Verification of Large Industrial Circuits Using SAT Based Reparameterization and Automated Abstraction-Refinement

Pankaj Chauhan
Ph.D. Thesis Proposal
Carnegie Mellon University

November 25, 2003

Abstract

Automatic formal verification of large industrial circuits with thousands of latches is still a major challenge today due to the state space explosion problem. Moreover, BDD based algorithms are very sensitive to the variable ordering. Satisfiability (SAT) procedures have become much more powerful over the last few years, and hence they have been used in formal verification of large circuits with techniques like automated abstraction refinement and ATPG. In the first part of my thesis, I propose to investigate the use of SAT based image computation procedures in symbolic model checking. The method does not use BDDs at all to represent the characteristic functions of sets of states. Instead, the sets of states and transition relation are represented in clausal form, which can be processed by SAT checkers. I propose a SAT based enumeration algorithm to generate the set of newly reached states in clausal form. This algorithm is demonstrated to be relatively immune to the number of variables that need to be existentially quantified in the enumeration.

However, all verification algorithms based on characteristic function representations are inherently limited. Symbolic simulation overcomes this limitation by implicitly representing the set of states as a vector of functions in *parametric form*, one function for each state variable. Techniques like bounded model checking (BMC) and various symbolic trajectory evaluation (STE) algorithms use this implicit representation. The image computation step is simple in symbolic simulation, as it just amounts to unwinding the transition relation of the system. Moreover, the size of the representation of the set of states is linear in the number of simulation steps. There are two main problems with existing algorithms: First, even a linear growth in the sizes of the formulae can be prohibitive for large circuits, thus limiting the simulation depth. BDD based symbolic simulation tries to address this issue by reparameterization, i.e., converting a parametric form into another another smaller parametric form. However, this is not applicable to BMC. More seriously, BDD based reparameterization algorithms are ineffective at handling all but small designs as the large number of variables that need to be quantified during reparameterization adversely affect the performance. The number of variables to be quantified grows linearly with the the simulation depth and the circuit size. I propose a novel SAT based reparameterization algorithm that solves these problems. The algorithm uses the SAT based enumeration procedure described earlier for quantification. First, the SAT based reparameterization algorithm builds Boolean expressions, which do not suffer from variable ordering problems, and can be used in BMC equally well. Next, the reparameterization algorithm is largely unaffected by the number of variables that are existentially quantified in the enumeration, thus enabling the simulation to go much deeper than existing techniques. I demonstrated a 3x improvement in the runtime and space requirement over existing BMC algorithm and BDD based symbolic simulator on large industrial circuits. The symbolic simulator using SAT based reparameterization algorithm forms the central part of my thesis. The reparameterization algorithm benefits from improvements in the SAT based enumeration procedure presented earlier. I will investigate nonclausal representations for the enumeration algorithm as there are known functions for which no concise clausal representations are possible. The size of a parametric representation is affected by the relative ordering of the state variables. I propose to investigate variable ordering techniques used in BDD based image computation for this purpose.

The integration of the reparameterization algorithm in a SAT based abstraction-refinement framework concludes my proposal. My algorithm can simulate much longer abstract counterexamples than currently possible. I propose to extract refinement information from the simulation, as was done in my earlier work [FMCAD'02]. Using the reparameterization algorithm in the SAT based abstraction-refinement framework will make the algorithm complete for safety properties. When the property is true, abstract model checking is used to infer that fact, and if the property is false, the symbolic simulator will provide a counterexample. Recently, Biere et al. have demonstrated how a liveness property can be reduced to safety property by a semantic translation and introduction of auxiliary variables in the transition system. Using reparameterization algorithm in the abstraction-refinement framework thus allows us to prove of both safety and liveness properties.

1 Introduction

State explosion [BCM+92] remains a major hurdle to the wide scale applicability of symbolic model checking [CGP00, CE81]. With the ever increasing complexity and enormous size of current industrial designs with thousands of state variables, automatic verification requires improvements on many fronts. Symbolic model checking traverses the state space of a system in a systematic manner in order to prove that a property is true of the system or exhibits a counterexample otherwise. Properties are given in some form of temporal logic, either linear temporal logic (LTL) or computation tree logic (CTL). The properties are either proved or disproved by fixed point algorithms that traverse state space of systems and compute set of states related to the property.

The first key element of a symbolic model checker is the state set representation. Traditionally, characteristic functions have been used to represent the set of states. For example, the set of states $\{10, 01, 11\}$ in two state bits v_1 and v_2 is given by the characteristic function $\xi(V) = v_1 \vee v_2$. Binary decision diagrams (BDDs) [Bry86] are the most commonly used data structures for representing boolean functions, including characteristic functions, transition relations, etc. However, the state explosion problem manifests itself in the form of unmanageable large BDDs [BCM+92]. BDD sizes are also sensitive to the variable ordering. Due to these limitations, BDD based model checking that uses characteristic functions is reliably applied to circuits with only a few hundred state variables. Alternative data structures, including a plethora of decision diagrams, are used to handle certain class of problems. Examples include BMDs (Binary Moment Diagrams, used for arithmetic circuits), ZDDs (zero suppressed diagrams), BEDs (Boolean expression diagrams), and so on. A fundamental limitation of characteristic function based approaches is the requirement that a set of states be represented by a single function. However, most industrial circuits are in *functional form*, i.e., the system is given as a collection of next state functions. In this form, computing a single characteristic function representing set of states becomes difficult, as the number of state variables grows. This motivates non-characteristic function based representation of a set of states, namely, *parametric form* [CM90]. In parametric representation, a set of states is implicitly represented by the range of a vector of n functions, where n is the number of state variables. For the set $\{01, 10, 11\}$, one representation is the vector $(f_1(p_1 \wedge p_2), f_2(\neg(p_1 \wedge p2) \vee p_2)$. Here, f_1 computes the values for the first state bit, while f_2 computes it for the second. Such an implicit representation is used in Bounded Model Checking (BMC) [BCC+99, BCCZ99] and various forms of symbolic trajectory evaluation (STE) [BBS91, SB95, YS00, YS02] algorithms. In BMC, the transition relation for a complex circuit and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure such as GRASP [SS96] or Chaff [MMZ+01]. The satisfiability of the SAT formula implies that the set of states reachable in a bounded number of steps violate the property of interest. In STE algorithms, Boolean expressions for state bits are represented by BDDs and algorithms for set union and fixed point detection are given in parametric forms.

The next key element of a symbolic model checker is *image computation*, i.e., computing the set of states that can be reached from a set of states in one transition. The reverse of image computation is pre-image computation, i.e., computation of the set of states that can reach a given set of states in one transition. The choice of state set representation dictates the method used for image computation. For characteristic function based approaches, the characteristic function for representing the image $Img(S)$ of a set S of states is computed by quantification of the present state variables and the input variables I as below:

$$\mathbf{Img}(S) := \{v' : \exists v.\exists I.R(v, I, v') \wedge S(v)\}.$$

Image computation is one of the major bottlenecks in BDD based verification. Often it is impossible to construct a single BDD for the transition relation R. Instead, R is represented as a *partitioned transition relation*, i.e., as the conjunction of several BDDs, each representing a piece of R. The problem is how to compute $\mathbf{Img}(S)$ without actually computing R. As the above definition of $\mathbf{Img}$ indicates, the process of image computation involves quantifying state variables. In the BDD representation, this amounts to quantifying over several Boolean state variables. *Early quantification* [BCL91b, TSL+90] is a technique which attempts to reorder the conjuncts so that the scope of each quantifier is minimized. The effect of early quantification is that the evaluation of single quantifiers can be done over *relatively small intermediate BDDs.* The success of early quantification hinges heavily upon the derivation and ordering of sub-relations. Significant effort has been directed over the last decade to this problem. Since the problem is known to be NP-hard, various heuristics have been proposed for the problem. For symbolic simulation, the image computation step is much simpler than that for characteristic functions. Since the set of state is implicitly represented as a vector of n functions, image computation is done by unwinding the circuit once. The current set of functions for state variables are applied as the present state variables in the circuit and fresh input variables are introduced for circuit inputs. A circuit can thus be simulated for a large number of steps by unwinding the circuit. If Boolean expressions are used for instead of BDDs, the size of representation can be guaranteed to be linear in the size of the circuit and the number of simulation steps. In BMC for example, the size of the SAT instance grows linearly with the unwinding depth. However, for very large circuits, even linear growth can be prohibitive: Either the formula already exceeds the memory limits, or the SAT instance is too hard for the SAT solver. No attempt is made to reduce the size of the representation.

Given the set representation and image computation algorithms, symbolic model checker than explores the state space for verification of the given property. This is the last key element of a symbolic model checker. Usually, the state space exploration is done only on the relevant portion of the system by abstraction. For characteristic function based approaches, various fixed point algorithms are given for computing the set of states satisfying a given property. In BMC, the set of states where a property is not computed, instead a SAT formula is constructed that denotes the failure of the property. If the formula is satisfiable, a counterexample can be extracted from the output of the SAT procedure. If the formula is not satisfiable, the circuit and its specification can be unwound more to determine if a longer counterexample exists. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [KS03]) or when the SAT procedure exceeds its time or memory bounds. BMC has been used successfully to find subtle errors in very large circuits [Sht00, CFF+01]. As noted earlier, even the linear growth in the formula sizes is prohibitive for large circuits. This severely restricts the depth to which a circuit can be unwound. As noted earlier, there can be multiple parametric representations for a set of states. For the set $\{01, 10, 11\}$, another smaller parametric representation is $(f_1(p_1), f_2(\neg p_1 \vee p_2))$. Thus, a method to compute smaller parametric representations would increase the capacity of bounded model checkers many fold. The increase in the capacity of bounded model checking is helpful in more than one way. BMC forms a central procedure for simulating concrete (as opposed to abstract) systems for a bounded depth in automatic abstraction refinement algorithms as described below.

Abstraction algorithms reduce the size of the design by focusing only on relevant portions of the design. A conservative abstraction is one which preserves all behaviors of a concrete system. Conservative abstractions benefit from a *preservation* theorem which states that the correctness of any universal (e.g. ACTL*) formulas on an abstract system automatically implies the correctness of the formula on the concrete system. However, a counterexample on an abstract system may not correspond to any real path, in which case it is called a *spurious* counterexample. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. It is obviously desirable to automate this procedure. Abstraction refinement usually involves the following process.

1. **Generation of Initial Abstraction.** It is desirable to derive the initial abstraction automatically.

2. **Model checking of abstract system**. If this results in a conclusive answer for the abstract system, then the process is terminated. For example, in case of existential abstraction, a "yes" answer for an ACTL* property in this step means that the concrete system also satisfies the property, and we can stop. However, if the property is false on the abstract system, an abstract counterexample is generated.

3. **Checking whether the counterexample holds on the concrete system**. If the counterexample is valid, then we have actually found a bug. Otherwise, the counterexample is *spurious* and the abstraction needs to be refined. In counterexample guided abstraction refinement (CEGAR), refinement of abstraction is based on the analysis of

There are many methods for abstraction refinement. In the counterexample guided abstraction refinement framework (CEGAR) [CGJ+00, CCS+02], an abstract counterexample is then used to constrain the states in a Bounded Model Checking SAT instance. If the constrained BMC SAT instance is satisfiable, the abstract counterexample can be simulated on the concrete model and a bug is found. If not, the abstraction is refined using various heuristics. In [MA03], this framework is changed as follows: An abstract counterexample is no longer obtained. The only information of interest is the *length* k of the abstract counterexample. This length k is then used as the bound for a normal, unconstrained BMC instance. If the BMC instance is satisfiable, a bug is found. If this is not the case, information from the SAT solver is used to generate the next abstract model. In [McM03], a new framework is introduced: The algorithm initially performs Bounded Model Checking for some k steps in order to refute the property. If this fails, the proof of unsatisfiability extracted from the SAT solver is used to simplify a fixed-point computation. The purpose of the fixed-point computation is to detect the case when the property actually holds. This may fail, and if so, the algorithm is repeated with an increased value of k.

All abstraction/refinement approaches therefore solely rely on the success of Bounded Model Checking to refute the property. The extensions are used to detect the case that the property is true. Thus any capacity constraints on BMC limits the applicability of these algorithms.

As noted earlier, there can be multiple parametric representations for the same set of states. The central part of this thesis is an algorithm that computes a small parametric representation for the set of states. The algorithm is periodically applied to reduce the size of state set representation. Thus, it enables the symbolic simulation to go much deeper than otherwise possible.

The re-parameterization algorithm is based on the SAT based enumeration procedure that we proposed. Moreover, using the symbolic simulator with reparameterization in abstraction refinement requires many issues to be solved.

1.1 Proposed Research

I propose the following research to address the problems. The general theme of the proposed research is the use of powerful SAT checkers like zChaff. First, at the core, I propose a SAT based image computation algorithm and provide a SAT based enumeration procedure. Building on this enumeration algorithm, I propose a novel SAT based reparameterization algorithm that increases the capacity of symbolic simulation by large extent. These improvements help in bounded model checking of large systems. Finally, the reparameterization algorithm is incorporated in a SAT based abstraction-refinement framework, thus providing completeness for safety and liveness properties. Thus, I propose improvements beginning at the core problem of image computation and continuing it way up to the abstraction-refinement.

1.1.1 SAT Based Image Computation

Satisfiability procedures have shown significant promise for symbolic simulation of large circuits, hence they have been used in many formal verification techniques, including automated abstraction refinement, ATPG etc. I propose how to use modern SAT solvers like Chaff and GRASP to compute images of sets of states and how to efficiently detect fixed point of the sets of states during reachability analysis. Our method is completely SAT based, and does not use BDDs at all. The sets of states and transition relation are represented in clausal form, which can be processed by SAT checkers. The SAT checker subsequently generates the set of newly reached states in clausal form as well. At the heart of our engine lie two efficient algorithms. The first algorithm shortens the cubes that the SAT checker generates by a static-analysis algorithm, which significantly reduces the number of cubes the SAT checker needs to enumerate. The second algorithm reduces the space required to store sets of states as a set of cubes by a recursive cube-merging procedure. The effectiveness of the SAT based image computation procedure on ISCAS sequential benchmarks for reachability. In particular, the algorithm does not have BDD size explosion surprises and deteriorates in a predictable manner. There are many improvements to be done to the enumeration procedure. A major improvement will be the use of non-clausal representation for state sets. The interpolation proof based approach [McM03] provides one possibility to explore. I also propose to explore is whether techniques like iterative squaring can be used with SAT to speed up fix-point detection in reachability. We want to extend this procedure for solving general quantified Boolean formulas (QBFs). Our algorithm was used in [LBC03] for solving QBFs with one existential quantifier that arise in symbolic predicate abstraction. We are also identifying other problem domains where our method can be applied. This approach is still limited by the single characteristic function representation for state sets.

1.1.2 SAT Based Reparameterization

I describe a SAT-based algorithm to perform the reparameterization step for symbolic simulation. The algorithm performs better than BDD-based reparameterization especially in the presence of many input variables. The algorithm takes arbitrary Boolean equations as input. Therefore, it does not require BDDs for the symbolic simulation. Instead, non-canonical forms that grow linearly with the number of simulation steps can be used. In essence, the SAT-based reparameterization algorithm computes a new parametric function for each state variable one at a time. In each computation, a large number of input variables are quantified by a single call to a SAT-based enumeration procedure [McM02, CCK03b]. The advantage of this approach is twofold: First, all input variables are quantified at the same time, and second, the performance of SAT-based enumeration procedure is largely unaffected by the number of input variables that are quantified. I demonstrated the efficiency of this new technique using large industrial circuits with thousands of latches. I compared it to both SAT-based Bounded Model Checking and BDD-based symbolic simulation. This new algorithm can go much deeper than a standard Bounded Model Checker can. Moreover, the overall memory consumption and the run times are, on average, 3 times less than the values measured using a Bounded Model Checker. The BDD-based symbolic simulator could not even verify most of the circuits that we used. There are various research problems to be solved for using this algorithm for complete verification of safety and liveness properties.

