Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1693-1730
submitted: 15/10/03, accepted: 14/8/04, appeared: 28/12/04 © J.UCS

Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals

Gianpiero Cabodi
(Politecnico di Torino, Dip. di Automatica e Informatica, Turin, Italy
gianpiero.cabodi@polito.it)

Sergio Nocco
(Politecnico di Torino, Dip. di Automatica e Informatica, Turin, Italy
sergio.nocco@polito.it)

Stefano Quer
(Politecnico di Torino, Dip. di Automatica e Informatica, Turin, Italy
stefano.quer@polito.it)

Abstract: Binary Decision Diagrams (BDDs) have been widely used in synthesis and verification. Boolean Satisfiability (SAT) Solvers, on the other hand, have been gaining ground only recently, with the introduction of efficient implementation procedures. Specifically, while BDDs have been mainly adopted to formally verify the correctness of hardware devices, SAT-based Bounded Model Checking (BMC) has been widely used for debugging.

In this paper, we combine BDD and SAT-based methods to increase the efficiency of BMC. We first exploit affordable BDD-based symbolic approximate reachability analysis to gather information on the state space. Then, we use the collected overestimated reachable state sets to restrict the search space of a SAT-based BMC. This is possible by feeding the SAT solver with a description that is the combination of the original BMC problem with the extra information coming from BDD-based symbolic analysis. We develop specific strategies to appropriately mix BDD and SAT efforts, and to efficiently convert BDD-based symbolic state set representations into SAT-oriented ones.

Experimental results prove the validity of our strategy to reduce the amount of variable assignments and variable conflicts generated by SAT solvers, with a subsequent significant performance gain. We gather results with four among the most used SAT solvers, namely Chaff, Limmat, BerkMin, and Siege. We could reduce the number of conflicts up to more than 100×, and the verification time up to 30×.

Key Words: binary decision diagrams (BDDs), satisfiability (SAT), bounded model checking (BMC), reachability analysis

Category: I.6.4, J.6

1 Introduction

Given a propositional formula, the *Boolean Satisfiability Problem* (commonly abbreviated as SAT) consists of determining a variable assignment such that the formula evaluates to true, or establishing that no such assignment exists.

Although SAT is an NP-complete problem, or at least no polynomial algorithm to solve it is known, large practical instances have been worked out thanks to efficient implementation procedures [Moskewicz et al. 2001, Limmat, Goldberg and Novikov 2002, Siege].

In the verification domain, SAT techniques are mainly used for *Bounded Model Checking* [Biere et al. 1999] (BMC) and *inductive verification* [Sheeran et al. 2000]. BMC formulates reachability checks as a series of satisfiability problems for paths of increasing (bounded) length. In practice, to see if a path of length n connecting initial and failure states exists, the transition relation of the system is unrolled n times, and the SAT solver is run on the generated problem.

Binary Decision Diagrams [Bryant 1986] (BDDs) are commonly used to implicitly represent large solution spaces in combinational and sequential problems that arise in synthesis and verification. BDDs may achieve an exponential compression rate, as the number of vertices and edges (graph size) is often exponentially lower than the number of paths from root to leaves. Nonetheless, even after almost two decades of intensive research in the area, BDDs have never been able to deal with real-world models and problem instances.

In the past few years, SAT and BDDs have been often presented as complementary techniques. In general, a BDD approach is more suitable for capturing all solutions of the problem simultaneously, whereas SAT decision trees, with no variable ordering restrictions, can potentially manage larger problems. Following this consideration, we propose a new way to trade of memory usage and run time by combining BDDs and SAT tools. Our target is to accelerate SAT-based BMC with the help of BDD-based reachability analysis. Symbolic BDD-based reachability analysis computes the set of states reachable from an initial set of states. This computation is applicable only to medium–small circuits because it may incur in an exponential blow-up. However, instead of computing an exact result we can adopt approximate techniques. Approximate reachability analysis estimates (in an under or overapproximated way) the reachable state set. It may also deal with larger circuits, as approximation techniques can be easily trade-off memory and time for the accuracy of the result. Unfortunately, the limit of approximate techniques, in verification, is that they are not complete. For example, with an overapproximate reachable state set it is possible to prove correctness[1], but it is not possible to disprove it[2]. As a consequence we need to appropriately adopt the SAT tool to complete the work started with the BDD in the symbolic domain. Our driving idea is to complement the initial overapproximate BDD-based state space visit, with a final SAT solver search. In our

[1] This happens any time there is no intersection between the estimate and the bad set of states.

[2] Whenever there is an intersection between the estimate and the bad set of states, it is not possible to establish whether the intersection lies in the exact reachable state set or only within the states added by overapproximating it.

method, the role of the overapproximation obtained with BDDs is essentially to prune and focus the SAT search. We proceed as follows.

In a first phase, we compute an overapproximate estimate of the traces connecting the initial state set to the set of bad states. We can compute this estimate starting from the initial set of states and proceeding in the forward direction, or starting from the bad set of states and proceeding backward. Then, the estimate is combined, as an additional constraint, with the original Bounded Model Checking problem. Notice that this estimate is redundant information already contained in the original problem formulation: The estimate is essentially an explicit time frame by time frame representation, i.e., a set of overapproximated state sets, of the behavior of the design in each time frame. This information can be seen as an explicit constraint for the SAT solver, which, in turn, does not have to imply it from the initial state set.

The effect we achieve somehow mimics the contribution of conflict clauses, generated by means of conflict analysis, in state-of-the-art SAT tools, where each new conflict clause individually represents a sub-set of the state space in which no solution exists. The information gathered in a conflict clause is redundant, as it was already contained in a set of clauses of the original problem.

Our estimate of the reachable state space is another formulation of the information already contained in the BMC problem. The advantage of using state sets is their ability to prune the SAT solver search space. We generate this extra information with an initial pre-processing and learning phase. Although we might lose some optimizations achievable through a tighter and more dynamic inter-leaving with the SAT solver, our methodology is compatible with any SAT solver, as, theoretically, we do not require any interaction with inner steps of SAT algorithms. Practically, we may need some interaction to avoid a performance degradation of the SAT tool due to a bad use of our additive state space information.

Our procedure mixes BDD-based and SAT-based algorithms, trading-off memory and time. As far as we know, this is the first time symbolic BDD-based overapproximate reachability analysis is used to prune a SAT solver search space.

A further contribution of our work is to introduce a set of strategies to convert BDDs (in a monolithic or conjoined form) to CNF formulas. The strategies we propose are compared in terms of their ability to generate a compact CNF problem (number of variables, literals and clauses), and benefit for the SAT engine (pruning efficiency).

The remainder of this paper is organized as follows. In Section 2, we introduce some preliminary concepts on notation, SAT techniques for verification, and reachability analysis. Section 3 summarizes the related work. Section 4 outlines our approach. Section 5 presents our top-level algorithm. Section 6 introduces our approximate reachability analysis routines, and Section 7 describes how to feed the symbolic BDD-based information to the SAT solver. Section 8 describes our technique to store BDDs as CNF problems. Section 9 presents our experimental results. Finally, Section 10 concludes the paper.

2 Background

2.1 Model, Notation, and Property Definition

The sequential systems we address are usually modeled as Finite State Machines (FSMs). Each FSM is described by a Transition Relation $\mathsf{TR}(s, y)$, which indicates its present–next state behavior, and an initial state set S. We will also use the C symbol to indicate the gate-level netlist of the system.

In our notation, B indicates the Boolean space. Symbols $\wedge$, $\vee$, $\neg$, and $\equiv$ are used for Boolean conjunction (AND), disjunction (OR), negation (NOT), and exclusive-nor (XNOR) respectively. The $\downarrow$ symbol denotes the generalized cofactor [Coudert et al. 1989] function, i.e., in $f \downarrow g$, g can be viewed as care set for f, which, as a consequence, can be arbitrarily simplified in the domain subspace where g is *false*. Set operations are indicated with $\cup$ and $\cap$. We make no distinction between the BDD representing a set of states, the characteristic function [Cerny et al. 1986] of that set, and the set itself. We thus use Boolean operators for set operations, implemented by Boolean operators on BDDs.

We use the notation $lfp\ v.f(v)$ to indicate a *least* fixed-point of f, where f is a formula and v is a propositional variable. The least fix-point of the formula $f(v)$ is any parameter b such that $b = f(b)$, and, if $c = f(c)$ for some c, then $b \subseteq c$ (see [McMillan 1994] for other details on this topic).

An invariant property[3] P is checked by attempting to prove (or disprove) the reachability of its complement T ($\mathsf{T} = \neg\mathsf{P}$) from S. We will use T to indicate the target, or the failure, or the bad set of states.

2.2 SAT-Based Model Checking

For an overview on SAT solvers and a list of references the reader is referred to the tutorials [Zhang and Malik 2002, Biere and Kunz 2002, Kautz and Selman 2003, Berre and Simon 2003].

SAT-based BMC considers only paths of bounded length n and builds a propositional formula f that is satisfiable *iff* there is a counter-example (a path from S to T) of the same length. For the above reason the technique works well in falsification and partial verification. Full verification is usually achieved by inductive proofs.

[3] Or AG CTL property.

SAT solvers generally operate on propositional formulas f specified in *Conjunctive Normal Form* (CNF). This form is a two-level decomposition: The logical AND of one or more *clauses*, each of which consists of the logical OR of one or more *literals*. A *literal* is merely an instance of a variable or its complement.

In order to decide if f is satisfied, most solvers adopt variants of the basic Davis-Putnam [Davis and Putnam 1960] recursive algorithm. At each recursive call, the algorithm proceeds through the following three steps:

- *Variable Decision*: Assign a value to an unassigned variable, so that the search space is recursively restricted.
- *Boolean Constraint Propagation*: Carry out all possible direct implications due to the previous assignment.
- *Conflict Analysis*: Check for conflicting clauses, i.e., clauses whose literals are all assigned the zero constant. In this phase, conflict clauses are added to the clause database for future early detection (and pruning) of bad decisions (variable assignments); *backtrack* is performed to properly re-start the search with a new variable decision.

2.3 BDD-Based Model Checking

A standard BDD-based forward reachability analysis procedure is a breadth-first visit of the state space that starts from $\mathsf{FR} = \mathsf{S}$ and proceeds through a least fix-point (lfp) iteration:

$$\mathsf{FR} = lfp\, \mathsf{FR}.(\mathsf{S}\ \vee (Img(\mathsf{TR}, \mathsf{FR}))$$

We indicate with FR_i the set of states reached *as far as* the i-th iteration, and with FR the array of sets until the fix-point, i.e., $\{\mathsf{FR_0}, \mathsf{FR_1}, \ldots, \mathsf{FR_n}\}$, where n is the sequential depth of the system. The method is based on the iterated application of the image function Img; $Img(\mathsf{TR}, \mathsf{FR})$ computes the set of states reachable from FR, by the model TR, in one single clock cycle.

As the target state set T may be reached before the fix-point, it is possible to avoid a full computation of FR with on-the-fly tests for intersection with T. Whenever an intersection with T is found, the FR array can be used to extract a (single) counter-example trace.

