

A Hybrid Model Checker

Michael L. Case Kelvin Lwin
Department of EECS, University of California, Berkeley
{casem, klwin}@eecs.berkeley.edu

Abstract—Any model checker is limited by the time needed to determine the set of reachable states. We propose a hybrid model checker that successfully combines two approximation methods for fast and robust reachability analysis. Experiments on both sequential equivalence problems and safety property verification problems illustrate a speedup over previous SAT-based unbounded verification methods.

I. INTRODUCTION

The low capacity of model checkers is limiting their use in industrial verification flows. The scalability is fundamentally limited by an explosion in the number of states as the number of registers increases.

Reachability analysis is at the core of most verification systems, but it is only practical for small designs. It explores the state space in a breadth-first manner, starting from the initial state. Generally, binary decision diagrams (BDDs) are used to represent the set of states that have been visited. While this approach is very fast when run on small designs, it is subject to the BDD memory explosion problem, resulting in unpredictability and failure to terminate.

Recent checkers are taking advantage of fast modern SAT solvers which increase the size of the designs that are verifiable. Broadly speaking, the solvers comes in three flavors: bounded model checkers (BMC), SAT-based unbounded methods, and hybrid SAT-BDD methods. It was shown by benchmarking 1000 industrial designs [19] that interpolation is the most robust of the SAT-based methods. However, it does not win on all types of designs and has room for improvement.

We can improve on interpolation by utilizing an implication routine that is a fast method to over-approximate the set of reachable states. Given the logic functions a and b at the output of two nodes in the network, it is inductively proved that $a \Rightarrow b$ in every reachable state. A large set of implications is proved, and the conjunction of the implications represents an over-approximation to the reachable state set. [20]

In this work we propose a hybrid model checker. This hybrid checker draws on the strengths of both interpolation and the implication routine. Implications are found as a preprocessing step, and this gives an over-approximation to the reachable state set. This over-approximation is used to prune the state space that the interpolation model checker must search, and it dramatically decreases the time needed for verification. This concept is illustrated in Figure 1.

Our contributions in this work are:

- Explored the theoretical aspects of forming a hybrid model checker.

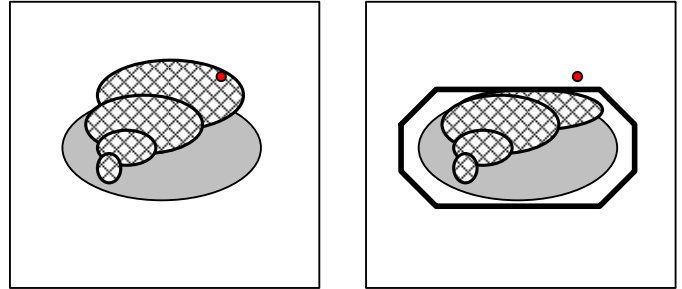


Fig. 1. Interpolation with (left) and without (right) a state-space over-approximation. The search starts in the set of initial states, and the set of explored states is allowed to grow. The search may erroneously leave the reachable state space (gray) and hit an unreachable bad state (red). If the search is confined to an over-approximation of the reachable state space then this will not happen.

- Implemented interpolation in the ABC framework [12].
- Implemented all intermediate code to connect implications to interpolation and form a hybrid model checker.
- Extensively benchmarked the two model checkers.

II. RELATED WORK

State traversal is typically performed through symbolic exploration of the system using BDDs to represent set of reachable states ([4], [10], [17]). Typical forms of exploration are forward traversal, back traversal, and mixed forward/backward traversal [7]. Trouble dealing with large systems lead to state space pruning techniques and approximate traversal techniques [5]. Several others proposed methods to partition the state transition relation before it is converted to a BDD ([1], [2], [6]). These methods result in a BDD-based implementation that scales better, yet still suffers from the fundamental BDD explosion problem and hence failure to produce a usable result in many designs.

SAT-based systems are becoming a viable alternative to BDDs due to advancements made in SAT solvers. There are several known methods of performing verification with a SAT solver:

- Bounded model checkers (BMC) try to find a counter-example of length k and only attempt to prove on a restricted model [18].
- ATPG-MC uses search strategies borrowed from test pattern generation to perform a backward search through the circuit represented in CNF [13].
- The interpolation model checker verifies models using an approximation to the image computation [14].

- Quantifier MC eliminates quantifications by enumerating all the possibilities of satisfying assignments in producing a SAT formula.