Safety property checking requires generation of a SAT formula from reparameterized form. For the symbolic simulator, the counterexample generation is nontrivial, since we do not keep the whole simulation. Periodically, we reparameterize the representation and hence lose the information about input variables up to that point. I propose algorithms for safety property checking and counterexample generation for the symbolic simulator. Invariant statements are often used to restrict the state space for verification. Such invariants are often called *verification conditions* [Jon99]. The technique described so far assumes a transition function. It does not allow an arbitrary transition relation. I propose an approach to handle invariant constraints.

The algorithm can also benefit from the improvements in the basic SAT enumeration algorithm. One limitation of the SAT enumeration algorithm is the clausal (CNF or DNF) representation it uses. There are certain class of Boolean functions which have no compact clausal representation. I plan to investigate non-clausal representations in SAT enumeration procedures. There has been some existing work in SAT based enumeration algorithms [McM02, KP03, SH03] , and I plan to investigate the efficiency of them in reparameterization. At present, the symbolic simulator handles only safety properties. Biere et al. [SB03] recently showed that checking for liveness properties can be done by a semantic translation of Note that, my symbolic simulator in its present from does not allow for proving that safety properties are correct.

1.1.3 Abstraction-Refinement and SAT Based Reparameterization

Abstraction-refinement algorithms that simulate concrete systems benefit immensely from a powerful simulator. I plan to use my symbolic simulation algorithm for simulating abstract counterexamples on concrete systems as in [CCS+02] or for doing BMC on concrete systems as in [MA03]. The refinement information in both approaches is obtained from the analysis of failed SAT instances (these correspond to spurious counterexamples, meaning the abstraction needs to refined). One issue with using reparameterization algorithm for refinement is that the SAT problem does not contain the information of the simulation from the initial state. The SAT problem only contains the trace from the last time frame when reparameterization was done to the length of the failed counterexample state. I propose to investigate the effect of doing refinement based on such truncated counterexamples. Integrating my symbolic simulation with abstraction-refinement will allow complete model checking of safety properties. Along with the semantic translation of liveness properties to safety properties as in [SB03], handling liveness properties should also be possible.

The final result of my research is an implementation in a software package **symbot** for **symbo**lic **t**raversal. The abstraction-refinement framework will use standard NuSMV model checker for model checking abstract models.

Outline

The rest of the proposal is organized as follows. In Section 2, I describe the problem of image computation and reachability analysis. The next Section 3 describes in detail the SAT based enumeration algorithm that we proposed, related work, the experimental results that were obtained for reachability analysis using this algorithm and the improvements that I propose for further research. The section also provides a brief description of SAT procedures. In Section 4, I briefly describe the earlier work that I did on quantification scheduling problem in the image computation problem. The techniques are applicable to the state variable ordering problem in reparameterization algorithm. The Section 5 forms the bulk of this proposal and I describe in detail the reparameterization algorithm, the results that we obtained for doing bounded model checking of some large industrial circuits with my tool, and the proposed research directions. In Section 6, I describe the SAT based counterexample guided abstraction-refinement framework and describe how my symbolic simulation algorithm can be used in the framework. I also briefly mention the translation from liveness properties to safety properties of [SB03] in section 6. I conclude with a timeline of my research in Section 7. I would like to point out that the work on SAT based image computation and reparameterization algorithm was done with Daniel Kroening.

2 Image Computation and Reachability Analysis

Computing the set of states reachable in one step from a given set of states under a transition relation forms the heart of many symbolic state exploration algorithms, including reachability analysis, model checking [CGP00, CE81, CES86], etc. This operation is called *image computation*. Let us consider a state transition relation T over the set of states S. The set of states is defined by the set of valuations over a vector of state variables $\mathbf{x}$. We denote a set or a vector of variables in a boldface. The transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ relates states defined by valuations of present state variables $\mathbf{x}$ and inputs $\mathbf{i}$ to states defined by valuations of next state variables $\mathbf{x}'$. Note that we are using characteristic functions $S(\mathbf{x})$ and $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ to represent set of states and set of transitions. The image of $S(\mathbf{x})$ under $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ is given by the following equation.

$$Img(S(\mathbf{x}')) = \exists \mathbf{x}.T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge S(\mathbf{x}) \tag{1}$$

Image computation is a major bottleneck in verification. For BDD based image computation, often it is infeasible to construct BDDs for the transition relation T and the set S. In SAT based image computation, we use propositional formulas in CNF with intermediate variables to represent these sets, often resulting in much smaller representation. The definition of image computation involves evaluation of a simple *quantified Boolean formula*.

In reachability analysis, beginning with the set of initial states S_0, images are repeatedly computed until the set of states does not grow any more, in other words, the least fixed point of the following equation is computed.

$$X = \mu X.(S_0 \cup Img(X)) \tag{2}$$

Following simple algorithm computes this fixed point.

REACHABILITY(S_0)
$S_{reach} \leftarrow \phi$
$i \leftarrow 0$
while $(S_i \neq \phi)$ {
    $S_{reach} \leftarrow S_{reach} \cup S_i$
    $S_{i+1} \leftarrow Img(S_i) \backslash S_{reach}$
    $i \leftarrow i + 1$
}
return S_{reach}

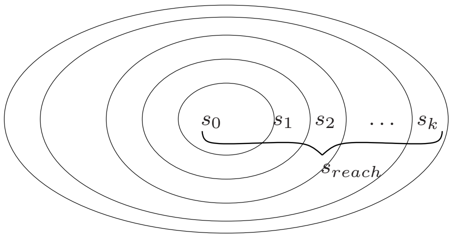

Figure 1: Reachability algorithm

In this algorithm, S_i denotes the set of newly discovered states in each iteration. Once there are no more states to be discovered, we have reached a fixed point.

As noted earlier, the sets of states and sets of transitions are traditionally represented by BDDs ([Bry86, CGP00, BCMD90, BCL91b]). It is well known that while BDDs are compact representations of many functions, they unfortunately suffer from size explosion for many circuits. A BDD based model checker is like a black box. A slight change in circuit or variable order can make model checking infeasible. Moreover, there are some functions like multipliers, where the BDDs are always exponentially large in the number of variables. BDD based model checkers do not have a gradual degradation in performance, and the performance is often not predictable. We offer a SAT procedure based image computation and fixed point detection mechanism that is robust and degrades gradually. The runtime of our algorithm depends on the size of the input circuit and the diameter of the circuit only.

3 SAT Based Reachability Analysis

3.1 SAT Procedures

A SAT solvers reads a formula in conjunctive normal form (CNF) and finds a satisfying assignment if there is any. If not, the solver returns that the formula is unsatisfiable. SAT solving is one of the classical NP-complete problems. Over the last 4 years, propositional SAT checkers have demonstrated tremendous success on various classes of SAT formulas. The key to the effectiveness of SAT checkers like Chaff [MMZ+01], GRASP [SS96], and SATO [Zha97] is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

SAT checkers have been successfully used for Bounded Model Checking (BMC) [BCCZ99], where the design under consideration is unrolled and the property is symbolically verified using SAT procedures. BMC is effective for showing the presence of errors. However, BMC is not at all effective for showing that a specification is true unless the diameter of the state space is known. Moreover, BMC performance degrades when searching for deep counterexamples. The basic problem with BMC is that there is no mechanism to detect whether a fixed point has been reached while exploring state space. A more serious problem is that the transition relation is unrolled for progressively increasing number of steps; hence, searching for deeper counterexamples becomes impractical. Our algorithm is used to detect fixed points in image computations and the SAT checker never has to deal with multiple unrollings of the transition relation. In each SAT checker run, only one step of the image computation is done at a time.

The efficiency of SAT procedures has made it possible to handle circuits with a few thousand of variables, much larger than any BDD based model checker is able to do at present.

```
while(1) {
   if (decide_next_branch()) {        // Branching
      while (deduce() == conflict) {  // Propagate implications
         blevel = analyse_conflict(); // Learning
         if (blevel == 0)
            return UNSAT;
         else
            backtrack(blevel);        // Non-chronological
                                      // backtrack
      }
   }
   else                               // no branch means all vars
                                      // have been assigned
      return SAT;
}
```

Figure 2: Basic DPLL backtracking search (used from [MMZ+01] for illustration purposes)

The basic framework for these SAT procedures, shown in Figure 2, is based on Davis-Putnam-Longeman-Loveland (DPLL) backtracking search. The function `decide_next_branch()` chooses the branching variable at current *decision level*. The function `deduce()` does *Boolean constraint propagation* to deduce further assignments. In the process, it might infer that the present set of assignments to variables does not lead to any satisfying solution. This is termed as a conflict, as at least one CNF clause remains unsatisfied. In case of a conflict, new clauses are learned by `analyze_conflict()` that hopefully prevent the same unsuccessful search in the future. The conflict analysis also returns a variable for which another value should be tried. This variable may not be the most recent variable decided, leading to a *non-chronological* backtrack. If all variables have been decided, then we have found a satisfying assignment and the procedure returns. The strength of various SAT checkers lies in their implementation of constraint propagation, non-chronological backtracking, decision heuristics, and learning.

Modern SAT checkers work by introducing conflict clauses in the learning phase and by non-chronological backtracking. Implication graphs are used for Boolean constraint propagation. The vertexes of this graph are literals, and each edge is labeled with the clause that forces the assignment. When a clause becomes unsatisfiable as a result of the current set of assignments (decision assignments or implied assignments), a conflict clause is introduced to record the cause of the conflict, so that the same futile search is never repeated. The conflict clause is learned from the structure of the implication graph. When the search backtracks, it backtracks to the most recent variable in the conflict clause just added, not to the variable that was assigned last.

In our enumeration algorithm, we use the Chaff SAT checker [MMZ+01], as it has been demonstrated to be the most powerful SAT checker on a wide class of problems.

3.2 High Level Description of the SAT Based Reachability Algorithm

SAT checkers like Chaff read propositional formulas represented in conjunctive normal forms (CNFs). We present an algorithm that does not use any BDDs. We assume that the transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ is already represented in as a set of CNF clauses. It is customary to convert any transition relation represented as a set of propositional formula to CNF form by introducing intermediate variables. This translation is polynomial in the size of the original circuit. We represent the set of newly reached states after each iteration of the reachability loop (S_i in Figure 1) as a set of disjunctive normal form (DNF) cubes. The set of all reachable states after each step (S_{reach}) is also represented in DNF. Since S_{reach} is in DNF, $\neg S_{reach}$ will be automatically in CNF. As Chaff needs CNF representation, we convert S_i from DNF to CNF by introducing intermediate variables. In each iteration i, we ask the SAT checker to find satisfying assignments to the formula below.

$$S_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge \neg S_{reach}(\mathbf{x}') \tag{3}$$

This formula corresponds to step 5 of the basic reachability algorithm (Fig. 1). This formula asks the SAT checker to compute a satisfying assignment such that the present state variables $\mathbf{x}$ and input variables $\mathbf{i}$ satisfy the predicate $S_{i-1}(\mathbf{x}) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$, i. e., the set of states reachable from the newly discovered states in the previous iteration. We conjoin with this the negation of the set of all accumulated states so far ($\neg S_{reach}$), thus we ask SAT checker to compute only the states that have not been seen so far. If the SAT checker concludes that the formula is unsatisfiable, then it means that the set of newly reached states S_i is empty, and we have reached fixed point. On the other hand, if the SAT checker finds a satisfying assignment to this formula in present state $\mathbf{x}$, input $\mathbf{i}$, intermediate and $\mathbf{x}'$ variables, the projection of this assignment on $\mathbf{x}'$ variables gives a subset of newly reached states. Note that this partial assignment to $\mathbf{x}'$ is consistent with the full assignment that the SAT checker finds to the Formula 3. The Formula 3 describes all constraints on the next set of states. Therefore, the projection is a valid state reachable from S_{i-1} following the transition relation T. Therefore, the following lemma easily follows.

/* It is assumed that S_0 is given in DNF form */
SATREACHABILITY(S_0)
1 $i \leftarrow 1$
2 $S_{reach} \leftarrow S_0$
3 **while** ($S_{i-1} \neq \phi$) {
4 $\quad S_i \leftarrow \phi$
$\quad$ /* $DNFtoCNF$ converts a formula to CNF by introducing intermediate variables */
5 $\quad$ **for** (each satisfying partial assignment $\mathbf{d}$ in $\mathbf{x}'$ to
$\quad\quad DNFtoCNF(S_{i-1}(\mathbf{x})) \wedge T(\mathbf{x}, \mathbf{i}, \mathbf{x}') \wedge \neg S_{reach}(\mathbf{x}')$)
$\quad\quad$ /* $\mathbf{d}$ contains only next state variables */
6 $\quad\quad \mathbf{d}' \leftarrow EnlargeCube(\mathbf{d})$
7 $\quad\quad$ Add $\neg\mathbf{d}'$ as a blocking clause
8 $\quad\quad S_i \leftarrow AddCube(S_i, next2current(\mathbf{d}'))$
9 $\quad\quad S_{reach} \leftarrow AddCube(S_{reach}, \mathbf{d}')$
10 $\quad$ **endfor**
11 }
12 **return** $next2current(S_{reach})$

Figure 3: Outline of SAT based reachability algorithm

Lemma 3.1 *The projection of any partial satisfying assignment to Equation 3 in $\mathbf{x}$, $\mathbf{i}$, $\mathbf{x}'$ and intermediate variables to just $\mathbf{x}'$ is a valid partial assignment in $\mathbf{x}'$ describing a newly discovered state reachable from S_{i-1} following T.*

We add this state to $S_i(\mathbf{x})$ as a DNF cube $\mathbf{d}$, after translating the next state variables in the cube to present state variables. The negation of $\mathbf{d}$ is a CNF clause, which is added as a conflict clause in the SAT engine. This clause $\neg\mathbf{d}$ is called a *blocking clause*. Thus after finding each satisfying assignment, the set $S_{reach}(\mathbf{x}')$ grows.

We present the high level algorithm in Figure 3. The algorithm has two loops. The outer loop carries out different steps of image computation, while the inner loop is implicit in the SAT checker, and enumerates sets of the newly reached states in a particular step.

Each satisfying cube $\mathbf{d}$ is added to S_i and S_{reach} after enlarging it to $\mathbf{d}'$ in step 6. The addition of $\mathbf{d}'$ to S_{reach} is done in the SAT checker when the blocking clause $\neg\mathbf{d}'$ is added. As noted earlier, negation of S_{reach} is automatically in CNF.

3.3 Efficient Implementation of SAT Based Reachability

This algorithm, as it is, is very inefficient and hence impractical. The first problem comes from the way the SAT checker computes satisfying assignments or cubes. Chaff SAT checker gives values to all variables in any assignment. We then project this assignment to $\mathbf{d}$, which assigns values to all next state variables $\mathbf{x}'$. Therefore $\mathbf{d}$ describes only one newly reached states. Enumerating states one at a time is clearly very inefficient. However, most times, only a partial assignment to all variables satisfies the clause database given to SAT. A partial assignment to $\mathbf{x}'$ describes more than one state at a time, the larger the set the few the number of assignments. This is advantageous in two ways, first the blocking clause for $\mathbf{d}$ prunes the SAT search space drastically, second, the number of state enumerations required go down considerably. Therefore, it is desired that the partial assignment be as small as possible. It is clearly to our advantage to get as small cubes as possible, since smaller cubes cover a larger number of assignments. Given a cube computed by Chaff, it may be possible to throw away certain assignments from the cube, and still get a satisfying cube. By a static analysis of the transition relation, we infer the unnecessary assignments in $\mathbf{d}$. This procedure *EnlargeCube* is called in line 6 on $\mathbf{d}$ to get a smaller cube $\mathbf{d}'$.