CTL model checking procedures are often implemented as backward traversals, which have a dual formulation, and compute the BR array in the backward direction. More in detail, a backward traversal is easily expressed by swapping the S and T sets, and changing the Img function with the PreImg one. The $PreImg(\mathsf{TR}, \mathsf{BR})$ procedure computes the set of states from which BR is reached, by the model TR, in one single clock cycle.

Approximate Traversals [Cho et al. 1996, Govindaraju and Dill 1998b] are a popular way to extend the applicability of reachability analysis to larger circuits. The approach is based on the *approximate image* (IMG$^+$) operator, returning overestimations of exact images:

$$Img^+(\mathsf{TR}, \mathsf{FR_i}) \supseteq Img(\mathsf{TR}, \mathsf{FR_i})$$

Notice that, although FR^+ includes more states than FR, its BDD representation is usually much smaller, and many mutual interactions and dependences among state variables disappear because of the approximation. Albeit approximate techniques are computationally cheaper and more scalable, they can provide sufficient but not necessary check, i.e., they can prove correctness but they cannot disprove it:

- Whenever FR^+ does not intersect the bad set of states, the property passes because the system is "safe" even within more states than the ones actually reached.
- Whenever there is an intersection between FR^+ and the bad set of states it is not possible to understand whether the intersection is due to the overapproximation or not.

3 Related Work

With the advent of SAT-based BMC tools a lot of researchers compared SAT-based methods with more traditional BDD-based ones [Bjesse et al. 2001, McMillan 2002]. As different researchers agree that the two approaches are essentially complementary, a lot of recent work concentrates on dovetailing the two approaches in a loose or strict fashion. In this section, we summarize previous research that is related to our approach.

3.1 Performing Satisfiability on BDDs

A few recent works performs satisfiability on BDDs or on mixed CNF/BDD representations, rather than on pure CNF. The rationale for that choice is simple: BDDs can be more compact, and they are more expressive, than an equivalent set of clauses. As a consequence there is room for trading-off memory and expressiveness.

In Franco et al. [Franco et al. 2003] BDDs, after a light pre-processing phase, are transformed into SMURFs, i.e., State Machine Used to Represent Functions. These SMURFs are essentially automata representing complete future information about the variable selection process necessary to satisfy the BDD. The search consists of extending partial assignments until either all SMURFs reach their end state, or until some contradiction arises among the partial assignments. In the former case, a solution is found. In the latter case, a backtrack occurs.

In Damiano et al. [Damiano and Kukula 2003], the authors build BDDs of moderate size starting from the original clauses. After that, they focus on BDD reasoning to obtain a smarter implication/learning process than the one performed by the SAT solver.

In Somenzi et al. [Jin and Somenzi 2004] the authors propose a SAT solver accepting as inputs a combination of CNF clauses, BDDs, and And-Inverter-Graphs [Kuehlmann et al. 2001]. Rather than converting all forms into one, the tools works on all representations, transforming, when appropriate, parts of the input from one of them to another.

3.2 Combining SAT and BDDs

Gupta et al. [Gupta et al. 2000, Gupta et al. 2001] perform BDD-based reachability analysis by using a SAT procedure within symbolic image computation. They call their approach *BDDs at SAT Leaves*. More specifically, they use BDDs to represent state sets and a CNF formula to represent the transition relation. Symbolic images of a state set are computed by exhaustive SAT search of all solutions within the space of primary inputs, present and next state variables. However, rather than using SAT to enumerate each solution all the way down to a leaf, the process switches to BDD-based computations at certain intermediate points within the SAT decision tree. The switch is done as a trade-off between space complexity of BDDs, and time complexity of full SAT enumeration. In a sense, the approach can be regarded as SAT providing a disjunctive decomposition for image computation into many sub-problems, each of which is handled symbolically using BDDs.

In Gopalakrishnan et al. [Gopalakrishnan et al. 2003], the previously described method is further developed with the aim of finding just one solution, instead of all solutions. The authors propose a new heuristic for variable decision, and an efficient implementation of the BDD section of the algorithm.

3.3 Learning from BDDs

Gupta et al. [Gupta et al. 2003a] propose an approach sharing similar goals with the one we present here. In that work, the authors start from the CNF representation of the problem. In order to improve the learning ability of the solver, they build BDDs of limited logic around selected structural points, thus learning useful information beyond the one usually gathered by conflict analysis. BDDs are then transformed back to CNF clauses and fed to the solver with a strategy similar to the one presented in Section 8.

Another work by Gupta et al. [Gupta et al. 2003b] may be considered as an extension of the present work, specifically oriented to induction verification. On one side, Gupta et al. use, as we do, overapproximate reachability analysis to constrain the BMC search. On the other, they use overapproximate information as an additional (non redundant) constraint for induction based unbounded verification [Sheeran et al. 2000].

3.4 Computing Overapproximations with BDDs

A lot of researchers have exploited approximate BDD-based reachability [Cabodi et al. 1994, Cho et al. 1996, Govindaraju and Dill 1998b] within techniques based on approximate, exact, forward, and backward traversals [Govindaraju and Dill 1998a, Moon et al. 1998].

Govindaraju et al. [Govindaraju and Dill 1998a] check the mutual reachability between initial and failure states, by iteratively combining overapproximate forward and backward traversals. Each new traversal increases the accuracy of the approximation. Whenever a forward or backward traversal reaches a fixpoint outside its target, initial and failure states are proved to be not mutually reachable, and this in turn proves the property.

Moon et al. [Moon et al. 1998] concentrate on the idea of using an overapproximation of the reachable state set to simplify exact verification while performing Model Checking.

3.5 Comparison with the Related Work

Our work shares several ideas with previous work.

We use both BDD and SAT-tools to cope with their contrasting limits, but we avoid a tight integration between them, thus enabling rapid prototyping and experimentation.

We make an effort to trade-off memory for expressiveness using BDDs to prune the search space. Our learning comes from an overapproximate symbolic reachability analysis, while other methods used a static netlist BDD reasoning.

We guide the final SAT solver search with constraints derived in previous cheaper and approximate searches performed with BDDs. Our method is exact, also if it partially relies on an approximated strategy, and it ends with a SAT solver call, rather than with a last BDD-based exact symbolic search.

As a final remark, notice that the work presented here is an extended and completely revised version of [Cabodi et al. 2003]. The top-level algorithm has been adapted to obtain a better tuning and trade-off between memory and time. Low-level algorithms have also been improved. Experimental results have been completely recollected, to add more detailed evidence on performance and comparison with various SAT solvers on a broader set of circuits.

4 Our Methodology to Combine BDDs and SAT

In this section we briefly overview our methodology. An in-depth detailed analysis is presented in the following sections.

The approach we propose can be viewed:

- As a way to partition a verification task between a BDD and a SAT engine (we perform a preliminary effort with BDDs, we conclude the task through a SAT solver).
- As an optimization of a SAT-based BMC, by means of redundant information learned by BDD pre-processing.

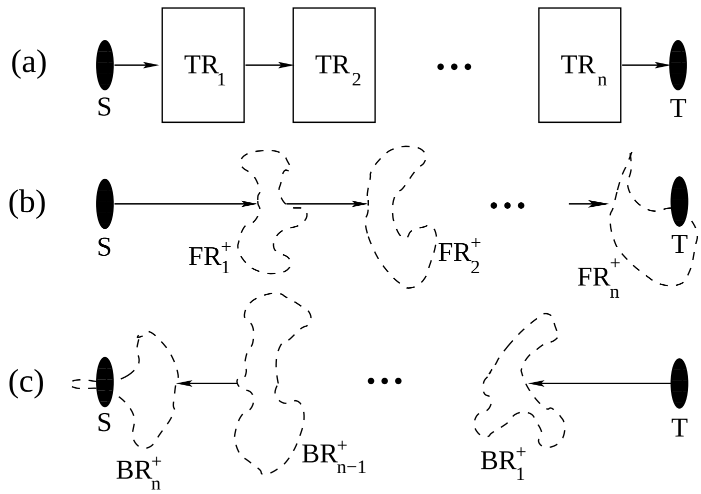


Figure 1: (a) Standard combinational unrolling for SAT-based BMC; (b) Approximate forward traversal from S to T; (c) Approximate backward traversal from T to S. S indicates the initial state set, T the target state set, TR the transition relation, FR_i^+ and BR_i^+ the overapproximate forward and backward reachable state sets, respectively.

Fig. 1 shows the main flow of our methodology. Fig. 1 (a) shows a graphical representation of standard SAT-based BMC. As introduced in Section 2, the goal is to find a path of length n between the start state S and target state T on the CNF representation of the problem. To produce this problem, a combinational unrolling (of length n) of the transition relation TR is first generated. During this phase we have to keep into account:

– A proper variable relabeling for the TR variables, i.e., each time frame has to be defined using different present and next state variables.

– A COI (Cone of Influence) reduction due to the property, i.e., only the logic strictly necessary to define the property under check has to be considered.

After that, the expressions for S and T are added to the previous formulation as CNF formula. Finally, the SAT-engine is run on the out-coming problem to solve it.

Our basic idea is to help the SAT solver with information coming from a BDD-based reachability analysis tool. The simplest approach performs an approximate forward breadth-first traversal, as the one represented in Fig. 1(b), or a backward breadth-first traversal, as the one sketched in Fig. 1(c). A tighter overapproximation is obtained with a forward-backward strategy, where forward estimates of the reachable state set constrain backward traversals and vice-versa. The result of this preliminary phase is an overestimate of the paths leading from S to T, such that all possible real paths are included in the overapproximation. Notice that, at this stage, we work with a BDD tool, and so each set of states is represented by means of BDDs[4].

Those BDDs contain redundant information representing constraints on the input space of each time frame in the combinational unrolling. More specifically, a constraint for the i-th time frame is already (and implicitly) present in the original formulation of the BMC problem, but it is represented in terms of all variables in all time frames of the combinational unrolling. Due to BDD preprocessing (and variable quantifications in image/pre-image operations) a state set provides constraints as a function of local state variables of time frame i. The effect is an enhanced ability to early detect invalid variable assignments at a given time frame, in order to better guide the search for a satisfying solution.

Fig. 2 depicts a state space interpretation of our pruning methodology. The square box represents the overall state space of the model where the SAT solver performs its search. Notice that the target of the SAT solver is to find one single path from S to T within the set of all existing paths P, shown in gray.

In our approach, before running the SAT tool, we perform our symbolic overestimated reachability analysis. We compute the overapproximate forward FR_n^+ or backward BR_n^+ reachable state set or both of them (actually, as previously stated, each estimate can be used to restrict the other estimate). These sets overestimate P and are indicated in the figure by two dotted shapes. Their intersection gives a tighter overestimation of P. The tighter the overestimation, the greater the pruning effect of the space search we may theoretically have by feeding it to the SAT solver.

[4] Depending on the kind of BDD representation/decomposition used, which we do not talk about, the state sets may be represented by monolithic, disjunctive or conjunctive forms.

