Iterative Abstraction using SAT-based BMC with Proof Analysis

Aarti Gupta, Malay Ganai, Zijiang Yang, Pranav Ashar

NEC Laboratories America
Princeton, NJ 08540, U.S.A.
{agupta, malay, jyang, ashar}@nec-labs.com

Abstract

Resolution-based proof analysis techniques have been proposed recently to identify a sufficient set of reasons for unsatisfiability derived by a CNF-based SAT solver. We have adapted these techniques to work with a hybrid SAT solver. We use the proof analysis technique with SAT-based BMC, in order to generate useful abstract models. Our abstraction procedure is used iteratively in a top-down framework, starting from the concrete design, where we apply BMC on increasingly more abstract models. We apply various SAT-based and BDD-based verification methods on these abstract models, in order to obtain proofs of correctness, or to perform deeper searches for counterexamples. We demonstrate the effectiveness of our prototype implementation on several large industry designs.

1. Introduction

Symbolic model checking techniques [1, 2], based on the use of Binary Decision Diagrams (BDDs) [3], offer the potential of exhaustive coverage and the ability to detect subtle bugs. However, these techniques do not scale very well in practice due to the state explosion problem. A recent alternative is Bounded Model Checking (BMC) [4], which focuses on the search for counterexamples of bounded depth. Effectively, the problem is translated to a propositional formula, such that the formula is satisfiable if and only if there exists a counterexample of depth *k*. In practice, the depth *k* can be increased incrementally to find the shortest counterexample. However, additional reasoning is needed to ensure completeness of the proof of correctness, when no counterexample can be found [4, 5].

The satisfiability check in the BMC method is typically performed by a backend SAT solver. Due to the many advances in SAT solving techniques [6-9], BMC can handle much larger designs than BDD-based methods. A related important development has been the use of resolution-based proof analysis techniques to check the unsatisfiability result of a SAT solver [10, 11]. As part of the check, these techniques also identify a set of clauses from the original problem, called the *unsatisfiable core*, such that the clauses are sufficient for implying the unsatisfiability. Similar SAT-based proof analysis techniques have also been proposed independently in the context of refinement, and abstraction-based verification methods [12, 13]. The existing resolution-based proof analysis techniques have been described for SAT solvers that use a CNF (Conjunctive Normal Form) representation of the Boolean problem. We have adapted these techniques to work with a circuit SAT solver [14], or a hybrid SAT solver [9].

We use the resolution-based proof analysis technique in the SAT solver used for checking BMC problems – we call this *SAT-based BMC with Proof Analysis*. Note that unsatisfiable SAT instances in BMC correspond to the absence of a counterexample of (or up to) a given depth. For each such depth, we identify an unsatisfiable core, and use it to generate an abstract model. In particular, we propose a latch-based abstraction, such that the resulting abstract models are guaranteed to not have a counterexample of (or up to) that depth.

Our overall verification methodology centers around a top-down *iterative abstraction* framework. Starting from the concrete design, we apply SAT-based BMC with Proof Analysis on a *seed model* in each iteration. Our abstraction, based on identification of unsatisfiable cores, is used to choose a seed model for the next iteration. Under certain practical conditions, we allow a refinement step, which can potentially increase the size of the seed model. In each iteration, we also generate existentially abstract models, again based on the unsatisfiable cores. These models are known to be conservative for LTL properties [1, 15]. We use various BDD-based and SAT-based methods for performing unbounded model checking on these models. A proof of correctness on any of these models guarantees correctness on the concrete design, while a counterexample may require a refinement, or going back to a previous iteration in our iterative flow. In practice, we iterate the loop until convergence of the seed model, or until a conclusive result is obtained on some abstract model.

The key contribution of our work is that the overall flow is targeted at reducing the size of the seed models across successive iterations. The potential benefit is that for properties that are false, BMC search for deeper counterexamples is performed on successively smaller models, thereby increasing the likelihood of finding them. For properties that are true, the successive iterations help to reduce the size of the abstract models, thereby increasing the likelihood of completing the proof by unbounded verification methods.

We have implemented these ideas in a prototype verification framework called *DiVer* [16], which includes other BDD-based and SAT-based verification methods. We report on our experience on some large industry designs. For some of these, we have been able to complete proofs of correctness for the first time ever. In most of these, we have been able to perform deeper searches for counterexamples. We have also observed that our iterative abstraction typically gives an order of magnitude reduction in the final model sizes. For many examples, this reduction was crucial in enabling the successful application of the unbounded verification methods.

2. Related Work

Our work is broadly related to the many efforts in verification that use abstraction and refinement [12, 17-20]. Most of these efforts are bottom-up approaches, where starting from a small abstract model of the concrete design, counterexamples found on these models are used to refine them iteratively. In practice, many iterations are needed before converging on a model where the proof succeeds. More frequently, the size of the refined abstract model grows monotonically larger, on which unbounded verification methods fail to complete.

The main reason for the popularity of the bottom-up approaches has been a lack of techniques that could extract relevant information from a relatively large concrete design. This is changing now with the use of proof analysis techniques for SAT solvers and other theorem-provers [13, 21], and use of interpolants that approximate reachable state sets [22]. We too have employed SAT proof analysis techniques effectively, to obtain abstract models in a top-down iterative abstraction framework based on use of BMC in the inner loop. In our approach, a lack of a counterexample provides an opportunity to perform further abstraction. On the other hand, the presence of a counterexample does not necessarily require a refinement based on that particular counterexample.