The second problem is that the representation of the sets S_{reach} and even S_i can grow too large. For example, if we consider a counter that counts up to 2^{30}, each iteration of the while loop will add only one state to S_{reach}. Thus we will have to represent 2^{30} clauses for S_{reach}. However, the DNF clause 1 represents all possible values of the counter. In other words, after a satisfying assignment to S_{reach} is found, we can combine multiple clauses to get a smaller partial assignment. For example, the DNF clauses $x_1 \wedge x_5 \wedge \neg x_6$ and $x_1 \wedge x_5 \wedge x_6$ can be combined to $x_1 \wedge x_5$. An efficient data structure is needed to support this *AddClause* operation, since many clauses may be added, and each clause can potentially be combined with more than one existing clause. We use a hash table, eacy entry of which contains a trie.

We give a cube enlargement heuristic procedure next, which is followed by a description of an efficient data structure that stores S_i and S_{reach}. The enlargement procedure reduces the number of set enumerations, hence the amount of time, while the seconf procedure reduces the space requirement.

3.3.1 Cube Enlargement

There are five types of variables that appear in the SAT Formula 3: present state variables $\mathbf{x}$, circuit inputs i, next state variables $\mathbf{x}'$, intermediate variables $\mathbf{i_s}$ introduced while converting S_{i-1} to CNF, and the intermediate variables $\mathbf{i_t}$ introduced while converting the transition relation $T(\mathbf{x}, \mathbf{i}, \mathbf{x}')$ to CNF. The SAT checker finds a satisfying assignment $\mathbf{c}$, possibly to all these variables. However, the cube $\mathbf{d}$ of line 7 in the algorithm (Fig. 3) is just in terms of $\mathbf{x}'$ variables. In order to reduce the number of assignments in $\mathbf{d}$, we present the following procedure. This procedure assumes that the transition relation is given in functional form, i.e., there is a transition function $f_i(\mathbf{x})$ for each next state variables x_i. Let $Supp(f_i)$ denote the support set of f_i, i.e., the variables that f_i depends on. This assumption is true for circuit descriptions. When an assignment to a next state variable x'_i can be ignored, we say that x'_i is *.

Free Variables

First, we describe the concept of free variables, i.e., the variables that are free to choose any value, despite SAT checker assigning them specific values. In order to detect whether the variable v is free or not, the following conservative tests are used. If v is an input variable or an intermediate variable, then it is definitely free. Moreover, for functional transition relations, we don't even need to check if an intermediate variable appears in other transition functions or not, since intermediate variables are generated from local translation for f_i. The only real problem is if v is a present state variable. The only constraints that are placed on the present state variables are from S_{i-1}. To see if the present satisfying assignment $\mathbf{c}$ restricts v or not, we can just check that assignment to v does not affect the present satisfying assignment. This can be efficiently detected as follows. While translating the DNF corresponding to S_{i-1} to CNF, we introduce one intermediate variable for each DNF cube. In essence, the truth value of each DNF cube is captured in the corresponding variable. Suppose $i_1, i_2, \ldots, i_k$ are the intermediate variables corresponding to DNF cubes $D_1, D_2, \ldots, D_k$ in S_{i-1}. The translation of the S_{i-1} constraint in Eqn. 3 looks like:

$$(i_1 \vee i_2 \vee \ldots \vee i_k) \wedge (i_1 \Leftrightarrow D_1) \wedge (i_2 \Leftrightarrow D_2) \wedge \ldots \wedge (i_k \Leftrightarrow D_k) \tag{4}$$

Each equality $i_j \Leftrightarrow D_j$ gives rise to a set of CNF clauses, which we haven't expanded for the sake of brevity.

If the satisfying assignment c makes any of intermediate variables true, the corresponding DNF cube is true, and we don't need to see if any other DNF cube is true or not, since the truth of only one DNF cube satisfies the S_{i-1} clauses. So we find the first intermediate variable i_l that is set to true. All present state variables *not mentioned in the DNF cube D_l are irrelevant* for satisfying S_{i-1} constraint, hence, they can be assumed to be free.

Free Transition Functions

Let us denote the set of free variables in the support of a transition function f_i as $FreeSupp(f_i)$.

The main idea is that if a transition function f_i (for x'_i) depends on a variable v (which is either a present state variable, input or an intermediate variable from $\mathbf{i}_t$), and the following conditions are satisfied, we can guarantee that x'_i can be set to either 1 or 0. Thus, the value of x'_i can be safely ignored from the present assignment.

1. Variable v is free.

2. Values of non-free variables in $Supp(f_i)$ are propagated and do not force f_i to a particular value. For example, when $f_i = x_1 \wedge x_2$, x_1 is free, f_i satisfies condition 3, but $x_2 = 0$ in S_{i-1}. This forces f_i to 0. So constant values of non-free variables are propagated first.

3. The function f_i does not share free support with any other transition function, i.e., $FreeSupp(f_i) \cap FreeSupp(f_j) = \phi, j \neq i$. Moreover, the variable v appears in the formula for f_i in exactly one place. In other words, the variable v does not appear in any other propositional function.

Note that there may be other conditions under which x'_i can still choose both values. However, this conditions allow us to do a static analysis of the circuit.

The third condition is too restrictive in practice. Usually, transition functions do share common variables. In order to infer that a next state variable x'_i can assume both values, we can simplify the transition functions by further constant-propagating values of some free variables as well (remember that we already constant-propagate the values of non-free variables). For example, suppose that $f_1 = x_1|i_2$ and $f_2 = x_1|i_3$, and x_1, i_1 and i_3 are free variables. Suppose the SAT assignment is $a = 0, i_1 = 0, i_2 = 1$. Since both f_1 and f_2 share the variable x_1, we can not safely say that both x'_1 and x'_2 are *. However, we can replace x_1 by 0, and propagate the effects, giving us $f_1 = i_2$ and $f_2 = i_3$. Now, both f_1 and f_2 become independent, as can be set to *. Note that since x_1 is a free variable, we could have chosen $x_1 = 1$, different from the SAT assignment. In order to determine which variables to assign values to, till the functions become independent, we use a greedy strategy. We order the variables by the number of times they appear in all transition functions. Beginning with the variable that occurs the most number of times, we keep replacing the variables by constants (from SAT assignment) and propagating the effect, until transition functions become independent and next state variables can be inferred to be *s. In the worst case, all transition functions become constants, whose values agree with the satisfying assignment.

Most analysis of this procedure can be done statically just once. The only changing part is detection of free present state variables. Note that this is just one alternative. There can be other options. For example, we considered using efficient approximate set cover algorithms to find out the literals that cover all clauses of Formula 3. Another option is to use BDD based symbolic simulation to infer multiple cubes. The given cube enlargement procedure produces one smaller cube. However, using BDDs for simulation of the circuit for one step and then applying constants to some of BDD variables to contain the BDD sizes can yield a set of many states at once.

3.3.2 Efficient Set Representation

The set of states are represented as a set of DNF cubes. However, it is easy to see that each new cube that is added to S_{reach} has a potential to be merged with other cubes to form shorter cubes. For example, the boolean function 1 is an exponentially compact representation than four DNF cubes $a \wedge b, \neg a \wedge b, a \wedge \neg b, \neg a \wedge \neg b$. We describe the following procedure to add a cube to the existing set of cubes. We assume that the variables in the cubes are ordered. The set is represented by a hash table, where each hash table entry stores a subset of cubes in a trie form. Each trie stores cubes that are made up of the same CNF variables. The hash table is indexed by the hash computed from a signature of a cube. In the following algorithm (Figure 4), assume that the DNF cube $\mathbf{d}$ is represented as a vector of integers, each integer identifying a particular propositional variable, negative if the literal is negative, positive otherwise. For example, if $a, b, c, d, e, f...$ are variables, then they are identified by the integers $1, 2, 3, 4, 5, 6, ...$ So a cube $(a \wedge \neg c \wedge \neg f)$ is represented by the vector $\mathbf{d} = [1, -3, -6]$. The function $ComputeSignature$ computes a bit string that is used to compute the hash value for a cube. The bit string is ordered, just as variables in the cubes are, and contains 1 for each variable (in any phase) present and 0 for the variables not present in the cube. The trailing $0s$ are removed to get a shorter bit string. So the signature for the cube $\mathbf{d}$ is 101001. Not that even though there may be variables numbered $7, 8, 9, ..$, the $0s$ corresponding to them do not appear in the signature.

Since each trie stores cubes made up of the same variables, the cubes are represented by bit strings of the same length as the number of variables in the cubes of the trie. Essentially, if a literal is positive, the bit corresponding to it is 1, and 0 otherwise. So the cube $\mathbf{d}$ is stored as 100.

The crux of the $AddCube$ procedure is between lines 5–19. Given an incoming cube $\mathbf{d}$, it tries to find all other cubes from the trie that differ from $\mathbf{d}$ in just one bit. For each such cube $\mathbf{d}'$ found (lines 10–13), the cube computed by merging $\mathbf{d}$ and $\mathbf{d}'$ is added to S by calling $AddCube$ recursively. The merged cube is essentially cube $\mathbf{d}$ with the matched bit (i^{th} bit) removed. If $\mathbf{d}$ doesn't match at the i^{th} bit, then the next bit is checked. Once the traversal over trie is done, we check if $\mathbf{d}$ was merged with anything. If it was, then we no longer keep $\mathbf{d}$. Line 18 just updates the hash table with potentially modified trie.

Note that the algorithm doesn't guarantee absolute minimum cubes. In fact, to do so, we may need to keep all cubes, even after they are merged, in the hopes of merging them with other future cubes. But the main focus of the algorithm is to reduce

```
ComputeSignature(d, m)
/* m is the size of the cube d, assume d is sorted */
j ← 1
for (i from 1 to d[m])
        if (i = |d[j]|) /* variable is present in d */
                s_d[i] ← 1
                j ← j + 1
        else
                s_d[i] ← 0
        endif
endfor
return s_d
```

```
AddCube(S, d, n, m)
/* n is the total number of variables, m is the number of variables in d */
s_d ← ComputeSignature(d, m)
/* T_d is the trie in which d will be stored */
T_d ← HashLookup(S, s_d)
/* compute the representation of d to store in T_d */
for (i from 1 to m)
        b[i] ← (d[i] > 0)?1 : 0
endfor
match ← false
T_d ← TrieInsert(T_d, b)
curr_node ← T_d
for (i from 1 to m)
        b[i] ← 1 − b[i] /* flip the i^th bit */
        if (TrieLookup(curr_node, b[i : m]) = true)
                /* match at the ith bit */
                T_d ← TrieDelete(T_d, b)
                /* insert the merged cube */
                S ← AddCube(S, d[1 : i − 1] :: d[i + 1 : m], n, m)
                match ← true
        endif
        b[i] ← 1 − b[i] /* flip it back to what it was */
        curr_node ← (b[i] = 1)?curr_node.right : curr_node.left
endfor
if (match = true)
        T_d ← TrieDelete(T_d, b)
/* update the trie for this cube */
S ← HashUpdate(S, s_d, T_d)
return S
```

Figure 4: Procedures $AddCube$ and $ComputeSignature$.

space, and not get the absolute smallest cubes. Another point to note is that since the SAT checker always finds new states that haven't been discovered so far, we assume that the trie T_d does not already contain $\mathbf{d}$.

The complexity of this algorithm in the worst case can be $O(n^2)$, not counting the recursive calls. Here n is the number of state variables. Each of line 1, 3–4, 6 and 17 cost $O(n)$, while hash lookups and updates on lines 2 and 18 are essentially constant time operations. Lines 10 and 11 cost $O(m) + O(m-1) + \ldots + O(1) = O(m^2) = O(n^2)$.

3.3.3 Complexity of the Set Representation

The set of DNF cubes representing S_i or S_{reach} possess a useful property. By the negation of S_{reach} in the SAT formula (Eqn. 3), we guarantee that no newly generated DNF cube shares a satisfying assignment with any existing cube in sets S_i or S_{reach}. Thus the set of DNF cubes representing these sets are disjoint, in that they do not have any common assignment. For example, the DNF cube $b \vee c \vee d$ cannot occur if the cube $a \vee b \vee c$ is already present, as they share a common assignment $a = b = c = d = 1$. However, cube $\neg a \vee b \vee c \vee d$ can occur. If we can detect that the set of cubes is a tautology, we can terminate the reachability, as we have reached all the states. Our cube addition algorithm is *online* in nature. We now actually prove that if the set of DNF cubes are given *a priori* that do not share a common a satisfying assignment, then detecting if it is a tautology is polynomial. The general problem of detecting DNF tautology is NP-complete, so is its dual CNF satisfiability.

We call the problem of tautology detection of a set of DNF cubes that do not share any common assignment pairwise an *R-TAUT* problem. The procedure for tautology detection works simply by counting the number of satisfying assignments. Suppose there are c DNF cubes made up from a total of m variables. The cubes do not share common assignments pair wise. Suppose cube i has literals $l_1, l_2, \ldots, l_{c_i}$. There are a total of 2^m possible assignments to variables, and if each assignment is a satisfying assignment, then the DNF cubes are a tautology. Cube i describes a total of 2^{m-c_i} assignments agreeing with the literals in cube i. As we know that no two cubes i and j share any common assignment, the total number of assignments that satisfy both cube i and j are precisely $2^{m-c_i} + 2^{m-c_j}$. The total number of satisfying assignments for the set of DNF cubes is just

$$2^{m-c_1} + 2^{m-c_2} + \ldots + 2^{m-c_c}.$$

Clearly, the additions can be carried out in time polynomial in m and c, the size of problem input. Hence the theorem

Theorem 3.1 *R-$TAUT$ is in P.*

Note that in our case, the set of DNF cubes are not given a priori. However, this procedure can be run periodically on S_{reach} to find out if all states have been reached, in which case we stop the search. We also use this counting mechanism to report the number of reached states.

3.4 Related Work for SAT Based Reachability

The first completely SAT based reachability algorithm was reported by McMillan in [McM02]. The main difference between our approaches is that we represent the set of states in DNF, while he presented it in CNF. A SAT based enumeration algorithm is used to compute a CNF formula equivalent to a given formula characterizing preimages. However, we use intermediate variables to convert DNF representation to CNF while running SAT. He used zero suppressed BDDs (zDDs) to store CNF clauses, and used a SAT conflict analysis based heuristic to enlarge the cubes. We did not compare the results reported in [McM02] as the set of benchmarks in [McM02] was not publicly available.

In [KP03], the authors used procedure similar to ours to compute preimages. They also intermediate variables to convert DNF cubes to CNF formula. However, they use offline Espresso [Rud] algorithm to reduce the number of cubes. Our cube storage procedure is on-line, in the sense that it processes the cubes as they are generated. Moreover, they do not have any algorithm to enlarge the cubes. We doubt their results. They also reported them only on two different circuits and for safety property checking only, which can be much easier than to do than computing reachability. Our attempts to contact them to get more information about the properties they checked failed.

In [SH03], the authors used ATPG instead of SAT to compute preimages. They used BDDS to store the resultant sets of states. ATPG allows reasoning directly on the circuits, hence they do not have any intermediate variables. They report results on only two circuits. These are known to be difficult circuits, however.

In [MS03], the only aim was to compute the sequential diameter, also called *diameter* of the circuit. They did not compute the reachable set of states at all. Their procedure was based on the Chaff SAT checker as well. Their procedures shared many similarities with bounded model checking (BMC) [BCC+99]. They build a SAT formula describing symbolic simulations of increasing length. In our approach, we explicitly compute the set of reachable states, and the SAT checker does not have to compute more than one step of symbolic simulation at a time. This we believe is a significant advantage that our method has over [MS03] and BMC. Using BMC for depth equal to circuit diameter is sufficient for $\mathbf{G}p$ kind of LTL properties. In [KS03], this notion is generalized to that of *completeness threshold* (CT). The authors describe a sorting network built on top of SAT formula for computing diameters.

In [GYAG00, GYA+01], a mixed BDD and SAT based approach to image computation is described. They use SAT procedure to derive a top level disjunctive decomposition of image computation and use BDD based image computation for each leaf subproblem.

