.