# SAT-Based Verification Methods and Applications in Hardware Verification

Aarti Gupta, Malay K. Ganai, and Chao Wang

NEC Laboratories America,
Princeton, USA
{agupta, malay, chaowang} at nec-labs dot com

**Abstract.** Verification methods based on *Boolean Satisfiability (SAT)* have emerged as a promising alternative to BDD-based symbolic model checking methods. This paper provides a tutorial on various SAT-based verification methods we have developed for verifying large hardware designs. We focus separately on methods for finding bugs and for finding proofs for correctness properties, along with highlighting the many common themes that benefit these methods. We also describe practical experiences with these methods implemented in our verification platform called *VeriSol* (formerly *DiVer*), which has been used successfully in industry practice.

## 1 Introduction

With the growing size and complexity of hardware designs, functional verification has become a bottleneck in the hardware and system development cycle. The cost of detecting bugs late in the design cycle is very high, both in terms of design re-spins and time lost to market. Simulation continues to be the primary workhorse for functional verification, primarily due to its scalability in performing dynamic analysis for a given testcase. However, its main problems are the prohibitive cost of exhaustive coverage and the practical difficulty in assessing good coverage. Formal verification techniques provide a complementary benefit to simulation, where static analysis is performed on a formal mathematical model of the given design, to check its correctness with respect to a given specification under all possible input scenarios. This provides exhaustive coverage, without any testcases, but at the expense of higher complexity of analysis.

Model checking is a formal verification technique for property checking, in which the design is typically modeled as a labeled state transition system, the correctness property is specified as a temporal logic formula, and verification is performed by checking whether the formula is true in the model provided by the design [1]. The practical application of model checking is limited by the state explosion problem, i.e. the state space to be searched grows exponentially with the number of state components in the design. Symbolic model checking techniques [2, 3] typically use methods based on Binary Decision Diagrams (BDDs) [4] to symbolically manipulate sets of states and transitions without explicit enumeration. Though this improves scalability to some degree, these techniques are unable to handle the large problems encountered in current industrial practice, mainly due to an explosion in memory requirements for BDDs. As an alternative, verification methods based on *Boolean Satisfiability (SAT)* have emerged as a promising solution. The success of SAT-based verification methods is due primarily to the many recent advances in SAT-solvers [5-8] that have enabled significantly larger problems to be solved in the last decade than ever before [9].

This paper provides a tutorial on the SAT-based verification methods we have developed for the purpose of verifying large scale industry designs. We focus mainly on property checking and model checking methods. (Sequential equivalence checking can be formulated in terms of model checking, and most of the methods described here can apply equally well in that context.) We describe the main technical ideas in relationship to SAT, along with pointers to related work. For a more comprehensive survey on SAT-based verification methods, the interested reader is referred to [10]; a discussion of SAT-based combinational equivalence checking techniques can be found in [11].

Our main classification is based on whether the methods are used for finding bugs (falsification) or for finding proofs (verification)[1]. In the former category, we discuss primarily Bounded Model Checking (BMC) [12] and its enhancements [13-15], while the latter category includes methods based on induction [16, 17], Unbounded Model Checking (UMC) [18-21], and proof-based iterative abstraction [22, 23]. In our experience, both falsification and verification methods are required for handling hardware verification problems in an industry setting. Furthermore, there are many common themes that benefit methods in both categories. In particular, we highlight the benefits of using an efficient circuit representation that supports on-the-fly simplifications [24]. We also exploit a hybrid SAT solver [25] that combines the benefits of circuit-based and CNF-based SAT solving techniques. It is useful for the circuit representation to also support symbolic manipulation through BDDs, and we describe many methods that combine SAT and BDDs to derive their complementary benefits in significantly enhancing the performance or scalability of the overall application. Another common theme is the use of bounded unrolling of the design (also called time frame expansion in related work on automatic test pattern generation [26]) to cast the associated problem as a SAT problem. This has been utilized by many falsification as well as verification methods. We use this feature most effectively for handling embedded memories, where our Embedded Memory Modeling (EMM) techniques abstract out the explicit memory elements, but add the data forwarding and other memory modeling constraints at each step of the unrolling [27]. This avoids the state space explosion of modeling the memory state, while preserving the accuracy of verification. Again, the EMM techniques have been used in both falsification [27] as well as verification methods [28].

We also describe practical experiences with these methods implemented in our verification platform called *VeriSol* (formerly *DiVer*), which has been used successfully in industry practice for the last four years to verify large hardware designs [29]. In addition to verifying correctness properties specified by a user on

[1] Although these categories correspond quite naturally to *bounded* and *unbounded* verification, respectively, we would like to avoid overloading the term *bounded*, since many verification methods we describe here use bounded design unrolling for unbounded verification.

RTL (Register Transfer Level) designs, it has also been used for checking automatically generated properties. Specifically, *VeriSol* has been integrated within a high level behavioral synthesis system called *Cyber* [30]. The *Cyber* system automatically generates RTL designs from high-level behavioral descriptions. In addition, it automatically generates correctness properties for these RTL designs also. The back-end property checking for *Cyber* can be performed by *VeriSol*. We have also added the capability of generating automatic correctness properties for typical Verilog RTL designs.

More recently, we have also used many of our SAT-based verification methods for performing the back-end verification in a system called *F-Soft*, which is targeted for verifying software programs [31, 32]. The *F-Soft* verification platform combines several recent advances in formal verification, including SAT-based verification, static analyses, and predicate abstraction. Basically, we accurately model the program behavior as a finite state symbolic model (under assumptions of finite data and control), use static analyses to reduce the size of the verification model, and perform back-end model checking using *VeriSol*. Although we will not describe the software verification applications in this paper, it is interesting to note that by adding several customized heuristics for software, our SAT-based methods in *VeriSol* could be applied successfully for verifying models generated from software programs.

Finally, we would like to note that there has been a lot of interest in applying SAT solver techniques to decision procedures for richer logics, such as quantifier-free fragments of first order logic [33, 34] . In our own work, we have proposed new solvers for difference logic, called SLICE [35] and SDSAT [36]. Since SAT solvers and such decision procedures provide essential components of many theorem provers, the SAT-based verification methods described here can be naturally extended to provide a bridge from model checking to theorem proving applications.

The paper is organized as follows. Section 2 briefly reviews the terminology and background for the basic SAT algorithms and model checking techniques. Section 3 describes the SAT-based verification methods, while their applications for hardware verification in a practical industry setting are described in Section 4. Finally, we conclude in Section 5.

## 2 Background

### 2.1 Boolean Satisfiability (SAT) Problem

The Boolean Satisfiability (SAT) problem is a well-known constraint satisfaction problem, with many applications in the fields of VLSI Computer-Aided Design (CAD) and Artificial Intelligence. Given a propositional formula, the Boolean Satisfiability problem is to determine whether there exists a variable assignment under which the formula evaluates to true. The SAT problem is known to be NP-Complete [37]. In practice, there has been tremendous progress in SAT solver technology over the years, summarized in a survey [9].

Most SAT solvers use a Conjunctive Normal Form (CNF) representation of the Boolean formula. In CNF, the formula is represented as a conjunction of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that in order for the formula to be satisfied, each clause must also be satisfied, i.e., evaluate to true. A Boolean circuit can be encoded as a satisfiability equivalent CNF formula [38]. Alternatively, for SAT applications arising from the circuit domain, the SAT solver may be modified to work directly on the Boolean circuit representation.

```
SAT_Solve(P=1) { // Check if constraint P=1 satisfiable?
     while(Decide()=SUCCESS) //Selects a new variable
       while(Deduce()=CONFLICT)//BCP till conflict/no-conflict
         if (Diagnose()=FAILURE) //Add conflict learnt clause(s)
           return UNSAT;//Conflict found at decision level 0
   return SAT;} //No more decision to make
```

**Fig. 1.** DPLL style SAT Solver

Most modern SAT solvers are based on a DPLL-style [39] as shown in Figure 1 with three main engines: *decision, deduction*, and *diagnosis*. All these engines have seen remarkable progress in the last few years, e.g. the VSIDS decision heuristic and the lazy two-literal watching scheme for deduction in Chaff [6], and the conflict analysis and conflict-driven learning in Grasp [5]. Conflict-driven learning results in addition of *conflict clauses* to the SAT problem in order to prevent the same conflict from occurring again during the search. Additionally, information recorded during conflict analysis has been used very effectively to provide a proof when a formula is determined to be unsatisfiable by the SAT solver. This proof can be independently checked to verify the SAT solver itself [40, 41]. These techniques can also be easily adapted to identify a subset of clauses from the original problem, called the *unsatisfiable core* [40, 42], such that these clauses are sufficient for implying unsatisfiability. The use of such techniques in verification methods are described in more detail in Section 3.2.2.

## 2.2 Circuit-Based and Hybrid SAT Solvers

SAT has many applications in the logic circuit domain, such as automatic test pattern generation (ATPG), verification, timing analysis. The Boolean problem in these applications is typically derived from the circuit structure. This has also led to interest in circuit-based SAT solvers [43-45] that work directly on the circuit structure, and use circuit-specific heuristics to guide the search. In general, attempts to include circuit structure information into CNF-based SAT solvers have been unsuccessful due to their significant overhead.

Before we compare circuit-based and CNF-based SAT solvers, it is instructive to consider how each performs Boolean Constraint Propagation (BCP) which constitutes the core of most deduction engines, and typically consumes 80% of the SAT runtime. Circuit-based BCP is typically performed by using a lookup table for fast implication propagation [45]. Based on the current values of the inputs and output of the circuit node, the lookup table determines the next “state” of the gate where the state encapsulates any implied values and the next action to be taken for the node. The implication algorithm is iterated over the entire circuit graph. For each vertex, it determines new implied values and the direction for further processing. As an example, Figure 2 (from [45]) shows some cases from the implication lookup table for a two-input AND gate. Note that only one case, a logical *0* at the output of an AND vertex, requires a new case split to be scheduled for justification. All other cases either cause a conflict and backtracking, or further implications, or a return to process the next element to be justified. Due to its low overhead, this table lookup-based implication algorithm is very efficient in practice.

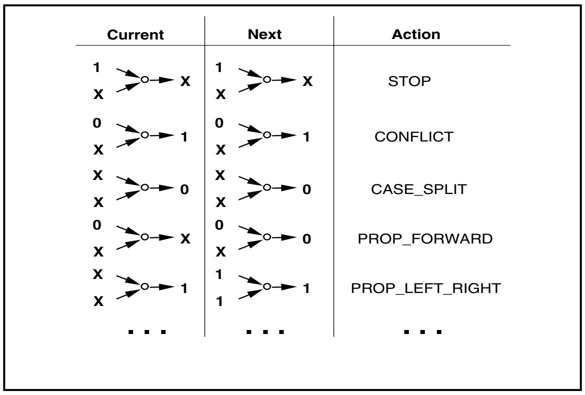
Current
Next
Action
1 X X
1 X X
STOP
0 X 1
0 X 1
CONFLICT
X X 0
X X 0
CASE_SPLIT
0 X X
0 X 0
PROP_FORWARD
X X 1
1 1 1
PROP_LEFT_RIGHT
. . .
. . .
. . .

**Fig. 2.** Lookup Table for Fast Implication Propagation on a 2-input AND Gate

For CNF-based BCP, consider the *lazy two-literal watching scheme* proposed by Chaff [6] :

- For each clause, only two literals are monitored for state change.
- The clause state is updated lazily when a variable is assigned, i.e., only when the two monitored literals coincide.
- It does not require state change for clauses during the backtracking process, thus unassigning a variable takes constant time.

For clauses with many literals, this lazy update works significantly better than other BCP schemes like those in SATO [7], and GRASP [5]. It avoids unnecessary lookups for a clause, thereby significantly improving the underlying cache behavior and the resulting performance.

To get back to the comparison between circuit-based and CNF-based BCP, note that there is an inherent overhead built into the translation of circuit gates into clauses. A two-input gate translates to three clauses in the CNF approach, while in the circuit-based approach a gate is regarded as a monolithic entity. Therefore, in the circuit approach an implication across a gate requires a single table lookup, while in the CNF approach it requires processing multiple clauses. In addition, the CNF-based BCP in Chaff does not keep track of the clauses that have been satisfied in order to reduce overheads. However, there is an inherent cost associated with visiting the satisfied clauses. Specifically, even if a clause gets satisfied due to an assignment to some un-watched literal, the watched literal pointers could still get updated. Overall, for the generally small clauses arising from circuit gates, these differences translate to significant differences in BCP time, usually in favor of the circuit-based approach.