3.5 Preliminary Experimental Results for SAT Based Reachability

We implemented our algorithms on top of the zChaff SAT solver. The SAT solver is modified to enumerate satisfying solutions by adding blocking clauses. The program is a part of a higher level verification system called **SATMC.**

We conducted our experiments on a 1.53GHz dual AMD Athlon processor machine with 3GB of memory running Linux. The memory cutoff was set to 1.5GB of resident program size, and the time cut off was set to 1000 seconds. The results are summarized in Table 1. For each circuit, we report the number of latches, the number of reachability steps that we could complete, the number of cubes stored in the representation for the reachable states, the number of cubes that were enumerated as blocking clauses (or how many times line 7 in algorithm of Figure 3 was called), the ratio of the number cubes v/s the number of blocking clauses added in percentage, and the total running time (user time+system time) in seconds. Note that the number of cubes in final set representation is much smaller than the total number of enumerated cubes, as evidenced by the %age size column. This asserts that the space saving data structure for storing cubes is effective. We compare our results with primarily that of [MS03]. We would like to emphasize that in [MS03], only the depth was computed, and the actual set of reachable states was not computed.

The circuits that we report come from mainly three sources, ISCAS'89 benchmarks, IU set of circuits from Synopsys, and some circuits that were translated from Verilog code available from various sources. The IU set of circuits are various abstractions of parts of a picoJava microprocessor implementation.

For a relatively small timeout value, we have been able to do reachability for many circuits. We have outperformed [MS03] by a large magnitude on all but one small completed benchmarks. They used a faster machine (2GHz v/s 1.53GHz) as well. The effectiveness of the cube merging procedure is evident from "%age space" column. The savings are dramatic for circuits that have counter like structures in them (iu**, s208.1, s420.1, s838.1, s635) and other circuits (s1512, s9234, s13207, s1423, s526, s526n). For other circuits, it can be inferred that the SAT checker and cube enlargement procedures generate many disjoint cubes that can not be merged with existing set of cubes. This may be dependent on circuit structure.

In [LBC03], our tool was used to enumerate symbolic solutions to certain predicate abstraction formulas. The characteristic of these formulas was that the number of variables to be quantified was much larger (an order of magnitude) than the number of variables representing the set of states. In our case, the number of variables to be quantified is $\mathbf{x}, \mathbf{I}$, and $\mathbf{i}_{int}$.

3.6 Ongoing and Future Research

I presented a completely SAT based image computation algorithm. The effectiveness of this algorithm is demonstrated for reachability analysis on many large circuits. This algorithm can be used for computing pre images equally well, hence it can be used in a general SAT based symbolic model checking algorithm. The novel features of our algorithm are an efficient data structure for storing sets of states as DNF cubes and a cube enlargement procedure based on static circuit analysis.

There are many improvements to the enumeration algorithm that can be done. The cube enlargement procedure can be improved by many different kind of heuristics, and I plan to investigate whether BDD based symbolic simulation can be used to reduce the clauses. There are different options for cube storage. I am investigating the use of some decision diagrams for this purpose, in the place of the hashing data structure that we have. The method of [McM03] provides a compact representation non-clausal of a given SAT formula in the form of *interpolation proof*. This represents the core of an unsatisfiable SAT formula. Modifying the interpolation algorithm to generate a representation of satisfiable assignments to a SAT formula generates a non-clausal representation. A related question to explore is whether techniques like iterative squaring can be used with SAT to speed up fixpoint detection in reachability. We want to extend this procedure for solving general quantified Boolean formulas (QBFs). Our algorithm was used in [LBC03] for solving QBFs with one existential quantifier that arise in symbolic predicate abstraction. We are also identifying other problem domains where our method can be applied.

Circuit	# latches	# steps	Space Requirement			Time (sec)	Comparison with [MS03]	
			# cubes	# blocking clauses	%age space		Max. Depth	Time (sec)
decss	86	85*	655	131304	0.50	1000.00		
iu30	30	4*	3343	72037	4.64	1000.00		
iu35	35	3*	1479	94424	1.57	1000.00		
iu40	40	2*	20	33168	0.06	1000.00		
iu45	45	1*	2294	165192	1.39	1000.00		
s208.1	8	255	8	255	3.14	0.56		
s298	14	18	33	217	15.21	0.33	18	19.3
s344	15	6	558	2624	21.27	15.30		
s349	15	6	546	2624	20.81	14.82		
s382	21	150	337	8864	3.80	7.71		
s386	6	7	6	12	50.00	0.21	7	0.18
s400	21	150	336	8864	3.79	7.81		
s420.1	16	65535	16	65535	0.02	213.97		
s444	21	150	341	8864	3.85	8.00		
s499	22	21	21	21	100.00	1.74	21	1.07
s510	6	46	10	46	21.74	0.47	46	144.81
s526	21	150	381	8867	4.30	9.35		
s526n	21	150	372	8867	4.20	9.21		
s635	32	125528*	66	125528	0.05	1000.00		
s641	19	6	321	1543	20.80	2.24	6	97.03
s713	19	6	363	1543	23.53	2.53	6	126.94
s820	5	10	11	24	45.83	0.48	10	2.51
s832	5	10	11	24	45.83	0.47		
s838.1	32	155441*	26	155441	0.02	1000.00		
s953	29	10	189	503	37.57	2.01	10	102.23
s967	29	10	177	548	32.30	3.12		
s1196	18	2	802	2615	30.67	6.79	2	232.84
s1238	18	2	849	2615	32.47	7.26		
s1269	37	1*	2136	4339	49.23	1000.00	7*	5000
s1423	74	3*	2652	55568	4.77	1000.00	26*	5000
s1488	6	21	19	47	40.43	0.87	21	96.87
s1494	6	21	19	47	40.43	0.87		
s1512	57	4*	2035	178175	1.14	1000.00		
s9234	228	8*	507	6651	7.62	1000.00		
s13207	669	2*	76	1824	4.17	1000.00		
s15850	597	5*	362	2558	14.15	1000.00		
s38584	1452	2*	58	452	12.83	1000.00		

Table 1: Experimental results on a set of circuits from various sources including ISCAS'89 and Synopsys. The comparison is against [MS03]. Note: (*)-reachability was not complete. Empty boxes denote results N/A.

4 BDD Based Image Computation

Computing the set of states reachable in one step from a given set of states, i.e. *image computation*, is a crucial step in several symbolic verification algorithms, including *model checking* and *reachability analysis*. In the first part of this project, we do an in-depth study of the image computation problem. We analyze and evaluate several new heuristics, metrics, and algorithms for this problem. The algorithms use combinatorial optimization techniques such as hill climbing, simulated annealing, and ordering by recursive partitioning to obtain better results than was previously the case. Theoretical analysis and systematic experimentation are used to evaluate the algorithms. In the second part of the project, I propose non-linear quantification scheduling. Until then, quantification scheduling in image computation, with a conjunctively partitioned transition relation, has been restricted to a linear schedule. This results in a loss of flexibility during image computation. We view image computation as a problem of constructing an optimal parse tree for the image set. The optimality of a parse tree is defined by the largest BDD that is encountered during the computation of the tree. We present dynamic and static versions of a new algorithm, *VarScore*, which exploits the flexibility offered by the parse tree approach to the image computation. We show by extensive experimentation that our techniques outperform the best known techniques so far.

At the core of all symbolic algorithms is *image computation*[1] i.e., the task of computing the set of successors $\mathbf{Img}(S)$ of a set of states S, where

$$\mathbf{Img}(S) := \{s' : \exists s. \exists w. R(s, w, s') \wedge s \in S\}.$$

Image computation is one of the major bottlenecks in verification. Often it is impossible to construct a single BDD for the transition relation R. Instead, R is represented as a *partitioned transition relation*, i.e., as the conjunction of several BDDs, each representing a part of R. The problem is to compute $\mathbf{Img}(S)$ without actually computing R.

The definition of $\mathbf{Img}$ involves evaluation of a *quantified Boolean formula*. In the BDD representation, this amounts to quantifying over several Boolean state variables. *Early quantification*is based on the following Boolean equation:

$$\exists y. f(x, y) \wedge g(x) \equiv (\exists y f(x, y)) \wedge g(x) \tag{5}$$

Early quantification results in smaller intermediate BDDs by reducing the scope of each variable to be quantified. The success of early quantification heavily depends upon the derivation and ordering of the sub-relations which comprise R. This problem has attracted significant attention over the last decade. Since the problem is known to be NP-hard, various heuristics have been proposed for the problem.

4.1 Background and related work

Notation: Every state is represented as a vector $b_1 \ldots b_n \in \{0, 1\}^n$ of Boolean values. The transition relation R is represented by a Boolean function $T(x_1, \ldots, x_n, w_1, \ldots, w_m\ x'_1, \ldots, x'_n)$. Variables $X = x_1, \ldots, x_n$, $X' = x'_1, \ldots, x'_n$ and $W = w_1, \ldots, w_m$ are *current state*, *next state* and *input* variables respectively. $T(x_1, \ldots, x_n, w_1, \ldots, w_m, x'_1, \ldots, x'_n)$ is abbreviated as $T(X, W, X')$. Similarly, functions of the form $S(X) = S(x_1, \ldots, x_n)$ describe sets of states. The set of variables on which f depends on is denoted by $Supp(f)$.

Example 4.1 [3 bit counter. (Running Example)] Consider a 3-bit counter with bits x_1, x_2 and x_3, where x_1 is the least significant and x_3 the most significant bit. The state variables are $X = x_1, x_2, x_3$, $X' = x'_1, x'_2, x'_3$. The transition relation of the counter can be expressed as

$$T(X, X') = (x'_1 \leftrightarrow \neg x_1) \wedge (x'_2 \leftrightarrow x_1 \oplus x_2) \wedge (x'_3 \leftrightarrow (x_1 \wedge x_2) \oplus x_3).$$

Note that the counter does not have any input variables. In later examples, we will compute the image $\mathbf{Img}(S)$ of the set $S(X) = \neg x_1$ which contains those states where the counter is even.

Partitioned BDDs: For most realistic designs it is impossible to build a single BDD for the entire transition relation. Therefore, it is common to represent the transition relation as a conjunction of smaller BDDs $T_1(X, W, X')$, $T_2(X, W, X'), \ldots,$ $T_l(X, W, X')$, i.e.,

$$T(X, W, X') = \bigwedge_{1 \leq i \leq l} T_i(X, W, X'),$$

[1]The techniques presented also apply to preimage computation. However, for ease of exposition, we restrict ourselves to image computation.

where each T_i is represented as a BDD. The sequence $T_1, \ldots, T_l$ is called a *conjunctively partitioned transition relation*. Note that T is *not actually computed*, and only the T_i's are kept in memory. Typically, these partitions are derived from the next state functions of state variables. However, if the BDD of a single next state function is too large, then the circuit for the next state function is further partitioned by introducing *cut-point* variables $C = c_1, \ldots, c_p$. These cut-points are then quantified away in the image computation. Let Q denote the variables to be quantified, which in our case is $Q = X \cup W \cup C$ and $|Q| = n+m+p$. The equation for image computation is then:

$$\begin{aligned} \mathbf{Img}(S(X)) &= \exists Q.(T(X,W,C,X') \wedge S(X)) && (6) \\ &= \exists Q.(\bigwedge_{1 \leq i \leq l} T_i(X,W,C,X') \wedge S(X)) && (7) \end{aligned}$$

Example 4.2 [3 bit counter, ctd.] For the 3 bit counter, a very simple partitioned transition relation is given by the functions $T_1 = (x'_1 \leftrightarrow \neg x_1), T_2 = (x'_2 \leftrightarrow x_1 \oplus x_2)$ and $T_3 = (x'_3 \leftrightarrow (x_1 \wedge x_2) \oplus x_3)$.

Early Quantification: Usually, the size of a BDD reduces by quantifying away a variable in its support. Loosely speaking, BDDs in the partition correspond to semantic entities of the design to be verified and it is expected that not all variables appear in all clusters. Therefore, by virtue of Equation 5, some of the quantifications in Equation 7 may be shifted over several BDDs as follows:

$$\mathbf{Img}(S(X)) = \exists Q_1 \cdot (T_1 \wedge \exists Q_2 \cdot (T_2 \ldots \exists Q_l \cdot (T_l \wedge S(X)))) \tag{8}$$

where Q_i is the set of variables which do not appear in $Supp(T_1) \cup \ldots Supp(T_{i-1})$. If we look at the parse tree of this equation, we see that it is a linear chain of conjunctions and quantifications. Generalizing this for an arbitrary parse tree, a variable can be quantified away at a subtree node as soon as it does not appear in the rest of the tree.

Quantification Scheduling: The size of intermediate BDDs and effectiveness of early quantification depends heavily upon the order in which BDDs are conjoined in Equation 8. For each linear ordering of the conjunctions, there is a unique order of variable quantifications. The problem of ordering the BDDs so as to minimize the size of intermediate BDDs is known as the *quantification scheduling* problem. The order of BDDs is known as the *conjunction schedule*. Traditionally, only linear conjunction schedules have been considered. We generalize this concept to arbitrary parse trees of the image computation equation. The problem of building the parse tree and scheduling the quantifications over them is called the *quantification scheduling problem*.

Related Work: Burch *et al.* [BCL91a] and Touati *et al.* [TSL+90] first recognized the importance of early quantification for image computation. Geist and Beer [GB94] proposed a simple heuristic algorithm, in which they ordered conjuncts in the increasing order of the number of support variables in the conjunct. Hojati *et al.* were the first to formulate the early quantification problem as an evaluation of a parse tree and proved the NP-completeness of the problem. They also offered a greedy strategy for evaluation of the parse tree by evaluating the node with the smallest support set next. However, they did not compare theire technique against other techniques, so the effectiveness of their algorithms was unclear. Traditional techniques for linear quantification schedules begin by first ordering the conjuncts, and then clustering them, and finally ordering the clusters again using the same heuristics. Ranjan *et al.* [RAP+95] proposed the first successful heuristics for this problem and Yang [Yan99] refined their technique. Their ordering procedure linearly orders the BDDs based on a heuristic score. The individual BDDs are then formed into clusters by conjoining them according to the linear order until BDD size grows beyond certain threshold. Finally, these clusters are ordered using the same algorithm. A recent paper by Moon and Somenzi [MS00] presents an ordering algorithm (henceforth referred to as FMCAD00) based on computing the *Bordered Block Triangular* form of the dependence matrix to minimize the *average active lifetime* of variables. Their clustering algorithm is based on the sharing of support variables or affinity between conjuncts. We extended their notion of lifetimes and used combinatorial algorithms to improve the performance [CCJ+01b].

Research has also been carried out in disjunctive decomposition of transition relation. In [MKRS00], the authors use dependency matrix to decide whether to introduce disjunctive decomposition. However, after the decomposition they use standard linear quantification schedules. Gupta *et al.*. [GYAG00] use SAT procedures to derive finer disjunctive decomposition of the transition relation and use a conjunctive schedule for the subproblems corresponding to the clauses. They also propose to use a non-linear quantification scheduling algorithm similar to the one proposed here for leaf image computation. They also use variable scoring mechanism to choose a variable. Then all BDD relations that the variable appears in are conjoined along with quantification of the chosen variable. However, they presented experimental results in the context of using SAT for decomposition of the overall problem, and it is difficult to make a fair comparison against a purely BDD based approach. Apart from this, there are some differences in the details of the algorithm. One difference is in the scoring mechanism. They use the product of the BDD sizes as the heuristic score, while we use the sum or the sum of the squares of the BDD sizes. In the worst case, it is true that the size of the result of the *apply* operation will the the product of the sizes of two BDDs, but this is a rather pessimistic estimate. Better estimation of BDD sizes as a function of the support set will improve the heuristics. We also believe that our algorithms provides more flexibility by conjoining only two smallest BDDs for a variable, unlike algorithm where they conjoin all the BDDs (which is used in [GYAG00]).