In comparison to bottom-up approaches, our top-down approach may need to handle much larger models. However, note that we do not require complete verification on these models for the purpose of abstraction. Instead, our abstraction method is based on SAT-based BMC, which we use up to some finite depth to check for counterexamples anyway. Furthermore, in practice, the first iteration of our iterative abstraction framework provides a significant reduction in model sizes, such that all successive iterations need to work on smaller models. Our broader goal is to systematically exploit proof analysis with SAT-based BMC wherever possible. Since it is unlikely that a purely top-down or a purely bottom-up approach will work best in practice, we are currently exploring combinations of the two.

The work by McMillan and Amla [12] is most closely related to ours. Though our work was done independently, there are some similarities. They also use SAT-based BMC, with extraction of unsatisfiable cores as the basis for performing automatic abstraction. Therefore, their abstract model also has the useful property of ruling out counterexamples below a certain depth. However, the specific abstraction that they use is slightly different from ours, in that it is not latch-based, but is based on individual gates. The biggest difference, however, is in the application setting. They do not use BMC on successively smaller abstract models within an iterative framework. This forces them to perform deeper searches with BMC on the concrete design itself, whenever a counterexample is found on an abstract model. This is avoided in our approach, by performing a deeper search on a less abstract model where possible. A related issue is that they do not use counterexample-driven refinement (other than noting its depth). In contrast, we do use refinement, though sparingly. Finally, our iterative abstraction method provides significant model size reductions in the final iteration. This frequently enables the application of unbounded verification methods, which often fail to complete on abstract models generated in the initial iteration (directly from the concrete design).

3. Proof Analysis for a Hybrid SAT Solver

There have been several independent efforts aimed at extracting proofs of unsatisfiability from a CNF-based SAT solver [10-13]. They are based on recording additional information during conflict analysis and conflict-driven learning [6], which form important components of modern SAT solvers [7-9]. We start by reviewing the basics.

3.1 Basics: Identification of An Unsatisfiable Core

Conflict analysis takes place whenever a conflict is discovered by a SAT solver during BCP (Boolean constraint propagation). The implications are typically stored in the form of an implication graph. In this graph, a node denotes a variable with a value (a literal), and an edge into a node denotes the clause, called the antecedent, which caused the implication on the node's variable. Nodes with no incoming edges denote decision variables. A conflict is obtained whenever the graph has two nodes for the same variable with opposite values. A small example of an implication graph leading up to a conflict is shown in Figure 1.

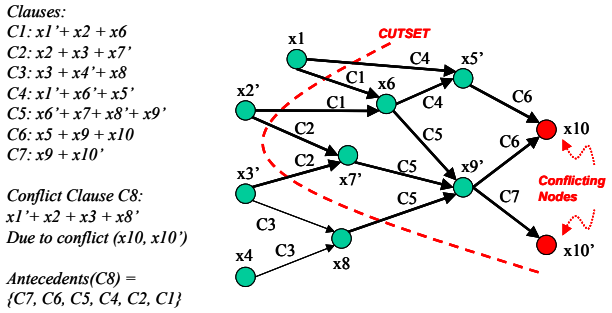

Figure 1: Example for Conflict Analysis

Conflict analysis takes place by following back the edges leading to the conflicting nodes up to any cutset in this graph. A conflict clause can then be derived from the variables feeding into the chosen cutset. A key feature of a learned conflict clause is that it is also the result of resolution on all the antecedent clauses, which are traversed up to the chosen cutset. For proof analysis, these antecedents (reasons) are recorded and associated with the learned conflict clause, as also shown in Figure 1.

When a SAT solver determines that a given problem is unsatisfiable, it does so because there is a conflict without any decisions being taken. A conflict analysis can again be used to record the antecedents for this final conflict. This time, the learned conflict clause, i.e. a resolution of all its antecedents, is an empty clause. Therefore, this final resolution tree constitutes a proof of unsatifiability [23], except that it may include some learned clauses. Recall that a learned clause is itself a resolution of the antecedents associated with it. Therefore, by recursively substituting the resolution trees corresponding to the learned clauses into the final resolution tree, a resolution proof only on the original clauses can be obtained. These original clauses constitute an *unsatisfiable core,* i.e. they are sufficient for implying the unsatisfiability. In practice, the resolution tree is created only if a check is needed for the unsatisfiability result [10, 11]. For the purpose of identifying an unsatisfiable core, a marking procedure is used, which starts from the antecedents of the final conflict graph, and recursively marks the antecedents

associated with any marked conflict clause. At the end, the set of marked original clauses constitutes an unsatisfiable core.