On the other hand, learned conflict clauses arising from conflict analysis are typically much larger than those arising from two-input gates. Adding a large learned clause as a gate tree can lead to a significant increase in the size of the circuit. This in turn, can increase the number of implications, thereby negating any potential gains obtained from circuit-based BCP. For such clauses, it is more useful to maintain them as monolithic clauses and take advantage of CNF-based two-literal watching and lazy update to process them efficiently.

Based on these observations, we proposed a hybrid SAT solver [25] to combine the relative benefits of CNF-based and circuit-based SAT solvers. In our scheme, the original circuit problem is represented as a gate-level netlist, while the learned conflict clauses are represented in CNF. The hybrid BCP engine consists of table lookups for the gates, and a Chaff-style two-literal watching scheme for the conflict clauses, thereby combining the advantages of both. Furthermore, a hybrid representation of the Boolean problem also allows exploitation of both circuit-based and CNF-based decision heuristics. In particular, we effectively exploit the justification frontier heuristic [43], which restricts the decision nodes to be those that justify the values on their fanout node. The use of this heuristic further improves performance in many practical instances of SAT problems arising in circuit applications. We typically obtained speedup by a factor of two, in comparison to a pure CNF-based approach. Since SAT is a core engine in many verification applications like equivalence checking and BMC, a consistent speedup can prove to be very significant in practice. Furthermore, any future improvements in the performance of circuit-based and CNF-based SAT solvers can directly translate into improvements of the hybrid SAT solver as well.

In related work, Kuehlmann *et al.* [46] used circuit-based SAT in combination with other useful techniques like BDD sweeping and dynamic circuit transformation for combinational equivalence checking. However, they did not propose any effective way to perform conflict-driven learning with conflict clauses represented as large OR-tree circuits. Other more recent efforts have also combined the advantages of multiple symbolic representations including circuit graphs, SAT, and BDDs [47, 48], and used these ideas along with additional conflict-driven learning, in order to improve the SAT solver performance [11].

## 2.3 Model Checking

In model checking [1], the design is typically modeled as a labeled state transition system, the property is specified as a temporal logic formula, and verification consists of checking whether the formula is true in that model. Temporal logics are very useful for specifying dynamic behavior over time. Different variants of temporal logics have become popular, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), depending on whether a linear or a branching view of time is considered, respectively. In this paper, we focus mainly on simple safety properties, denoted as *AGp*. This formula specifies that on *all(A)* paths of a system, *globally (G)* in each state of the path, the property *p* holds. Such properties can be verified by an exhaustive traversal of the state space to check that $p$ holds in every reachable state. This state space traversal forms the computational core of most model checking techniques.

Explicit state model checkers, such as SPIN [49], use an explicit representation of the states and transitions in the system, and enumerate all reachable states explicitly. They utilize many additional techniques such as state hashing for compaction of state representations, and partial order methods to avoid exploring all interleavings of concurrent processes. The scalability issue in explicit state enumeration makes these checkers unsuitable for hardware designs, although they have found practical success in verification of controllers and software. In contrast, symbolic model checkers, such as SMV [3], avoid an explicit enumeration of the state space by using symbolic representations of sets of states and transitions. They typically use BDDs, which provide a canonical representation of Boolean formulas and efficient symbolic manipulation algorithms. For hardware designs, where these symbolic representations effectively capture the regularity in the state-space, symbolic model checking has significantly extended the ability to handle large state spaces.

The core steps in symbolic model checking are the *image/pre-image computation*s, which compute the set of states reachable in one step from/to a given set of states via the transition relation, as follows:

$$Img(Y) = S_N(Y) = \exists X,W.\ \ S_C(X) \wedge T(X,Y,W) \tag{1}$$

$$PreImg\ (X) = S_C(X) = \exists Y,W.\ \ S_N(Y) \wedge T(X,Y,W) \tag{2}$$

Here, the variable sets $X$, $Y$, $W$, denote the present state, next state, and primary input variables, respectively; and $S_C$ , $S_N$ and $T$ denote the next states, the current states, and the transition relation, respectively. When these state sets and the transition relation are represented symbolically as BDDs (or its variants), these computations can be performed symbolically by using the BDD-based operations for conjoining and existential quantification. However, for many large designs, these BDD-based operations can cause a blow up in memory size.

A basic algorithm for symbolic model checking simple safety properties can be formulated as shown in Figure 3. Let $B$ be the set of bad states, in which property $p$ does not hold, and $I$ the set of initial states. It represents sets of states symbolically, and searches for bad states in breadth first order starting from the initial states.

```
1. model-check(I,T,B)
2.      $S_C$ =∅; $S_N$ = I;
3.      **while** $S_C \neq S_N$ **do**
4.           $S_C = S_N$ ;
5.           **if** $B \cap S_C \neq \emptyset$ **then**
6.                **return** ``found counter-example'';
7.           $S_N = S_C \cup Img(S_C)$;
8.      **done**;
9.    **return** ``no bad state reachable'';
```

**Fig. 3.** Forward model checking algorithm for simple safety properties

This forward model checking algorithm starts at the initial states and searches forward along the transition relation, relying on the symbolic image computation described earlier. Similarly, there are backward model checking algorithms, which search backward from the bad states, and rely on the symbolic pre-image computation.

## 3 SAT-Based Verification Methods

We start by describing methods specialized for finding bugs in hardware designs. Most of these are based on the Bounded Model Checking (BMC) framework proposed by Biere *et al.* [12]. In the second half of this section, we focus on methods targeted for finding proofs. Many of these use our efficient BMC framework and add techniques on top to provide completeness of verification. We also describe additional techniques for proof-based abstraction and unbounded model checking using SAT.

### 3.1 Methods for Finding Bugs

Despite the considerable benefits of symbolic model checking using BDDs, the basic *verification* approach of exhaustive analysis does not scale well in practice. An alternative is the use of *falsification* approaches which focus primarily on the search for finding bugs. One of the most popular falsification approaches is Bounded Model Checking (BMC) [12]. We describe the basic BMC framework, followed by various performance enhancements, and a distributed BMC framework useful for overcoming memory limitations of a single workstation environment. We also describe EMM techniques which allow efficient handling of embedded memories in hardware designs for BMC applications.

#### 3.1.1 Bounded Model Checking (BMC)

In BMC, the problem of searching for a counter-example of length $k$ is translated to a Boolean formula such that the formula is *satisfiable* if and only if there exists a counter-example of length $k$. Effectively, the translation to a Boolean formula is performed by unrolling the transition relation of the design for $k$ time frames, and adding appropriate constraints due to the property. The satisfiability check is typically performed by using a Boolean SAT solver in the back-end.

In this paper, we consider the following notation and formulation of BMC. The design is described as a *Kripke* structure $M = (S, I, T, L)$, with a finite set of states $S$, a set of initial states $I$, a transition relation between states $T$, and a labeling $L$ of states with atomic propositions. Let $T(x,y,w,z)$ denote the symbolic transition relation in terms of present state variables $x$, next state variables $y$, primary input variables $w$, and intermediate variables $z$. Let $y_k$ denote the symbolic state (in terms of latch variables) after $k$ time frames in the unrolled design. In addition, we can also consider environmental constraints on variables (signals) $e$ in the design, which are required to hold in every time frame.

***Definition 1.*** We use the following Boolean formula, denoted $BMC(M,f,k)$, to check the existence of a $k$-length witness for property $f$:

$$BMC(M, f, k) = I(y_0) \wedge_{1 \le j \le k} [\, T(x_j, y_j, w_j, z_j) \wedge (\, y_{j-1} = x_j\,)] \wedge_{\, 0 \le j \le k} [Env(e_j)] \wedge \langle f \rangle_k$$

Here, the different sets of constraints are described as follows:

1. $I(y_0)$ : Initial state constraints on initial state $y_0$
2. $T(x_j, y_j, w_j, z_j)$ : Transition relation constraints for time frame $j$, $1 \le j \le k$
3. $(y_{j-1} = x_j)$ : Latch interface propagation constraints, $1 \le j \le k$, which capture the propagation of latch inputs in one time frame to the latch outputs in the next time frame
4. $Env(e_j)$ : Environmental constraints on signals $e$ in each time frame $j$, $0 \le j \le k$
5. $\langle f \rangle_k$ : Constraints due to property translation of formula $f$ in time frames up to $k$

Verification typically proceeds by looking for witnesses or counter-examples (CE) of increasing length until some *completeness threshold* [12, 50] is reached. The overall algorithm of a SAT-based BMC procedure for checking (or falsifying) a simple safety property *AG p* is shown in Figure 4, where *Unroll* corresponds to an unrolling of the symbolic transition relation of the design, and *P* corresponds to the circuit representation of the proposition *p*. Note that the SAT problems generated by the BMC translation procedure grow bigger as *k* increases. Therefore, the practical efficiency of the backend SAT solver becomes critical in enabling deeper searches to be performed.

```
BMC(k,P){//Falsify safety property AG p within bound k
   for (int i=0; i<=k ; i++) {
      P_i=Unroll(P,i);//Get property p at i^th unrolled frame
       if (SAT_Solve(P_i=0)=SAT) return CE; //Try to falsify
   }
 return NO_CE; } //No counter-example found
```

**Fig. 4.** SAT-based BMC for Safety Property *AG p*

Since BMC was first proposed, several methods have improved upon the basic BMC framework described above. These include use of variable ordering techniques to guide the decision heuristics of the back-end SAT solver [51], use of incremental SAT solvers to exploit the learning across related problems in SAT-based BMC [52, 53], as well as improved techniques for finding completeness thresholds [54, 55]. We now describe in detail our BMC enhancements, along with related work.

### 3.1.2 Performance Enhancements for BMC

**Circuit Simplification.** In our BMC implementation, we use circuit simplification techniques to build the transition relation of the design and for unrolling it during the course of property checking. The main motivation is to simplify the generated SAT problems in order to reduce the overall verification time. Furthermore, we have found that circuit simplification techniques are more efficient in handling of constants, in comparison to constant propagation within CNF-based SAT decision procedures. Such constants arise due to initial state and environmental constraints involving constant values on flip-flops ($I(y_0)$ and $Env(e_j)$ constraints in Definition 1 above), and learned constant constraints added during property checking. Circuit simplification is achieved by using a non-canonical two-input AND/INVERTER graph representation [56], and an on-the-fly reduction algorithm [24, 45] on such a graph representation.

**Hybrid SAT Solver.** We also use a hybrid SAT solver (described in Section 2.2) as the back-end SAT solver for our BMC applications. In addition to the performance benefits, it also provides us memory savings, since it avoids the need for maintaining a separate CNF representation of the unrolled design. We have also implemented incremental SAT solving techniques in our hybrid SAT solver to take advantage of our incremental formlations of the BMC problem, described below.

**Customized Property Translations.** We use customized property translations for the LTL formulas ($\langle f \rangle_k$ constraints in Definition 1 above) [15]. Rather than generating a monolithic SAT formula as in standard BMC, our property translations can be viewed as building the SAT formula incrementally, by lazily indexing over the bounded conjunctions/disjunctions, terminating early when possible. Though the standard BMC procedure also partitions the overall problem into separate k-instances, in our customized translation we further partition each k-instance problem into multiple, smaller SAT sub-problems. To mitigate the overhead due to multiple problems, we use an incremental SAT solver in the back-end. Additionally, our property partitioning is formulated in a way that facilitates the following kinds of SAT-based learning:

- **Learning from shared constraints (L1):** Given two SAT instances $S_1$ and $S_2$, conflict clauses that are deduced solely from the set of constraints shared between $S_1$ and $S_2$ can be used as learned clauses while solving $S_1$ *or* $S_2$ [52, 57].
- **Learning from satisfiable results (L2):** Suppose $\{\varphi_{1,\dots,}\varphi_n\}$ represents a series of SAT problems where problem $\varphi_i$ is built incrementally by adding and removing constraints to and from $\varphi_{i-1}$ respectively. The satisfying solution for $\varphi_{i-1}$ (if it exists) can be used to guide SAT decision engine to solve $\varphi_1$ [57].
- **Learning from unsatisfiable results (L3):** Let a SAT problem $\Phi$ be a disjunction of sub-problems $\{\varphi_{1,\dots,}\varphi_n\}$. Instead of solving $\Phi$ as a monolithic problem, one can solve $\varphi_i$ starting from $i=1$ with the additional constraint $C_i = \neg\varphi_1 \wedge \dots \wedge \neg\varphi_{i-1}$ where each $\varphi_j$ is unsatisfiable for all $j < i$. More benefit will be potentially obtained when the problem $\varphi_i$ shares more with the additional constraint $C_i$, thereby, allowing *L1* learning.