Recently industrial examples were used to compare nine SAT-based model checking techniques [19], and the most robust system is shown to be interpolation [14].

Researchers have tried to combine SAT and BDD engines to better cope with different types of problems [11]. Proof-based Abstraction Refinement iterates through MC based on BDD and BMC using SAT. Another approach is to prune the search space for SAT through computation using BDDs [16]. Other researchers have addressed sequential optimization without reachable state analysis. Mehrotra et. al. [3] present one such approach in which they use recursive learning to discover redundancies across latch boundaries. Van Eijk [8] presents a method to inductively discover nodes between two designs that are equivalent in every reachable state. The equivalences are used to simplify the equivalence check, bypassing the need for reachability analysis. Bjesse and Claessen [9] generalized this technique and improved its performance, but they are limited to equivalence relations between two designs.

III. THEORY

It is well known that verifying a safety property can be reduced to checking that a set of bad states are not reachable, and so verification can be done by state reachability analysis. Two methods of finding reachable states are explicit state traversal and proving invariants.

In explicit state traversal, the reachable state set is developed in a breadth-first search starting from the initial states. If a fixed point is reached before a bad state is encountered then all reachable states have been explored and the design is verified. In practice, the large size of the state space often makes this method impractical.

The interpolation method is an abstraction of explicit state traversal. Instead of explicitly computing the image of a set of states under the transition relation, it finds an over-approximation to this set. This over-approximation allows it to proceed more quickly through the reachable state set.

Proving invariants is a competing method for finding the set of reachable states. In this method, the state graph is not traversed. Instead, an invariant over the set of states is proved inductively. The proof is done so that every reachable state satisfies the invariant, and therefore the image of the invariant is an over-approximation to the set of reachable states.

In this work, we harness the power of both reachability techniques to create a hybrid model checker that is more powerful than either of its two constituents.

A. The Interpolation Method

The interpolation method is a way of doing unbounded model checking.

1) *A Brief Overview:* Suppose we have a design we wish to verify, and we do a bounded model check (BMC) on that design. If no bugs are found then the underlying SAT problem appears unsatisfiable. In this case, we can extract a

Boolean expression called the interpolant from the proof of unsatisfiability. The interpolant is an over-approximation of the states reachable in 1 step from the initial state used in the BMC.

This concept can be used to construct an unbounded model checker like so:

```

for (bmcDepth = 2; true; bmcDepth += 10) {
  startStates = initialStates;
  while (true) {
    // if the model looks broken
    if (! bmc(startStates)) {
      // if the counterexample is real
      if (startStates == initialStates)
        return COUNTEREXAMPLE;
      // counterexample may be spurious
      else break;
    } else {
      startStates += getInterpolant();
      if (fixedPoint(startStates))
        return VERIFIED;
    }
  }
}

```

The interpolant is used to over-approximate the image of a set of states, and this over-approximated image is used in a manner similar to what is done in explicit state traversal. If a fixed point is reached, then every reachable state has been explored and we know that the design is verified.

If a bad state is reached, the counterexample is only considered valid if the bounded model checker started at the true set of initial states. If it started from an interpolation result then this interpolant could contain an unreachable state. The fact that a bad state is reachable in `bmcDepth` transitions from an unreachable state is not interesting, and to avoid this the bounded model checking depth is increased and the proof begins anew in an attempt to find a concrete counterexample.

2) *Analysis of the Interpolation Method:* Interpolation is a unbounded model checking algorithm that is both complete and sound. It is the most robust verification method known to date [19]. However, it is not without its flaws. The main defect is that if an interpolant includes an unreachable state then the algorithm will begin exploring traces starting at this unreachable state. These traces add to the overall runtime unnecessarily, and reducing these erroneous and extraneous traces would directly improve the efficiency of interpolation.

B. The Implication Method

The implication method is a way of quickly finding an over-approximation to the reachable state space.

1) *A Brief Overview:* Let a and b be the Boolean expressions present at the output of two nodes in the logic network, and consider the simple invariant $a \Rightarrow b$. Such simple invariants are considered because they form an implication graph which can be minimized, and so the set of invariants to be proved can be reduced to a minimal set [20].

In the base case of the inductive proof of $a \Rightarrow b$, the implication is tested in each of the initial states. If it fails

to hold in one of these states then we know that it doesn't hold in every reachable state.