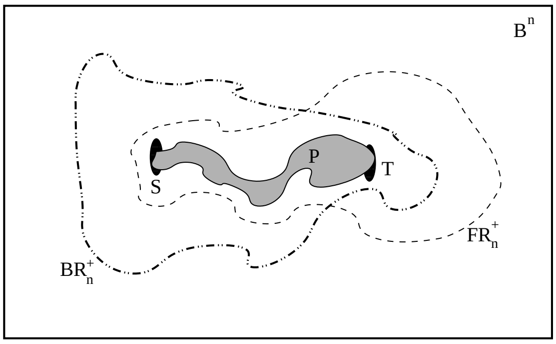


Figure 2: State Space Pruning Interpretation. B^n indicates the overall Boolean state space for the model, S represents the initial state set, T the target state set, P the set of paths from S to T, FR_n^+ and BR_n^+ the overapproximate forward and backward reachable state sets, respectively.

To give a rough idea let us take the ISCAS'89 circuit `s15850.1`. It has 534 memory elements and, as a consequence, it has a total number of states equal to $2^{534} = 5.62 \cdot 10^{160}$. Cho et al. [Cho et al. 1996] report an estimated reachable state set equal to $1.51 \cdot 10^{78}$ for this circuit[5]. As a consequence, we can potentially reduce the SAT solver's search space from about 10^{160} to about 10^{78}: For each time frame we impose a state space restriction which otherwise the SAT solver should imply from the initial state space S.

A minor contribution of applying approximate reachability analysis before satisfiability analysis is to identify early terminations of the BMC procedure saving computation time. In fact, whenever the overestimation of the reachable state set does not intersect the target set of states T (or the initial one S, in the backward direction) the property will pass. Those are the cases on which approximation works at its best and the hardest to be proved by the SAT engine (as, at least theoretically, he state space has to be completely explored). Reachability would then consist of a preliminary check within the overall verification procedure. Moreover, the approximate search may also be useful to identify a lower limit for the value of the bound n. To this respect, the paths found by the overapproximation are always shorter or equal to the exact ones. This can avoid useless searches with invalid values of n. As a consequence, the overapproximation reveals to be particularly useful for bugs with very long counter-examples or holding properties [McMillan 2004].

[5] This value was computed using an algorithm called TMBM. This algorithm mixes the two algorithms, i.e., MBM and FBF, described in Section 6.

5 Top-Level Algorithm

Fig. 3 describes our high-level algorithm, BDD&SATVERIFICATION. The procedure iterates through a cycle. Each iteration is characterized by an increment of the BMC bound, and a growth of the effort to solve the problem. During each iteration BDD and SAT operations are interleaved.

```
BDD&SATVERIFICATION (C, S, T, time_th, mem_th)
    TR ← GENERATETR (C)
    bound ← 0
    while (LIMITCHECK (time_th, mem_th) = TRUE)
        R ← OVERAPPROXFWDRA (TR, S, T, bound, time_th, mem_th)
        if (R = ∅) then
            return (PASS)
        else
            bound ← SETSIZE (R)
            R ← OVERAPPROXBWDRA (TR, S, T, R, bound, time_th, mem_th)
            if (R ≠ ∅) then
                Ω ← BMCCLAUSES (C, S, T, bound)
                Ω̂ ← BDD2CNF (R, bound)
                res ← SATCHECK (Ω ∪ Ω̂, time_th)
                if (res = FAIL) then
                    return (res)
            bound ← bound + Δ_bound
            time_th ← time_th + Δ_time
            mem_th ← mem_th + Δ_mem
    return (UNKNOWN)
```

Figure 3: Our BMC algorithm combining a SAT solver with BDDs information for state traversal.

Function OVERAPPROXFWDRA, introduced in Section 6.1, performs an overestimated forward reachability analysis. It looks for a set of paths, of the shortest possible length, but not shorter than *bound*, connecting S to T. When this set does not exist, the verification process ends up with a PASS result, without resorting to the SAT solver engine. In the opposite case, the process moves forward to the overestimated backward traversal phase, OVERAPPROXBWDRA.

Function OVERAPPROXBWDRA, introduced in Section 6.2, proceeds in the backward direction to refine the previous set of paths R, i.e., the array of forward reachable state sets FR_i. If its search ends up with a valid array of state sets, i.e., R is a proper cone connecting S to T (coming from refining the former R array, FR, with the BR array) we proceed with the SAT solver search. In this case we proceed through three steps.

In the first step, function BMCCLAUSES, starting from the bound *bound* and the circuit C, generates the CNF problem Ω. It essentially duplicates the circuit C a number of times equal to the bound. Different tools can be used for this step, as we will outline in the experimental result section.

Then, in the second step, function BDD2CNF, setting out from the array R, generates the CNF constrain $\widehat{\Omega}$, as will be described is Section 8. Here, the starting point is to use all sets R_i to increment the original SAT problem on each time frame. Nevertheless, we proceed heuristically (see Section 7) to reduce the overhead in terms of added clauses and variables.

Finally, in the third step, SATCHECK runs the SAT solver on the original CNF problem Ω augmented with the learned clauses $\widehat{\Omega}$. Whenever the SAT solver discovers that the property fails, the procedure terminates reporting this failure. Otherwise, the algorithm proceeds to the next iteration, after the minimum number of steps *bound*, the time threshold $time_{th}$, and the memory threshold $memory_{th}$ have been incremented.

Notice that the value of the bound is mainly driven by the forward reachability analysis phase to avoid useless and repeated experiments with increasing value of the bound. The value Δ_{bound} may be equal to 1 or larger in case the user is not interested in finding the shortest possible counter-example. Moreover, time and memory limits are checked within OVERAPPROXFWDRA, OVERAPPROXBWDRA, SATCHECK, and the main verification loop. When a limit is exceeded within OVERAPPROXFWDRA or OVERAPPROXBWDRA the reachable state set collected up to that point is returned and used to constrain only the "corresponding" part of the combinational unrolling. When a limit is encountered within the SATCHECK function we simply try to increase the accuracy of the reachable state phase. When a limit is encountered in the main cycle the process is stopped returning "unknown" as a result.

As a final remark, it is possible to observe that for any iteration of the main loop, except for the first one, the OVERAPPROXFWDRA routine does not necessarily have to perform full reachability from S to T. Usually, it is sufficient to proceed toward T from the last set R_l (i.e., FR_l) of the array R, where R_l was found in the previous iteration. Full reachability may still be performed to increase or to decrease the accuracy of previously performed visits.

6 Approximate Reachability Analysis

The approximate reachability analysis phase is very important as far as the SAT search space simplification is concerned.

When the approximate traversal is more accurate and the sets of states reached are closer to the exact ones, the SAT solver search space is potentially reduced more drastically. This is motivated by the consideration that our estimates are directly added to each time frame and do not have to be directly derived from the initial state set by the SAT solver. Nevertheless, better approximate traversal accuracy implies larger BDDs representing the reached sets, and the translation process of these BDDs into CNF formulas (see Section 8) is more likely to introduce a larger amount of temporary variables and clauses. As a consequence, there is a trade-off between the accuracy and the effectiveness of the reachable state sets. Moreover, these factors are balanced by the cost of computing the overestimations themselves.

From a general point of view, standard approximate reachability routines [Cho et al. 1996] proceed in two main steps:

1. They perform a *State Space Decomposition*, i.e., they evaluate a partition of the state variables of the model. Each partition corresponds to a Boolean subspace, and to a sub-FSM of the original FSM.

2. Given the above partition, they calculate a super-set of the reachable state of the model by performing separate traversal of the previously created sub-FSMs.

Different strategies can be used to perform the separate traversals and to model the interaction among them.

The *Machine-by-Machine* (MBM) approach processes each sub-FSM *serially* and *iteratively*, i.e., sub-FSMs are treated one at a time during one entire least fix-point computation. The order in which sub-FSMs are traversed is important to obtain good performance: They are often treated in an event-driven fashion, i.e., a sub-FSM is traversed again only when the reached set of one of its fanin FSM's has changed.

The *Frame-by-Frame* (FBF) approach handles all sub-FSMs in *parallel*, i.e., it performs a traversal step on each sub-machine. In the FBF algorithm, interaction between the sub-FSMs is more fine-grained, since the base time unit of interaction is a time frame rather than an entire least fixed-point calculation of a sub-FSM. As a consequence, FBF is usually more expensive but it results in tighter estimate of the reachable state set.

In our framework we need the overapproximation of the reachable state set at the same traversal level for all sub-machines. As a consequence we need a variant of the original FBF algorithm.

Moreover, as we have an initial and a target sets of states, we may proceed both in the forward and in the backward direction. As introduced in Section 5 we generalize this idea, see also [Govindaraju and Dill 1998a, Cabodi et al. 2002], by adopting an iterative refinement process based on a sequence of alternate forward and backward phases. This method can produce more accurate estimates, and, in our framework, it can be accomplished with the appropriate (desired) computational effort.

In the sequel, we describe our approximate forward and backward reachability analysis routines.

6.1 Algorithm for Forward Approximate Reachability

Fig. 4 shows our *forward* routine. More specifically, it is an overapproximate visit (notice the use of the $\text{I}\small{\text{MG}}^+$ function) proceeding in a breadth-first way. The function may end because:

- The least fix-point is reached without intersection with the T set of states, i.e., ($\mathsf{New} \neq \emptyset$) is false. This means that there are no paths from S to T within the overestimated set of reachable states, and, as a consequence, the property will hold and PASS.
- There is an on-the-fly intersection between the explored state space and T, i.e., $((\mathsf{FR}_l^+ \wedge \mathsf{T}) \neq \emptyset)$. In this case, the property is violated before the fix-point and we can avoid the need to fully compute the overestimation of the state space.
- The *bound* on the iteration count has been reached, i.e., $l \geq bound$. This condition identifies the lower admissible value for the SAT solver's bound.

The set FR^+ is returned by the procedure.

```
OverApproxFwdRA (TR, S, T, bound, time_th, mem_th)
    l ← 0
    FR_l^+ ← S
    New ← S
    while (New ≠ ∅)
        if ((FR_l^+ ∧ T) ≠ ∅ AND l ≥ bound) then
            return (FR^+)
        Next ← Img^+ (TR, FR_l^+)
        New ← Next ∧ ¬ FR_l^+
        l ← l + 1
        FR_l^+ ← Next
    return (∅)
```

Figure 4: Forward Approximate Reachability Analysis within BMC.

As already described in Section 5, OVERAPPROXFWDRA does not always have to perform full reachability from S to T. Anyhow, this possibility is not indicated by the pseudocode for sake of simplicity.

As a final remark, notice that all BDD operations are scheduled with run-time checks on memory and time limits (see also Section 5) even if this is not explicitly described by the pseudocode. The parameters $time_{th}$ and mem_{th} are implicitly used by all the BDD-based procedures. The run is stopped when limits are exceeded. In this case the reachable state set collected up to that point is returned and used to constrain only the "corresponding" part of the combinational unrolling.