In a standard BMC procedure, there is a considerable overlap of circuit constraints due to unrolled transition relations between a $k$-instance and a $(k+1)$-instance of the BMC problem. Some researchers have applied L1 learning across $k$-instances [52, 57] while some researchers have used L3 learning for safety properties to reduce the overall verification time [58]. In our customized BMC translations, sharing occurs not just between the circuit constraints due to the unrolled transition relation, but also between the constraints arising from our property translations and between constraints learned from unsatisfiable SAT sub-problems. In order to take advantage of incremental SAT techniques, our property translations are geared toward an incremental formulation, i.e. reuse of variables and constraints wherever possible.

This allows use of incremental SAT learning techniques L1, L2 and L3 in the SAT solver very effectively. In contrast, the standard property translation in BMC does not provide such an incremental formulation of the property constraints.

**Experimental Results.** We briefly describe the results of some experiments (performed on a workstation with 2.8 GHz Xeon Processors with 4GB running Red Hat Linux 7.2) to show the benefits of our customized translation for checking liveness properties on industry designs D1-D3. These are bus core designs with multiple masters and slaves. The properties are of the type “request should be eventually followed by acknowledge or error”. We compare various learning schemes i.e. NL (no learning), L1+L3, L2+L3, and L1+L2+L3 in our customized approach, called custom. We also compare them against standard BMC in VIS [59], called standard. For fair comparison, we use the same SAT heuristics in our SAT solver as those used in the backend SAT solver (zChaff [6]) in VIS BMC. We present the comparison results in Table 1. Columns 2-4 show the characteristics of the designs in Column 1, i.e., number of flip-flops, primary inputs and gates, respectively; Column 5 shows the length of counter example (CEX); Columns 6-9 and 10 report the time taken by (in seconds) our customized translation with different learning schemes, and the standard translation, respectively. Clearly, our customized translation is able to find counterexamples far quicker than the standard monolithic translation. Moreover, the various learning schemes improve the performance of BMC significantly.

**Table 1.** BMC Performance Improvements due to Customized Translations

| D | #FF | #PI | #G | CEX (D) | Custom (DiVer) | | | | Standard (VIS) |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | NL | L1,3 | L2,3 | L1,2,3 | |
| D1 | 2316 | 76 | 14655 | 19 | 2.3 | 2.2 | 2.3 | 2 | 77 |
| D2 | 2563 | 88 | 16686 | 22 | 11.2 | 8.9 | 11.7 | 8 | 201 |
| D3 | 2810 | 132 | 18740 | 28 | 730 | 290 | 862 | 240 | 2728 |

**BDD Learning.** As demonstrated by the recent SAT solvers, learned clauses play a crucial role in determining performance, both by pruning the search space, and by affecting the choice of decision variables. In addition to the incremental SAT learning opportunities described above, we use an additional technique [13] for using learned clauses automatically generated from a BDD-based analysis. We call this BDD Learning. Essentially, a BDD is used to capture the relationship between Boolean variables of (a part of) the SAT problem, in the form of a characteristic function. In such a BDD, each path to a “0” (false) node denotes a conflict. A learned clause corresponding to this conflict is easily obtained by negating the literals that define the path. Since a BDD captures all paths to 0, i.e. all possible conflicts, the potential advantage is that multiple learned clauses can be generated and added to the SAT solver at the same time. In contrast, a SAT solver typically analyzes a single conflict at a time, thereby generating a single learned clause. An example with multiple learned clauses generated from a BDD is shown in Figure 5.

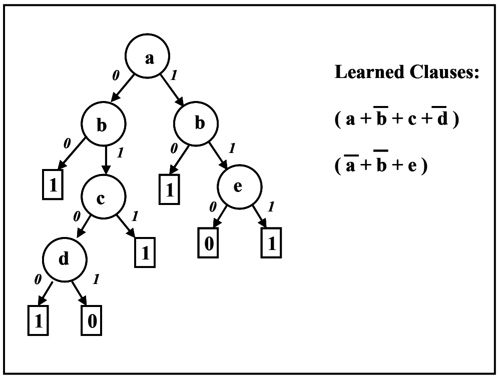
a
b
b
c
e
d
0
1
Learned Clauses:
$(a+\overline{b}+c+\overline{d})$
$(\overline{a}+\overline{b}+e)$

**Fig. 5.** Example of Learned Clauses from BDD Learning

In BMC, or any circuit-based application involving time frame expansion, the bulk of the constraints that define the SAT problem arise from the $k$-times unrolled transition relation of the design. Therefore, the transition relation is a natural candidate for the application of BDD Learning, i.e. BDDs are generated for "seed" nodes in the circuit structure graph corresponding to the transition relation (either before or after unrolling). At the same time, there is an overhead associated with the addition of each learned clause, e.g. during BCP. Due to this potential overhead of adding too many learned clauses, the choice of the seed nodes becomes crucial. To ensure a good tradeoff between the usefulness and the overheads of adding learned clauses, we experimented with the following kinds of learning:

- *Static learning*: seed nodes are selected using static information, and learned clauses are added statically, before the SAT solver starts the search.
- *Dynamic learning*: seed nodes are selected using dynamic SAT information, and learned clauses are added on-the-fly, during the SAT search.

In static learning, the seed selections will tend to reflect only the circuit structure information, while for dynamic learning they will also reflect the dynamic state of the SAT solver. Clearly, it is easier to integrate static learning with a SAT solver. On the other hand, we found that dynamic learning is crucial in attaining a good balance in the tradeoff between the usefulness and overhead of adding learned clauses. In our experimental results, we obtained up to 73% reduction in runtime, allowing us to perform deeper searches (with up to 45% more time frames) within the allotted time.

**Additional Constraints from BDD-based Reachability Analysis.** Another BDD-based technique that we have used very successfully with SAT-based methods is the addition of external constraints (not necessarily from conflicts) generated by performing BDD-based reachability analysis. While BDD-based reachability analysis provides a complete verification method, it works only on small models. On the other hand, SAT-based BMC can handle much larger models, but it is incomplete in practice. Conservative abstractions, i.e. abstractions that over-approximate the paths in concrete models, are the key in providing a link between the two. In particular, we use conservative approximations of reachable state sets, computed as BDDs, as additional constraints in BMC. Other related efforts have also used BDD-based enlarged targets [54], and BDD-based approximate reachability state sets [60], but only in searching for counterexamples with BMC. Our application of this idea in helping derive proofs by induction is described later in Section 3.2.1.

Since the BDD constraints for BMC are required to be over-approximations, we obtain "existential" abstractions of the design by considering some latches in the concrete design as pseudo-primary inputs. For example, we abstract away latches farther in the dependency closure of the property signals, identified by localization techniques [61]. Given a conservative abstract model, and a correctness property, we use exact or approximate symbolic model checking techniques to generate the BDD constraints. For example, for simple safety properties, we store the union of the state sets computed iteratively by the pre-image operation, backwards from the set of bad states, as shown in Figure 6 (a). We also store the union of the state sets in the forward reachability analysis, starting from the initial state, as shown in Figure 6 (b).

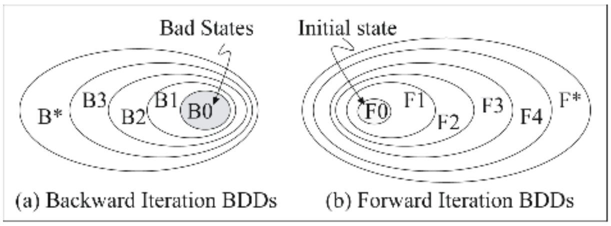

**Fig. 6.** Generation of BDD Reachability Constraints

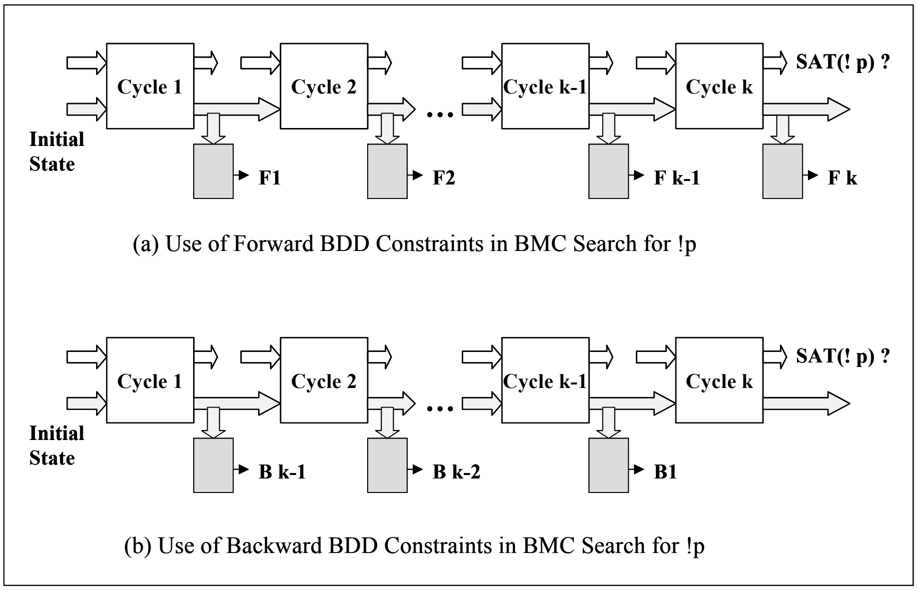

**Fig. 7.** Use of BDD Constraints in BMC

We convert each BDD to either a gate-level circuit or a CNF formula, where each internal BDD node is regarded as a multiplexor controlled by that node's variable. The size of the resulting circuit or CNF formula is linear in the size of the BDD, due to introduction of extra CNF variables for each BDD node. We use BDD reordering heuristics as well as over-approximation methods to keep down the BDD sizes. Finally, we use the BDD constraints as additional constraints on the state variables at each (or some) cycle of the unrolled design during BMC, as shown by dark boxes in Figure 7. Note that while these constraints are redundant, in practice they can improve the efficiency of the back-end SAT solver and allow deeper searches for bugs.

### 3.1.3 Distributed SAT and Distributed BMC

In SAT-based BMC the problems get larger as the depth of unrolling increases. Sometimes, the memory limitation of a single server, rather than SAT solver performance, can become a bottleneck for doing deeper BMC search for bugs. We have proposed methods for distributing SAT and distributing the BMC computation over a network of workstations (e.g. connected by Ethernet LAN) to overcome this memory limitation, albeit at increased communication cost and some performance overhead [14].

Our distributed methods use a master/client model where each client has an exclusive partition of the SAT problem and uses knowledge of the distributed partition topology to communicate with other clients. Due to the design unrolling, a BMC problem provides a natural linear partitioning of the overall SAT problem, thereby suggesting a linear topology for configuing the computing resources. An example topology using one master and several clients is shown in Figure 8. Each client $C_i$ hosts a part of the unrolled design in BMC, e.g., from $n_i$+1 to an $n_{i+1}$ where $n_i$ represents the partition depth. Each $C_i$ (except for the terminal clients) is connected to $C_{i+1}$ and $C_{i-1}$. The master is connected to each of the clients. Using a linear topology, we can also distribute parts of the unrolled design dynamically over additional clients as and when memory resources on current clients get close to exhaustion.