3.2 Adaptation to Hybrid SAT Solver

We have extended the proof analysis technique to work with a hybrid SAT solver. A hybrid SAT solver uses hybrid representations of Boolean constraints, e.g. where a circuit netlist is used to represent the original circuit problem, and CNF is used to represent the learned constraints [9]. Conflict analysis in a hybrid SAT solver also traverses back from the conflicting nodes in an implication graph. However, edges in such a graph may correspond to hybrid representations of constraints. For example, while performing BCP directly on a circuit netlist, edges might correspond to nodes in the circuit. We record the reasons for the conflict, in their hybrid representations, and associate them with the learned constraint (corresponding to the conflict clause). When the final conflict is found, indicating the unsatisfiability, a marking procedure is started from its antecedents. Again, reasons for any learned constraints are marked recursively. At the end of this procedure, the marked constraints from the original problem constitute an unsatisfiable core. For example, given an unsatisfiable circuit problem due to external constraints, this procedure identifies a set of nodes in the circuit that are sufficient for implying the unsatisfiability.

3.3 Reducing the Size of the Unsatisfiable Core

By iterating SAT checking on the unsatisfiable core, the number of original clauses/variables needed for unsatisfiability can be reduced significantly [10]. Also, the constraints constituting the unsatisfiable core are related to the particular conflict clauses learned by a SAT solver. These, in turn, depend upon other heuristics in the SAT solver, e.g. decision heuristics, heuristics for choosing a cutset during conflict analysis [24] etc. We vary these heuristics in different runs of the SAT solver, in order to obtain a potentially smaller unsatisfiable core.

4. Generation of Abstract Models

We use a proof analysis technique in the SAT solver used to check BMC problems. For each depth checked, the lack of a counterexample corresponds to an unsatisfiable SAT instance. We identify an unsatisfiable core in each such instance.

4.1 SAT-based BMC with Proof Analysis

For ease of exposition, consider that we use SAT-based BMC to search for counterexamples of increasing depth k, $1 \leq k \leq k_{max}$. If there is no counterexample of depth k, i.e. the check for depth k is unsatisfiable, we obtain the unsatisfiable core, and denote it *R(k)* (Reasons for depth k). Note that this corresponds to a set of clauses for a CNF-based SAT solver, or a set of circuit nodes and external constraints for a hybrid SAT solver using a circuit-netlist representation for the given design. Our techniques described in the rest of this paper do not depend upon which representation is actually used.

4.2 Sufficient Abstract Models

Given an unsatisfiable core *R(k)*, which was identified by BMC with Proof Analysis at depth k, we generate an abstract model which preserves the unsatisfiability implied by constraints in *R(k)*. At the same time, we want to keep the size of the abstract model small. Rather than optimize at the level of each gate in the original design, our abstraction tries to minimize the number of latches to include in the abstract model, while still retaining the useful property that there is no counterexample of depth k.

4.2.1 Latch Interface Abstraction

We use the following notation to describe our abstraction. For a node v in the unrolled design, let *F(v)* denote the node corresponding to it in the transition relation[1] of the design. For a given node e in the unrolled design, let *Ext(e)* denote the (possibly empty) set of external constraints imposed on node e. For a given latch L in the transition relation of the design, let *IF(k,L)* denote the set of its *latch interface constraints* in the unrolled design up to depth k. The set *IF(k,L)* consists of constraints corresponding to equality of the latch output at time frame i, with the latch input at time frame $i-1$, for $1 \leq i \leq k$. Note that this set includes the initial state constraint. Note also that any k-depth unrolling of the design would necessarily include these constraints for each latch, either explicitly, or implicitly, in the problem representation. We propose a *latch interface abstraction*, which works as follows:

Step 1: Given *R(k)*, we first mark a node v in the unrolled design, if variable v appears in some constraint in *R(k)*. For each such v, we say that node v is *marked*.

Step 2: For each marked node e, such that some constraint in *Ext(e)* belongs to *R(k)*, we perform a backward DFS traversal starting from e, through only marked nodes, using the procedure shown in Figure 2. Note that the recursive traversal on marked nodes is terminated at any unmarked node, but is otherwise continued through the fanin nodes. Any marked node which is visited during such a traversal is called *visited*.

```
rec_dfs_through_marked_nodes(node n) {
  if (visited(n)) return;
  if (!marked(n)) return;
  visited(n)=1;
  for each m in fanin(n) {
    rec_dfs_through_marked(m);
  }}
```

Figure 2: Recursive DFS through Marked Nodes

Step 3: For each latch L, we say that L is *visited* if any of the nodes denoting its output at time frame i, $0 \leq i \leq k$, is visited.

Step 4: We extract the combinational fanin cones of all latches that are visited. We also extract combinational fanin cones of all nodes *F(e)* such that *Ext(e)* is not empty. These fanin cones represent the transition relation of our abstract model. In particular, all latches that are not visited are abstracted away as pseudo-primary inputs.

The resulting abstract model is called a *sufficient model* for depth k, denoted *SM(k)*. Since it is generated by abstracting away some latches as pseudo-primary inputs, it is known to be conservative for LTL properties [1, 15], i.e. truth of a property

[1] We use the term *transition relation* to denote the entire combinational logic of the design, including next-state logic for the latches, as well as output logic for the external constraints, either due to the property, or enforced by the designers.

on the abstract model guarantees its truth on the given design. It has an additional useful property, stated in the following theorem.

Theorem 1: The sufficient model SM(k) generated using the latch interface abstraction does not have any counterexample of depth k.