The inductive hypothesis is that if the implication holds in one state, then it should also hold in every next state. If the inductive hypothesis holds then $a \Rightarrow b$ holds in every reachable states.

In our method we pick a and b at random and attempt to prove the implication. We do this in a loop, proving as many implications as we can before a timeout is reached. This is summarized in the pseudocode below:

```
while (!timeout) {
    attempt to inductively prove an implication;
}
construct the reachable state space approximation;
```

When the timeout is reached (or all possible a , b are explored), we take the conjunction of all proved implications. This forms an invariant that is true for every reachable state. The set of states that satisfy this invariant form an over-approximation to the reachable states.

2) *Analysis of the Implication Method:* The main strength of this method is that we can trade runtime for quality of results. As the runtime is increased, more implications can be proved. Since the invariant is the conjunction of the proved implications, as more implications are proved the set containing states which satisfies the invariant shrinks. This will give a tighter over-approximation to the reachable states.

This method is dramatically faster at finding a reachable state approximation than explicit state traversal methods or approximate traversal methods like interpolation. The reason for this is that the proofs are based on induction and so there is no dependence on the structure of the underlying state transition graph.

The main weakness of this method is that there is no good way to select the nodes a and b for which we will attempt to prove $a \Rightarrow b$. Previously we had considered the following heuristic which was later found to be not useful:

```
// given: a set of bad states
while (!timeout) {
    select an implication a->b
    if (forall bad states, a->b does not hold)
        attempt to inductively prove a->b;
}
construct the reachable state space approximation;
// bad states are proven unreachable
```

In practice, this heuristic did not perform well because it was found that most implications hold in at least one bad state. As a result, no useful invariant is computed because essentially no implications can be proved. Therefore this heuristic is not considered in this work and the implications are selected at random.

3) *Application to Model Checking:* The implication method can verify a safety property by checking that all bad states do not satisfy the invariant. If this is true then we have proved

that all bad states are unreachable.

If we find that a bad state satisfies the invariant, we do not know if this state is truly reachable because the invariant represents an over-approximation to the set of reachable states. Furthermore, since the implication method does not traverse the state space, no counterexample trace can be generated to show how the system reaches the bad state.

In summary, the implication method can form a fast yet incomplete model checker. However, by itself it is ill-suited to this task.

C. Combining the Strengths of the Two Methods

Consider the following method which combines the strengths of both the interpolation and implication methods.

```
// Design preprocessing
for (i = 0; i < N; i++) {
    findImplications(5 seconds);
    extract equivalence classes;
    collapse equivalence classes into one node;
    combinationally re-synthesize;
}
// Model checking
interpolate();
```

In this procedure, the design is preprocessed with implications before it is model checked with interpolate.

Each preprocessing step uses 5 seconds of processing time attempting to prove implications. Node equivalence classes are then extracted from the implications that have been proved. These classes are each collapsed into a single class representative, and then the design is combinationally re-synthesized.

Each preprocessing step helps later procedures in the following ways:

Sequential resynthesis is performed in each step. Nodes that are equal in every reachable state are collapsed. Because these nodes may not be equal in the unreachable states, this is a sequential transformation. The authors acknowledge that this is a very simple transformation, and this method could be made more powerful with a resynthesis technique that uses all implications, not just those involved in equivalence classes. The sequential (and subsequent combinational) resynthesis helps later algorithms by reducing the size of the problem.

Implication candidates are changed because combinational resynthesis changes the individual nodes in the design. The Boolean functions that are candidates for implications are changing, and this makes it possible to express implications that may have been previously inexpressible.

The reachability over-approximation is refined. Each iteration computes an invariant, and the conjunction of these invariants gives a reachability approximation that gets tighter with every pre-processing iteration.

After the preprocessing, every bad state that violates the safety property might lie outside the reachability over-approximation. In this case, the design is verified and we terminate. If a single bad state satisfies our over-approximation

TABLE I
THE ISCAS89 SEC BENCHMARKS.