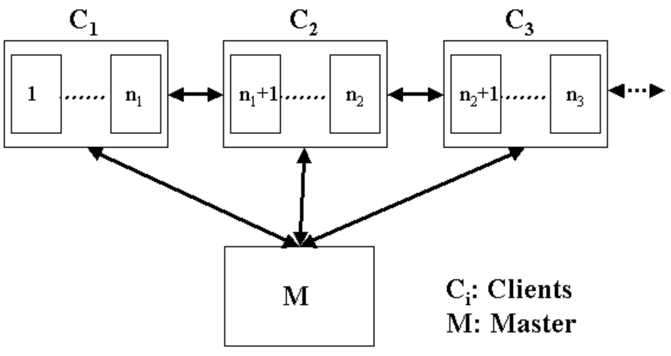

**Fig. 8.** Distributed BMC Configuration with Master/Clients

For distributed SAT, the master controls the overall execution between the parallelized decision, deduction, and diagnosis engines. The decision engine is distributed in a way such that each client selects a good local variable and the master then chooses the globally best variable to branch on. During the deduction phase, each client performs BCP on its exclusive local partitions, and communicates implications on the common variables to its neighboring clients. The master performs BCP on the global pool of learned conflict clauses, and also communicates the implications to all the clients. Diagnosis is performed by the master, and each client performs a local backtrack when requested by the master. Note that the master does not keep all problem clauses and variables; however, the master maintains the global assignment stack and the global state for diagnosis. This requires much less memory than the entire problem data.

Though this work is closely related to [62], there are some important differences. In that approach, although each client has a disjoint set of clauses, the variables are not shared in any set pattern. Therefore, each client broadcasts its implications to all other clients after completing the local BCP. In a communication network implementation like ours (unlike the application specific processor framework considered in [62]) broadcasting all these implications can be a significant overhead. In our improved distributed BCP, however, each client has knowledge of the topology of the SAT-problem partitioning, and uses it to communicate with other clients. This ensures that the receiving client has to never read a message that is not meant for it. To ensure proper execution of the parallel algorithm, each client is required to be synchronized. In addition, we use several optimization schemes to reduce the effect of communication overhead on performance in general-purpose networks by identifying and executing tasks in parallel while messages are in transit. For a large industry design with ~13K flip-flops and ~0.5Million gates, our distributed BMC approach enabled us to search up to a depth of 323 with only 30% communication overhead, while we could analyze only up to 120 time frames in a single workstation environment before running out of memory.

### 3.1.4 Efficient Memory Modeling (EMM) for Finding Bugs

Designs with large embedded memories have wide application in the industry. However, such designs add further complexity to formal verification tasks due to an exponential increase in the state space for each additional memory bit. For BMC, with each time frame unrolling of a design, the search space becomes prohibitively large to analyze beyond a reasonable depth. In order to make BMC more useful, it is important to have some abstraction of the memories. However, for finding real bugs, it is sufficient that the abstraction techniques capture the memory semantics [63] without explicitly modeling each memory bit.

To capture the memory semantics, Burch and Dill introduced the interpreted *read* and *write* operations in their logic of equality with un-interpreted functions (EUF) [63] instead of an un-interpreted abstraction of memories. These interpreted functions are used to represent the memory symbolically by creating nested *if-then-else* (ITE) expressions to record the history of writes to the memory. Such interpretated functions have also been exploited in later derivative verification efforts [64, 65]. More recently, Bryant *et al.* proposed the logic of Counter arithmetic with Lambda expressions and Un-interpreted functions (CLU) to model infinite-state systems and unbounded memory in the UCLID system [66]. Memory is modeled as a functional expression whose body changes with time step. Similar to [63], the memory is represented symbolically by creating nested ITE expressions, which are translated by the CLU decision procedure and handled by a back-end SAT solver.

Our EMM approach [27] is similar to the abstract interpretation of memory that captures its data forwarding semantics, i.e., a data read from a memory location is same as the most recent data written at the same location. We construct a BMC problem instance as follows. For our discussion, assume a single port memory with the following interface signals: Address Bus (*Addr*), Write Data Bus (*WD*), Read Data Bus (*RD*), Write Enable (*WE*), and Read Enable (*RE*). Assume that the *write* phase of the memory requires one clock cycle, i.e., in the current clock cycle when the data value is assigned to *WD* bus, write address location is assigned to *Addr* bus and *WE* signal is made active, then the new data is available in the next clock cycle. The *read* phase of memory is regarded as a same cycle event, i.e., when the read address location is assigned to *Addr* bus and the *RE* is made active, the read data is assigned to *RD* bus in the same clock cycle. Assume that we unroll the design up to depth *k*. Let $S^j$ denote a memory interface signal variable *S* at time frame *j*. Let the Boolean variable $E^{ij}$ denote the address comparison between time frames *i* and *j*, defined as $E^{ij}=(Addr^i=Addr^j)$. Then the forwarding semantics of the memory can be expressed as:

$$RD^k = \{WD^j \mid E^{jk}=1 \wedge WE^j=1 \wedge RE^k=1 \wedge \forall_{j<i<k}(E^{ik}=0 \vee WE^i=0)\}, \text{ where } j<k \tag{3}$$

In other words, data read at depth *k* equals the data written at depth *j* if memory addresses are equal at *k* and *j*, write enable is active at *j*, read enable is active at *k*, and for all depths *i* between *j* and *k*, either the address at *i* is different from that at *k* or the write enable at *i* is inactive.

Note that the BMC problem instance is generated by eliminating memory arrrays but retaining the memory interface signals in the design, and adding memory-modeling constraints on those signals at every depth of unrolling to preserve the semantics of the memory. The additional novelty of our EMM approach is that the formulation of the memory-modeling constraints (not shown here) capture the exclusivity of a read and write pair explicitly, i.e., when a SAT-solver decides on a valid read and write pair, other pairs are *implied invalid immediately*, thereby reducing the SAT solve time. Furthermore, we use a hybrid circuit-based and CNF-based representation of these constraints, where their size depends linearly on the bus widths of memory interface signals and quadratically on the number of memory accesses. Since the unrolling depth of BMC bounds the number of memory accesses, the size of these constraints is significantly smaller than an explicit modeling of the memory state in BMC. We can also apply the EMM approach for handling multiple memories, with multiple read and write ports [28].

The modified BMC algorithm using EMM is shown in Figure 9. The memory modeling constraints are generated by the procedure *EMM_Constraints*, which is invoked after every unrolling. The updated constraints $C^i$ in line 5 capture the forwarding semantics of the memory up to depth i very efficiently using hybrid symbolic representations, i.e., 2-input gates and CNF clauses, in order to improve the SAT solve time. We have experimented with our EMM method on a number of hardware and software designs with large embedded memories, where we have obtained at least an order of magnitude improvement (both in time and space) using our method over explicit modeling of each memory bit. We have also shown that the particular form of our memory modeling constraints boosts the performance of the SAT solver significantly, in comparison to the conventional way of modeling these constraints as nested *if-then-else* expressions [63, 66].

```
BMC-EMM (n, P) { // BMC with EMM
  $C^{-1}=\phi$; // initialize memory modeling constraints
  for (int i=0; i<=n ; i++) {
      $P^i$ = Unroll(P,i); // get property node at $i^{th}$ unrolling
      $C^i = C^{i-1} \cup$ EMM_Constraints(i); // update the constraints
      if (SAT_Solve($I \wedge \neg P^i \wedge C^i$)=SAT) return CE;}
  return NO_CE; } // no counter-example found
```

**Fig. 9.** SAT-based BMC with Efficient Memory Modeling (EMM)

## 3.2 Methods for Finding Proofs

In this section we describe SAT-based methods targeted for finding proofs. These methods can prove the correctness of a property on a design, as well as find counter-examples for failing properties. The method may or may not be complete, i.e. it may not be able to prove every correct property. We describe induction-based methods that have been found useful for proving safety properties. Next, we describe abstraction-refinement methods, where SAT-based BMC is used primarily for abstraction or refinement, and is supplemented by other techniques for obtaining proofs on the abstract models. We also describe the application of induction and proof-based abstraction in the context of our EMM approach, for finding proofs for embedded memory designs. Finally, we describe SAT-based methods that directly implement unbounded model checking. Many of these replace or supplement the use of BDDs by SAT solvers in traditional symbolic model checking techniques. In principle, completeness of verification can also be achieved by making the transition from SAT to solvers for Quantified Boolean Formulas (QBF) – we do not discuss QBF-based techniques here.

### 3.2.1 SAT-Based Induction

Induction with increasing depth *k*, and restriction to loop-free paths, provides a complete proof technique for safety properties and has been proposed for use with SAT solvers [16, 67]. Induction with depth *k* consists of the following two steps:

- Base step: to prove that the property holds on every *k*-length path starting from the initial state.

- Inductive step: to prove that if the property holds on a *k*-length path starting from any arbitrary state, then it also holds on all its extensions to a (*k*+1)-length path.

The restriction to loop-free paths imposes the constraints that no two states in a path are identical.

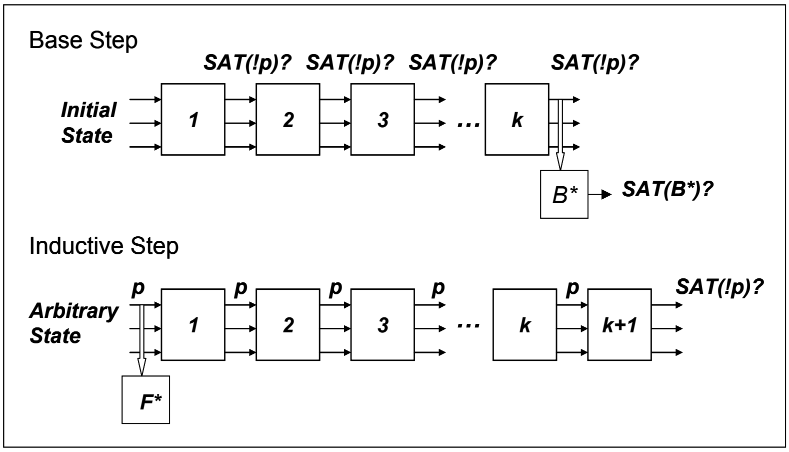

**Fig. 10.** Proof by Induction using SAT-based BMC (with BDD Constraints)

We use the BMC framework for performing a proof by induction, as shown pictorially in Figure 10 for the safety property *AGp*. (For the moment, disregard the boxes labeled *B** and *F**.) For the base step, the satisfiability of *negated p* is checked at each cycle up to *k*, starting from the initial state. If it is satisfiable, then the property is proved to be false. If it is not, then the inductive step is carried out. For the inductive step, it is assumed that the property *p* holds at the first *k* cycles, starting from an arbitrary state, and satisfiability of *negated p* is checked at the *(k+1)*$^{th}$ cycle. If it is unsatisfiable, then the induction proof is complete, and the property is proved to be true. However, if it is satisfiable, then the verification is incomplete. In this case, the depth of induction (*k*) can be increased, and the proof steps repeated, until the longest loop-free paths have been examined.

Note that the base step includes use of the initial state constraint, but the inductive step does not. Therefore, the inductive step may search through unreachable states also. In practice, this may not allow the induction proof to go through at small depths, i.e. the proof may need stronger induction invariants than the property itself. In general, any circuit constraints known by the designers can be used to strengthen the induction invariant. We use BDD-based reachability constrains, described earlier in Section 3.1.2.

In our enhanced BMC method for induction [17], we use the BDD-based reachability constraints not as redundant constraints, but as additional constraints to the SAT problem in order to facilitate the proof by induction. This use is shown pictorially by the boxes labeled *B** and *F** in Figure 10. For the base step, after we have checked the unsatisfiability of the negated property at each cycle up to *k* cycles, we can additionally check the satisfiability of the $B^*$ BDD constraint after $k$ cycles. If it is unsatisfiable, then the property is proved to be true, without performing any more inductive steps by BMC. In a sense, the inductive step has been already performed by the BDD-based fixpoint operation on the abstract model. However, if $B^*$ is satisfiable, we proceed with the inductive step. For the inductive step, we use the $F^*$ BDD to constrain the arbitrary state at the start of the $k+1$ cycles. This provides an additional reachability invariant, which can potentially allow the inductive step to succeed with BMC. It is intructive to note that the base step of an induction proof proceeds in the forward direction, and therefore the $B^*$ constraint derived from the backward BDD-based analysis complements BMC to provide completeness. In contrast, the inductive step constitutes a backward style of reasoning. Therefore, the $F^*$ constraint derived from the forward BDD-based analysis complements BMC to provide completeness.