Proof: Recall that constraints in *R(k)* are sufficient to generate implications (without taking any decisions), which lead to a conflict at some node in the unrolled design, corresponding to the final conflict graph. The latch interface abstraction uses circuit connectivity information to prune away those constraints in *R(k)* which are not needed to obtain the final conflict. Consider a marked node that does not have a transitive fanout path, through other marked nodes, to a marked node with an external constraint in *R(k)*. We claim that such a node is not needed to obtain the final conflict. This is because implications only on inputs of a circuit node cannot cause a final conflict on the output of that node. The same reasoning can be used to show that there can be no final conflict on any of its marked but unvisited transitive fanouts. Therefore, at the end of Step 2, the set of visited nodes and their associated constraints are guaranteed to lead to a final conflict.

Due to the structure of the unrolled design, all transitive paths connecting visited nodes in different time frames have to go through latch interfaces between those time frames. Therefore, each visited node is contained in the combinational fanin cone of some visited latch output node, or some visited node on which an external constraint is in *R(k)*. The abstract model *SM(k)* includes all such combinational fanin cones in its transition relation (Step 4). Therefore, all visited nodes and their constraints are included in a *k*-depth unrolling of the abstract model. Therefore, we are guaranteed to get a final conflict without any decisions, thereby proving that the abstract model *SM(k)* cannot have any counterexample of depth *k*. □

We also use an alternative abstraction, where we skip the recursive DFS traversal in Step 2 altogether. We consider a latch *L* to be marked, if any of its output nodes in any time frame is marked (not visited), i.e. if any constraint in *IF(k, L)* belongs to *R(k)*. In this case, the abstract model consists of the combinational fanin cones of all marked latches, and all external constraint nodes. The reasoning in our proof works also for this cheaper (to compute) abstraction. Indeed, the proof of sufficiency works, while pruning *R(k)*, for any subset *S* of latches, such that $\{L|\ visited(L)\} \subseteq S \subseteq \{L|\ marked(L)\}$. In the remainder of this paper, we denote the abstract model corresponding to any such set *S* as *SM(k),* since it is guaranteed to not have a counterexample of depth *k*.

4.2.2 Pruning due to Latch Interface Abstraction

The unsatisfiable core *R(k)* can include a node in the unrolled design, on which constraints are not needed to generate the final conflict. The pruning obtained by the latch interface abstraction in Step 2 is geared at throwing away these nodes, without losing sufficiency for the unsatisfiability.

A small example of how this can happen is shown in Figure 3. Part (a) of this figure shows the implication graph at the time of learning the conflict clause *C1: (a'+b)*, and the associated antecedents. In Part (b), we show a final conflict graph, where implications from an external constraint on node *e* imply variable *a* to *1*, which leads to use of the conflict clause to imply *b* to *1*. This further leads to *c* and *d* being implied to *1*, with the implication on *d* leading to a final conflict on node *v*. Given this final conflict graph, the recursive marking procedure for *R(k)* starts by including antecedents for implications from the circuit clauses (from *e* to *a*, from *b* to *c*, … etc.). Furthermore, it substitutes the antecedents of the conflict clause, leading to *R(k)* as shown in the figure.

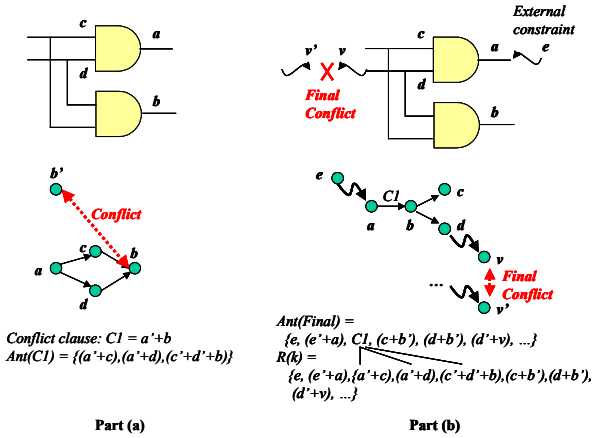

Figure 3: Example for Pruning the Set R(k)

Suppose there is no implication from any external constraint on the value of variable *b*, i.e. it does not matter what value *b* takes. The reason that constraints involving *b* even appear in *R(k)* is that the required values on *c* and *d*, needed for the final conflict, are used to imply a consistent value on their fanout variable *b*. As noted in our proof, implication values existing only on inputs of node *b* cannot lead to a conflict on *b*. Therefore, our abstraction prunes away the constraints associated with *b*. Note from the circuit shown at the top of Part (b), that when the gate corresponding to variable *b* is removed, the implication from *a* to *d* can still be used to obtain the final conflict. Indeed, if the SAT solver had used this implication directly, the conflict clause *C1* may never have been used at all. However, in general, we cannot rely on the SAT solver to use the implication from *a* to *d*, instead of the transitive implications from *a* to *b*, and *b* to *d*. In case *d* is far away from *a*, say through a chain of buffers, it may actually be faster for the SAT solver to use the learned clause *C1*, than not to use it (which is how conflict clauses help to improve SAT solver performance).