6.2 Algorithm for Backward Approximate Reachability

Fig. 5 shows the pseudocode of the backward procedure. It proceeds computing reachable state sets BR_i^+ in the backward direction using function PreImg$^+$. During the computation, we simplify the sets BR_i^+ by restricting the search with the forward FR state sets. This operation is performed using cofactor based simplification: The backward estimates have to be included in the forward estimates and must have the most possible compact BDD representation. We also try to force the procedure to perform a number of backward steps equal to the number of steps performed in the forward direction. Whenever the set BR_i^+ becomes the empty set the procedure terminates returning an empty set of states. Similarly to function OverApproxFwdRA, the set BR^+ is returned by the procedure, and on-the-fly checks on memory and time are performed by all BDD-based procedures.

```
OverApproxBwdRA (TR, S, T, FR⁺, l, time_th, mem_th)
    BR⁺_l ← T ↓ FR⁺
    i ← l
    while (i > 0)
        if (BR⁺_i = ∅) then
            return (∅)
        BR⁺_{i−1} ← PreImg⁺ (TR, BR⁺_i) ↓ FR⁺
        i ← i − 1
    return (BR⁺)
```

Figure 5: Backward Approximate Reachability Analysis within BMC.

Run-time checks on memory and time limits are performed and dealt with as described in Section 6.1.

6.3 Approximate Image Computation

As indicated in Sections 6.1 and 6.2, our approximate traversal procedures rely on an approximate image (Img$^+$) or pre-image (PreImg$^+$) computation. These routines are characterized as follows.

We perform an initial state space decomposition (see beginning of Section 6). In this phase, we decide how to decompose the FSM into sub-FSMs, and in which order to deal with them. Within each sub-FSM, we adopt a standard threshold-based clustering and reordering technique [Ranjan et al. 1995], i.e., we start from the next state functions of the sub-FSM, we build BDD clusters up to a certain BDD size threshold, and we order the clusters using an heuristic based on the support and the size of the BDD representing the clusters.

The initial state space decomposition (as well as the clusters) may be *dynamically* modified during the traversal process, depending on previous image computations. In particular, we adopt a learning approach using two BDD size thresholds to trim the computation effort. During each image computation of each sub-FSM we record the maximum BDD size involved in the computation:

- Sub-FSMs that are easy to traverse, i.e., whose BDDs are smaller than the first size threshold, are merged together.
- Sub-FSMs that are difficult to traverse, i.e., whose BDDs are larger than the second size threshold, are further decomposed into two subsequent sub-FSMs.

We found this methodology useful to correctly trim computational effort, especially within very deep traversal operations. We also use overlapping projections as presented by Govindaraju et al. [Govindaraju and Dill 1998b]. This technique is based on the idea of adopting not necessarily disjoint sub-FSMs, and it is very useful to increase the accuracy of the estimate keeping the computational cost under control.

7 Using Symbolic State-Set Information within the SAT Solver

The overestimation computed in the previous section is used in a straightforward way to prune the SAT solver's search. The BDD representing each estimate R (see Fig. 3), in a monolithic or decomposed way, is directly, i.e., time frame by time frame, and incrementally added to the problem formulation as a further set of CNF clauses. The simplest approach is to use all components R_i of R to increment each time frame of the original SAT problem. Nevertheless, to reduce the SAT solver's overhead (in terms of added clauses and variables) we adopt some further optimization heuristics. On one side, we know:

- The size of the original problem in terms of clauses and variables.
- How much each R_i differ from the previous one in terms of number of represented states.

– How many clauses/variables each R_i would generate.

Given these data, we keep the added overhead below a certain threshold: We add our constraints not on all time frames but only once every k time frames. The value of k is selected heuristically using the above information.

A more complex approach consists in explicitly simplifying, e.g., using cofactor based techniques, the circuit representation with each corresponding time frame estimate. The final CNF problem would be generated using the simplified time frame instances.

While this solution would possibly give better simplifications, it would be far more expensive and applicable only up to medium sized verification instances. For this reason, we rely on the efficiency of standard SAT solvers to obtain a proper search pruning effect.

Notice that, in all the cases, the counter-example eventually obtained possibly includes some temporary variables generated by the BDD-to-CNF translation process in addition to the variables usually present in the BMC problem. As a consequence, we need to bring each counter-example back to the original representation space, by quantifying out the temporary variables (see Section 8) when counter-examples are computed.

8 Converting BDDs to CNF Formulas

In our framework it is particularly important to generate compact SAT formula, i.e., to introduce as few clauses and intermediate variables as possible.

For this reason, given a BDD representing a function f in monolithic or conjunctive form, we develop three possible ways to store it as a CNF formula. We analyze them in the following three sections.

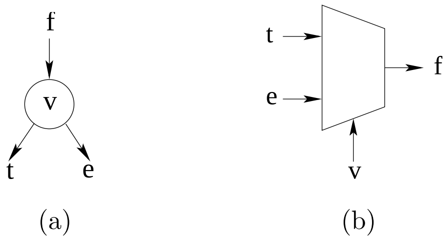
8.1 Single-Node-Cut Method

The first method, which we call *Single-Node-Cut*, models each BDD node, except for the ones with both constant children, as a multiplexer. Fig. 6 shows the strategy adopted. Each node, i.e., each multiplexer, Fig. 6(a) and (b), is characterized by:

– Two data inputs: one for the "then" or 1 child t, and one for the "else" or 0 child e.

– A selection input, i.e., the node variable v.

– One output, i.e., the function value f (whose value is assigned to an additional CNF variable).



$$(\neg f \lor v \lor e) \land$$
$$(f \lor v \lor \neg e) \land$$
$$(\neg f \lor \neg v \lor t) \land$$
$$(f \lor \neg v \lor \neg t)$$

(c)

Figure 6: BDD to CNF: Multiplexer Interpretation.

The final number of variables is equal to the number of original BDD variables plus the number of internal nodes of the BDD. Fig. 6(c) shows the clauses obtained representing a single multiplexor. Simplifications are possible when one of the children is the terminal node[6].

Fig. 7 shows the pseudocode used to generate this format. It is essentially a recursive post-order visit of the BDD. Function PrintNodeAsCnf generates, for the BDD node f, the CNF representation represented in Fig. 6(c).

```
SingleNodeCut (f)
    if (IsConstant (f)) then
        return
    if (IsVisited (f)) then
        return
    SingleNodeCut (f.t)
    SingleNodeCut (f.e)
    SetVisited (f)
    PrintNodeAsCnf (f)
    return
```

Figure 7: Pseudocode for the Single-Node-Cut Method.

The following example shows how the Single-Node-Cut method works on a small BDD.

Example 1. Fig. 8(a) represents a BDD with 4 nodes. BDD variables are named after integer numbers ranging from 1 to 4. To translate this BDD into a CNF formulation we insert a temporary variable for each node but the one with two terminal children. The new variables are named 5, 6 and 7 and are indicated in the square boxes. Procedure SingleNodeCut generates the CNF formulation

[6] Notice that we represent BDDs using only one terminal node, i.e., the node 1, while the other is obtained pointing to the previous terminal node with an inverted edge.

as indicated in Fig. 8(b). Appropriate simplifications are applied to the general case of Fig. 6(c) when one child of the node is the terminal node. For example, for node 3 the child e is equal to 0. Then the CNF representation of Fig. 6(c) becomes equal to

$$(\neg f \vee v \vee 0) \wedge (f \vee v \vee 1) \wedge (\neg f \vee \neg v \vee t) \wedge (f \vee \neg v \vee \neg t)$$

Keeping into account that

$$(\neg f \vee v) \wedge (\neg f \vee \neg v \vee t) = (\neg f \vee v) \wedge (\neg f \vee t)$$

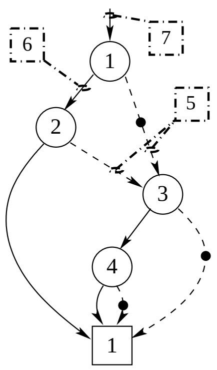
the formulation can be simplified to

$$(\neg f \vee v) \wedge (\neg f \vee t) \wedge (f \vee \neg v \vee \neg t)$$

which in turn gives

$$(\neg 5 \vee 3) \wedge (\neg 5 \vee 4) \wedge (5 \vee \neg 3 \vee \neg 4)$$

as in Fig. 8(b).



(a)

$(node3)$	$(\neg 5 \vee 3) \wedge$
	$(\neg 5 \vee 4) \wedge$
	$(5 \vee \neg 3 \vee \neg 4) \wedge$
$(node2)$	$(6 \vee \neg 2) \wedge$
	$(6 \vee \neg 5) \wedge$
	$(\neg 6 \vee 2 \vee 5) \wedge$
$(node1)$	$(7 \vee 1 \vee 5) \wedge$
	$(\neg 7 \vee 1 \vee \neg 5) \wedge$
	$(7 \vee \neg 1 \vee \neg 6) \wedge$
	$(\neg 7 \vee \neg 1 \vee 6) \wedge$
(f)	(7)

(b)

Figure 8: BDD to CNF Translation: Single-Node-Cut Algorithm.

8.2 No-Cut Method

The *No-Cut* method creates clauses expressing (covering) the zeros of a function f. This idea is motivated by the consideration that a CNF formulation is essentially a product-of-sum, which in turn can be derived by expressing the off-set of f. Within the BDD for f, such clauses are found by following all the paths from the root node to the constant node 0. Fig. 9 shows the pseudocode for this operation. It is a recursive BDD visit from the root node to the terminal 0 node[7]. As in the previous section, $f.t$ indicates the "then" or 1 child of f, and $f.e$, the "else" or 0 child. Once the terminal node 0 is reached, function PrintClause prints all the literals in the set clause. clause is initially the empty set, and is augmented with the variables selected when searching for a path to the terminal node 0.

```
NoCut (f, clause)
    if (IsConstantZero (f)) then
        PrintClause (clause)
    else
        NoCut (f.t, clause ∪ ¬ f.v)
        NoCut (f.e, clause ∪ f.v)
    return
```

Figure 9: Pseudocode for the No-Cut Method.

The final number of CNF variables is equal to the number of BDD variables.

Example 2. Fig. 10 shows an application of function NoCut to the example used in Fig. 8. In Fig. 10 (a) the three paths to the constant 0[8] are indicated on the BDD as ($p1$), ($p2$) and ($p3$). Fig. 10(b) reports the corresponding clauses.

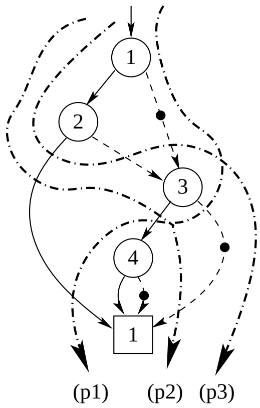
8.3 Auxiliary-Cut Method

The *Auxiliary-Cut* method is a trade-off between the previous two strategies. The BDD is logically decomposed into sub-trees by selectively inserting cut-points. Each cut point implies the insertion of a new CNF variable. As a consequence, this method coincides with the No-Cut algorithm when no cut point is inserted, and with the Single-Node-Cut method when a cut point is inserted for each BDD node.