Circuit Statistics			Reachable States (%) After Preprocessing Step					Verification ²	Runtime (sec) ³		Verification
Benchmark	Nodes ¹	Latches	#1	#2	#3	#4	#5	Complete	Interpolate	Hybrid	Result
s27	17	6	12.50	12.50	-	12.50	12.50	true	0.97	0.43	pass
s208	99	16	0.39	100.00	0.39	-	0.39	true	1200.00	0.49	pass
s298	187	45	1.66	1.66	0.10	0.10	0.05	false	456.60	257.09	pass
s344	235	56	3.44	0.18	0.01	0.02	0.00	false	1200.00	1200.00	-
s349	235	56	3.71	1.25	0.22	0.01	0.01	false	1200.00	1200.00	-
s382	223	54	3.07	0.34	2.17	0.04	0.04	false	1200.00	1200.00	-
s400	232	56	9.06	0.42	0.10	0.01	0.00	false	1200.00	1200.00	-
s444	233	53	2.48	0.73	0.73	0.29	0.23	false	1200.00	1200.00	-
s641	301	39	7.98	0.92	0.00	0.00	0.00	true	57.69	0.66	pass
s713	303	39	31.25	8.59	1.66	0.59	-	false	40.86	9.26	fail
s420	229	39	39.06	39.06	13.06	20.62	9.84	false	1200.00	1200.00	-
s386	268	28	0.01	0.00	0.01	-	0.00	true	30.80	0.76	pass
s526n	267	59	42.40	2.94	0.93	0.05	0.01	false	1200.00	1200.00	-
s526	268	67	1.05	1.05	1.05	0.01	0.00	false	1200.00	1200.00	-
s510	434	31	5.13	2.79	0.57	1.27	0.34	false	110.56	91.30	pass
s820	570	19	61.68	36.74	20.13	23.47	23.47	false	140.73	184.03	pass
s838	442	72	56.25	56.25	31.64	28.13	14.94	false	1200.00	1200.00	-
s832	577	22	73.33	42.77	26.88	8.86	5.01	false	179.89	157.05	pass
s953	466	39	84.38	35.01	19.54	7.40	2.84	true	0.80	0.82	pass
s1423	896	163	100.00	53.58	100.00	20.59	-	false	1200.00	1200.00	fail
s1196	685	37	40.63	40.63	40.63	20.31	13.49	false	3.38	2.70	pass
s1238	784	37	43.75	23.44	12.89	23.44	3.55	false	5.08	3.27	pass
s1488	1217	54	47.03	8.76	1.10	1.10	0.24	false	360.64	262.92	pass
s1494	1236	23	54.27	19.09	14.87	11.91	10.25	false	114.10	161.11	pass
s5378	1963	421	1.93	11.06	1.93	1.93	2.64	false	21.51	1.40	pass
s9234	2675	497	82.03	41.02	-	100.00	-	false	7.97	0.25	fail
s13207.1	4321	1606	25.36	25.36	25.36	-	100.00	false	22.00	20.31	fail
s13207	3660	1831	2.81	2.81	2.81	2.81	2.81	false	2.97	2.80	pass
s15850	5331	1612	21.88	21.88	21.88	21.88	21.88	false	3.60	3.40	pass
s38417	16684	4041	-	100.00	100.00	100.00	100.00	false	88.01	0.24	fail
s38584.1	18705	4672	100.00	99.95	99.90	28.80	99.90	false	109.79	103.69	fail
s38584	20276	4747	100.00	100.00	100.00	100.00	100.00	false	16.89	31.35	fail

¹ Circuit statistics represent the product machine. Nodes reported are the number of and nodes in an and-inverter graph.

² Verification using only the invariant.

³ 1200 second timeout.

invariant then we call a modified version of interpolation that does not explore states violating the invariant. The interpolation should be dramatically sped-up because it is presented both with a smaller problem and also a useful reachability over-approximation.

IV. EXPERIMENTAL RESULTS

The implication procedure, the interpolation procedure, and all intermediate tools necessary to form a hybrid solver were implemented in C++ in the ABC Logic Synthesis and Verification framework [12]. Two sets of benchmarks were then run through the system: a set of sequential equivalence checking problems over the ISCAS89 benchmarks, and a set of safety property benchmarks from [15]. Safety benchmark contains examples from Cadence SMV, CMU SMV, SMV case studies, NuSMV, VIS, Texas97, ISCAS89 and IRST model checking group. A 3 GHz Pentium 4 machine was used to generate all the results.

A. Sequential Equivalence Checking

To explore the properties of our hybrid solver, we created a set of synthetic verification benchmarks from the ISCAS89 test suite. We retimed each design and then built a product machine and miter to verify that the outputs of the original machine

match those of its retimed counterpart. If the retiming is done correctly, the output of the miter will be 0 in every reachable state. The synthetic benchmarks were chosen because they represent hard verification problems with many latches, and as such they stress the interpolation method.