**Experiment Results.** Our results for using this technique for checking safety properties on some large industrial designs are shown in Table 2. Here, Columns 2 – 5 report the results for BDD-based analysis on the abstract model (number of flip-flops #FF, number of gates #G, the CPU time taken for traversal, the number of forward iterations, and the final size of the BDD $F^*$, respectively). Columns 6 – 9 report the results for a BMC-based proof by induction on the concrete design, with use of the BDD constraints (the verification status, and the time and memory used by the BMC engine, respectively). We obtained the abstract models automatically from the unconstrained designs, by abstracting away latches farther in the dependency closure of the property variables. Due to the small size of the abstract models, we could keep the resource requirements for BDDs fairly low. The important observation is that despite gross approximations in the abstract models, the BDD reachability invariants were strong enough to let the induction proof go through successfully with BMC in each case. Though neither the BDD-based engine, nor the BMC engine, could individually prove these safety properties, their combination allowed the proof to be completed very easily (in less than a minute).

**Table 2.** Experimental Results for Proof by Induction using BDD-based Reachability Invariants

| | BDD-based Abstract Model Analysis | | | | Induction Proof with BDD Constraints on Concrete Design | | | |
|---|---|---|---|---|---|---|---|---|
| | #FF / #G | Time(s) | Depth | Size of F* | #FF / #G | Status | Time(s) | Mem(MB) |
| D1-p1 | 41 / 462 | 1.6 | 7 | 131 | 2198 / 14702 | TRUE | 0.07 | 2.72 |
| D2-p2 | 115 / 1005 | 15.3 | 12 | 677 | 2265 / 16079 | TRUE | 0.11 | 2.84 |
| D3-p3 | 63 / 1001 | 18.8 | 18 | 766 | 2204 / 16215 | TRUE | 0.1 | 2.85 |

### 3.2.2 Proof-Based Iterative Abstraction (PBIA)

In order to handle large designs, there has been a great deal of interest in the use of abstraction and refinement techniques for verification. Most efforts are refinement-based approaches, where starting from a small abstract model of the concrete design, counterexamples found on these models are used to refine the model iteratively until either a conclusive result is obtained by conservative model checking, or the resources are exhausted [61, 68]. In a separate development, resolution-based proof analysis techniques for SAT solvers [40, 42] can be used to identify a set of original clauses from an unsatisfiable SAT problem, called the *unsatisfiable core*, that are sufficient for implying unsatisfiability. These techniques have been independently used for verification applications also – counterexample guided refinement [69], proof-based abstraction [22, 42], unbounded model checking using interpolants [70]. Here we describe the details of our proof-based iterative abstraction (PBIA) technique [22, 23], based on our SAT-based BMC framework. It can be used to generate small abstract models, on which proofs can be derived using any unbounded verification methods. In the next section, we describe its use in conjunction with EMM techniques to find proofs for designs with embedded memories.

**Proof-based Abstraction using SAT-based BMC.** A proof-based abstraction technique works as follows. BMC is performed for increasing depths on the concrete design. Recall that when there is no counterexample at (or up to) a given depth of unrolling, the Boolean formula *BMC(M,f,k)* (Definition 1 in Section 3.1.1) is unsatisfiable. In this case, an unsatisfiable core consisting of a subset of constraints is identified using resolution-based proof analysis techniques. Then a latch-based [22] (or a gate-based [42]) abstraction is used to generate a conservative abstract model that is guaranteed to have more behaviors than the concrete design. In addition, due to the sufficiency property of the unsatisfiable core, the abstract model is guaranteed to preserve correctness of the related property up to the given depth. In many cases, the abstract model can also be proved correct for *all* depths, by using unbounded model checking techniques, such as those based on BDDs or SAT-based unbounded model checking. Since the abstract model has more behaviors than the concrete design, this also proves correctness for the concrete design. The usefulness of the proof-based abstractions stems from the empirical evidence that for typical verification applications, the unsatisfiable cores and the corresponding abstract models are much smaller than the concrete designs, thereby making it easier to apply unbounded verification methods for deriving proofs.

**Proof-based Iterative Abstraction.** We further use an iterative abstraction framework [22], where a proof-based abstraction is used in the inner loop (unlike other proof-based methods [42]). This is shown in Figure 11. Given a concrete model *M*, and an LTL formula *f* (negation of the given correctness property), the shown procedure either finds a true counterexample for the correctness property, or returns a final abstract model on which bounded or unbounded verification can be performed.

Starting from the concrete model, an outer loop indexed by *i* (line 3) performs BMC on increasingly more abstract models, denoted *M[i]*. In each iteration of the outer loop, BMC is performed on model *M[i]* for increasing bound *k* (line 6). If the BMC formula is satisfiable, the counterexample is checked to see if it is spurious. If it is not spurious, the correctness property is shown to be false. If it is spurious, a new model *M'* is derived, either by using proof-based refinement, or by deriving a proof-based abstraction from *M[i-1]* for a bound *d* > *k* (line 8). The current iteration is started again for the new model *M'* (line 10). On the other hand, if the BMC formula is unsatisfiable for bound *k*, an abstract model *A[k]* is derived by using a proof-based abstraction on the unsatisfiable core (line 14), e.g. the latch interface abstraction. If the abstract model is stable, e.g. if its size does not change over the last few time frames (indexed by *k*), then a new iteration is started with this model chosen as

*M[i+1]* (line 16). Otherwise, BMC is performed again on the current model *M[i]* after incrementing *k* (line 19). As shown here, the outer loop is iterated up to convergence of the size of the abstract model *M[i]* (line 21). Alternatively, unbounded verification can be attempted for any of the abstract models *M[i]*, where a true result is conclusive, while a counterexample is handled the same way as a satisfiable BMC formula (lines 6-12).

```
 1  Proof_Based_Iterative_Abstraction (M,f) {
 2    M[1] = M; // start with concrete design
 3    for (i=1; i; i++) { // multiple iterations
 4      k = 1;
 5      do {
 6       if (BMC(M[i], f, k)_is_satisfiable)  // counterexample
 7           if (counterexample_is_spurious) {
 8             M' = get_refined_model(M[i], M[i-1], f, k);
 9             M[i] = M';
10             k = 1; }  // restart iteration i with model M'
11           else
12             return(property_is_false);
13       else {  // unsatisfiable BMC formula
14           A[k] = get_abstract_model(M[i], f, k);
15           if (model_A[k]_is_stable) {
16             M[i+1] = A[k];
17             break; }  // out of do-while loop
18           else
19              k++; }
20      } while (1);
21      if (size(M[i+1]) = = size(M[i]))
22        break;  // out of loop on i
23    }  // end of for loop
24    return(M[i]);  // final abstract model for verification
25  }
```

**Fig. 11.** Pseudo-code for Iterative Abstraction

The overall flow in iterative abstraction is targeted at reducing the size of the abstract models across successive iterations. The potential benefit is that for properties that are false, BMC search for deeper counterexamples is performed on successively smaller models, thereby increasing the likelihood of finding them. For properties that are true, the successive iterations help to reduce the size of the abstract models, thereby increasing the likelihood of completing the proof by unbounded verification methods. We obtained typically two orders of magnitude reduction in the size of the abstract model across all iterations. In our experience, this reduction was crucial for successful verification of large industry designs.

**Lazy Constraints in Proof-based Abstraction.** We further proposed the idea of lazy constraints [23], where the main motivation is to delay propagating the effect of values implied by certain constraints, in order to derive smaller unsatisfiable cores for proof-based abstractions. In a standard DPLL-based SAT solver, the BCP procedure treats all constraints as eager constraints, i.e. implications due to the constraints are performed as soon as possible (modulo some ordering). Rather than modify the SAT solver, we change the CNF representation of certain single-literal constraints to achieve the desired lazy effect. This allows us to directly use, without any modification, the latest improvements in SAT solver technology. In particular, a 1-literal eager constraint (x) can be converted to a lazy constraint by replacing it with (x+y)(x+y'), where y is a fresh variable not appearing in the remaining formula.

We use lazy constraints in SAT-based BMC for handling initial state constraints for latches and for environmental constraints provided by the designer (*$I(y_0)$ and $Env(e_j)$*, respectively, in Definition 1, Section 3.1.1). Intuitively, lazy initial state constraints provide a way to get away from the "irrelevant" initial state values, which may otherwise get forced into the unsatisfiable core due to eager implication at the time of pre-processing in the SAT solver. Delaying such irrelevant implications (so that they potentially never take effect) can often lead to a much smaller invariant abstract model. Similarly, lazy constraints provide a way to delay enforcing "irrelevant" environmental constraints, potentially leading to a smaller set being actually used in the proof of unsatisfiability.

**Experimental Results.** Our experimental results for proof-based iterative abstraction are shown below in Table 3. The first three columns report the design name, the number of flip-flops and gates in the concrete design, respectively. The next three columns report the results for use of PBIA without lazy constraints, with number of flip-flops in the final abstract model, the number of iterations it took to derive the model, and the total time taken (in seconds), respectively. The last three columns report these data for use of PBIA with lazy constraints.

Note first that in all examples, the use of PBIA automatically generates much smaller abstract models than concrete designs. This greatly facilitates the use of unbounded verification methods on the abstract models to prove properties on the concrete design. With respect to use of lazy constraints, note that delaying implications by using lazy constraints can result in a performance penalty in some cases, since the efficiency of a SAT solver depends upon performing more implications in order to avoid search. On the other hand, it often results in further reduction in the sizes of the abstract models. In this sense, use of lazy constraints can be regarded as a heuristic targeted at deriving a smaller abstract model using proof-based abstraction. In practice, we also identify a *sufficient* set of environmental constraints for a given property, in order to generate smaller abstract models. In many cases, these techniques are crucial in being able to complete unbounded verification on the derived abstract models.

**Table 3.** Experimental Results for Proof-Based Iterative Abstraction (PBIA)

| D | Concrete Model | | Abstract Models from PBIA | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | No Lazy | | | With Lazy | | |
| | #FF | # G | #FF | # I | T(s) | #FF | # I | T(s) |
| D1 | 3378 | 28384 | 522 | 9 | 60476 | 294 | 4 | 11817 |
| D2 | 4367 | 36471 | 1223 | 8 | 80630 | 1119 | 9 | 64361 |
| D3 | 910 | 13997 | 433 | 5 | 11156 | 166 | 10 | 29249 |
| D4 | 12716 | 416182 | 369 | 4 | 1099 | 71 | 6 | 1310 |
| D5 | 2714 | 77220 | 187 | 2 | 17 | 3 | 3 | 21 |
| D6 | 1635 | 26059 | 228 | 6 | 5958 | 148 | 3 | 4102 |
| D7 | 1635 | 26084 | 244 | 3 | 3028 | 155 | 5 | 2768 |
| D8 | 1670 | 26729 | 149 | 3 | 25 | 148 | 3 | 28 |
| D9 | 1670 | 26729 | 162 | 3 | 40 | 147 | 3 | 44 |
| D10 | 1635 | 26064 | 159 | 2 | 12 | 146 | 3 | 30 |
| D11 | 1670 | 26729 | 149 | 3 | 25 | 148 | 3 | 28 |
| D12 | 1670 | 26729 | 183 | 4 | 2119 | 182 | 4 | 2376 |
| D13 | 1670 | 26729 | 180 | 2 | 63 | 154 | 3 | 71 |
| D14 | 1635 | 26085 | 190 | 3 | 1352 | 154 | 5 | 1480 |
| D15 | 1635 | 26060 | 153 | 3 | 125 | 153 | 3 | 142 |

### 3.2.3 EMM for Finding Proofs

In this section, we describe extensions of our EMM method described earlier (for finding bugs, in Section 3.1.4), to providing correctness proofs for embedded memory systems. The novelties of our extended EMM approach [28] are:

- Modeling an arbitrary memory state precisely, and thereby, providing the ability to derive induction proofs using SAT-based BMC
- Combining EMM with proof-based abstraction to find irrelevant memory modules/ports that do not affect correctness of a given property

**Induction with EMM.** To model a memory with an arbitrary initial state, we introduce new symbolic variables at every time frame. Note that for a $k$-depth analysis of a design, there can be at most $k$ different memory read accesses from a read port; out of which at most $k$ accesses can be to un-written memory locations. Therefore, in total we need to introduce $k$ symbolic variables for the different data words in memory. However, these variables are not completely independent, and simply introducing new variables introduces spurious behaviors in the verification model. Therefore, we need to identify a sufficient set of constraints on these fresh variables that capture the arbitrary memory state precisely. This is done by identifying pairs of read operations from the same address, which should forward the same data, provided no other write operations have been performed on the same address in between the two reads. Again, we use a hybrid representation of these constraints and add them to the BMC problem at each depth of unrolling.