We experimented with two cut point selection strategies. In the first one, a new CNF variable is inserted on shared nodes, i.e., BDD nodes which have more than one incoming edge. This technique, reduces the total number of literals

[7] The reader should remember that the value of the constant leaf reached in a BDD depends on the number of negations found along the path from the root of the BDD to the leaf itself. The value is 1 when the number of inverted edges is even, and 0 when is odd.

[8] Or constant 1 through an odd number of inverted edges.



(a)

$(p1)\quad (1 \vee \neg 3 \vee \neg 4) \wedge$
$(p2)\quad (\neg 1 \vee 2 \vee \neg 3 \vee 4) \wedge$
$(p3)\quad (\neg 1 \vee 2 \vee 3)$

(b)

Figure 10: BDD to CNF Translation: No-Cut Algorithm.

stored, but can produce clauses with many literals[9]. The second method tries to avoid this drawback by introducing further cut points in addition to the previously defined ones. The new cut points are used to break the length of each path between two cut points to a maximum (user) selected value.

Fig. 11 shows the pseudocode for this procedure.

It proceeds in two phases. During the first phase, function GenerateCut performs one or more in-depth visits of the BDD, and sets a flag on each BDD node on which a cut point has to be inserted. When the first heuristic is used, function GenerateCut simply counts the number of incoming edges for each node, and sets the flag when this number is greater than 1. When the second strategy is used, it firstly proceeds as in the previous case. After that, it measures the length of each path between two cut points, and sets a flag on a BDD node every time this value is larger than the selected threshold.

During the second phase, function AuxiliaryCutRecur, generates all the clauses with one in-depth visit of the original BDD. For each cut point, determined in the previous phase, it implicitly decomposes the BDD into two BDDs. Let us suppose that the BDD representing function $f(x)$ has a cut point $\widehat{x}$ representing subfunction $g(x)$. Then $f(x)$ can be decomposed as

$$f(x) = f(x, \widehat{x}) \wedge (\widehat{x} \equiv g(x))$$

which expresses $f(x)$ by augmenting its support with the cutting point variable $\widehat{x}$, and states that this new variable should have the same value as subfunction $g(x)$. After that, the strategy relies on visiting both BDDs for $f(x, \widehat{x})$ and for

[9] The number of literals is bound by the number of variables of the BDD, i.e., the longest path from the root to the terminal node.

```
AuxiliaryCut (f)
    GenerateCut (f)
    AuxiliaryCutRecur (f, ∅)
    return

AuxiliaryCutRecur (f, clause)
    if (IsConstantZero (f)) then
        PrintClause (clause)
        return
    if (IsACut (f)) then
        PrintClause (clause)
        if (IsNotVisited (f)) then
            ResetCut (f)
            AuxiliaryCutRecur (f, ¬ f.cut)
            AuxiliaryCutRecur (¬ f, f.cut)
            SetCut (f)
            SetVisited (f)
        return
    AuxiliaryCutRecur (f.t, clause ∪ ¬ f.v)
    AuxiliaryCutRecur (f.e, clause ∪ f.v)
    return
```

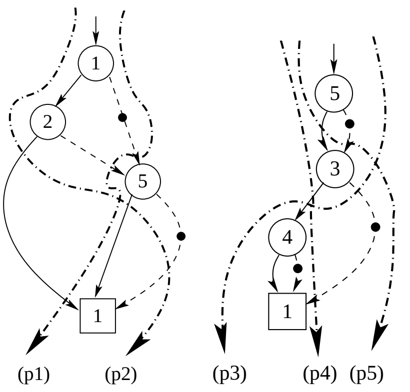
Figure 11: Pseudocode for the Auxiliary-Cut Method.

$(\widehat{x} \equiv g(x))$, and storing them as clauses using the No-Cut strategy. The BDD representing $g(x)$ can be further decomposed recursively. Notice that the procedure does not create any new BDD node. The BDDs for $f(x, \widehat{x})$ and $(\widehat{x} \equiv g(x))$ are not created as they are implicitly generated and visited during the depth-first visit of f. Every time the function encounters a cut point $\widehat{x}$, i.e., procedure $IsACut(f)$ evaluates to true, it prints a clause with the PrintClause procedure. Visiting f till the cut point is equivalent to visiting $f(x, \widehat{x})$. Once this is done, it recurs on f and on $\neg f$. In the first case the clause set is initialized to $\neg f.cut$, and in the second case to $f.cut$, where $f.cut$ is the variable for the cut point $\widehat{x}$. These two recursive calls implicitly generate $(\widehat{x} \equiv g(x))$, i.e., $(\neg\widehat{x} \wedge \neg g(x)) \vee (\widehat{x} \wedge g(x))$

In the pseudocode, as in the previous sections, for each node f, $f.t$ indicates the "then" child, $f.e$ the "else" child, $f.v$ the BDD variable, and $f.cut$ the CNF (or cut point) variable of f.

Example 3. We start from the same BDD used in Fig. 8 (a). Let us suppose function GenerateCut inserts a cut point on node 3, i.e., the only node with two incoming edges. After that, the BDD is *logically* decomposed into two BDDs:

– The first one contains the auxiliary variable to replace the underlying function (the one starting at node 3). This BDD is represented in Fig. 12 (a) where the auxiliary variable is named 5.



(a) (b)

$(p1)$ $(1 \vee \neg 5) \wedge$
$(p2)$ $(\neg 1 \vee 2 \vee 5) \wedge$
$(p3)$ $(5 \vee \neg 3 \vee \neg 4) \wedge$
$(p4)$ $(\neg 5 \vee \neg 3 \vee 4) \wedge$
$(p5)$ $(\neg 5 \vee 3)$

(c)

Figure 12: BDD to CNF Translation: Auxiliary-Cut Algorithm.

– The second one, represented in Fig. 12 (b), is the XNOR of the auxiliary variable just introduced with the BDD that this variable represents.

Both BDDs are visited using the No-Cut strategy to generate CNF clauses. Notice again that neither the BDD in Fig. 12 (a) nor the BDD in Fig. 12 (b) are built: To save memory and computation time the procedure does not produce any new node, but we represent these BDDs to illustrate how the clauses are generated.

8.4 Comparison of the Three Previously Described Methods

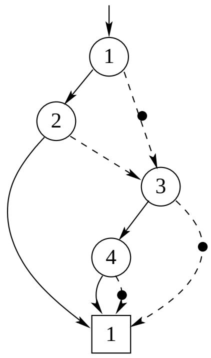
All the methods described in this section can be brought back to the basic idea of possibly breaking the BDD through the use of additional cutting variables and generating a single clause for each path between the root of the BDD, the cutting variables and the terminal nodes. Such internal cutting variables are added always (for each node), never or sometimes, respectively.

While the Single-Node-Cut method minimizes the length of the clauses produced, it also requires more CNF variables than the other two methods. On the contrary, the No-Cut technique minimizes the number of CNF variables required, but, in the worst case, the number of clauses (and literals) produced is exponential in the BDD size (in terms of number of nodes). The application of this method is then limited to the cases in which the off-set of the represented function f has a small cardinality[10]. The Auxiliary-Cut strategy is a trade-off

[10] We define the cardinality of the off-set of a function $f : B^n \rightarrow B$ as he number of points in the Boolean space B^n for which f is false.

between the first two methods, and it is the one which often produces the more compact CNF representation.

Example 4. Fig. 13 shows a final example of how our procedures work to generate the DIMACS CNF format for satisfiability problems.



(a)

```
p cnf 7 11
-5 3 0
-5 4 0
5 -3 -4 0
6 -2 0
6 -5 0
-6 2 5 0
7 1 5 0
-7 1 -5 0
7 -1 -6 0
-7 -1 6 0
7 0
```

(b)

```
p cnf 4 3
1 -3 -4 0
-1 2 -3 4 0
-1 2 3 0
```

(c)

```
p cnf 5 5
-5 1 0
5 -1 2 0
-3 -4 5 0
3 -5 0
-3 4 -5 0
```

(d)

Figure 13: (a) BDD; (b) Single-Node-Cut format; (c) No-Cut format; (d) Auxiliary-Cut format.

Fig. 13 (a) represents the same BDD used in all the previous examples. Fig. 13 (b), (c) and (d) show the corresponding CNF representations generated by our three methods. As in the standard DIMACS CNF format, the problem line is denoted as:

$$p \;\; cnf \quad < \#variables > \;\; < \#clauses >$$

where "p" denotes the problem line, "cnf" means that the file is in CNF format, and $< \#variables >$ and $< \#clauses >$ indicate the total number of variables used (4 is the minimum value as the BDD itself has 4 variables), and the total number of clauses in the instance. This line must appear before any line describing a clause. CNF variables are named after integer numbers ranging from 1 to 4, to respect the format and have a 1-to-1 correspondence with the BDD variables.

As we have already described the No-Cut method does not add any new variable (and uses only the original 4 variables), the number of added variable is equal to 3 (variables 5, 6 and 7) for the Single-Node-Cut method, and equal to 1 (variable 5) for the Auxiliary-Cut method. In this last case, we simply add a variable for each node with an incoming number of edges greater than two as all the clauses are short enough.

As a final remark notice that for this specific example the No-Cut approach is the one which gives the most compact CNF representation but also the clause with the largest number of literals (4).

9 Experimental Results

This section describes the experimental data gathered with our strategy and four publicly available SAT solvers, namely Chaff [Moskewicz et al. 2001], Limmat [Limmat], BerkMin [Goldberg and Novikov 2002], and Siege [Siege, Ryan 2004]. We start this section by describing our experimental setting. We move on by reporting results related with the BDD-to-CNF translation, and the reachability analysis phase. Finally, we analyze the verification results.

9.1 Circuits, Properties and Experimental Setting

For our experiments, we start from Verilog or Blif netlists. From these source files we generate the BMC-CNF formulation of the problem using three publicly available tools (VIS [Brayton et al. 1996], NuSMV [Cimatti et al. 1999], BMC [BMC]) and our home-made software. As far as our package is concerned it is able to generate CNF formulas both from the original network of the circuit and from its transition relation representation. The generated CNF problem is stored as a standard DIMACS CNF file. All our experiments run on a 1.7 GHz Pentium 4 Workstation with 1 GByte main memory, running RedHat Linux 7.1. We present results on both standard benchmarks, i.e., ISCAS'89 and the ISCAS'89–addendum [Brglez et al. 1989], and industrial designs, i.e., the IBM Formal Verification Benchmark Library [IBM Library].

The IBM verification benchmark suite includes 75 circuits in BLIF format with a size ranging from 95 to 917 memory elements. For each circuit only one property is reported in the description. This property specifies that the single output of the circuit has to be an identity. Among *all* BMC problems, we consider the ones requiring more than 1000 seconds of CPU time.

For ISCAS'89 and ISCAS'89–addendum benchmarks we make a similar selection. In this case, invariant properties are generated with the strategy originally adopted in [Cabodi et al. 2002]. Following this approach, target state sets T are sets with increasing *Hamming* distance[11] from the initial state set S.