Table I shows the results of the sequential equivalence checker (SEC) benchmarks. We first ran interpolation on these designs and got the mix of passing, failing, and undecidable properties shown in the table. Note that the retiming method we use does have bugs. Next we applied our hybrid model checker.

We preprocessed the design 5 times. Each preprocessing step takes 5 seconds and computes an invariant which over-approximates the set of reachable states. In the table, one can see that this over-approximation gets tighter as more preprocessing steps are run and the apparent set of reachable states shrinks. In practice we find that we can shrink the over-approximation to an arbitrarily small size by running an appropriate number of preprocessing steps.

After the invariant is derived, we run interpolation with this invariant. In this configuration we have a hybrid solver that pulls from the strength of both invariants (implications) and interpolation.

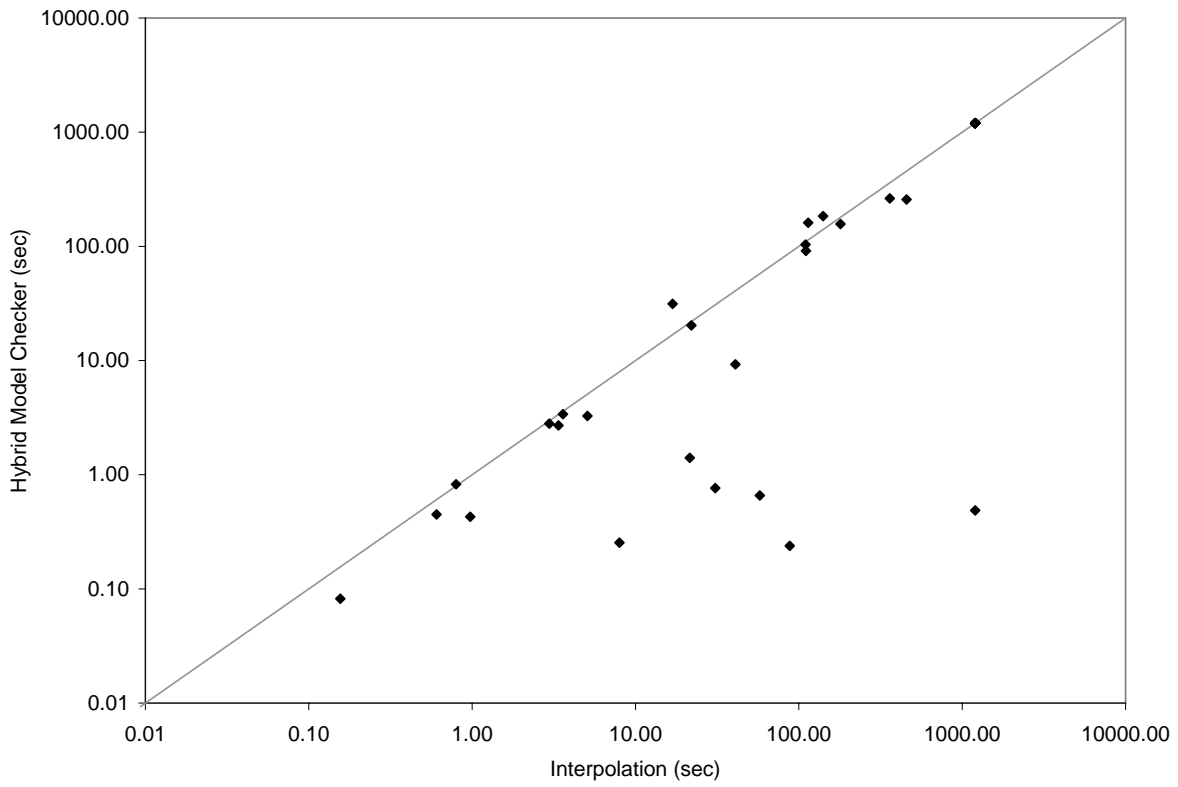


Fig. 2. Scatter plot of the runtimes for the sequential equivalence problems of Table I. Model checking was run with a timeout of 1200 seconds.