4.2 Image Computation by Heuristic Methods and Dependency Matrices

We propose and analyze several new techniques for efficient image computation using the partitioned representation. The main contributions of this work are the following:
• We extend and analyze image computation techniques previously developed by Moon *et al.*These techniques are based on the *dependence matrix* of the partitioned transition relation. We explore various *lifetime metrics* related to this representation and argue their importance in predicting costs of image computation. Moreover, we provide effective heuristic techniques to optimize these metrics.
• We show that the problem of minimizing the lifetime metric of is NP complete. More importantly, the reduction used to prove this result explains the close connection between efficient image computation and the well studied problem of computing the *optimal linear arrangement* for an undirected graph.
• We model the interaction between various sub-relations in the partitioned transition relation as a weighted graph, and introduce a new class of heuristics called *ordering by recursive partitioning*.
• We have performed extensive experiments which indicate the effectiveness of our techniques. By implementing these techniques, we have also contributed to the code base of the symbolic model checker NuSMV.

The main conclusion to be drawn from our analysis is the following: For complicated industrial designs, the effort initially spent on ordering algorithms is clearly amortized during image computation. In other words, the benefits of good orderings outweigh the cost of slow combinatorial optimization algorithms.

4.3 Image Computation by Non-linear Quantification Scheduling

We propose a more flexible approach to image computation by viewing the image computation equation as a *symbolic expression evaluation* problem. The main contributions are as follows:

- We formulate the problem of image computation as an expression evaluation problem where the goal is to reduce the size of the intermediate BDDs. This approach provides significantly more flexibility than the traditional *linear* approach for ordering BDDs during image computation. We show how this approach subsumes the *linear* approaches.
- We provide the *VarScore* heuristics for evaluating the parse tree of image computation equation to reduce the size of the intermediate BDDs. Our heuristics are based on scoring the variables that need to be quantified and restructuring the parse tree according to the heuristic. We provide *dynamic* and *static* versions of our *VarScore* heuristics. In the dynamic version, the parse tree is built for each image computation, while in the static version, a single parse tree is built in the beginning and is used for all subsequent image computations.
- We compare our dynamic and static heuristics to the best known techniques based on linear ordering of BDDs. We show that even with a simple heuristic such as *VarScore*, we achieve impressive results. We have also contributed to the code base of the symbolic model checker NuSMV by implementing our techniques.

We have demonstrated that our simple yet flexible approach yields better experimental results for many reachability analysis and model checking problems.

4.4 Summary of BDD Based Image Computation

We have proposed simple yet effective quantification scheduling algorithms for the image computation problem. We view the problem of quantification scheduling for symbolic image computation as a quantified Boolean formula evaluation problem. We have also proposed heuristic algorithms based on scoring of quantification variables to reduce the size of the intermediate BDDs. We have demonstrated that our simple yet flexible approach yields better experimental results for many reachability analysis and model checking problems.

We can view the problem of building an optimal parse tree as a combinatorial optimization problem and apply techniques like simulated annealing to get better quantification schedules. A middle ground between dynamic and static techniques seems promising. For example, we believe that beginning with some static schedule, the schedule can be tuned for a particular image computation with a little effort. We also want to investigate techniques of approximating sizes of BDDs based on the size of support sets, which will definitely improve all image computation heuristics. We would also like to apply the techniques developed for quantification scheduling to other related problems, like splitting orders in SAT checkers [BCCZ99] and hierarchical model checking [MA01].

I plan to use the ordering algorithms based on dependency matrices for ordering the state variables. The representative function of a state variable will be given the by the corresponding transition function. This does not directly indicate the complexity of the parametric function for the particular state variable, but is often related. I present some experimental evidence to this effect in Section 5.

5 SAT Based Reparameterization Algorithm for Symbolic Simulation

Symbolic simulation is a widely applied technique for analysis of complex transition systems and synchronous circuits in particular. In symbolic simulation, the transition relation is unwound m times into an equation that represents the set of states that is reachable in exactly m steps. The simulator keeps separate equations for each state variable. They are parameterized in the initial state and the inputs of the circuit. Thus, the set of states is stored in a *parametric representation*.

An efficient way to store and manipulate this parametric representation of the set of states is crucial for the performance of the algorithm. Such a representation describes a set of states as a vector $(f_1, f_2, \ldots, f_n)$ of functions in parameters $P = \{p_1, p_2, \ldots, p_m\}$. Each parametric function gives the value of one state variable. For example, the set of states $S = \{10, 01\}$ is represented parametrically as $(p_1, \neg p_1)$. In this case, there is only one parameter p_1.

Most implementations use BDDs to represent these functions [CM90, Jon99, Goe03, YS02]. These BDDs may grow exponentially in the number of simulation steps, as the number of variables grows. In order to address this problem, symbolic simulators compute a new, equivalent parametric representation. The new representation can be significantly smaller since it usually requires fewer variables. This step is done as soon as one of the BDDs becomes too large. The process of converting one parametric representation to another is called *reparameterization*. In [CM90] and [Jon99], the reparameterization algorithm first converts the parametric representation into characteristic function form and then parameterizes this form. In [Goe03], an algorithm is given for computing set union in parametric form. Algorithms for reparameterization and quantification are given that are based on this set union algorithm. However, the reparameterization is done using BDDs, hence as the number of simulation steps grows, the algorithm quickly becomes very expensive. This is due to the fact that each simulation step introduces more input variables, which need to be quantified during reparameterization.

Contribution We describe a SAT-based algorithm to perform the reparameterization step for symbolic simulation. The algorithm performs better than BDD-based reparameterization especially in the presence of many input variables. The algorithm takes arbitrary Boolean equations as input. Therefore, it does not require BDDs for the symbolic simulation. Instead, non-canonical forms that grow linearly with the number of simulation steps can be used.

In essence, the SAT-based reparameterization algorithm computes a new parametric function for each state variable one at a time. In each computation, a large number of input variables are quantified by a single call to a SAT-based enumeration procedure [McM02, CCK03b]. The advantage of this approach is twofold: First, all input variables are quantified at the same time, and second, the performance of SAT-based enumeration procedure is largely unaffected by the number of input variables that are quantified.

We demonstrate the efficiency of this new technique using large industrial circuits with thousands of latches. We compare it to both SAT-based Bounded Model Checking and BDD-based symbolic simulation. Our new algorithm can go much deeper than a standard Bounded Model Checker can. Moreover, the overall memory consumption and the run times are, on average, 3 times less than the values measured using a Bounded Model Checker. The BDD-based symbolic simulator could not even verify most of the circuits that we used.

Other Techniques *Model checking* [CGP00, CE81] techniques suffer from the state explosion problem. In case of BDD-based symbolic model checking this problem manifests itself in the form of unmanageably large BDDs [BCM+92]. This problem is partly addressed by a formal verification technique called *Bounded Model Checking* (BMC) [BCC+99, BCCZ99]. In BMC, the transition relation for a complex circuit and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure such as GRASP [SS96] or Chaff [MMZ+01]. If the formula is satisfiable, a counterexample can be extracted from the output of the SAT procedure. If the formula is not satisfiable, the circuit and its specification can be unwound more to determine if a longer counterexample exists. This process terminates when the length of the potential counterexample exceeds its completeness threshold (i.e., is sufficiently long to ensure that no counterexample exists [KS03]) or when the SAT procedure exceeds its time or memory bounds. BMC has been used successfully to find subtle errors in very large circuits [Sht00, CFF+01].

In BMC, the size of the SAT instance grows linearly with the unwinding depth. However, for very large circuits, even linear growth can be prohibitive: Either the formula already exceeds the memory limits, or the SAT instance is too hard for the SAT solver. No attempt is made to reduce the size of the representation.

BMC is not at all effective for showing that a property is true unless m exceeds the completeness threshold for the design and the property. Since this completeness threshold is, in most cases, prohibitively large, several extensions to BMC have been proposed in order to detect the absence of counterexamples:

1. In the counterexample guided abstraction refinement framework (CEGAR) [CGJ+00, CCS+02], model checking is performed on a safe abstraction of the model. Thus, if the property holds on the abstract model, it also holds on the concrete model. If this is not so, an abstract counterexample is obtained from the model checker. This abstract counterexample is then used to constrain the states in a Bounded Model Checking SAT instance. If the constrained BMC SAT instance is satisfiable, the abstract counterexample can be simulated on the concrete model and a bug is found. If not, the abstraction is refined using various heuristics.

2. In [MA03], this framework is changed as follows: An abstract counterexample is no longer obtained. The only information of interest is the *length* m of the abstract counterexample. This length m is then used as the bound for a normal, unconstrained BMC instance. If the BMC instance is satisfiable, a bug is found. If this is not the case, information from the SAT solver is used to generate the next abstract model.

3. In [McM03], a new framework is introduced: The algorithm initially performs Bounded Model Checking for some m steps in order to refute the property. If this fails, the proof of unsatisfiability extracted from the SAT solver is used to simplify a fixed-point computation. The purpose of the fixed-point computation is to detect the case when the property actually holds. This may fail, and if so, the algorithm is repeated with an increased value of m.

All three approaches therefore solely rely on Bounded Model Checking to refute the property. The extensions are used to detect the case that the property is true. We briefly describe how our algorithm can be used within these frameworks as a replacement for SAT-based BMC.

Notations and Conventions

We will use the following notations and conventions. Sets will be denoted by capital letters, as in S for the set of states, V for the set of state variables, I^m for the set of input variables, and P for the set of parametric variables. We use a superscript of m for input variables to denote input variables accumulated over m steps of symbolic simulation. An ordered tuple of lower case letters denotes a vector of variables. For example, the state variable vector with n state variables is $(v_1, v_2, \ldots, v_n)$. The vector is denoted by using a bar over the symbol. For example, a state vector will be denoted by $\bar{v}$ or in full form by $(v_1, v_2, \ldots, v_n)$. A particular parametric assignment is given by $\bar{p} = (p_1, p_2, \ldots, p_m)$. The set of all possible 2^n vectors of n state variables is $\mathcal{S}_n$, the set of all possible 2^m assignments to m parameters is $\mathcal{P}_m$, and the set of all possible input vectors is $\mathcal{W}^m$. Other uppercase calligraphic letters denote subsets of these sets. When the number of components in a vector is clear, we will often drop the subscripts, and just use $\mathcal{S}, \mathcal{P}$, and so on. Functions will be denoted by lower case symbols, e.g., $f(I^m)$. In the brackets after a function symbol, the list of variables on which the function depends (the support set) is given, e.g., $h_i^1(p_1, p_2, \ldots, p_i)$. The value of a function for a particular assignment to its support variables is given as $h_i^1(p_1, p_2, \ldots, p_i)$ or in short $h_i^1(\bar{p})$. A vector of functions will be denoted by a bar over the top of the function symbol. For example, a vector of parametric functions is $\bar{h}(P) = (\bar{h}_1(P), \bar{h}_2(P), \ldots, \bar{h}_n(P))$. The symbols α and β will denote the constants 0 or 1.

5.1 Parametric Representation

Characteristic functions and parametric representations are two well known methods of representing a set of Boolean vectors. A set of Boolean vectors over the state variables represents a set of states. Consider a set S of vectors over the variables

$V = \{v_1, v_2, \dots, v_n\}$. As described above, $\bar{v} = (v_1, v_2, \dots, v_n)$ denotes a particular vector or a particular assignments to the variables in V. If the characteristic function $\xi(V)$ represents the set S of vectors, then

$$S \quad = \quad \{\bar{v} \in \mathcal{S}_n \mid \xi(\bar{v}) = 1\}. \tag{9}$$

Example The following example will be used throughout this section. Let v_1 and v_2 be two Boolean state variables. Consider the set of states $\{01, 10, 11\}$. This set of states has the characteristic function $\xi(V) = v_1 \vee v_2$.

On the other hand, if S is represented parametrically with a vector of n functions $\bar{f}(P) = (f_1(P), f_2(P), \dots, f_n(P))$ over m parameters $P = \{p_1, p_2, \dots, p_m\}$, then

$$S \quad = \quad \{\bar{v} \in \mathcal{S}_n \mid \exists \bar{p} \in \mathcal{P}_m [v_1 = f_1(\bar{p}) \wedge v_2 = f_2(\bar{p}) \wedge \ldots \wedge v_n = f_n(\bar{p})]\}. \tag{10}$$

Informally, the set of vectors in S is given by the range of the vector of functions $(f_1(\bar{p}), f_2(\bar{p}), \dots, f_n(\bar{p}))$, where $\bar{p}$ ranges over all possible Boolean vectors in $\mathcal{P}_m$. For the running example, one possible parametric representation with three parameters $P = \{p_1, p_2, p_3\}$ is

$$(f_1(P) = p_1 \wedge p_2, f_2(P) = \neg(p_1 \wedge p_2) \vee p_3).$$

Note that, in general, $m \neq n$. For the particular case of symbolic simulation that we will discuss later, the number of parameters will be equal to the number of input variables to the circuit times the number of simulation steps, which can be much larger than n.

A parametric representation can be easily converted to a characteristic function by using the following equation:

$$\xi(V) = \exists \bar{p} [(v_1 \leftrightarrow f_1(\bar{p})) \wedge (v_2 \leftrightarrow f_2(\bar{p})) \wedge \ldots \wedge (v_n \leftrightarrow f_n(\bar{p}))]. \tag{11}$$

In other words, $\xi(\bar{v})$ is true if the there exists an assignment $\bar{p}$ to the parameters such that the parametric function $f_1(\bar{p})$ evaluates to v_1, $f_2(\bar{p})$ evaluates v_2, and so on. This is what is desired, since ξ is supposed to be true exactly for the vectors in the set. In the case of symbolic simulation, $\bar{p}$ consists of the initial state and the inputs on the path to the state $\xi(\bar{v})$.

Note that the conversion to characteristic function involves Boolean quantification over the parameters. If the functions are represented by BDDs, then this quantification becomes harder as the number of parameters m and the number of state variables n increase. A similar quantification problem occurs in BDD-based image computation when a transition relation is represented in conjunctively decomposed form. In that case, the variables to be quantified are the present state and input variables of the circuit, while the next state variables are not quantified. Considerable effort has been devoted to making image computation faster. *Early quantification* [BCL91b, TSL+90, RAP+95, CCJ+01b, CCJ+01a] is a well known technique, in which the quantifiers are pushed inside conjunction as far as possible. The order in which conjunctions are carried out usually influences the effectiveness of early quantification.

Consider a circuit C with p inputs and n state variables. Suppose the circuit is symbolically simulated for m steps, by building Boolean expressions that represent the values of each of the state bits. After the m-step simulation, suppose each state bit v_i is given by a Boolean expression denoted by the function $f_i(I^m)$. The variables I^m appearing in each function $f_i(I^m)$ are the $p \cdot m$ inputs plus the n initial values of the state variables. Thus, $|I^m| = p \cdot m + n$. We will denote the set of input vectors over I^m variables by $\mathcal{W}^m$ and a particular input vector by $\bar{\iota}$. Powerful symbolic simulators can simulate a large number of steps, making $p \cdot k \gg n$. The set of reachable states in m steps, as a set of state vectors in V variables, is given by

$$S \quad = \quad \{\bar{v} \in \mathcal{S}_n \mid \exists \bar{\iota} \in \mathcal{W}^m [v_1 = f_1(\bar{\iota}) \wedge v_2 = f_2(\bar{\iota}) \wedge \ldots \wedge v_n = f_n(\bar{\iota})]\}.$$

Thus symbolic simulation builds a parametric representation of the set of states reachable in exactly m steps, where the parameters are input variables I^m.

Usually, the number of parameters $|I^m|$ is very large. The number of possible valuations of these variables is $2^{|I^m|}$, while the number of possible valuations of the state variables is 2^n. Therefore, many vectors in I^m variables will map to the same state vector. Hence, it should be possible to reduce the number of parameters. We aim at finding new functions $h_1(P), h_2(P), \dots, h_n(P)$ in new parameters P, where $|P| \ll |I^m|$. This is why reparameterization is useful. Obviously, a set of vectors in n variables can be represented by parametric functions of n variables. Hence, $|P| \leq n$. This process of converting from one parametric representation to another is called *reparameterization* [CM90, Goe03].