4.2.3 Other Applications

It is instructive to recall that the set *R(k)* is sufficient, but not necessary, for unsatisfiability. As described in the previous section, the latch interface abstraction prunes the given sufficient set *R(k)* further, to yield another sufficient set. We can use this pruned sufficient set for other applications as well. In particular, we can use it to identify refinement candidates for counterexample guided abstraction refinement [12]. Rather than choosing all latch nodes at the failure interface marked by *R(k)*, we can use the latch nodes in the pruned sufficient set as refinement candidates. This can potentially reduce the number of candidates.

Another application is in verification of safety properties. If the pruned set of constraints does not include any constraint due to initial state of a latch, then it represents an inductive invariant. Note that though initial state constraints are enforced in the BMC SAT problem at depth *k*, if the pruned set of sufficient constraints does not contain any, then this constitutes a proof of unsatisfiability when starting from an arbitrary initial state. This corresponds to an inductive step in a proof by induction with increasing depth [5, 16]. By separately checking the basis step, a complete proof by induction can be obtained for the safety property.

4.3 Accumulated Sufficient Abstract Models

When using BMC on increasing depth *k*, it is useful to identify the *accumulated* unsatisfiable core for depth *k*, denoted *AR(k)*. (It can also be identified from a single unsatisfiable BMC problem, which checks the existence of a counterexample of any depth *up to k*.)

We use an abstraction similar to that defined in the previous section, where the accumulated unsatisfiable core *AR(k)* is used in place of *R(k)*. The resulting model is called an *accumulated sufficient model* for depth *k*, denoted *ASM(k)*. Following a similar reasoning as in the proof of Theorem 1, it can be shown that the model *ASM(k)* does not have any counterexample of depth less than or equal to *k*.

4.4 Insufficient Abstract Models

The main purpose of generating abstract models is to enable use of complete verification methods, such as symbolic model checking [1, 2], or a proof by induction for safety properties [5, 16]. Typically, such methods do not work well on large models. Therefore, if the abstract models resulting from the entire unsatisfiable core are too large, we may not be able to apply these methods. This is typically the case for many industry designs, especially when *k* gets large (some data are provided in Section 6).

The latch interface abstraction already includes some pruning of the set *R(k)* (or *AR(k)*), which is guaranteed to retain the unsatisfiability at (or up to) depth *k*. It is also possible to arbitrarily pick any subset of visited latches required by the latch interface abstraction. The choice can be dictated by heuristic criteria such as – at what depth was its output node visited, at how many depths was its output node visited, etc. The abstract model derived by retaining some, but not all, of the visited latches is called an *insufficient model*. It is not guaranteed to exclude a counterexample of any length. However, it can potentially exclude many in practice. The important point is that it is still conservative for verification of LTL properties. In comparison to models derived from localization reduction [17], which is based on a static cone of influence analysis, an insufficient model based on proof analysis may better capture the needed invariant for all depths. We are currently exploring heuristics for obtaining useful insufficient abstract models.

5. Iterative Abstraction Framework

Our verification methodology is centered around a novel iterative abstraction framework, based on the use of BMC with Proof Analysis and the related abstractions in the inner loop. The overall flow is shown in Figure 4.

5.1 Inner Loop of the Framework

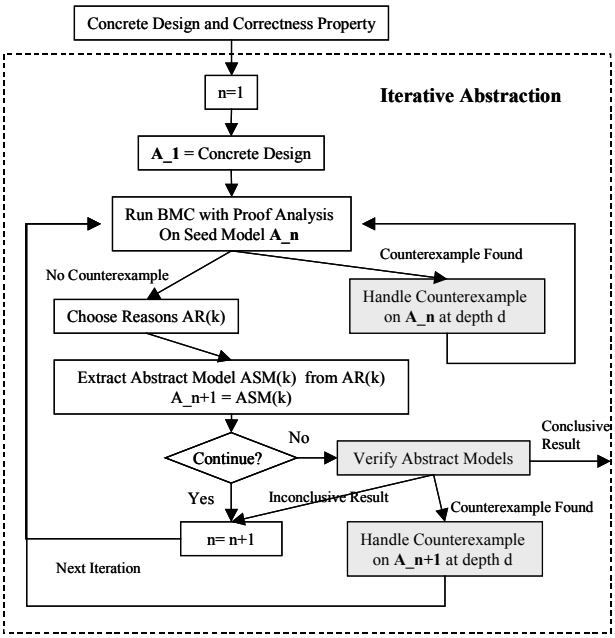

Figure 4: Iterative Abstraction Framework

Each iteration of our framework, indexed by *n* as shown in the figure, consists of applying SAT-based BMC with Proof Analysis on a given *seed* model A_n. The seed model for the initial iteration is the concrete design. In each iteration, we run BMC with Proof Analysis up to some fixed depth (potentially different for each iteration). The proof analysis technique is used to identify the unsatisfiable cores for each depth *k* when there is no counterexample. If a counterexample is found at some depth *d*, it is handled as described in the next section. The result of such handling is that we may obtain a new seed model A_n' potentially larger than A_n, and we repeat the current iteration.