Example 5. As an example, let us suppose to have a model with 5 state variables $(d_0, d_1, d_2, d_3, d_4)$, and initial state $\mathsf{S} = (0,0,0,0,0)$. We have just one state with Hamming distance 5 from S, i.e, $(1,1,1,1,1)$, we have 6 states at Hamming distance ≥ 4, i.e, $(1,1,1,1,1)$, $(0,1,1,1,1)$, $(1,0,1,1,1)$, $\ldots$, $(1,1,1,1,0)$, and

[11] We compute the Hamming distance between two points p_1 and p_2 in the Boolean space B^n, as the number of bits which assume a different value in p_1 and p_2.

so on. Our invariant property requires the T states to be unreachable. As a consequence, for the state $(1, 1, 1, 1, 1)$, our invariant property is $\neg(d1 \wedge d2 \wedge d3 \wedge d4 \wedge d5)$, while for the state $(0, 1, 1, 1, 1)$ it is $\neg(\neg d1 \wedge d2 \wedge d3 \wedge d4 \wedge d5)$, and so on.

Our experience is that the shorter the Hamming distance from the initial state, the easier it is to falsify the property, whereas most properties generated using high Hamming distances are proved correct (i.e., state space regions distant from initial state are unreachable).

We adopted automatic invariant property generation for two reasons:

1. There is no standard set of properties, and no specific knowledge about the functionality of the ISCAS benchmarks.

2. We aim at measuring the ability of a verifier to explore arbitrary properties in the state space.

Notice that, albeit artificially generated properties may not be as meaningful as designer given ones, this is a common way to cope with the lack of publicly available and suitable benchmarks. For any given property we performed a COI (Cone of Influence) reduction before starting the verification phase.

Tab. 1 contains the description of the circuits and the properties used. # SV is the number of state variables in the model. For each circuit we checked different properties P_i. All the properties selected are false properties, so that we are able to report two separate results with two successive values of the bound: In the first case the property is verified and it passes, while in the second one it is always falsified and fails. The SAT section of the table reports the number of clauses, variables, and literals of the SAT problem generated.

As far as the generation of the CNF formulation of the problem is concerned, NuSMV and VIS produce similar results. BMC generates CNF files on average 20-30% more compact both in terms of clauses and (intermediate) variables. Our tool is somehow in between and starting from the BDD representation of the circuit can sometimes give some advantage. Tab. 2 reports some evidence on that. More specifically, the four tools, i.e., VIS, NuSMV, BMC, and our home-made generator, are used to generate the BMC problem of length 1 for the property P_1 of circuit s35932. Note that the size of the CNF formulation of the BMC problem, grows practically linearly with the value of the bound as it can be deduced by comparing the data of Tab. 2 with the corresponding one of Tab. 1.

As a final remark, notice that the BMC tool does not store the variable coding used along each time frame. As we need this information to be congruent with the data coming from our reachability analysis, we do not report further experiments with the BMC tool and we always rely on the other three to generate the CNF problem.

Model	# SV	Property		Bound	SAT Problem # Clauses [×10^3]	# Vars [×10^3]	# Lits [×10^3]
s1512	57	P_1	Pass	66	92	34	223
			Fail	67	94	35	226
		P_2	Pass	130	181	68	439
			Fail	131	183	68	442
		P_3	Pass	259	381	141	917
			Fail	260	383	142	920
s9234	211	P_1	Pass	80	414	148	1002
			Fail	81	419	150	1015
		P_2	Pass	240	1190	427	2886
			Fail	241	1195	429	2898
		P_3	Pass	240	1192	428	2890
			Fail	241	1197	430	2902
s15850.1	534	P_1	Pass	75	1057	387	2572
			Fail	76	1071	391	2607
s13207.1	638	P_1	Pass	55	465	188	1151
			Fail	56	474	191	1172
		P_2	Pass	109	920	369	2278
			Fail	110	929	372	2299
s35932	1728	P_1	Pass	31	1495	537	3630
			Fail	32	1543	554	3747
31_1_batch_2	123	P_1	Pass	22	264	91	615
			Pass	25	300	103	698
20_batch	148	P_1	Pass	29	421	145	980
			Fail	30	435	150	1014
13_batch_1	179	P_1	Pass	20	718	242	1674
22_batch	191	P_1	Pass	40	722	248	1681
11_batch_2	296	P_1	Pass	35	866	297	2016
26_batch	657	P_1	Pass	100	3283	1144	7637

Table 1: Analyzed Properties. # SV indicates the number of state variables in the model.

Tool	# Clauses [×10^3]	# Vars [×10^3]	# Lits [×10^3]	File Size [kB]
VIS	53	21	120	1127
NuSmv	53	21	129	1121
BMC	37	15	87	804
Our Tool	39	18	94	857

Table 2: Comparison of different tools used to generate the CNF formulation of the BMC problem: Circuit s35932, property P_1, bound 1.

9.2 BDD-to-CNF Transformation

In this section, we report some data regarding the BDD-to-CNF transformation described previously. Tab. 3 reports data on circuit s35932 using BDDs representing its transition relation and its reachable state sets.

The DDDMP tool is used for all the experiments. It is both publicly available [Cabodi and Quer, Cabodi et al. 1996] and distributed within the Colorado University Decision Diagram (CUDD) package.

Within Tab. 3 the meaning of the columns is the following. Column # Nodes indicates the number of nodes of the BDD used as a starting point for the trans-

	BDDs			CNF				
	# Nodes	Mem [kB]	File Size [kB]	D.M.	# Clauses [$\times 10^3$]	# Vars [$\times 10^3$]	# Lits [$\times 10^3$]	File Size [kB]
1	4764	76	75	S	14	6	33	282
				N	5	2	12	143
				A	5	2	15	164
2	10039	161	163	S	37	13	105	794
				N	20	3	120	733
				A	20	7	62	504
3	75785	1213	159	S	231	78	551	3856
				N	*ovf*	*ovf*	*ovf*	*ovf*
				A	93	8	306	1864
4	339668	5435	461	S	1038	341	2471	18107
				N	*ovf*	*ovf*	*ovf*	*ovf*
				A	418	32	1364	8112

Table 3: Performance of the routine BDD2CNF for the next state functions and a few reachable state sets of circuit s35932. D.M. represents the storing techniques: "S" indicates the Single-Node-Cut, "N" the No-Cut, and "A" the Auxiliary-Cut method.

formation. Mem reports the memory used by the CUDD package to represent this BDD (considering only the space taken by the node of the BDD, and not the extra-space necessary for other data structures, such as the computed tables, etc.). File Size indicates the space required to store the BDDs using the binary (compressed) format available within the DDDMP package. These data may give a better idea of the compactness/efficiency of the CNF format. D.M. represents the method we used to store the BDDs as CNF formulas ("S" stands for the Single-Node-Cut method, "N" for the No-Cut and "A" for the Auxiliary-Cut). The subsequent columns report statistics (i.e., number of clauses, variables, literals, and memory used) on the CNF representation of the BDD.

ovf indicates that the generated file, containing the CNF formulation of the problem, was too large to be appropriately managed by the operating system (i.e., was larger than 1 GBytes).

It is evident from the table that the Auxiliary-Cut method usually gives the best performance in terms of CNF compactness. On the contrary the No-Cut method has to be used with extreme care, even if it can produce compact representation when the off-set of the function has a small cardinality.

9.3 Approximate Reachability Analysis

After the previous phase, we generate the set of approximate reachable state sets for the circuit. For this operation we use both the VIS tool, and again a home-made tool called FBV (Forward Backward Verifier). Albeit VIS implements almost all the approximate traversal algorithms presented in the literature, we need the overapproximation of the reachable state set at the same bound level for all sub-FSMs. Our tool, implemented on top of the Colorado University Decision Diagram (CUDD) package, implements the approximation verification method presented in Section 6. The different thresholds introduced during the description of the algorithms are essentially selected by the user, although some automatic tuning is dynamically performed (see for example Section 6.3).

Model	Property	Modified SAT+BDD Problem D.M.	# Clauses [%]	# Vars [%]	# Lits [%]
s1512	P_1 Pass	A	+21.7	+7.5	+72.1
	Fail	A	+22.0	+7.6	+73.2
	P_2 Pass	A	+53.4	+20.0	+168.5
	Fail	A	+53.4	+20.0	+169.4
	P_3 Pass	A	+115.0	+40.0	+317.7
	Fail	A	+115.3	+40.0	+318.6
s9234	P_1 Pass	S	+14.6	+13.4	+14.5
	Fail	S	+14.6	+13.4	+14.5
	P_2 Pass	S	+3.8	+0.3	+21.7
	Fail	S	+3.8	+0.3	+21.7
	P_3 Pass	S	+24.3	+0.5	+220.2
	Fail	S	+24.2	+0.5	+219.9
s15850.1	P_1 Pass	A	+2.0	+0.4	+13.0
	Fail	A	+2.0	+0.4	+13.0
s13207.1	P_1 Pass	A	+21.8	+5.4	+76.7
	Fail	A	+21.8	+5.4	+76.6
	P_2 Pass	A	+21.2	+5.3	+74.5
	Fail	A	+21.2	+5.3	+75.2
s35932	P_1 Pass	A	+4.4	+1.0	+33.9
	Fail	A	+4.3	+1.0	+33.2
31_1_batch_2	P_1 Pass	A	+15.0	+6.0	+32.6
	Pass	A	+13.4	+5.4	+29.1
20_batch	P_1 Pass	A	+0.6	+0.3	+1.1
	Fail	A	+0.7	+0.3	+1.3
13_batch_1	P_1 Pass	A	+1.3	+0.6	+2.7
22_batch	P_1 Pass	A	+1.4	+0.6	+2.7
11_batch_2	P_1 Pass	A	+1.8	+0.8	+3.4
26_batch	P_1 Pass	A	+21.8	+20.8	+23.4

Table 4: Statistics for the produced SAT problems. D.M. represents the storing techniques: "S" indicates the Single-Node-Cut, "N" the No-Cut, and "A" the Auxiliary-Cut method. All datas indicate the percentage increment over the original SAT problem formulation reported in Tab. 1.

Once we have generated the BDDs for the overapproximation, we convert them into CNF following the methodology reported in Section 8, and we generate the final problem with this new space restriction, as outline in Section 4. Tab. 4 reports statistics on the CNF problems generated in this way, i.e., it indicates the size increase of the new CNF formulation over the size of the original problem (reported in Tab. 1).

As previously described, we have to trade-off between the accuracy of the result and its efficacy to prune the SAT solver search space. As a consequence, we have to control the size of the out-coming CNF problem formulation. Tab. 4

reports for each experiment the result that leads to the best trade-off. A more sophisticated analysis of the impact of the accuracy of the estimate on the SAT solver performance is reported in the following section. Notice that our strategy works at its best also if we produce a modest (at least compared with conflict analysis) increment in the number of clauses, varying from 2–3% to about 100%. As a consequence, the increment in the clause database size is not a bottleneck of the procedure. Moreover, as previously introduced, the strategy Auxiliary-Cut usually gives the best performance and it is the strategy we use as a default.