For the example above, another parametric representation in just two parameters $P = \{p_1, p_2\}$ is $(h_1(P) = p_1, h_2(P) = \neg p_1 \vee p_2)$.

There has been some work on reparameterization using BDDs. The most complete description can be found in [Jon99, Goe03]. The BDD-based method quantifies the input variables one at a time from the parametric representation $\bar{f}(I^m)$. Each quantification involves a parametric union of the two sets, each of which could require a number of BDD operations, linear in the number of state bits. The BDD-based algorithm has $|I^m|$ variable eliminations in the outer loop, and the inner loop iterates over all state bits. Thus, to eliminate all I^m variables, $|I^m| \cdot n$ BDD operations are needed [Goe03, Jon99].

We present a SAT-based reparameterization algorithm. Our SAT-based algorithm does this in one pass over the state bits. The outer loop iterates over the state bits, and the inner computation quantifies all I^m variables in one run of the SAT checker. The details of the algorithm are described in the next section.

5.2 Reparameterization using SAT

Background

The algorithm computes functions $h_1(P), h_2(P), \dots, h_n(P)$ in parameters P, where $|P| \leq n$. Thus, the number of parameters is at most equal to the number of state variables. Moreover, the functions h_i will have a specific structure, in that the function h_i will only depend on the variables $\{p_1, p_2, \dots, p_i\}$. This will be explicitly denoted by $h_i(p_1, \dots, p_i)$. We will derive these functions in the order $h_1, h_2, \dots, h_n$. Intuitively, each new parameter p_i allows for the free choice of the i^{th} state bit v_i. Let $h_i^1(p_1, \dots, p_{i-1})$ denote the Boolean condition under which the state bit v_i is forced to take value 1, and let $h_i^0(p_1, \dots, p_{i-1})$ denote the Boolean condition under which the state bit v_i is forced to take value 0, and $h_i^c(p_1, \dots, p_{i-1})$ denote the Boolean condition under which v_i is free to choose a value (is not forced to either 0 or 1).

For the set $\{01, 10, 11\}$ in the running example, suppose we let the first bit be represented by free parameter p_1. If the first bit is 0, then the second bit is forced to be 1 in the set. Thus, the Boolean condition under which v_2 is forced to 1 is $h_2^1(p_1) = \neg p_1$. Moreover, if the first bit is 1, then the second bit is free to be either 0 or 1. Thus, $h_2^c(p_1) = p_1$. Note that $h_2^0(p_1) = 0$, since the second bit is not forced to 0 in any condition.

The following decomposition of h_i was introduced in [Goe03, GB03]:

$$h_i(p_1, \dots, p_i) \quad = \quad h_i^1(p_1, \dots, p_{i-1}) \vee (p_i \wedge h_i^c(p_1, \dots, p_{i-1}))\,. \tag{12}$$

Intuitively, Equation 12 is interpreted as follows. The value of bit v_i is 1 precisely under the condition h_i^1, hence the first term in the equation. If the parameters p_1 to p_{i-1} do not force the bit v_i to be 1, then the bit is given by the free parameter p_i under the free choice condition h_i^c.

The three conditions h_i^0, h_i^1 and h_i^c are mutually exclusive and complete, thus

$$h_i^c \quad = \quad \neg(h_i^1 \vee h_i^0) = \neg h_i^1 \wedge \neg h_i^0. \tag{13}$$

Continuing our example, we get $h_2(p_1, p_2) = \neg p_1 \vee (p_2 \wedge p_1)$, which is equivalent to the smaller parametric representation $\neg p_1 \vee p_2$ we presented in the previous section. It should be evident that h_i^0, h_i^1, and h_i^c depend only on the parameters p_1 to p_{i-1}. Assigning some specific value to a bit restricts the set of choices for the following bits. In our example, choosing $v_1 = 0$ restricts the value of the bit v_2 to 1. In this special form of a parametric representation, the parametric function h_i is restricted only by the choices made for the earlier bits. Thus, the critical part of computing h_i is computing the three conditions h_i^1, h_i^0 and h_i^c, which we describe now.

Computing h_i^1 and h_i^c

Let us recall the meaning of h_i^1: It denotes the Boolean condition in variables $\{p_1, \dots, p_{i-1}\}$ under which the i^{th} bit v_i is forced to take the value 1. In the given representation $\bar{f}(I^m)$, bit v_i is constrained by other bits in what values it can take. Initially, these constraints are given by the common variables I^m. We want to re-express these constraints in P variables. Let $\bar{p} = (p_1, p_2, \dots, p_{i-1})$ be a specific assignment which makes the Boolean condition $h_i^1(p_1, \dots, p_{i-1})$ true. Then all input vectors $\bar{\iota} \in \mathcal{W}^m$, for which the functions $f_1, \dots, f_{i-1}$ evaluate to the same value as $h_1, \dots, h_{i-1}$, are said to be confirming to the assignment $(p_1, p_2, \dots, p_{i-1})$. In essence, the evaluation of the new parametric functions and the old parametric functions is the same for these input vectors. We introduce *restriction function* $\rho_i(p_1, \dots, p_{i-1}, I^m)$ to find this set of conforming inputs. The function ρ_i restricts the set of input vectors $\mathcal{W}^m$ to only those that conform with the given assignment to the parameters. Formally, it can be written as

$$\rho_i(p_1, \dots, p_{i-1}, I^m) \quad = \quad \bigwedge_{j=1}^{i-1} h_j(p_1, \dots, p_j) = f_j(I^m). \tag{14}$$

Note that $\rho_1 = 1$. Now the condition h_i^1 can be easily expressed as follows: We want a Boolean condition in $\{p_1, \dots, p_{i-1}\}$ variables under which v_i is forced to take the value 1. So if an assignment $(\bar{p}_1, \bar{p}_2, \dots, \bar{p}_{i-1})$ makes h_i^1 true, then that means that for all input vectors $\bar{\imath}$ that conform with this assignment, the function $f_i(\bar{\imath})$ evaluates to 1. Hence,

$$h_i^1(p_1, \dots, p_{i-1}) = \forall I^m . \left(\rho_i(p_1, \dots, p_{i-1}, I^m) \Rightarrow f_i(I^m) = 1\right). \tag{15}$$

Analogously, h_i^0 can be expressed as

$$h_i^0(p_1, \dots, p_{i-1}) = \forall I^m . \left(\rho_i(p_1, \dots, p_{i-1}, I^m) \Rightarrow f_i(I^m) = 0\right). \tag{16}$$

Equation 13 can be used to compute h_i^c, given both h_i^1 and h_i^0. Thus h_i can be easily computed. Note that $h_1 = p_1$, unless the bit v_1 is always 1 or 0, in which case $h_1 = 1$ or $h_1 = 0$. This follows automatically from $\rho_1 = 1$.

Thus, Equations 12 to 16 give us the following high level reparameterization algorithm, that we call ORDEREDREPARAM.

// Input: Parametric Representation $\bar{f}(I^m) = (f_1(I^m), f_2(I^m), \dots, f_n(I^m))$.
// Output: Parametric Representation $\bar{h}(P) = (h_1(P), h_2(P), \dots, h_n(P))$.
ORDEREDREPARAM($\bar{f}(I^m) = (f_1(I^m), f_2(I^m), \dots, f_n(I^m))$)
for $i = 1$ **to** n
  $\rho_i \leftarrow 1$
  for $j = 1$ **to** $i - 1$
    $\rho_i \leftarrow \rho_i \wedge (h_j = f_j)$
  endfor
  $h_i^1 \leftarrow \forall I^m.(\rho_i \Rightarrow f_i = 1)$
  $h_i^0 \leftarrow \forall I^m.(\rho_i \Rightarrow f_i = 0)$
  $h_i^c \leftarrow \neg(h_i^1 \vee h_i^0)$
  $h_i \leftarrow h_i^1 \vee (p_i \wedge h_i^c)$
endfor
return $(h_1(P), h_2(P), \dots, h_n(P))$

Figure 5: High Level Description of the Reparameterization Algorithm

The following theorem states that the algorithm is correct. It states that the set of state vectors $\mathcal{Y}$ given by the new parametric representation is exactly the same as that given by the original set of state vectors $\mathcal{X}$. The proof is given in the technical report [CCK03a].

Theorem 1 *Suppose beginning with the parametric representation $\mathcal{X} = \{\bar{v} \in \mathcal{S} \mid \exists \bar{\imath} \in \mathcal{W}^m . \bar{v} = \bar{f}(\bar{\imath})\}$, we obtain $\mathcal{Y} = \{\bar{v} \in \mathcal{S} \mid \exists \bar{p} \in \mathcal{P} . \bar{v} = \bar{h}(\bar{p})\}$ by following the algorithm* ORDEREDREPARAM. *Then $\mathcal{X} = \mathcal{Y}$.*

Computing h_i^1 and h_i^0 from Equations 15 and 16 involves universally quantifying a large number of I^m variables. This is especially expensive with a BDD-based representation. Moreover, representing parametric functions with BDDs becomes very expensive as the number of simulation steps becomes larger as the number of variables $|I^m|$ increases. BDDs can blow up due to variable ordering problems, and the size of BDDs can become exponential in $|I^m|$. However, if the parametric functions are represented by Boolean expressions, the size of each expression is bounded by the circuit size times the number of simulation steps. Therefore, symbolic simulators that use non-canonical Boolean expressions can go much deeper. Thus, we seek to compute h_i when the functions are given as Boolean expressions.

In the previous section, I reported an efficient procedure to quantify existentially a large number of variables from a Boolean formula. The procedure essentially uses powerful SAT checkers like Chaff to enumerate cubes (partial assignments) given in terms of the variables that are not to be quantified and stores these cubes in an efficient data structure. It is worthwhile to note that the complexity of the procedure is mostly related to the number of variables **not** quantified and not to the number of variables to be quantified. If the formula is not given in CNF, intermediate variables can be used to convert it to CNF. In essence, the variables to be quantified are treated in the same way as the intermediate variables.

We intend to use the same procedure to compute h_i^α (where α is either 1 or 0). However, note that we need to universally quantify I^m variables, while the procedure does existential quantification. So we re-express h_i^α as

$$\begin{aligned} h_i^\alpha(p_1, \dots, p_{i-1}) &= \forall I^m.\rho_i(p_1, \dots, p_{i-1}, I^m) \to f_i(I^m) = \alpha & (17)\\ &= \neg\exists I^m.\neg(\rho_i(p_1, \dots, p_{i-1}, I^m) \to f_i(I^m) = \alpha) & (18)\\ &= \neg\exists I^m.\rho_i(p_1, \dots, p_{i-1}, I^m) \wedge f_i(I^m) \neq \alpha & (19)\end{aligned}$$

Thus, the existential quantification can be carried out by our SAT-based enumeration procedure to compute $\neg h_i^\alpha$. The formula $\rho_i(p_1, \dots, p_{i-1}, I^m) \wedge f_i(I^m) \neq \alpha$ is given to the SAT checker in CNF, which is done by introducing intermediate variables. The large number of I^m variables poses no problem, as they are treated just like intermediate variables by our SAT-based enumeration procedure. The procedure computes $\neg h_i^\alpha$ in disjunctive normal form (DNF) over $\{p_1, \dots, p_{i-1}\}$ variables.

After computing h_i^1 and h_i^0 (thus in CNF), h_i^c is given by $\neg h_i^1 \wedge \neg h_i^0$. This can be converted to CNF, if required for the SAT checker, by again introducing intermediate variables. This allows us to derive h_i using Equation 12. It appears that for computing each h_i, two SAT-based enumerations are required, hence a total of $2n$ SAT-based enumerations. In the next section, we show that there are a number of optimizations. First, we show that a single SAT-based enumeration can be used to compute both $\neg h_i^1$ and $\neg h_i^0$. Moreover, we show that successive SAT runs are similar to earlier runs and how to use this similarity to improve the performance of the SAT checker.

Computing h_i^0 and h_i^1 in a single SAT run

While enumerating cubes in variables $\{p_1, \dots, p_{i-1}\}$ for computing h_i^1 and h_i^0, we note that the SAT formulas are very similar to each other. In fact, the only difference is whether $f_i(I^m)$ equals 0 or 1. In order to merge these two computations, we ask the SAT-based enumeration procedure to enumerate cubes in $\{p_1, \dots, p_{i-1}\}$ variables for the following formula:

$$\rho_i(p_1, \dots, p_{i-1}, I^m) \tag{20}$$

For each solution enumerated (in p_1 to p_{i-1} and I^m), we check the value of $f_i(I^m)$. We do this check by just evaluating $f_i(I^m)$ using the assignment to the I^m variables computed by the SAT checker. Note that we have to do this evaluation a large number of times, hence it should be made as fast as possible. Since this is just a function evaluation, techniques such as compiled simulation can be used to do this much faster than what we do at present. Another option is to use the SAT checker itself to do this evaluation, rather than using a separate function evaluator. This can be done as follows: Instead of asking SAT to enumerate the formula $\rho_i(p_1, \dots, p_{i-1}, I^m)$, we ask it to enumerate on

$$t_i(p_1, \dots, p_{i-1}, I^m) = \rho_i(p_1, \dots, p_{i-1}, I^m) \wedge (f_i(I^m) = \beta). \tag{21}$$

Here, β is a new intermediate variable. This does not place any constraints on the solution space. However, since the SAT checker assigns values to all variables, the value it assigns to β is the evaluation of the function $f_i(I^m)$. It appears that we are unnecessarily adding CNF clauses to the SAT instance. However, as we will see in the next subsection, these additional clauses can be used when doing SAT-based enumeration for computing h_{i+1}^α.

If $f_i(I^m)$ evaluates to 0, then we know that the cube found by the SAT checker cannot belong to h_i^1. This is because we found at least one consistent assignment to I^m variables that leads to the value 0 for $f_i(I^m)$, hence bit i is not forced to 1 for all consistent assignments to I^m. Thus, the cube in $\{p_1, \dots, p_{i-1}\}$ is added to $\neg h_i^1$. Similarly, if $f_i(I^m)$ evaluates to 1, then the cube is added to $\neg h_i^0$. Thus, both $\neg h_i^0$ and $\neg h_i^1$ are computed in a single SAT run, and then h_i^c is computed as given in Equation 13.

Incremental SAT

The optimized SAT formula for computing $h_i^\alpha, \alpha \in \{0, 1\}$ (Equation 20) is very similar to the formula given to the SAT checker for computing h_{i-1}. Since $\rho_i = \bigwedge_{j=1}^{i-1}(h_j = f_j)$, the following recurrence is evident:

$$\rho_i(p_1, \dots, p_{i-1}, I^m) = \rho_{i-1}(p_1, \dots, p_{i-2}, I^m) \wedge (h_{i-1}(p_1, \dots, p_{i-1}) = f_{i-1}(I^m)) \tag{22}$$

Thus, an incremental SAT checker can be used, provided we delete the clauses that were added as blocking clauses and the conflict clauses inferred from them while enumerating h_{i-1}^α. An incremental SAT checker keeps all the conflict clauses learned while enumerating solutions to ρ_{i-1}. This is correct because of the recurrence above.

We have implemented an incremental SAT checker on top of zChaff along with the cube enumeration. This SAT checker allows us to remove the blocking clauses added in the previous SAT run. The advantage of incremental SAT checking is that all the learning done while computing ρ_{i-1} comes for free when checking ρ_i. Only the clauses corresponding to $h_{i-1} = f_{i-1}$ need to be added, and only the blocking clauses need to be deleted.

Suppose the SAT checker is used to evaluate $f_i(I^m)$ when enumerating on t_i (Equation 21), as described in the previous section by adding clauses corresponding to $f_i(I^m) = \beta$. In the next iteration we have to add an equality between h_i and f_i to get t_{i+1}, so we just add $(\beta = h_i(p_1, \ldots, p_i)) \wedge (f_{i+1} = \gamma)$ to the SAT formula t_i to get the SAT formula t_{i+1}. Here, γ is again a new intermediate variable. Thus the clauses corresponding to $f_i(I^m) = \beta$ are used in the next iteration.