**EMM with Proof-based Abstraction.** Recall that EMM can significantly reduce the size of the verification model by eliminating explicit modeling of memory state. However, for checking the correctness of a given safety property, we may not require all the memory modules or the ports in the memory subsystem of a design. To further reduce the model, we can abstract out *irrelevant* memory modules or ports completely. In this case, we do not need to add the memory modeling constraints for the irrelevant memory modules or ports, thereby further reducing the BMC problem size.

For the purpose of automatically identifying irrelevant memory modules and ports, we use a technique combining EMM constraints with proof-based abstraction. We use SAT-based BMC with memory-modeling constraints added according to our EMM approach. Again, we derive an unsatisfiable core whenever the BMC problem is unsatisfiable at a given depth $k$. The latches appearing in the unsatisfiable core are accumulated over the depths in a set LR. If a latch in the control logic of a memory module (the logic driving its memory interface signals) does not appear in the set LR, we do not add the EMM modeling constraints for that memory module. In other words, since the correctness of the property does not depend on that memory module (up to depth $k$), we abstract it out completely. Note this corresponds to a conservative abstraction, i.e. a proof of correctness on the abstract model implies correctness on the original design. This reduction in the EMM constraints reduces the BMC problem size and significantly improves the performance in our experience.

### 4.2.4 SAT-Based Unbounded Model Checking (UMC)

Due to the success of SAT solvers in bounded model checking, there has been growing interest in their use for unbounded model checking as well. Here, the crucial non-trivial operation is quantifier elimination, which converts a QBF to a propositional Boolean formula. This is shown below for the image operation and pre-image operations, which form the computational core of symbolic model checking methods (also described in Section 2.3):

$$Img\ (Y) = S_N(Y) = \exists X,W,Z.\ \ S_C(X) \wedge T(X,Y,W,Z) \tag{4}$$

$$PreImg(X) = S_C(X) = \exists Y,W,Z.\ \ S_N(Y) \wedge T(X,Y,W,Z) \tag{5}$$

As before, the variable sets $X$, $Y$, $W$, denote the present state, next state, and input variables respectively; and $S_C$ , $S_N$ and $T$ denote the next states, the current states, and the transition relation, respectively. Note that there is an additional set of variables $Z$ denoting the internal variables needed for a CNF representation of the transition relation. In addition to the issue of quantification of these additional variables, there is also the issue of what representation to use for the state sets. There have been many efforts with different approaches for handling these issues. We first discuss methods that use SAT in combination with other methods, and then methods that use purely SAT for symbolic model checking.

**SAT with Other Symbolic UMC Methods.** Symbolic model checking based on combining non-canonical decision diagrams like Reduced Boolean Circuits (RBCs) [71] and Boolean Expression Diagrams (BEDs) [72] with SAT-solvers were proposed to alleviate the problems seen in pure BDD-based approaches. However, the use of circuit-based existential quantification ($\exists xf = f_x + f_{x'}$) in pre-image/image computation results in a formula size that grows exponentially with the number of quantified variables, in spite of all heuristic attempts to mitigate the growth. These approaches are, therefore, limited to designs with a small number of primary inputs.

We have also proposed a combined method [18], which integrates BDD-based techniques tightly into the SAT decision procedure. We represent the transition relation $T$ in CNF, and the set of states $S_C$ and $S_N$ as BDDs. For image computation, quantifier elimination is performed by using SAT techniques to enumerate all solutions to the CNF formula, and by projecting each solution on the set of image variables ($Y$). The search for solutions is constrained by the BDD for $S_C$, using a technique called *BDD Bounding*, whereby any partial solution in SAT which is inconsistent with the BDD is regarded as a conflict. This technique is also used effectively to avoid repeating image set solutions by bounding against the BDD representing the currently computed image set $S_N$.

Another novelty of our method is that in order to avoid enumeration of all solutions in SAT, we generate BDD-based subproblems on-the-fly for a partial assignment in SAT. The decision about when to generate a BDD subproblem is heuristically determined, e.g. when the number of assigned variables in SAT is some fraction of the total number of variables. The BDD subproblem corresponds to the transition relation cofactored with the current partial assignment in SAT. Indeed, each BDD subproblem is very similar to the standard BDD formulation of image computation, except that the CNF representation of the (cofactored) transition relation provides a much finer-grained conjunctive partition of the transition relation. This provides additional opportunities for early quantification of primary input variables. The overall image computation can be viewed as the SAT decision tree providing a disjunctive partitioning into multiple BDD subproblems at its leaves, as shown below in Figure 12. This combination of decision heuristics and implications provided by SAT, with the efficient computation of smaller subproblems using BDDs, is highly beneficial in handling large designs. We enhanced this basic image computation procedure by adding circuit structure information to the CNF formula of the transition relation, in order to dynamically detect and remove redundant clauses [19]. We also added partition-based SAT decision heuristics to further improve its performance [20].

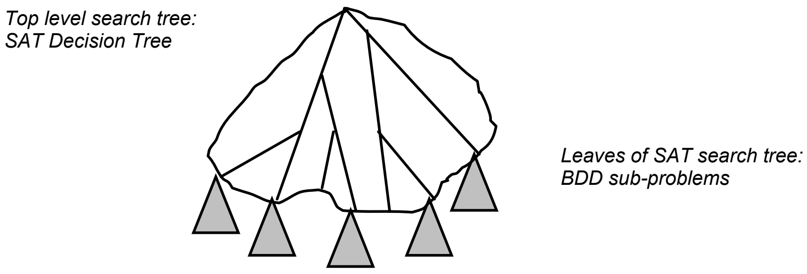

**Fig. 12.** Symbolic Image Computation Combining SAT and BDDs

**Purely SAT-based UMC Methods.** In purely SAT-based approaches, the transition relation is maintained in a conjunctive normal form (CNF) and a SAT procedure is used to enumerate all solutions for the required state set. Typically, the solution cubes are enumerated by the SAT solver for computing the pre-image state set [73, 74], and a blocking clause representing the negation of the enumerated state cube is added at each step in order to prevent repeating the solutions. In [73] a redrawing of the SAT solver's implication graph is carried out to enlarge the state cube. In [74], a two-level minimizer is used to compact the CNF formula after addition of new blocking clauses. Note that in both approaches, only a single state cube is captured at any enumeration step. Since the number of required enumerations is bounded below by the size of a two-level prime and irredundant cover of the solution set, quantifier elimination based on cube-by-cube enumeration tends to be expensive. In yet another approach [47], an ATPG solver is used as the search engine, and state cube enlargement is achieved using a separate justification procedure once a satisfying result is found by the SAT solver. This approach is also limited by its cube-wise enumeration strategy.

A different model checking approach based on use of SAT techniques and *Craig interpolants* has been proposed by McMillan [70]. Given an unsatisfiable Boolean problem, and a proof of unsatisfiability derived by a SAT solver, a Craig interpolant can be efficiently computed to characterize the interface between two partitions of the Boolean problem. In particular, when no counterexample exists for depth $k$, i.e., the SAT problem in BMC for depth $k$ is found to be unsatisfiable, a Craig interpolant is used to obtain an over-approximation of the set of states reachable from the initial state in 1 step (or any fixed number of steps). This provides an approximate image operator, which can be used iteratively to compute an over-approximation of the set of reachable states, i.e., till a fixpoint is obtained. If at any point, the over-approximate reachable set is found to violate the given property, then the depth $k$ is increased for BMC, till either a true counterexample is found, or the over-approximation converges without violating the property. The main advantage of the interpolant-based method is that it does not require an enumeration of satisfying assignments by the SAT solver. Indeed, the proof of unsatisfiability is used to efficiently compute the interpolant, which serves directly as an over-approximated state set. However, this approach can computes approximate reachable states, which are sufficient for proving safety properties, but are harder to apply for exact model checking.

Our SAT-based quantifier elimination method is considerably different from the other efforts. It dramatically reduces the number of enumerations of satisfying solutions in comparison to the cube-based approaches, thereby, significantly improving the performance of pre-image and fixed-point computation in SAT-based UMC [21]. Our SAT-based method is shown in Figure 13:

Note that we too use a SAT solver to enumerate solutions, but the novelty in our approach is that we use cofactoring after each enumeration, as follows:

- Get the satisfying assignment $\alpha$, which could be a cube (line 4)
- Pick a satisfying input minterm m by choosing an assignment on the unassigned input variables in the satisfying cube $\alpha$ (line 5)
- Cofactor the function $f$ with respect to the satisfying input minterm m (line 6)

Note that the cofactored function $f_m$ is used to represent the set of satisfying states derived from a single solution enumerated by the SAT solver. In general, a cofactor can capture not just a single cube of the state solution set, but several cubes. Specifically, our cofactoring approach is *guaranteed* to contain the set of new states captured in each enumeration step by the cube-based approaches. Therefore, our approach is also guaranteed to require a smaller number of enumerations by the SAT solver. In our experience, cofactoring-based quantification greatly reduces the total number of solutions enumerated by the SAT solver, sometimes by several orders of magnitude, in comparison to cube-wise enumeration approaches. Furthermore, we do not require a redrawing of the implication graph [73], or an enlargement of the enumerated cube as a post-processing step [47].

```
SAT-EQuant($f$,A,B) { // calculate $\exists_B$ $f(A,B)$
  C = $\emptyset$; //initialize constraint
  while (SAT_Solve($f$=1$\wedge$C=0) is SAT) {
    $\alpha$ = get_satisfying_cube();
    m = get_satisfying_input_minterm($\alpha$,B);
    $f_m$ = cofactor_cube($f$, m);
    C = C $\vee$ $f_m$; }
  return C; } // return when no more solution
```

**Fig. 13.** SAT-based existential quantification using Circuit Cofactoring

Additionally, we use circuit cofactoring techniques based on efficient circuit graphs [46] for representing the solution states, which is more robust in practice than CNF-based blocking clauses or BDD-based representations. We also use our hybrid SAT solver [25] to directly work on these representations. We also observed that the total number of enumerations depends, though less severely, on the choice of values chosen on the unassigned input variables in the satisfying cube assignment which determine the cofactor. We also proposed several SAT solver heuristics to choose an input minterm that enlarges the set represented by $f_m$.

Our circuit cofactoring-based quantification technique can be used to compute exact image/pre-image state sets, unlike the interpolant-based technique (described earlier), which computes approximate state sets. It has been used in SAT-based unbounded symbolic model checking to handle many difficult industry examples, which could not be handled by either BDDs or blocking-clause based SAT approaches. We have also used it in SAT-based UMC formulations (greatest fixed point, and least fixed point computations) for determining completeness bounds for liveness properties, which are then used in our customized translations with BMC to find proofs [15]. More recently, we have combined our SAT-based UMC technique with symmetry reduction [75] to obtain further performance improvements, due to both the reduced state space and simplification in the resulting SAT problems.

**Experimental Results.** We show experimental results on 102 examples for checking safety properties on designs from the VIS verification benchmark suite (VVB) [76] with a time limit of 1000s for each example. We present the comparison of our Circuit Cofactoring approach (CC) with the Blocking Clauses (BC) approach, and the BDD and BMC approaches in VIS, as scatter plots in Figure 14.

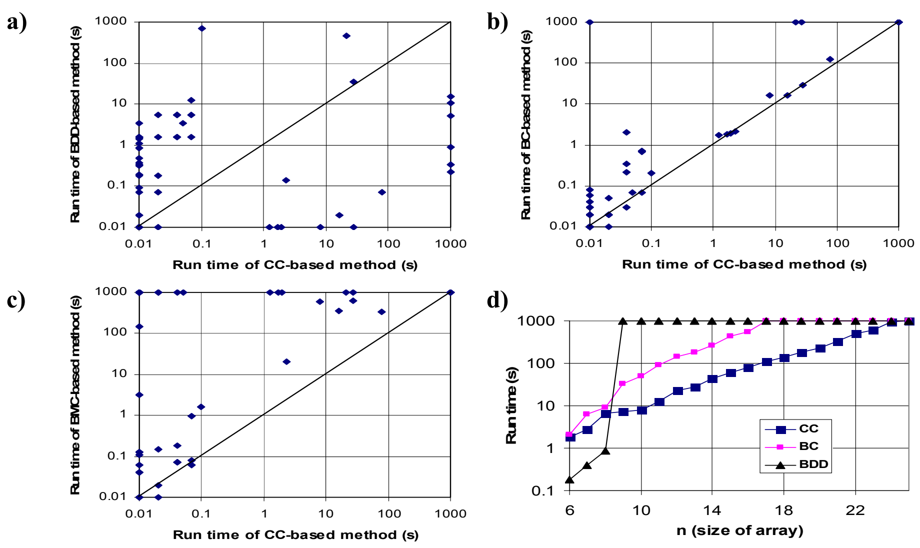

**Fig. 14.** Experimental Results for Circuit Cofactoring-based UMC Method (CC): (a) CC vs. BDD (b) CC vs. BC (c) CC vs. BMC (d) Swap example results

As shown in the scatter plot of CC/BDD in Figure 14 (a), our approach performed better in 68 cases while BDD performed better in 16 cases. We observe here the complementary strengths of BDD-based and SAT-based approaches. As shown in the scatter plots of CC/BC and CC/BMC in Figure 14 (b) and 14 (c), respectively, our CC approach is almost always better than both BC and SAT-based BMC.