On the other hand, if no counterexample is found by BMC, we heuristically choose one of the sets *AR(k)* at some depth *k*. For example, we can choose a set that remains unchanged for a certain number of time frames. Then we use any abstraction technique which is guaranteed to exclude all counterexamples of depth less than or equal to *k*, in order to generate the corresponding accumulated sufficient model *ASM(k)*. Specifically, we can use any version of the latch interface abstraction proposed in this paper, or the abstraction proposed by others [13]. The *ASM(k)* model is used as the seed model A_{n+1} for the next iteration. If we don't wish to continue, e.g. if the seed model A_{n+1} is small enough, or if model A_{n+1} is unchanged from model A_n, we attempt to verify the abstract models generated in this iteration. This is described in more detail in Section 5.4. The result of such verification is that we can get a counterexample (handled as described next), or a conclusive result (we can stop), or an inconclusive result. In the last case, we try to reduce the size of the seed model by performing another iteration.

5.2 Handling Counterexamples

Our scheme for handling counterexamples is shown in Figure 5. Given a counterexample on a model A_n at depth *d*, we first check

if model A_n is the concrete design. If it is, then we have found a true counterexample. However, for $n>1$, the counterexample could be spurious, since it was obtained not on the concrete design, but on an abstract model.

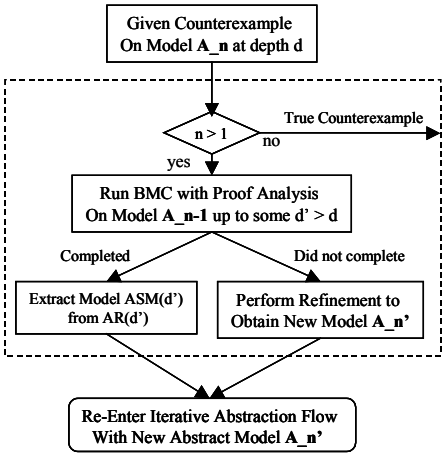

Figure 5: Handling Counterexamples

In case the counterexample is spurious, we run a deeper search on seed model A_{n-1} from the previous iteration, in order to choose *AR(d')*, at some depth $d' \geq d$. Note that, in practice, it may not always be possible to run BMC deeper than *d*. If we can, we extract the corresponding model *ASM(d')*. Otherwise, we perform a resolution-based refinement to remove the counterexample in case it is spurious [12]. For refinement, we use the pruned set of sufficient constraints, rather than the entire set *R(k)*, to identify suitable refinement candidates (as described in Section 4.2.3). Finally, we re-enter the iterative abstraction flow with the new model A_n'. This model is guaranteed to exclude at least the given counterexample, but is potentially larger than the model A_n.

5.3 Potential Benefits of Iterative Abstraction

Due to the sufficiency property of the abstractions, the seed model in each iteration can have counterexamples only at depths strictly greater than the depth *d*, from which it was generated in the previous iteration. Furthermore, it is no bigger than the seed model of the previous iteration, provided there are no refinements (which we use only when we cannot do a deeper search with BMC). The combined effect is that for properties that are false, BMC search for deeper counterexamples is performed on successively smaller models, thereby increasing the likelihood of finding them. For properties that are true, the successive iterations help to reduce the size of the abstract models, thereby increasing the likelihood of completing the proof by unbounded verification methods.

5.4 Verification of Abstract Models

We apply various complete, as well as incomplete, verification methods on the abstract models generated in any iteration of the iterative abstraction flow. We prefer to perform verification on the seed models (*ASM(d)* models), in order to derive benefits of iterative abstraction. However, in practice, if these models are too large, we use either the *SM(k)* models, or the insufficient abstract models, derived from any depth *k* checked by BMC.

We briefly mention the techniques we use, and highlight the benefit of using them on smaller abstract models.

- We use symbolic model checking techniques [1, 2] on the abstract model. If the correctness property is proved true, it is guaranteed to be true on the concrete design as well. Due to the limited capacity of such methods, they are more likely to work on smaller abstract models.
- We also use symbolic traversal techniques to perform a reachability analysis on the abstract model [1, 2, 25]. The computed reachable set corresponds to an over-approximate reachable set for the concrete design. These are used as additional reachability constraints for the BMC methods described below.
- We use BMC (optionally with reachability constraints [16]) to search for a counterexample [4]. If there is no counterexample up to depth *d*, then it is guaranteed that there is no counterexample up to depth *d* in the concrete design either. In many examples, BMC can complete deeper searches on the smaller abstract models than on the larger concrete design.
- For safety properties, we use BMC (optionally with reachability constraints [16]) to perform a proof by induction with increasing depth [5]. In many cases, a smaller model enables a proof of increased depth, which may succeed.

6. Experimental Results

We have implemented the iterative abstraction framework in a prototype verification platform called *DiVer* [16]. The BMC engine in *DiVer* has been successfully used for verification of many large industry designs. For our experiments here, we chose the most difficult of these designs, ranging in size up to 416k gates and 12k flip-flops in the static cone of influence. We verified safety properties, i.e. the BMC search was for simple counterexamples without loops. All experiments were performed on a 2.2 GHz Dual Xeon processor machine, with 4 GB memory, running Linux 7.2.

6.1 Results for Iterative Abstraction

The results for use of iterative abstraction are summarized in Table 1. The size of the concrete design is listed in Column 2 in terms of number of flip-flops (#FF), and number of gates (#Gates). The results for the different iterations are shown in the remaining columns, where for each iteration, we report the size of the abstract model (#FF, number of flip-flops), the depth at which it was derived (k), and the total CPU time taken by BMC with Proof Analysis to check up to that depth (T(s), in seconds).