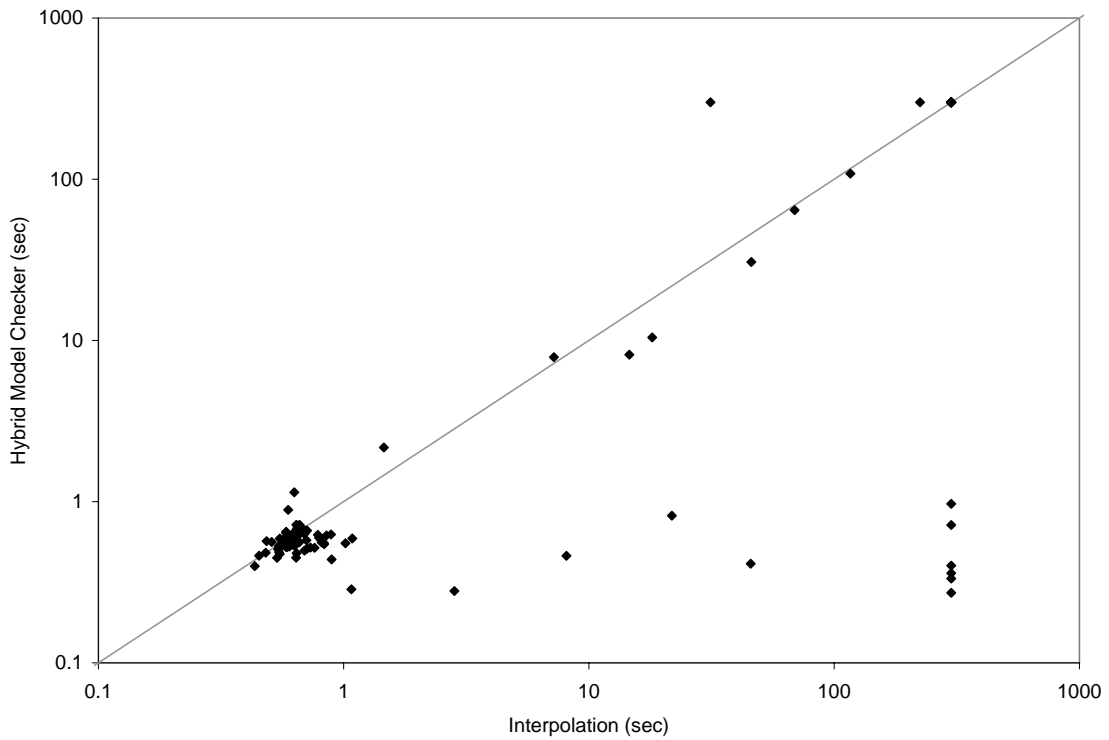


Fig. 3. Runtimes for a selection of safety property benchmarks. Shown are 74 passing properties, 15 failing properties, and 14 undecidable properties on designs ranging from 10 to 506 latches. Model checking was run with a timeout of 300 seconds.

If the interpolation proves the property, and if the proof of unsatisfiability of the bounded model checking problem does not depend on the initial state, then we know that from any state, no bad states are reachable. This means that the invariant reduced the search space so much that model checking was trivial. In fact, the invariant is telling us that every bad state is unreachable. This phenomena is documented in the “Verification Complete” column located after the preprocessor columns in the table.

We see from the table that the use of the invariant significantly speeds up the interpolation. The runtimes from Table I are represented in the scatter plot shown in Figure 2. It is important to note here that the hybrid solver runtimes do not include the 25 second overhead introduced by preprocessing. This is a constant-time overhead that will be insignificant when this method is applied to industrial benchmarks. This time was omitted from the table because the speed increase in the interpolation procedure is all that is important.

B. Safety Benchmarks

For our next experiment, we took 103 verification benchmarks from various checkers [15]. These benchmarks each have one safety property to be checked. We ran interpolation and the hybrid model checker on these designs, and the results are shown in Figure 3.

This empirical evidence shows that our hybrid technique dramatically speeds up interpolation.

V. FUTURE WORK

We are very encouraged by our experimental results, and we feel that this method has tremendous potential. However, the data does reveal a few flaws in our methods.

The first problem is that the implication procedure used as a preprocessor is not very stable. The dashes in the preprocessor section of Table I denote cases where the code crashed. Something about the sequential equivalence benchmarks is difficult for the implication procedure, and this problem requires further study.