We also experimented on a design *swap* [73] that swaps non-deterministically consecutive elements in an array of length *n*. The correctness criterion is that all elements of the array are distinct. We compared our approach CC with BC and BDD (VIS) for varying *n*, each with a time limit of 1000s. The performance results are shown in Figure 14(d). Note that there is a time-out for BDDs for n>8 (also noted by others). With our CC approach, we can successfully analyze the design up to *n*=24, while with BC the analysis can complete only up to *n*=16. Note that our approach is about an order of magnitude faster than the *BC* approach on this example.

## 4 SAT-Based Applications in Hardware Verification

We have implemented our various SAT-based methods in a verification platform called *VeriSol* (formerly *DiVer*), targeted at verification of large scale hardware designs in the industry [29]. Due to an efficient and flexible infrastructure, *VeriSol* provides a very productive platform for research and development. A view of *VeriSol* as a "wheel of verification engines" is shown in Figure 15, where all these engines and their novel features have been described earlier in Section 3.

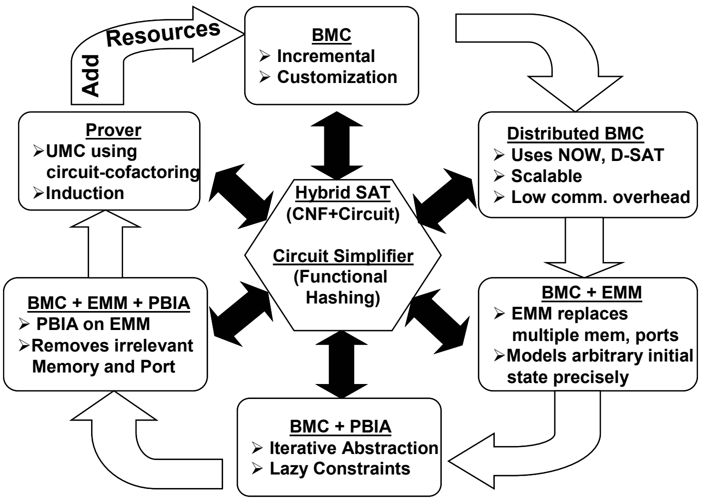

**Fig. 15.** Wheel of Verification Engines in *VeriSol*

In addition to the verification methods, *VeriSol* also has the ability to handle industry design features including multiple clocks, multiple phase and gated clocks, embedded memories with multiple read and write ports, environmental and fairness constraints. *VeriSol* has matured over the last four years, and has been used successfully by designers in our company to discover bugs that either could not have been found or were missed by functional simulation.

## 4.1 Selected Case Studies

Using selected case studies from the industry, we demonstrate the role of various engines in *VeriSol* at each step of the verification. It is interesting to note that without an interplay between the engines, we could not have verified any of these designs. The first two case studies do not contain large embedded memories, and use the verification flow shown in Figure 16(a). The next two case studies have large embedded memories and use the verification flow shown in Figure 16(b). All experiments were performed on a server with 2.8 GHz Xeon processors with 4GB running Red Hat Linux 7.2.

**Industry Design I.** The design has 13K flip-flops (FFs), ~0.5M gates in the cone of influence of a safety property. Using BMC, we showed that there was no witness up to depth 120 (in 1643s) before we run out of memory. Using d-BMC, we showed no witness up to depth 323 (in 8643s) using 5 workstations (configured as 1 Master and 4 Clients and connected with 1Gps Ethernet LAN), with a communication overhead of 30% and scalability factor of 0.1 (i.e, potentially we could do a 10 times deeper analysis than that on the single server.) We hypothesized that the property is correct. We used the PBIA engine to obtain an abstract model with 71 FFs and ~1K gates in 6 iterations taking ~1200s. By using standard BDD-based symbolic model checking on

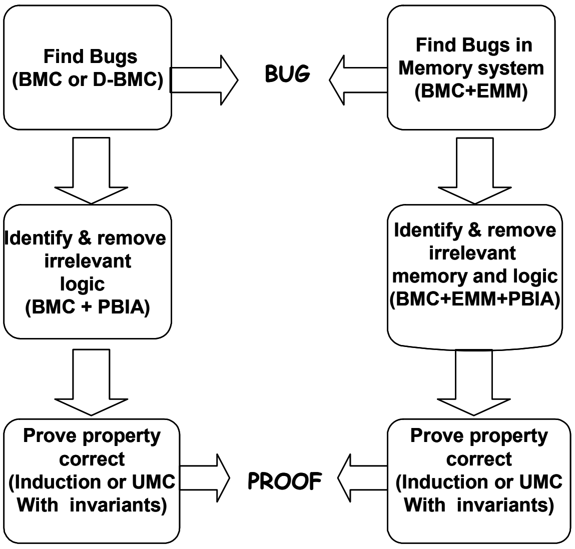
Find Bugs (BMC or D-BMC)
BUG
Find Bugs in Memory system (BMC+EMM)
Identify & remove irrelevant logic (BMC + PBIA)
Identify & remove irrelevant memory and logic (BMC+EMM+PBIA)
Prove property correct (Induction or UMC With invariants)
PROOF
Prove property correct (Induction or UMC With invariants)

**Fig. 16.** (a) Standard Verification Flow (b) Verification Flow with EMM

the conservative abstract model, we proved the property correct in 30s. This implied correctness on the original (much larger) design.

**Industry Design II.** The design with environmental constraints has 3.3K FFs and ~28K gates for a safety property. Using BMC, we showed there was no witness up to depth 113 (in ~3hr, 720MB). Again, we hypothesized the correctness of the property. We used PBIA to obtain an abstract model A1 with 163 FFs and ~2K gates in 4 iterations taking 9000s. Without the environmental constraints, the abstract model A2 has only 66 FFs and ~1K gates. We computed a reachability invariant on the A2 model (in ~4s) and used this invariant with SAT-based UMC on the A1 model to obtain a correctness proof in ~60s. Interestingly, for this example, none of the other UMC techniques (SAT-based induction, BDD-based symbolic model checking) was successful on the A1 model, even though the model was fairly small.

**Industry Design III.** The design has 756 FFs (excluding the memory registers), and ~15K gates. It has two memory modules, both having address width, AW = 10 and data width, DW = 8. Each memory module has 1 write and 1 read port, with the memory state initialized to 0. There are 216 reachabality properties to be checked. Using BMC+EMM, we found witnesses for 206 of the 216 properties, taking ~400s and 50Mb. The maximum depth over all witnesses was 51. Using exact memory modeling of the state, we required 20540s (~6hrs) and 912Mb to find witnesses for all 206 properties. By using induction with BMC+EMM, we proved the remaining 10 properties in <1s (this took 25 s with exact modeling of the memory state).

**Quicksort.** A hardware implementation of this algorithm has two memory modules: an un-initialized array with AW=10, DW=32, 1 read and 1 write port; and an un-initialized stack (for implementing the recursive function calls) with AW=10, DW=24, 1 read and 1 write port. The design has 167 FFs (excluding memory registers), and ~9K gates for array size of 5. The correctness property states that after return from a recursive call, the program counter should go to a recursive call on the right partition or return to the parent on the recursion stack. Using BMC+EMM+PBIA, we reduced the model to 91 FFs and ~3K gates, and also identified that the array module was irrelevant for this property (since the array data does not affect the control structure of the algorithm). On this reduced model we proved correctness using forward induction (proof diameter = 59) in 2.3Ks, 116MB. Without the proof-based abstraction, the induction proof in BMC+EMM takes ~5Ks, 400MB. When modeling the exact memory state, however, we could obtain neither a proof nor an abstract model in 3 hours.

## 4.2 Integration of *VeriSol* with Hardware Design Methodology

Although checking properties on the hardware design can be made (almost) fully automatic, *VeriSol* relies on the designers or verification engineers to provide correctness properties. Properties must capture the design intent and at the same time be expressed in some kind of formal syntax [77]. This requires the users to have both detailed knowledge of the design and some background in formal methods. Despite the growing popularity of formal verification and assertion-based verification, many designers are often reluctant to specify properties due to lack of either interest or training. We believe this has been one of the major hurdles in the wider application of formal verification in hardware design

In terms of our efforts in this direction, *VeriSol* has been integrated within a high level behavioral synthesis system called *Cyber* [30] developed in NEC Japan, which is used in-house by many system design groups. The *Cyber* system automatically generates RTL designs from high-level behavioral descriptions. In addition, it also automatically generates correctness properties for these RTL designs. The back-end property checking for *Cyber* can be performed by *VeriSol.*

We have also added techniques to automatically generate verification properties from RTL Verilog designs. The properties include both simple Boolean assertions and temporal property checkers (those describing the behavior of the model over time). Although they are generated with absolutely no user effort, these properties often capture some important aspects of the design intent and cover corner case conditions that are critical for the design to be correct. *VeriSol* can be used to either formally verify these properties using its various SAT-based and BDD-based engines, or generate simulation monitors from these properties. Our automatic checker generation procedure targets properties in the categories of checking register usage (write before read, read before initialized), memory usage (unused, read-only, write-only, multiple drivers), full case / parallel case, unreachable or constant condition branches, etc.

Automatic property generation can significantly increase *VeriSol's* ability in capturing subtle corner case errors in early RTL design stages with absolutely no user effort, and significantly ease the burden that has traditionally been put solely on designers or verification engineers. Once the users start benefiting from these automatically generated properties, they can be motivated to start thinking of new properties to cover more subtle design errors. In this sense, it also helps in promoting formal verification techniques to a wider design community in real industry settings. Our target users include both designers using the *Cyber* design synthesis flow and designers using Verilog only. We have also developed an IDE (Integrated Design Environment) for *VeriSol* in the Verilog design flow, utilizing an Eclipse-based GUI and Icarus Verilog [78] in the tool front-end.

## 5 Conclusions

In this paper we have described SAT-based verification methods we have developed for verifying large hardware designs. Due to many recent advances in SAT solver technology, these methods have emerged as a promising alternative to the traditional BDD-based symbolic model checking methods. At the same time, we have found in our experience on industry designs that many of these methods need to be combined with each other, and also with other BDD-based and circuit-based methods, in order to successfully find bugs or proofs for correctness properties in practice.

While a lot of progress has been made, much remains to be done to make these methods more robust in order to apply them reliably to large scale industry designs. In terms of industry applications, there is also a need for more effort in integrating these methods and the related tools within practical design methodology flows, in order to more fully realize the potential of formal verification.

## Acknowledgements

We would like to thank Pranav Ashar for his numerous and valuable contributions to the methods described here, and Kazutoshi Wakabayashi and Akira Mukaiyama for their contributions and help in applying them in an industry context.

## References

1. E.M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. 1999: MIT Press.
2. R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D. Dill, *Symbolic model checking for sequential circuit verification.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1994. **13**(4): p. 401-424.
3. K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem.* 1993: Kluwer Academic Publishers.
4. R.E. Bryant, *Graph-based algorithms for Boolean function manipulation.* IEEE Transactions on Computers, 1986. **C-35(8)**: p. 677-691.
5. J.P. Marques-Silva and K.A. Sakallah, *GRASP: A Search Algorithm for Propositional Satisfiability.* IEEE Transactions on Computers, 1999. **48**: p. 506-521.
6. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, *Chaff: Engineering an Efficient SAT Solver*, in *Proceedings of Design Automation Conference*. 2001.
7. H. Zhang, *SATO: An efficient propositional prover*, in *Proceedings of International Conference on Automated Deduction*. 1997. p. 272-275.
8. E. Goldberg and Y. Novikov, *BerkMin: A Fast and Robust SAT-Solver*, in *Proceedings of Conference on Design Automation & Test Europe (DATE)*. 2002. p. 142-149.