Typically, we used a 3-hour time limit for each iteration. Within each iteration, we used either the last depth completed by BMC, or a simple heuristic to pick the depth, from which to generate the seed model for the next iteration. The heuristic we used was to look for a stable set *AR(k)* which did not change for 10 time frames. For these experiments, we used the cheaper latch interface abstraction, which skips the recursive DFS traversal. Also, we iterated over the inner loop till the size of the seed model converged.

Note that the first iteration was quite successful in generating small abstract models. For most designs, we obtained a magnitude of order reduction, in comparison to the size of the

	Concrete Design	Abstract Models Generated by Iterative Abstraction																				
		Iteration 1			Iteration 2			Iteration 3			Iteration 4			Iteration 5			Iteration 6			Iteration 7		
	#FF / #Gates	#FF	k	T(s)	#FF	k	T(s)	#FF	k	T(s)	#FF	k	T(s)	#FF	k	T(s)	#FF	k	T(s)	#FF	k	T(s)
D1	12.7k / 416.1k	1269	63	32815	541	63	486	439	63	140	259	63	75	212	63	34	118	61	20	113	60	5
D2	4.2k / 37.8k	523	47	10043	451	56	2540	445	41	4163	444	43	2293									
D3	5.2k / 46.4k	1530	30	10515	1468	28	2763	1434	29	4354	1406	27	6812	1356	28	8650						
D4	910 / 18k	476	80	6274	420	78	3250	405	80	8765	397	85	9588	396	78	2660						
D5	4.2k / 37.8k	330	43	882	303	34	27															
D6	3.6k / 155k	105	15	1786	103	15	7															

Table 1: Results for Iterative Abstraction

concrete design. Typically, the first iteration was also the most expensive in CPU time. Next, note that for some designs, we can clearly see a reduction across the iterations also. In particular, for the design D1, iterative abstraction allowed the size of the abstract model to be reduced from 1269 (Iteration 1) to 113 (Iteration 7) flip-flops. Though other designs did not exhibit the same level of reduction, we did manage to reduce their sizes as well.

This effect is related to that observed by others, i.e. when proof analysis techniques are applied iteratively, the final unsatisfiable core can be much smaller than the original problem [10]. However, note that we are not applying the iterative technique to the unsatisfiable core at each depth. We are choosing the core at a particular depth, and applying proof analysis on the resulting abstract model at all depths in the next iteration. We are currently experimenting with combining the two, to see if we can reduce the sizes of the abstract models even further.

6.2 Results for Verification of Abstract Models

After we performed iterative abstraction, we tried verification of the generated abstract models. These results are summarized in Table 2, and discussed in more detail in this section.

For each design, we ran basic BMC with a 3-hour time limit, and these results are shown in the first set of columns. We report the size of the concrete design (#FF / #Gates), the maximum depth for which BMC search was completed (Depth), and the total CPU time taken for searching all depths up to the maximum (T(s), in seconds). Note that our basic BMC engine is able to search fairly deep even for large designs. However, we were unable to find a counterexample for any of these designs.

The next set of columns shows results for basic BMC on an abstract seed model generated during iterative abstraction. Again, we report the size of the abstract model (#FF / #Gates), the maximum depth searched by BMC (Depth), and the total CPU time taken to search up to that depth (T(s), in seconds). For all designs except D3, we were able to search deeper on the abstract models than on the concrete designs. For some, there was an increase by an order of magnitude in the maximum depth searched. This is due to an improvement in the SAT checking time on smaller problems, and the ability to unroll the model deeper with bounded memory resources. Since no counterexample were still found, these results constituted at least an increased level of confidence in the correctness.

The last set of columns report the results for complete verification of the abstract models. We report the status of the verification (Status), i.e. whether or not we were able to prove the property correct. Note that we were able to prove the correctness of 3 of the 6 designs. For successful instances, we also report the time taken (T(s), in seconds) and the verification method used.

For design D1, we were able to prove the property correct on the abstract model in 40 seconds, by using standard BDD-based symbolic model checking. In fact, we had not been able to prove this design correct using any other techniques so far. For design D5, we were given external constraints by the designers, which needed to be enforced at every cycle. However, these constraints were not enough to help a proof by induction. Therefore, we performed a BDD-based reachability analysis on a much smaller abstract model derived from depth 4, with 40 flip-flops, which took 1 second. The computed reachable state set was used as a reachability invariant by the BMC engine [18], to successfully perform a proof by induction on the concrete design, in less than 1 second. Similarly, for design D6 also, we performed a reachability analysis on the shown model with 103 flip-flops, taking 2737 seconds. Again, with the BDD-based reachability invariant, our BMC engine was able to successfully prove the property on the concrete design in less than 1 second. So far, we