The second problem is much more profound. The invariant is not represented in a natural way. Invariants help to prune the SAT search space, but we must use additional clauses to represent the invariants. This means that the cost of representing an invariant can outweigh its benefit. We see evidence of this in two places: in the preprocessor over-approximation, and in the runtime of the hybrid solver. Note that occasionally, a preprocessor step produces an over-approximation that is worse than the proceeding step. Each preprocessor step uses the current invariant, and if this invariant is excessively large then the SAT formulation becomes complicated. In a time-constrained environment, the preprocessor finds fewer implications and forms a larger over-approximation. The second place we see this invariant problem is in the runtime of the hybrid solver. Note that there are instances where the hybrid solver is slower than the normal interpolation routine. This should never happen and can be credited to the complexity of the invariant.

Our future work will be to develop a natural way to represent invariants in ABC [12]. This should encourage node sharing between the invariant and the design, and it should minimize the number of clauses allocated only for the invariant. Another way to approach this problem is to quantify the benefit of the invariant and to only use a useful subset in the SAT formulation.

VI. CONCLUSION

We presented a hybrid checker that strengthens the interpolation method by pruning the state search space. Our experimental results show a nearly consistent improvement to the original interpolation method. Because interpolation is the most robust model checking strategy, by extension the hybrid checker presented here is better than all other SAT-based methods.

REFERENCES

- [1] J.R. Burch and E.M. Clarke and D.E. Long and K.L. McMillan and D.L.Dill, “Symbolic model checking for sequential circuit verification,” in *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, April 1994.
- [2] C. Hyunwoo and E. Macii and B. Plessier and F. Somenzi and G. Hachtel, “Algorithms for approximate FSM traversal based on state space decomposition,” in *DAC*, 1996.
- [3] A. Mehrotra and S. Qadeer and V. Singhjal and R. Brayton and A. Aziz and A. Sangiovanni-Vincentelli, “Sequential optimisation without state space exploration,” in *IEEE Intl. Conf. Computer-Aided Design*, Nov. 1997.
- [4] J. Jang and I. Moon and G. Hachtel and F. Somenzi and J. Yuan and C. Pixley, “Iterative Abstraction-based CTL Model Checking,” in *DAC*, 1998.
- [5] K. Ravi and K. McMillan and T. Shiple and F. Somenzi, “Approximation and Decomposition of Binary Decision Diagrams,” in *DAC*, 1998.
- [6] I.H. Moon and J.Y. Jang and G.D. Hachtel and F.Somenzi and J. Yuan and C. Pixley, “Approximate Reachability Don’t Care for CTL model checking,” in *ICCAD, Digest of Technical Papers*, Nov. 1998.
- [7] S. Govindaraju and D. Dill, “Verification by Approximate Forward and Backward Reachability,” in *ICCAD*, 1998.
- [8] C. van Eijk, “Sequential equivalence checking based on structural similarities,” in *IEEE Trans. Computer-Aided Design*, July 2000.
- [9] P. Bjesse and K. Claessen, “SAT-based Verification without State Space Traversal,” in *FMCAD*, 2000.
- [10] J. Jang and I. Moon and G. Hachtel, “Iterative Abstraction-based CTL Model Checking,” in *DATE*, 2000.
- [11] D. Wang and P. Ho and J. Long and J. Kukula and Y. Zhu and T. Ma and R. Damiano, “Formal property Verification by Abstraction Refinement with Formal, Simulation and Hybrid Engines” in *DAC*, 2001.
- [12] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, <http://www.eecs.berkeley.edu/alanmi/abc/>
- [13] M. Iyer, G. Parthasarathy, and K.T. Cheng, SATORI- an efficient sequential SAT solver for circuits, in *ICCAD*, 2003
- [14] K. McMillan, “Interpolation and SAT-based Model Checking”, in *CAV*, July 2003.
- [15] Collection of Benchmarks, <http://www.cs.chalmers.se/een/Tip/>
- [16] G. Cabodia and S. Nocco and S. Querz, “Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals,” in *DATE*, 2003.
- [17] C. Wang and G. Hachtel and F. Somenzi, “The Compositional Far Side of Image Computation,” in *ICCAD*, 2003.
- [18] K. McMillan, “Don’t-care Computation using k-clause Approximation,” in *IWLS*, 2005.
- [19] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. McMillan, “An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment”, in *CHARME*, October 2005.
- [20] M. L. Case and R. K. Brayton and A. Mishchenko, “Inductively Finding a Reachable State Space Over-Approximation,” to appear in *IWLS*, 2006.