9. L. Zhang and S. Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, in *Proceedings of the International Conference on Computer Aided Verification*. 2002, Springer. p. 17-36.
10. M. Prasad, A. Biere, and A. Gupta, *A survey of recent advances in SAT-based formal verification*. Software Tools for Technology Transfer, 2005. **7**(2): p. 156-173.
11. F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna, *A signal correlation guided ATPG solver and its applications for solving difficult industrial cases*, in *Proceedings of the Design Automation Conference*. 2003. p. 436-441.
12. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu, *Symbolic Model Checking without BDDs*, in *Proceedings of Tools and Algorithms for Analysis and Construction of Systems (TACAS)*. 1999.
13. A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. *Learning from BDDs in SAT-based bounded model checking*. in *Design Automation Conference*. 2003.
14. M. Ganai, A. Gupta, Z. Yang, and P. Ashar, *Efficient distributed SAT and SAT-based distributed bounded model checking*, in *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME)*. 2003. p. 334-347.
15. M. Ganai, A. Gupta, and P. Ashar, *Beyond safety: Customized SAT-based model checking*, in *Proceedings of the Design Automation Conference*. 2005. p. 738-743.
16. M. Sheeran, S. Singh, and G. Stalmarck, *Checking Safety Properties using Induction and a SAT Solver*, in *Proceedings of Conference on Formal Methods in Computer-Aided Design*. 2000.
17. A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar, *Abstraction and BDDs Complement SAT-based BMC in DiVer*, in *Proceedings of International Conference on Computer Aided Verification*. 2003.
18. A. Gupta, Z. Yang, P. Ashar, and A. Gupta, *SAT-based Image Computation with Application in Reachability Analysis*, in *Proceedings of Conference on Formal Methods in Computer-Aided Design*. 2000. p. 354-371.
19. A. Gupta, A. Gupta, Z. Yang, and P. Ashar, *Dynamic detection and removal of inactive clauses in SAT with application in image computation*, in *Proceedings of Design Automation Conference*. 2001.
20. A. Gupta, Z. Yang, P. Ashar, L. Zhang, and S. Malik, *Partition-Based Decision Heuristics for Image Computation using SAT and BDDs*, in *Proceedings of International Conference on Computer-Aided Design*. 2001.
21. M. Ganai, A. Gupta, and P. Ashar, *Efficient SAT-based unbounded symbolic model checking using circuit cofactoring*, in *Proceedings of the International Conference on Computer-Aided Design*. 2004. p. 510-517.
22. A. Gupta, M. Ganai, J. Yang, and P. Ashar, *Iterative Abstraction using SAT-based BMC with Proof Analysis*, in *Proceedings of International Conference on Computer Aided Design (ICCAD)*. 2003.
23. A. Gupta, M. Ganai, and P. Ashar, *Lazy constraints and SAT heuristics for proof-based abstraction*, in *Proceedings of the International Conference on VLSI Design*. 2005. p. 183-188.
24. M. Ganai and A. Kuehlmann, *On-the-fly compression of logical circuits*, in *Proceedings of International Workshop on Logic Synthesis*. 2000.
25. M. Ganai, L. Zhang, P. Ashar, and A. Gupta, *Combining Strengths of Circuit-based and CNF-based Algorithms for a High Performance SAT Solver*, in *Proceedings of the Design Automation Conference*. 2002.
26. M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*. 1990: Computer Science Press.

27. M. Ganai, A. Gupta, and P. Ashar, *Efficient modeling of embedded memories in bounded model checking*, in *Proceedings of the International Conference on Computer Aided Verification*. 2004, Springer. p. 440-452.
28. M. Ganai, A. Gupta, and P. Ashar, *Verification of embedded memory systems using efficient memory modeling*, in *Proceedings of Design Automation and Test Europe (DATE)*. 2005. p. 1096-1101.
29. M. Ganai, A. Gupta, and P. Ashar, *DiVer: SAT-based model checking platform for verifying large scale systems*, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2005, Springer. p. 575-580.
30. K. Wakabayashi, *Cyber: High level synthesis system from software into ASIC*, in *High Level VLSI Synthesis*, R. Camposano and W. Wolf, Editors. 1991, Kluwer Academic Publishers. p. 127-151.
31. F. Ivancic, J. Yang, M. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar, *F-Soft: Software Verification Platform*, in *Proceedings of the International Conference on Computer Aided Verification*. 2005, Springer. p. 301-306.
32. F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang, *Model checking C programs using F-Soft*, in *Proceedings of the International Conference on Computer Design*. 2005. p. 297-308.
33. C. Barrett, D. Dill, and A. Stump, *Checking satisfiability of first-order formulas by incremental translation to SAT*, in *Proceedings of the International Conference on Computer Aided Verification*. 2002. p. 236-249.
34. S. Seshia, S. Lahiri, and R.E. Bryant. *A hybrid SAT-based decision procedure for separation logic with uninterpreted functions*. in *Design Automation Conference*. 2003.
35. C. Wang, F. Ivancic, M. Ganai, and A. Gupta, *Deciding separation logic formulae by SAT and incremental negative cycle elimination*, in *Proceedings of Logic for Programming, Artificial Intelligence and Reasoning*. 2005. p. 322-336.
36. M. Ganai, M. Talupur, and A. Gupta, *SDSAT: Tight integration of small domain encoding and lazy approaches in a separation logic solver*, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. 2006.
37. M.R. Garey and D.S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*. 1979: W. H. Freeman and Co.
38. T. Larrabee, *Test pattern generation using Boolean Satisfiability.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1992. **11**(1): p. 4-15.
39. M. Davis, G. Longeman, and D. Loveland, *A Machine Program for Theorem Proving.* Communications of the ACM, 1962. **5**: p. 394-397.
40. L. Zhang and S. Malik, *Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications*, in *Proceedings of Conference on Design Automation & Test Europe (DATE)*. 2003.
41. E. Goldberg and Y. Novikov, *Verification of Proofs of Unsatisfiability for CNF Formulas*, in *Proceedings of Conference on Design Automation & Test Europe (DATE)*. 2003.
42. K.L. McMillan and N. Amla, *Automatic Abstraction Without Counterexamples*, in *Proceedings of Tools for Algorithms for Construction and Analysis of Systems (TACAS)*. 2003.
43. H. Fujiwara and T. Shimono, *On the Acceleration of Test Generation Algorithms.* IEEE Transactions on Computers, 1983. **C-32**(12): p. 265-272.
44. P. Goel, *An implicit enumeration algorithm to generate tests for Combinational circuits.* IEEE Transactions on Computers, 1981. **C-30**(3): p. 215-222.
45. A. Kuehlmann, M. Ganai, and V. Paruthi, *Circuit-based Boolean Reasoning*, in *Proceedings of Design Automation Conference*. 2001.

46. A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, *Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002. **21**(12): p. 1377-1394.
47. M. Iyer, G. Parthasarthy, and K.-T. Cheng, *SATORI -- A fast sequential SAT engine for circuits*, in *Proceedings of the International Conference on Computer-Aided Design*. 2003. p. 320-325.
48. H.S. Jin, M. Awedh, and F. Somenzi, *CirCUs: A satisfiability solver geared toward bounded model checking*, in *Proceedings of the International Conference on Computer Aided Verification*. 2004, Springer. p. 519-522.
49. G.J. Holzmann, *The Model Checker SPIN.* IEEE Transactions of Software Engineering, 1997. **23**(5): p. 279-295.
50. D. Kroening and O. Strichman, *Efficient Computation of Recurrence Diameters*, in *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*. 2003, Springer Verlag.
51. O. Shtrichman, *Tuning SAT Checkers for Bounded Model Checking*, in *Proceedings of International Conference on Computer-Aided Verification*. 2000.
52. O. Shtrichman, *Pruning Techniques for the SAT-based bounded model checking*, in *Proceedings of Workshop on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*. 2001.
53. N. Een and N. Sorensson, *Temporal induction by incremental SAT solving*, in *Proceedings of the First International Workshop on Bounded Model Checking (BMC)*. 2003, Elsevier.
54. J. Baumgartner, A. Kuehlmann, and J. Abraham, *Property Checking via Structural Analysis*, in *Proceedings of CAV*. 2002.
55. M. Mneimneh and K. Sakallah. *SAT-based sequential depth computation*. in *Proceedings of the First International Workshop on Constraints in Formal Verification*. 2002.
56. A. Kuehlmann and F. Krohm, *Equivalence Checking using Cuts and Heaps*, in *Proceedings of Design Automation Conference*. 1997.
57. J. Whittemore, J. Kim, and K. Sakallah, *SATIRE: A new incremental SAT engine*, in *Proceedings of the Design Automation Conference*. 2001.
58. M. Ganai and A. Aziz, *Improved SAT-based Bounded Reachability Analysis*, in *Proceedings of VLSI Design Conference*. 2002.
59. R. Brayton, F. Somenzi, and others, *VIS: Verification Interacting with Synthesis, http://vlsi.colorado.edu/~vis*. 2002.
60. Cabodi, S. Nocco, and S. Quer, *Improving SAT-based bounded model checking by means of BDD-based approximate traversals*, in *Proceedings of Design Automation and Test Europe*. 2003. p. 898-903.
61. R.P. Kurshan, *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*. 1994: Princeton University Press.
62. Y. Zhao, *Accelerating Boolean Satisfiability through Application Specific Processing*, in *Electrical Engineering*. 2001, Princeton University.
63. J.R. Burch and D. Dill, *Automatic verification of pipelined microprocessor control*, in *Proceedings of the International Conference on Computer Aided Verification*. 1994.
64. R.E. Bryant, S. German, and M.N. Velev, *Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic*, in *Proceedings of the International Conference on Computer Aided Verification*. 1999.
65. M.N. Velev, *Automatic abstraction of memories in the formal verification of superscalar microprocessors*, in *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2001.

66. R.E. Bryant, S. Lahiri, and S. Seshia, *Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions*, in *Proceedings of Conference on Computer Aided Verification*. 2002.
67. P. Bjesse and K. Claessen, *SAT-based verification without state space traversal*, in *Proceedings of Conference on Formal Methods in Computer-Aided Design*. 2000.
68. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Counterexample-guided abstraction refinement*, in *Proceedings of Conference on Computer Aided Verification*. 2000. p. 154-169.
69. P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang, *Automated Abstraction Refinement for Model Checking Large State Spaces using SAT based Conflict Analysis*, in *Proceedings of Conference on Formal Methods in CAD (FMCAD)*. 2002.
70. K.L. McMillan, *Interpolation and SAT-based Model Checking*, in *Proceedings of Conference on Computer-Aided Verification*. 2003.
71. P.A. Abdulla, P. Bjesse, and N. Een, *Symbolic Reachability Analysis based on {SAT}-Solvers*, in *Proceedings of Workshop on Tools and Algorithms for the Analysis and Construction of Systems (TACAS)*. 2000.
72. P. Williams, A. Biere, E.M. Clarke, and A. Gupta, *Combining Decision Diagrams and SAT Procedures for Efficient Symbolic Model Checking*, in *Proceedings of International Conference on Computer-Aided Verification*. 2000. p. 124-138.
73. K.L. McMillan, *Applying SAT methods in unbounded symbolic model checking*, in *Proceedings of the International Conference on Computer Aided Verification*. 2002, Springer.
74. H.-J. Kang and I.-C. Park, *SAT-based unbounded symbolic model checking*, in *Proceedings of the Design Automation Conference*. 2003.
75. D. Tang, S. Malik, A. Gupta, and N. Ip, *Symmetry reduction in SAT-based model checking*, in *Proceedings of the International Conference on Computer Aided Verification*. 2005. p. 125-138.
76. *VIS Home page. http://www-cad.eecs.berkeley.edu/~vis*.
77. A. Gupta, A.A. Bayazit, and Y. Mahajan, *Verification Languages*, in *The Industrial Information Technology Handbook*. 2005, CRC Press.
78. S. Williams, *Icarus Verilog. http://www.icarus.com/eda/verilog*.