	Concrete Design Basic BMC Performance			Abstract Model Basic BMC Performance			Proof of Property Using Abstract Model		
	#FF / #Gates	Depth	T (s)	#FF / #Gates	Depth	T (s)	Status	T (s)	Verification Method
D1	12.7k / 416.1k	96	10230	113 / 1.5k	1012	10788	Yes	40	BDD-based model checking
D2	4.2k / 37.8k	64	7519	451 / 14.5k	115	7129	No		
D3	5.2k / 46.4k	32	8667	1356 / 20.8k	30	7513	No		
D4	910 / 18k	89	9760	396 / 6k	96	10134	No		
D5	4.2k / 37.8k	82	3968	303 / 12.4k	211	10603	Yes	2 *	BDD Constraints for BMC-based Induction
D6	3.6k / 155k	307	3099 **	103 / 17.3k	3034	2635 **	Yes	2738	BDD Constraints for BMC-based Induction

Table 2: Results for Verification of Concrete and Abstract Models

(Notes *: BDD analysis performed on a different abstract model, with 40 FF, derived from depth 4.
**: mem-out within 3-hour time limit.)

have not been able to conclusively verify the remaining designs D2, D3, and D4. We are currently experimenting with abstract models generated from lower depths, and hope to obtain further improvement.

We are also investigating the comparison of our results against use of counterexample-driven refinement methods [12]. It would be interesting to compare the sizes of the abstract models, which allow complete proofs for the successful instances among these designs, and the total time and number of iterations required. Though our overall iterative abstraction framework allows refinement to be used, so far we have focused more on the top-down abstraction. We also plan to examine more flexible combinations of the two approaches.

7. Conclusions

We have presented a novel approach for utilizing SAT-based proof analysis in an abstraction-refinement flow using BMC. Our contributions are in the specific abstraction we generate from the proof analysis, and the manner in which we iterate through successive abstractions, leading to successively smaller abstract models. The small abstract models enable deeper BMC search for counterexamples. They also enable a conclusive proof of the correctness property using BDD-based model checking, or SAT-based inductive methods. We are able to show significant model reductions on industry designs, including a 100x reduction in the number of flip-flops leading to a conclusive proof for a large production design.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*: MIT Press, 1999.

[2] K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*: Kluwer Academic Publishers, 1993.

[3] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35(8), pp. 677-691, 1986.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Proceedings of Workshop on Tools and Algorithms for Analysis and Construction of Systems (TACAS)*, vol. 1579, *LNCS*, 1999.

[5] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties using Induction and a SAT Solver," in *Proceedings of Conference on Formal Methods in Computer-Aided Design*, 2000.

[6] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, pp. 506-521, 1999.

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of Design Automation Conference*, 2001.

[8] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," in *Proceedings of Conference on Design Automation & Test Europe (DATE)*, 2002, pp. 142-149.

[9] M. Ganai, L. Zhang, P. Ashar, and A. Gupta, "Combining Strengths of Circuit-based and CNF-based Algorithms for a High Performance SAT Solver," in *Proceedings of the Design Automation Conference*, 2002.

[10] L. Zhang and S. Malik, "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications," in *Proceedings of Conference on Design Automation & Test Europe (DATE)*, 2003.

[11] E. Goldberg and Y. Novikov, "Verification of Proofs of Unsatisfiability for CNF Formulas," in *Proceedings of Conference on Design Automation & Test Europe (DATE)*, 2003.

[12] P. Chauhan, E. M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis," in *Proceedings of Conference on Formal Methods in CAD (FMCAD)*, 2002.

[13] K. L. McMillan and N. Amla, "Automatic Abstraction Without Counterexamples," in *Proceedings of Tools for Algorithms for Construction and Analysis of Systems (TACAS)*, 2003.

[14] A. Kuehlmann, M. Ganai, and V. Paruthi, "Circuit-based Boolean Reasoning," in *Proceedings of Design Automation Conference*, 2001.

[15] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," in *Proceedings of Conference on Principles of Programming Languages*, 1992.

[16] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, "Abstraction and BDDs Complement SAT-based BMC in DiVer," in *Proceedings of International Conference on Computer Aided Verification*, vol. 2725, *LNCS*, 2003.

[17] R. P. Kurshan, *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*: Princeton University Press, 1994.

[18] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of Conference on Computer Aided Verification*, vol. 1855, *LNCS*, 2000, pp. 154-169.

[19] D. Wang, P.-H. Ho, J. Long, J. Kukula, Y. Zhu, H. Keung, T. Ma, and R. Damiano, "Formal Property Verification by Abstraction Refinement with formal, simulation, and hybrid engines," in *Proceedings of Design Automation Conference*, 2001.

[20] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman, "SAT based abstraction-refinement using ILP and machine learning techniques," in *Proceedings of Conference on Computer Aided Verification*, 2002.

[21] R. Majumdar, T. A. Henzinger, R. Jhala, and G. Sutre, "Lazy Abstraction," in *Proceedings of Conference on Principles of Programming Languages*, 2002.

[22] K. L. McMillan, "Interpolation and SAT-based Model Checking," in *Proceedings of Conference on Computer-Aided Verification*, vol. 2725, *LNCS*, 2003.

[23] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of ACM*, vol. 7, pp. 201-214, 1960.

[24] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Proceedings of the International Conference on Computer-Aided Design*, 2001.

[25] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, "Partition-Based Decision Heuristics for Image Computation using SAT and BDDs," in *Proceedings of International Conference on Computer-Aided Design*, 2001.