5.3 Safety Property Checking

Symbolic simulation with reparameterization works as follows: Beginning with the initial states, the circuit is simulated up to a certain depth, say k, when the functions become too large. At this point, reparameterization is applied, and a smaller parametric representation $\bar{h}^k(P^k) = (h_1^k(P^k), h_2^k(P^k), \ldots, h_n^k(P^k))$ is computed representing the set of states reached in exactly k steps. The superscript here just emphasizes the fact that this parametric representation is for step k. After that point, symbolic simulation continues using $\bar{h}^k(P^k)$ as the set of initial states in parametric form. This is continued until a bug is found or the time limit is exceeded. In this section, we describe the method used for finding violations of safety properties.

Let us assume that $S_0(V)$ is the initial state predicate and $Bad(V)$ is the predicate describing the set of states that violate the safety property of interest. For the initial states, we generate a parametric representation from the predicate $S_0(V)$ using the algorithm in [Jon99]. The initial state predicates are usually small, hence this is not very expensive. The parametric variables for initial state will be part of the I^m variables, as described earlier. If $(h_1(P), h_2(P), \ldots, h_n(P))$ is the parametric representation at some step of the simulation, then the SAT checker is asked to provide an assignment to the parameters such that the state vector satisfies the $Bad(V)$ predicate. Formally, the SAT checker is asked to find a satisfying assignment for

$$v_1 = h_1(P) \wedge v_2 = h_2(P) \wedge \ldots \wedge v_n = h_n(P) \wedge Bad(V) \tag{23}$$

If the SAT checker generates a satisfying assignment, then we know that the property fails, and a counterexample needs to be generated.

5.4 Counterexample Generation

For our symbolic simulator, the counterexample generation is nontrivial, since we do not keep the whole simulation. Periodically, we reparameterize the representation and hence lose the information about input variables up to that point. In order to generate counterexamples, we need to store all intermediate parametric representations and the simulated functions that these representations are derived from. This storage can be done on a disk, offline. We pick up one state that violates the safety property and ask the SAT checker to provide an assignment to the input variables that lead from the most recent parameterized representation to the bug. Since the simulated functions are stored on the disk, they can be directly used in the SAT checker, rather than unrolling the circuit again. Once we get a state at the step when the last reparameterization was done, we choose one state from that step and repeat the whole process again. This is similar to the strategy that standard BDD-based model checkers use. They begin with one bad state, and then keep on intersecting pre-images with the frontier state sets, until they get to an initial state.

5.5 Experimental Results for Symbolic Simulation and BMC Using Reparameterization

I am building a tool **symbot** that will incorporate sat based enumeration algorithm, reparameterization and abstraction-refinement. I implemented the reparameterization algorithm as part of this tool. We report our experimental results on a 1.3 GHz AMD Athlon processor machine with 1 GB of main memory running RedHat Linux 7.1. We set a memory limit of 0.7GB. We report experimental results (Table 2) on large industrial circuits. These circuits are taken from various processor designs. Both the circuits were used in [CCS+02], where SAT-based abstraction-refinement was done for verification of safety properties. All D series circuits have a counterexample, while both properties hold on the IU circuit. IUp1 and IUp2 are the same circuit, but checked with different properties.

We compare our algorithm against a BMC algorithm implemented in the NuSMV model checker ([MMZ+01]) with the zChaff SAT checker and the abstraction-refinement results in [CCS+02]. BMC keeps on unwinding the transition relation, while we periodically reduce the size of representation with reparameterization. Therefore, comparing against BMC is fair.

Our algorithm is not yet complete for safety properties, in that it cannot prove properties true without resorting to abstraction-refinement. However, as we will describe later, we can combine abstraction-refinement with our symbolic simulator to make the property checking complete.

Circuit	# latches	# inputs	bug length	Run time			sym max length	sym max time	# reparame-terizations
				BMC	abs-ref [CCS+02]	sym			
$D2^+$	94	11	15	18	79	32	221	1000	8
$D5^+$	343	7	32	15	38.2	17	127	1000	13
D24	223	47	10	5	8	7	543	1000	9
D6	161	16	20	289	833	145	64	1000	5
D18	498	247	28	6834	9955	1698	56	3000	7
D20	532	30	14	2349	1947	574	89	3000	9
IUp1	4494	361	true	3000*	3350	-	183	3000	45
IUp2	4494	361	true	3000*	712	-	183	3000	45

Table 2: Experimental Results on Large Industrial Benchmarks. Times reported are in seconds. BMC was able to complete just 39 steps and then ran out of memory for IUp1 and IUp2.

In Table 2, the column marked "bug length", denotes the length of the shortest counterexample to the safety property, if the property is false. The "bmc time" column records the amount of time the BMC algorithm required for finding the bug, the "abs-ref time" records the amount of time the abstraction-refinement algorithm took to find the bug or to prove the property, and the column marked "sym time" denotes the amount of time our algorithm takes to simulate up to the bug and find the bug. Since IUp1 and IUp2 did not have any bug, we did not record the time for these two circuits in the "sym bug time" column.

To show that our algorithm can go deeper than BMC, we continue simulating these circuits past the bug and record the maximum length we can reach within the time limit. The "max len." column denotes the maximum length that we simulate the circuit for in the time given in the column " max time". The last column marked "# reparams" records the number of reparameterizations done for simulating up to the maximum length.

We would like to point out that in [CCS+02], a spurious counterexample of length 72 was found, which could not even be simulated with SAT on a machine with 3 GB of memory. However, we could simulate it for 72 steps in 987 seconds on the smaller machine with our algorithm.

It is evident from the results that our algorithm is more powerful than the plain BMC algorithm. We are able to go much deeper and can do it in shorter amount of time. In fact, we were even able to do better than the results obtained with abstraction. It should be noted that multiple refinement steps are required in abstraction-refinement, and in each step, a spurious counterexample is simulated using SAT. Therefore, abstraction-refinement can be slower in many cases.

The BDD-based reachability program of [Goe03] does property checking and can also do fixed points. However, it was able to find bugs for D2 and D5 circuits only. For the rest of the circuits, it either exceeded the time or memory limit.

5.6 Ongoing and Future Research

5.6.1 An Approach to Proving the Safety Properties True

For proving that a safety property is true, the BDD-based symbolic simulators perform a fixed-point detection using efficient set union algorithms. Implementing set union in our framework is feasible as described below, and as the representation after reparameterization is canonical, the fixed point could be detected by comparing the last two parametric representations.

Set Union

Suppose two sets of states S_1 and S_2 are given with the following parametric representations $\bar{h}(P) = (h_1(P), h_2(P), \ldots, h_n(P))$ and $\bar{g}(Q) = (g_1(Q), g_2(Q), \ldots, g_n(Q))$ respectively. Suppose $P \cap Q = \emptyset$. If that's not the case, the parameters can just be renamed to make them disjoint. We define $\uplus$ as the following operator for two parametric representations.

$$\bar{h}(P) \uplus \bar{g}(Q) = (z?h_1(P) : g_1(Q), z?h_2(P) : g_2(Q), \ldots, z?h_n(P) : g_n(Q)).$$

Here, the expression $z?h_i(P) : g_i(Q)$ is just a short form for $(z \wedge h_i(P)) \vee (\neg z \wedge g_i(Q))$ and z is a new parameter. The claim below that $\bar{h}(P) \uplus \bar{g}(Q)$ represents $S_1 \cup S_2$ is easy to prove.

Theorem 2 *If S_1 and S_2 are given by parametric representations $\bar{h}(P)$ and $\bar{g}(Q)$, then $S_1 \cup S_2$ is given by the parametric representation $\bar{h}(P) \uplus \bar{g}(Q)$.*

This set union operation can be generalized to take the union of n different parametric representations by using $\lceil log_2 n \rceil$ new parameters.

The number of parameters after set union of two sets is $|P| + |Q| + 1$. This representation can be reparameterized by our SAT based algorithm to get a parametric form in n parameters. Since our algorithm generates canonical forms, the fixed point could be detected by comparing the last two representations. Thus, fixed point detection would require a reparameterization run after each step of simulation. This would nullify the performance gained by the new algorithm, which benefits from performing the reparameterization only when the equations become too big.

Hence, the fixed point detection algorithm, while a theoretical possibility, should not be used for performance reasons. In the next section, I propose the use of counterexample guided abstraction-refinement to make the reparameterization based symbolic simulation algorithm complete for model checking safety property.

The user of Bounded Model Checking has the same problem: the Bounded Model Checker only guarantees the absence of bugs up to the bound. As described in the introduction, there are several techniques to detect that the property holds. Thus, we propose to use SAT-based symbolic simulation as a replacement for BMC within these frameworks. Thus, the symbolic simulator is used to disprove the property only.

5.6.2 Invariant Constraints

Invariant statements are often used to restrict the state space for verification. Such invariants are often called *verification conditions* [Jon99]. An invariant is a predicate $C(V)$ on state variables and the state exploration is restricted to only those states satisfying $C(V)$. The technique described so far assumes a that the transition relation of the system is a conjunction of transition functions of individual state variables. It does not allow an arbitrary transition relation. The following approach can be used for handling such invariant constraints. We first need to convert the invariant $C(V)$ in a parametric form $(c_1(Q), c_2(Q), \ldots, c_n(Q))$, where Q are parameters. Assuming that the invariants are not too complicated, BDD based algorithms can be used to get this parametric form as described in [CM90] or [Jon99]. Once this is done, for each state variable v_i, if the next state function is $f_i(V)$, then a constraint of the form $f_i(V) = c_i(Q)$ is added to every time step. In the SAT formula for symbolic simulation, each $f_i(V)$ is represented by an intermediate variable, say vf_i. Similary, $c_i(Q)$ is also represented by another intermediate variable vc_i, so the equality constraint $vf_i = vc_i$ is added to the SAT formula. These equality constraints are added for each state variable in each step of the simulation. It should be clear that the parameters Q are new for each time step. Finally, when reperameterization is done, the Q parameters added for invariants along with I^m variables are removed.

For the special case of counterexample guided abstraction refinement, an abstract counterexample, which is just an assignment to a subset of state variables at each time step, needs to be simulated on concrete machine. In which case, the abstract counterexample already gives constraints in parameterized form to be applied to symbolic simulation.

5.6.3 Effect of State Variable Ordering

The static ordering techniques for quantification scheduling in BDD based image computation place state variables whose transition functions are closely related next to each other. I propose to use the same heuristic for ordering the state variables in the reparameterization algorithm. It is not clear that the size of the parametric functions is related to the size of the transition functions. I measured the correlation between the sizes of transition functions and the sizes of parametric functions, before and after reparameterization for different circuits. For some circuits (IU, D24, D6), very high correlation ($> 85\%$)was observed while for some circuits (D2, D5, D18), the correlation was minimal ($< 60\%$). This indicates that static ordering based on transition function sizes can be useful. This was done over multiple timesteps. More work is needed to understand the relationship better. It should be noted that the ordering problem for parametric representation is not as severe as it is in quantification scheduling. However, some improvements can be achieved by better orderings.

6 SAT Based Abstraction Refinement

Abstraction reduces the size of the design by focusing only on relevant portions of the design. A conservative abstraction is one which preserves all behaviors of a concrete system. Conservative abstractions benefit from a *preservation* theorem which states that the correctness of any universal (e.g. ACTL*) formulae on an abstract system automatically implies the correctness of the formula on the concrete system. However, a counterexample on an abstract system may not correspond to any real path, in which case it is called a *spurious* counterexample. To get rid of a spurious counterexample, the abstraction needs to be made more precise via refinement. It is obviously desirable to automate this procedure. In this section, we briefly describe the SAT based abstraction-refinement framework of [CCS+02].

6.1 Introduction

Our abstraction function is based on hiding irrelevant parts of the circuit by making a set of variables *invisible*. This simple abstraction function yields an efficient way to generate minimal abstractions, a source of difficulty in previous approaches. We describe two techniques to produce abstract systems by removing invisible variables. The first is simply to make the invisible variables into input variables. This is shown to be a minimal abstraction. However, this leaves a large number of input variables in the abstract system and, consequently, BDD based model checking even on this abstract system becomes very difficult [WHL+01]. We propose an efficient method to pre-quantify these variables on the fly during image computation. The resulting abstract systems are usually small enough to be handled by standard BDD based model checkers. We use an enhanced version [CCJ+01a, CCJ+01b] of NuSMV [CCGR99] for this. If a counterexample is produced for the abstract system, we try to simulate it on the concrete system symbolically using a fast SAT checker (Chaff [MMZ+01, ZMMM01] in our case).

The refinement is done by identifying a small set of invisible variables to be made visible. We call these variables the *refinement variables*. Identification of refinement variables is the main focus of this work. Our techniques for identifying important variables are based on analysis of effective *boolean constraint propagation (BCP)* and *conflicts* [MMZ+01] during the SAT checking run of the counterexample simulation.

The efficiency of SAT procedures has made it possible to handle circuits with a few thousand of variables, much larger than any BDD based model checker is able to do at present. Our approach is similar to BMC, except that the propositional formula for simulation is constrained by assignments to visible variables. This formula is *unsatisfiable* for a spurious counterexample. We proposed heuristic scores based on backtracking and conflict clause information, similar to VSIDS heuristics in Chaff, and conflict dependency analysis algorithm to extract the reason for unsatisfiability. Our techniques are able to identify those variables that are critical for unsatisfiability of the formula and are, therefore, prime candidates for refinement. The main strength of our approach is that we use the SAT procedure itself for refinement. We do not need to invoke multiple SAT instances or solve separation problems as in [CGKS02].

Thus the main contributions of our work are, (a) use of SAT for counterexample validation, (b) refinement procedures based on SAT conflict analysis, and, (c) a method to remove invisible variables from the abstract system for computational efficiency.

6.2 Abstraction in Model Checking

We give a brief summary of the use of abstraction in model checking and introduce notation that we will use in the remainder of the section (refer to [CGP00] for a full treatment). A transition system is modeled by a tuple $M = (S, I, R, \mathcal{L}, L)$ where S is the set of states, $I \subseteq S$ is the set of initial states, R is the set of transitions, $\mathcal{L}$ is the set of atomic propositions that label each state in S with the labeling function $L : S \rightarrow 2^{L}$. The set I is also used as a predicate $I(s)$, meaning the state s is in I. Similarly, the transition relation R is also used as a predicate $R(s_1, s_2)$, meaning there exists a transition between states s_1 and s_2. Each program variable v_i ranges over its non-empty domain D_{v_i}. The state space of a program with a set of variables $V = \{v_1, v_2, \ldots, v_n\}$ is defined by the Cartesian product $D_{v_1} \times D_{v_2} \times \ldots \times D_{v_n}$.

In existential abstraction [CGP00] a surjection $h : S \rightarrow \hat{S}$ maps a concrete state $s_i \in S$ to an abstract state $\hat{s}_i = h(s_i) \in \hat{S}$. We denote the set of concrete states that map to an abstract state $\hat{s}_i$ by $h^{-1}(\hat{s}_i)$.

Definition 6.1 The ***minimal existential abstraction*** $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{\mathcal{L}}, \hat{L})$ corresponding to a transition system $M = (S, I, R, \mathcal{L}, L)$ and an abstraction function h is defined by:

1. $\hat{S} = \{\hat{s} | \exists s. s \in S \wedge h(s) = \hat{s}\}$.
2. $\hat{I} = \{\hat{s} | \exists s. I(s) \wedge h(s) = \hat{s}\}$.
3. $\hat{R} = \{(\hat{s}_1, \hat{s}_2) | \exists s_1 . \exists s_2 . R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2\}$.
4. $\hat{\mathcal{L}} = \mathcal{L}$.
5. $\hat{L}(\hat{s}) = \bigcup_{h(s)=\hat{s}} L(s)$.