9.4 SAT Solvers

We present here the core part of our experimental section, i.e, the results obtained with Chaff (both MChaff and ZChaff), Limmat, BerkMin (version 561) and Siege (version v4).

Chaff, by Moskewicz et al. [Moskewicz et al. 2001], is a complete DPLL deterministic solver. It is carefully engineered with non-chronological backtracking, learning (clause recording), restarts, randomized branching heuristic and an innovative notion of heuristic learning (VSIDS).

Limmat, by Biere [Limmat], is a ZChaff–like SAT solver with an early detection of conflicts in the BCP queue, a constant time lookup of the other watched literal, and an optimized ordering of decision variables and a robust code through sophisticated test framework.

BerkMin, by Goldberg and Novikov [Goldberg and Novikov 2002], inherits such features of Grasp, Sato, and Chaff as clause recording, fast BCP, restarts, and conflict clause aging. At the same time BerkMin introduces a new decision making procedure and a new management of the database of the conflict clauses. One of its key novelties is that this database is organized as a chronologically sorted stack, i.e., the algorithm always tries to satisfy clauses at the top and to remove clauses at the bottom.

Siege, by Ryan [Siege,Ryan 2004], is a recent SAT solver which improved over previous version using a more powerful resolution strategy [Berre and Simon 2003].

Tab. 5 reports runs with the Chaff SAT engine on the two problem instances, i.e., the original problem formulation and the one generated by merging with it the information coming from the reachability analysis phase. We use both the MChaff and the ZChaff versions. On our verification instances there is no clear winner among the two engines. As a consequence, to be conservative, we always present results with the faster tool among the two on the original problem instance. In all the cases Chaff is run with the default settings. In Tab. 5 # Decs and # Confl. represent the total number of decisions taken and conflicts produced by the SAT solver. Mem. indicates the maximum amount of memory (in MB) necessary to complete the entire verification task. As the memory requirement of

Model	Property		Original SAT Problem # Decs [$\times 10^3$]	# Confl. [$\times 10^3$]	Mem. [MB]	Time [s] Search	Modified SAT+BDD Problem # Decs [$\times 10^3$]	# Confl. [$\times 10^3$]	Mem. [MB]	Time [s] Setup	Time [s] Search	Time [s] Total	Speedup
s1512	P_1	Pass	106	57	28	74	3	1	17	2	2	4	18.5
		Fail	155	97	28	169	20	1	17	2	5	7	24.1
	P_2	Pass	566	318	85	899	157	73	51	5	104	109	8.3
		Fail	707	415	84	1326	149	59	50	5	76	81	16.4
	P_3	Pass	3469	2018	172	11330	415	163	146	19	390	409	17.7
		Fail	4502	2579	299	18677	609	230	97	19	603	622	30
s9234	P_1	Pass	167	133	123	517	23	10	95	14	27	41	12.6
		Fail	275	221	122	950	35	14	71	14	45	59	16.1
	P_2	Pass	–	–	ovf	–	353	221	186	74	1100	1174	∞
		Fail	–	–	ovf	–	924	172	125	67	600	667	∞
	P_3	Pass	–	–	ovf	–	486	287	212	14	2818	2832	∞
		Fail	–	–	ovf	–	1996	1247	245	14	18465	18479	∞
s15850.1	P_1	Pass	25	20	175	184	8	6	143	53	35	88	2.1
		Fail	475	120	140	1227	173	43	141	53	273	326	3.8
s13207.1	P_1	Pass	22	11	55	22	24	16	74	54	53	107	0.4
		Fail	641	83	46	375	758	35	59	54	116	170	2.2
	P_2	Pass	315	230	193	4287	98	51	133	96	176	272	15.8
		Fail	640	358	158	3210	2348	109	115	96	432	528	6.1
s35932	P_1	Pass	1	1	205	9	1	1	205	0	9	9	1
		Fail	206	40	201	1156	119	25	181	175	760	935	1.2
31_1_batch_2	P_1	Pass	460	239	156	3474	36	9	26	312	30	342	10.2
		Pass	881	453	164	ovf	85	22	51	312	124	436	∞
20_batch	P_1	Pass	204	56	99	653	206	53	101	61	698	759	0.9
		Fail	396	125	106	1859	277	60	104	61	735	796	2.3
13_batch_1	P_1	Pass	893	676	138	ovf	30	5	54	28	16	42	∞
22_batch	P_1	Pass	1270	170	209	2641	962	109	185	424	1463	1887	1.4
11_batch_2	P_1	Pass	684	83	160	1415	402	46	133	280	577	857	1.7
26_batch	P_1	Pass	2001	11	485	1831	1302	9	290	657	197	854	2.1

Table 5: Comparison between output statistics produced by Chaff for the original CNF problem and the one incremented with symbolic BDD-based information. Memory limit: 1 GBytes. Time limit: 8 hours for ISCAS'89 benchmarks, and 2 hours for the IBM Formal Verification Benchmarks Library. ovf indicates (memory or time) overflow.

the symbolic reachability analysis phase is always smaller, these numbers always coincides with the memory used by the SAT solver. Time indicates the CPU time (in seconds) taken by the BDD, the SAT tools, or both. The Setup is the one to perform the reachability analysis phase and the one to generate the new problem formulation. As introduced in Section 5 one single BMC step includes two steps of traversal, the first one forward and the second one backward. The forward step is the more expensive one, and produces strong space restrictions. The backward step appears to have a constraint effect which decreases as long as the backward reachability analysis moves from T to S. It effects is then more remarkable within the first 4–5 steps of the backward visits. We typically dedicate from 70% to 90% of our traversal time to perform the forward analysis and the remaining 10–30% for the backward phase. Speedup indicates the ratio between the column Search Time of the SAT side of the table and the column Total Time of the SAT+BDD side. Tab. 5 shows a reduction in terms of verification time ranging from a factor of 2 to a factor of 30. Memory usage is also reduced. Although BDD information increases the number of total literals by adding temporary variables to the original CNF problem, also the number of decisions and conflicts is smaller in the SAT+BDD side of Tab. 5 than in the SAT side.

BDD		SAT						
# Nodes [$\times 10^3$]	Time [sec]	# Clauses [%]	# Vars [%]	# Lits [%]	# Decs [%]	# Confl. [%]	Mem. [%]	Time [%]
7	55	+12.9	+3.4	+19.2	+13.4	+29.2	+32.4	+39.4
8	80	+13.0	+3.4	+19.1	+11.2	+19.2	+26.3	+27.5
9	68	+17.1	+4.9	+28.7	+14.9	+11.4	+13.8	+19.8
15	80	+35.3	+11.2	+58.3	+10.7	+8.4	+6.2	+23.2
15	*96*	*+21.2*	*+5.3*	*+74.5*	*0*	*0*	*0*	*0*
16	101	+36.8	+11.7	+61.6	+9.2	+6.9	+6.3	+9.3
26	239	+40.7	+15.5	+52.3	+5.1	+3.3	+6.3	+10.3
71	434	+71.7	+22.0	+149.7	+11.0	+14.7	+2.9	+12.9
75	286	+50.0	+14.4	+92.7	+12.4	+22.3	+18.4	+16.4
29	411	+49.1	+15.1	+90.6	+15.9	+48.6	+16.7	+16.9
47	357	+55.3	+17.2	+108.9	+27.2	+38.9	+45.2	+35.4
47	362	+55.6	+17.4	+110.0	+47.4	+110.0	+22.9	+21.5
98	739	+144.8	+40.4	+410.1	+49.4	+90.1	+21.0	+29.1

Table 6: Dependency of the SAT solver's results from the accuracy of the over-estimated reachability analysis. Circuit s13207.1, property P_2, Pass case. Datas indicate the percentage increment over the approximation used in Tab. 5 and reported in this table in italics.

Tab. 6 studies the impact of the accuracy of the reachable state on the size of the CNF problem generated. We concentrate on circuit s13207.1 and property P_2, Pass case. We present data with increasing accuracy, and size, of the reachable state set estimate. All the data are given as a percentage variation with respect to the case reported within the SAT+BDD section of Tab. 5 and reported in italics in this table.

For coarse estimates (top part of the table) the reachability analysis phase tends to be useless, i.e., it does not help to prune the SAT solver's search. For precise estimates (bottom part of the table) it becomes inappropriate, i.e., too expensive compared with the SAT solver's costs. As a result, memory and time performance degraded toward the top and the bottom of the table. For the central part, i.e., not too coarse or too precise estimates, Tab. 6 indicates a quite flat behavior.

We also tried to guide the SAT solver variable selection, in the Variable Decision phase, using the variable order adopted to represent BDDs during traversal. This order includes all present and next variables (of a single time frame) of the model. From it we heuristically deduce a variable order including all state and auxiliary (generated during the BDD-to-CNF transformation) variables and feed it to the SAT solver. Also in this case we had slight variations in the SAT engine performances, in the order of about 10%. This conclusion seems somehow in contrast with what other researchers discovered and it is possibly motivated by our limited analysis. For this reason we do not report evidence on that.

To run experiments with Limmat [Limmat], we slightly modified the tool in the following way. As described in the previous sections, our final CNF problem formulation is obtained by mixing the original BMC problem with information coming from BDDs. This process implies the introduction of auxiliary or cut-point variables, added while converting BDDs to CNF formulas (see Section 8). With Limmat we discovered that any decision on cut-point variables, i.e., non-original problem variables, has a wrong impact on the entire satisfiability process. As a consequence, we forced the tool to use a variable order (for the decision phase) in which cut point variables may only be implied and may never be selected. This improved the performance of the tool on our benchmark. Limmat proved to be sometimes faster and sometimes slower than Chaff, with no clear winner among the two. For that reason we do not report any evidence with it.

Tables 7 and 8 report runs with BerkMin [Goldberg and Novikov 2002] (version 561) and Siege [Siege, Ryan 2004] (version v4) respectively.

As these tools proved to be more efficient than the other two SAT solvers used for our experiments, we report only the verification instances which still are hard-enough to motivate the application of our technique. We do not report the memory used by Siege as the tool does not report its exact memory usage, and we did not have access to the source code. Anyhow, some rough measurements indicate that Siege's memory requirements appear to be similar to the ones obtained with BerkMin. As far as the statistics reported in the table are concerned, Siege seems to perform better than BerkMin on our set of experiments. It takes fewer decisions, encounter fewer conflicts, and it is a little faster both on the original SAT than on the modified SAT+BDD problem. For both the tools we could reduce the verification time up to about a 15.6×.

Model	Property		Original SAT Problem				Modified SAT+BDD Problem						Speedup
			# Decs $[\times 10^3]$	# Confl. $[\times 10^3]$	Mem. [MB]	Time [s] Search	# Decs $[\times 10^3]$	# Confl. $[\times 10^3]$	Mem. [MB]	Time [s] Setup	Search	Total	
s1512	P_3	Pass	4635	464	188	2223	3449	244	168	19	741	760	2.9
		Fail	4813	525	216	2795	3084	234	194	19	881	900	3.1
s9234	P_2	Pass	1880	537	117	2989	364	48	96	74	253	327	9.1
		Fail	2736	888	129	6348	501	88	102	67	339	406	15.6
	P_3	Pass	1545	398	197	1802	471	100	139	14	400	414	4.4
		Fail	1795	473	216	2208	582	135	187	14	593	607	3.6
s13207.1	P_2	Pass	234	145	102	345	217	115	95	76	114	190	1.8
		Fail	246	168	119	295	196	106	99	76	98	174	1.7

Table 7: Comparison between output statistics produced by BerkMin for the original CNF problem and the one incremented with symbolic BDD-based information. Memory limit 1 GBytes.

Model	Property	Original SAT Problem			Modified SAT+BDD Problem					Speedup
		# Decs	# Confl.	Time [s]	# Decs	# Confl.	Time [s]			
		[$\times 10^3$]	[$\times 10^3$]	Search	[$\times 10^3$]	[$\times 10^3$]	Setup	Search	Total	
s1512	P_3 Pass	2366	242	682	469	48	24	110	134	5.1
	Fail	2292	235	641	895	101	24	244	268	2.4
s9234	P_2 Pass	688	297	1164	290	30	8	92	100	11.6
	Fail	659	309	917	356	88	9	214	223	4.1
	P_3 Pass	792	337	1226	494	166	45	470	515	2.4
	Fail	538	212	558	490	164	45	444	489	1.1
s13207.1	P_2 Pass	740	113	458	129	43	78	88	166	2.8
	Fail	810	107	365	2131	63	78	302	380	1.0

Table 8: Comparison between output statistics produced by Siege for the original CNF problem and the one incremented with symbolic BDD-based information. Memory limit 1 GBytes.

10 Conclusions

In this paper we dovetail two of the most widely used techniques within formal verification: BDD-based symbolic reachability analysis and SAT-based Bounded Model Checking.

We proposed to exploit inexpensive BDD-based symbolic approximate forward and/or backward reachability analysis to restrict the overall search space of a SAT solver engine. We develop specific strategies to appropriately mix BDD and SAT efforts, and to efficiently convert BDD-based symbolic state set representations into SAT-oriented ones. We showed experimentally the potential of the BDD-based symbolic reachability estimate to reduce the number of decisions taken and conflicts discovered by four state-of-the-art SAT tools.

Experiments shows a reduction in the number of conflicts up to more than a 100×. This reduction has a direct beneficial effect on the memory and time used by the SAT-tool to solve the verification problem. We could reduce the verification time up to a 30×, with an average speedup of about 8×.

References

[Berre and Simon 2003] D. Le Berre and L. Simon. Results from the SAT'03 SAT Solver Competition. In Enrico Giunchiglia and Armando Tacchella, editors, *The Sixth International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, S. Margherita Ligure - Portofino, Italy, May 2003. Springer-Verlag.

[BMC] A. Biere. Bounded Model Checker: BMC, http://www-2.cs.cmu.edu/-modelcheck/bmc.html/.

[Limmat] A. Biere. Limmat SAT Solver, http://www.inf.ethz.ch/personal/biere/-projects/limmat/.

[Biere et al. 1999] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking using SAT procedures instead of BDDs. In *Proc. 36th Design Automat. Conf.*, pages 317–320, New Orleans, Louisiana, June 1999.

[Biere and Kunz 2002] A. Biere and W. Kunz. SAT and ATPG: Boolean Engines for Formal Hardware Verification. In *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, November 2002.

[Bjesse et al. 2001] P. Bjesse, T. Leonard, and A. Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In Gérard Berry, Hubert Comon, and Alan Finkel, editors, *Proc. Computer Aided Verification*, volume 2102 of *LNCS*, pages 454–464, Paris, France, July 2001. Springer-Verlag.

[Brglez et al. 1989] F. Brglez, D. Bryan, and K. Koźmiński. Combinatorial Profiles of Sequential Benchmark Circuits. In *Proc. IEEE ISCAS'89*, pages 1929–1934, May 1989.

[Bryant 1986] R. E. Bryant. Graph–Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, C–35(8):677–691, August 1986.

[Cabodi et al. 1994] G. Cabodi, P. Camurati, and S. Quer. Efficient State Space Pruning in Symbolic Backward Traversal. In *Proc. Int'l Conf. on Computer Design*, pages 230–235, Cambridge, Massachussetts, October 1994.

[Cabodi et al. 1996] G. Cabodi, P. Camurati, and S. Quer. Improved Reachability Analysis of Large Finite State Machine. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 354–360, San Jose, California, November 1996.

[Cabodi et al. 2002] G. Cabodi, P. Camurati, and S. Quer. Can BDDs compete with SAT solvers on Bounded Model Checking? In *Proc. 39th Design Automat. Conf.*, New Orleans, Louisiana, June 2002.

[Cabodi et al. 2003] G. Cabodi, S. Nocco, and S. Quer. Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals. In *Proc. Design Automation & Test in Europe Conf.*, pages 898–903, Munich, Germany, March 2003.

[Cabodi and Quer] G. Cabodi and S. Quer. URL: http://staff.polito.it/-{gianpiero.cabodi,stefano.quer}/.

[Cerny et al. 1986] E. Cerny and M. A. Marin. An approach to unified methodology of combinational switching circuits. *IEEE Trans. on Computers*, C–26(8):745–756, August 1986.

[Cho et al. 1996] H. Cho, G. D. Hatchel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for Approximate FSM Traversal Based on State Space Decomposition. *IEEE Trans. on Computer-Aided Design*, 15(12):1465–1478, December 1996.

[Cimatti et al. 1999] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifyer. In *Proc. Computer Aided Verification*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, July 1999.

[Coudert et al. 1989] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Function Vectors. In *Proc. IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, volume 1, pages 111–128, November 1989.

[Damiano and Kukula 2003] R. Damiano and J. Kukula. Checking Satisfiability of a Conjunction of BDDs. In *Proc. 40th Design Automat. Conf.*, pages 818–823, Anaheim, CA, 2003.

[Davis and Putnam 1960] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[Brayton et al. 1996] R. K. Brayton et al. VIS. In Mandayam Srivas and Albert Camilleri, editors, *Proc. Formal Methods in Computer-Aided Design*, volume 1166 of *LNCS*, pages 248–256, Palo Alto, California, November 1996. Springer-Verlag.

[Franco et al. 2003] J. Franco, M. Kouril, J. S. Schlipf, J. Ward, S. Weaver, M. Dransfield, and W. M. Vanfleet. SBSAT: A State-based BDD-based Applications of Satisfiability Testing. In *Proc. 6th International Conference on the Theory and Application of Satisfiability Testing*, pages 151–158, S. Margherita, Italy, 2003.

[Goldberg and Novikov 2002] E. Goldberg and Y. Novikov. BerkMin: a Fast and Robust SAT-Solver. In *Proc. Design Automation & Test in Europe Conf.*, pages 142–

149, Paris, France, February 2002.

[Gopalakrishnan et al. 2003] S. Gopalakrishnan, V. Durairaj, and P. Kalla. Integrating CNF and BDD Based SAT Solvers. In *IEEE International High-Level Design Validation and Test Workshop (HLDVT'03)*, November 2003.

[Govindaraju and Dill 1998a] S. G. Govindaraju and D. L. Dill. Verification by Approximate Forward and Backward Reachability. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 366–370, San Jose, California, November 1998.

[Govindaraju and Dill 1998b] S. G. Govindaraju, D. L. Dill, A. Hu, and M. A. Horowitz. Approximate Reachability Analysis with BDDs using Overlapping Projections. In *Proc. 35th Design Automat. Conf.*, pages 451–456, San Francisco, California, June 1998.

[Gupta et al. 2003a] A. Gupta, M. Ganai, C. Wang, A. Yang, and P. Ashar. Learning from BDDs in SAT-based Bounded Model Checking. In *Proc. 40th Design Automat. Conf.*, pages 824–829, Anaheim, CA, 2003.

[Gupta et al. 2003b] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Abstraction and BDDs Complement SAT–Based BMC in *Diver.* In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proc. Computer Aided Verification*, volume 2725 of *LNCS*, pages 206–209, Boulder, CO, USA, July 2003. Springer-Verlag.

[Gupta et al. 2000] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT–Based Image Computation with Application in Reachability Analysis. In *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, Austin, TX, USA, 2000.

[Gupta et al. 2001] A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik. Partition–Based Decision Heuristic for Image Computation using SAT and BDDs. In *Proc. Int'l Conf. on Computer-Aided Design*, San Jose, California, November 2001.

[Jin and Somenzi 2004] H. Jin and F. Somenzi. CirCUs: A Hybrid Satisfiability Solver. In *The Seventh International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, Canada, May 2004.

[Kautz and Selman 2003] H. Kautz and B. Selman. Ten Challenges Redux: Recent Progress in Propositional Reasoning and Search. In F. Rossi, editor, *Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*. Springer-Verlag, September 2003.

[Kuehlmann et al. 2001] A. Kuehlmann, M. K. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proc. Design Automat. Conf.*, Las Vegas, Nevada, June 2001.

[IBM Library] IBM Formal Verification Benchmark Library. http://www.haifa.il.ibm.com/projects/verification/rb_homepage/fv-benchmarks.html.

[McMillan 1994] K. McMillan. *Symbolic Model Checking.* Kluwer Academic, Boston, Massechusset, 1994.

[McMillan 2002] K. L. McMillan. Applying SAT Methods in Unbounded Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. Computer Aided Verification*, volume 2404 of *LNCS*, pages 250–264, Cophenagen, Denmark, 2002.

[McMillan 2004] K. L. McMillan. An Interpolating Theorem Prover. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 16–30, Barcelona, Spain, March 2004. Springer-Verlag.

[Moon et al. 1998] I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan, and C. Pixley. Approximate Reachability Don't Cares for CTL Model Checking. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 351–358, San Jose, California, November 1998.

[Moskewicz et al. 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th Design Automat. Conf.*, Las Vegas, Nevada, June 2001.

[Ranjan et al. 1995] R. K. Ranjan, A. Aziz, R. K. Brayton, B. Plessier, and C. Pixley.

Efficient BDD Algorithms for FSM Synthesis and Verification. In *Proc. Int'l Workshop on Logic Synthesis*, Lake Tahoe, California, May 1995.

[Siege] L. Ryan. Siege SAT Solver, http://www.cs.sfu.ca/∼loryan/personal/.

[Ryan 2004] L. Ryan. Efficient Algorithms for Clause Learning SAT Solvers. Master's thesis, Simon Fraser University, February 2004.

[Sheeran et al. 2000] M. Sheeran, S. Singh, and G. Stålmarck. Checking Safety Properties Using Induction and SAT Solver. In W. A. Hunt and S. D. Johnson, editors, *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 108–125. Springer-Verlag, November 2000.

[Zhang and Malik 2002] L. Zhang and S. Malik. The Quest for Efficient Boolean Satisfiability Solvers. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. Computer Aided Verification*, volume 2404 of *LNCS*, pages 17–36, Cophenagen, Denmark, 2002.