Condition 3 can be stated equivalently as

$$\exists s_1, s_2(R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Leftrightarrow \hat{R}(\hat{s}_1, \hat{s}_2) \tag{24}$$

An atomic formula f *respects* h if for all $s \in S$, $h(s) \models f \Rightarrow s \models f$. Labeling $\hat{L}(\hat{s})$ is *consistent*, if for all $s \in h^{-1}(\hat{s})$ it holds that $s \models \bigwedge_{f \in \hat{L}(\hat{s})} f$. The following theorem from [CGJ+00, Lon93] is stated without proof.

Theorem 6.1 *Let h be an abstraction function and ϕ an ACTL* specification where the atomic sub-formulae respect h. Then the following holds: (i) For all $\hat{s} \in \hat{S}$, $\hat{L}(\hat{s})$ is consistent, and (ii) $\hat{M} \models \phi \Rightarrow M \models \phi$.*

This theorem is the core of all abstraction refinement frameworks. However, the converse may not hold, i.e., even if $\hat{M} \not\models \phi$, the concrete model M may still satisfy ϕ. In this case, the counterexample on $\hat{M}$ is said to be spurious, and we need to refine the abstraction function. Note that the theorem holds even if only the right implication holds in Equation 24. In other words, even if we add more transitions to the minimal transition relation $\hat{R}$, the validity of an ACTL* formula on $\hat{M}$ implies its validity on M.

Definition 6.2 An abstraction function h' is a ***refinement*** for the abstraction function h and the transition system $M = (S, I, R, \mathcal{L}, L)$ if for all $s_1, s_2 \in S, h'(s_1) = h'(s_2)$ implies $h(s_1) = h(s_2)$. Moreover, h' is a ***proper refinement*** of h if there exist $s_1, s_2 \in S$ such that $h(s_1) = h(s_2)$ and $h'(s_1) \neq h'(s_2)$.

In general, ACTL* formulae can have *tree-like* counterexamples [CJLV02]. In this paper, we focus only on safety properties, which have finite path counterexamples. It is possible to generalize our approach to full ACTL* as done in [CJLV02]. The following iterative abstraction refinement procedure for a system M and a safety formula ϕ follows immediately.

1. Generate an initial abstraction function h.
2. Model check $\hat{M}$. If $\hat{M} \models \phi$, return `TRUE`.
3. If $\hat{M} \not\models \phi$, check the generated counterexample $\widehat{T}$ on M. If the counterexample is real, return `FALSE`.
4. Refine h, and goto step 2.

Since each refinement step partitions at least one abstract state, the above procedure is complete for finite state systems for ACTL* formulae that have path counterexamples. Thus the number of iterations is bounded by the number of concrete states. However, as we will show in the next two sections, the number of refinement steps can be at most equal to the number of program variables.

We would like to emphasize that we model check abstract system in step 2 using BDD based symbolic model checking, while steps 3 and 4 are carried out with the help of SAT checkers.

6.3 Generating Abstract State Machine

We consider a special type of abstraction for our methodology, wherein, we hide a set of variables that we call *invisible* variables, denoted by $\mathcal{I}$. The set of variables that we retain in our abstract machine are called *visible* variables, denoted by $\mathcal{V}$. The visible variables are considered to be important for the property and hence are retained in the abstraction, while the invisible variables are considered irrelevant for the property. The initial abstraction and the refinement in steps 1 and 4 respectively correspond to different partitions of V. Typically, we would want $|\mathcal{V}| \ll |\mathcal{I}|$. Formally, the value of a variable $v \in V$ in state $s \in S$ is denoted by $s(v)$. Given a set of variables $U = \{u_1, u_2, \ldots, u_p\}, U \subseteq V$, let s^U denote the portion of s that corresponds to the variables in U, i.e., $s^U = (s(u_1)s(u_2)\ldots s(u_p))$. Let $\mathcal{V} = \{v_1, v_2, \ldots, v_k\}$. This partitioning of variables defines our abstraction function $h : S \rightarrow \hat{S}$. The set of abstract states is $\hat{S} = D_{v_1} \times D_{v_2} \ldots \times D_{v_k}$ and $h(s) = s^{\mathcal{V}}$.

In our approach, the initial abstraction is to take the set of variables mentioned in the property as visible variables. Another option is to make the variables in the cone of influence (COI) of the property visible. However, the COI of a property may be too large and we may end with a large number of visible variables. The idea is to begin with a small set of visible variables and then let the refinement procedure come up with a small set of invisible variables to make visible.

We also assume that the transition relation is described not as a single predicate, but as a conjunction of bit relations R_j of each individual variable v_j. More formally, we consider a sequential circuit with registers $V = \{v_1, v_2, \ldots, v_m\}$ and inputs $I = \{i_1, i_2, \ldots, i_q\}$. Let $s = (v_1, v_2, \ldots, v_m)$, $i = (i_1, i_2, \ldots, i_q)$ and $s' = (v'_1, v'_2, \ldots, v'_m)$. The primed variables denote the next state versions of unprimed variables as usual. Thus the bit relation for v_j becomes $R_j(s, i, v'_j) = (v'_j \leftrightarrow f_{v_j}(s, i))$.

$$R(s, s') = \exists i \bigwedge_{j=1}^{m} R_j(s, i, v'_j) \tag{25}$$

Two methods for hiding invisible variables are described in [CCS+02]. One is called input abstraction, where the logic corresponding to the invisible latches is removed. This might potentiall leave a large number of input variables to quantify, as follows:

$$\hat{R}(\hat{s}, \hat{s}') = \exists s^{\mathcal{I}} \exists i \bigwedge_{v_j \in \mathcal{V}} R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v'_j) \tag{26}$$

On the other hand, if some of the invisible variables and original input variables are prequantified, the abstract model has fewer input variables. Thus, the abstract transition system, which is an approximation, is given by:

$$\tilde{R}(\hat{s}, \hat{s}') = \exists s^{W} \bigwedge_{v_j \in \mathcal{V}} \exists s^{Q} R_j(s^{\mathcal{V}}, s^{\mathcal{I}}, i, v'_j) \tag{27}$$

Here, $Q \subseteq \mathcal{I} \cup I$ denotes the set of variables to be pre-quantified and $W = (\mathcal{I} \cup I) \setminus Q$ denotes the set of variable that are not pre-quantified. Since the BDDs for state sets do not contain input variables in the support, this is a safe step to do. This does not violate the soundness of the approximation, i.e., for each concrete transition in R, there will be a corresponding transition in $\hat{R}$, as stated below.

Theorem 6.2 $\exists s_1, s_2(R(s_1, s_2) \wedge h(s_1) = \hat{s}_1 \wedge h(s_2) = \hat{s}_2) \Rightarrow \tilde{R}(\hat{s}_1, \hat{s}_2)$.

The other direction of this implication does not hold because of the approximations introduced.

6.4 Checking the Validity of an Abstract Counterexamples

Given an abstract model $\hat{M}$ and a safety formula ϕ, we run the usual BDD based symbolic model checking algorithm to determine if $\hat{M} \models \phi$. Suppose that the model checker produces an abstract path counterexample $\bar{s}_m = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_m \rangle$. To check whether this counterexample holds on the concrete model M or not, we symbolically simulate M beginning with the initial state $I(s_0)$ using a fast SAT checker. At each stage of the symbolic simulation, we constrain the values of visible variables only according to the counterexample produced. The equation for symbolic simulation is:

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \wedge (R(s_{m-1}, s_m) \wedge (h(s_m) = \hat{s}_m)) \tag{28}$$

Each $h(s_i)$ is just a projection of the state s_i onto visible variables. If this propositional formula is satisfiable, then we can successfully simulate the counterexample on the concrete machine to conclude that $M \not\models \phi$. The satisfiable assignments to invisible variables along with assignments to visible variables produced by model checking give a valid counterexample on the concrete machine.

If this formula is not satisfiable, the counterexample is *spurious* and the abstraction needs refinement. Assume that the counterexample can be simulated up to the abstract state $\hat{s}_f$, but not up to $\hat{s}_{f+1}$ ([CGJ+00, CGKS02]). Thus Formula 29 is satisfiable while Formula 30 is *not* satisfiable, as shown in Figure 6.

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \wedge (R(s_{f-1}, s_f) \wedge (h(s_f) = \hat{s}_f)) \tag{29}$$

$$(I(s_0) \wedge (h(s_0) = \hat{s}_0)) \wedge (R(s_0, s_1) \wedge (h(s_1) = \hat{s}_1)) \wedge \ldots \wedge (R(s_f, s_{f+1}) \wedge (h(s_{f+1}) = \hat{s}_{f+1})) \tag{30}$$

Using the terminology introduced in [CGJ+00], we call the abstract state $\hat{s}_f$ a *failure state*. The abstract state $\hat{s}_f$ contains many concrete states given by all possible combinations of invisible variables, keeping the same values for visible variables as given by $\hat{s}_f$. The concrete states in $\hat{s}_f$ reachable from the initial states following the spurious counterexample are called the *dead-end* states. The concrete states in $\hat{s}_f$ that have a reachable set in $\hat{s}_{f+1}$ are called *bad* states. Because the dead-end states and the bad states are part of the same abstract state, we get the spurious counterexample. The refinement step then is to

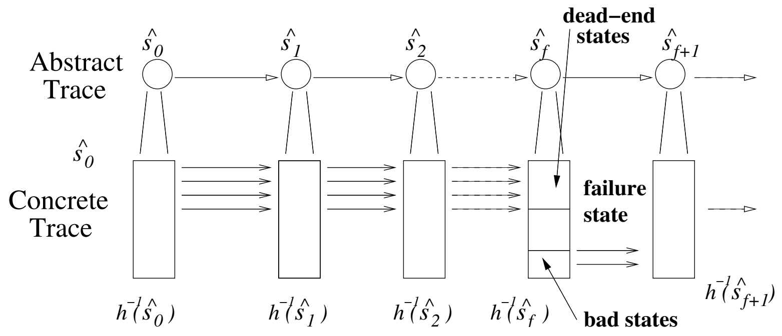

Figure 6: A spurious counterexample showing failure state [CGKS02]. No concrete path can be extended beyond failure state.

separate dead-end states and bad states by making a small subset of invisible variables visible. It is easy to see that the set of dead-end states are given by the values of state variables in the f^{th} step for all satisfying solutions to Equation 29. Note that in symbolic simulation formulas, we have a copy of each state variable for each time frame.

We do this symbolic simulation using the SAT checker Chaff [MMZ+01]. We assume that there are concrete transitions which correspond to each abstract transition from $\hat{s}_i$ to $\hat{s}_{i+1}$, where $0 < i \leq f$. It is fairly straightforward to extend our algorithm to handle spurious abstract transitions. In this case, the set of *bad* states is not empty. Since $\bar{s}_f$ is the shortest prefix that is unsatisfiable, there must be information passed through the invisible registers at time frame f in order for the SAT solver to prove the counterexample is spurious. Specifically, the SAT solver implicitly generates constraints on the invisible registers at time frame f based on either the last abstract transition or the prefix $\bar{s}_f$. Obviously the intersection of these two constraints on those invisible registers is empty. Thus the set of invisible registers that are constrained in time frame f during the SAT process is sufficient to separate *deadend* states and *bad* states after refinement. Therefore, our algorithm limits the refinement candidates to the registers that are constrained in time frame f.

Equation 28 is exactly like symbolic simulation with Bounded Model Checking. The only difference is that the values of visible state variables at each step are constrained to the counterexample values. Since the original input variables to the system are unconstrained, we also constrain their values according to the abstract counterexample. This puts many constraints on the SAT formula. Hence, the SAT checker is able to prune the search space significantly. We rely on the ability of Chaff to identify important variables in this SAT check to separate dead-end and bad states, as described in the next section.

6.5 SAT Based Refinement Heuristics

In this section, we briefly mention the two refinement heuristics that we proposed.

Refinement Based on Scoring Invisible Variables

We score invisible variables based on two factors, first, the number of times a variable gets backtracked to and, second, the number of times a variable appears in a conflict clause. Note that we have adjust the first score by an exponential factor based on the decision level a variable is at, as the variable at the root node can potentially get just two back tracks, while a variable at the decision level dl can get 2^{dl} backtracks globally. Every time the SAT procedure backtracks to an invisible variable at decision level dl, we add the following number to the *backtrack_score*.

$$2^{\frac{|\mathcal{I}|-dl}{c}}$$

We use c as a normalizing constant. For computing the second score, we just keep a global counter *conflict_score* for each variable and increment the counter for each variable appearing in any conflict clause. The method used for identifying conflict clauses from conflict graphs greatly affects SAT performance. As shown in [ZMMM01], we use the most effective method called the *first unique implication point* (1UIP) for identifying conflict clauses. We then use weighted average of these two scores to derive the final score as follows.

$$w_1 \cdot \textit{backtrack\_score} + w_2 \cdot \textit{conflict\_score} \tag{31}$$

Note that the second factor is very similar to the decision heuristic VSIDS used in Chaff. The difference is that Chaff uses these per variable global scores to arrive at local decisions (of the next branching variable), while we use them to derive global information about important variables. Therefore, we do not periodically divide the variable scores as Chaff does.

We also have to be careful to guide Chaff not to decide on the intermediate variables introduced while converting various formulae to CNF form, which is the required input format for SAT checkers. This is done automatically in our method.

Refinement Based on Conflict Dependency Graph

The choice of which invisible registers to make visible is the key to the success of the refinement algorithm. Ideally, we want this set of registers to be small and still be able to prevent the spurious trace. Obviously, the set of registers appearing in the conflict graphs during the checking of the counterexample could prevent the spurious trace. However, this set can be very large. We will show here that it is unnecessary to consider all conflict graphs.

Dependencies Between Conflict Graphs

We call the implication graph associated with a conflict a *conflict graph*. At least one conflict clause is generated from a conflict graph.

Definition 6.3 Given two conflict graphs A and B, if at least one of the conflict clauses generated from A labels one of the edges in B, then we say that conflict B ***directly depends*** on conflict A.

For example, consider the conflicts depicted in the conflict graphs of Figure 7. Suppose that at a certain stage of the SAT checking, conflict graph A is generated. This produces the conflict clause $\omega_9 = (\neg x_9 + x_{11} + \neg x_{15})$. We are using the first UIP (1UIP) learning strategy [ZMMM01] to identify the conflict clause here. This conflict clause can be rewritten as $x_9 \wedge \neg x_{11} \rightarrow \neg x_{15}$. In the other conflict graph B, clause ω_9 labels one of the edges, and forces variable x_{15} to be 0. Hence, we say that conflict graph B directly depends on conflict graph A.

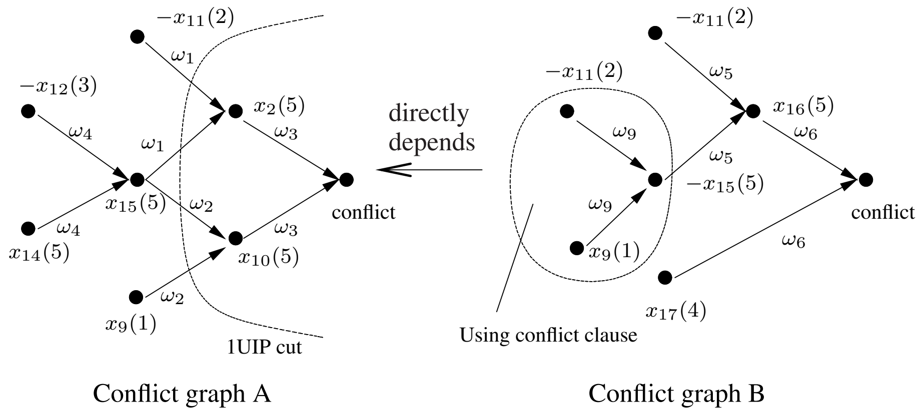

Figure 7: Two dependent conflict graphs. Conflict B depends on conflict A, as the conflict clause ω_9 derived from the conflict graph A produces conflict B.

Given the set of conflict graphs generated during satisfiability checking, we construct the *unpruned conflict dependency graph* as follows:

- **Vertices** of the unpruned dependency graph are all conflict graphs created by the SAT algorithm.
- **Edges** of the unpruned dependency graph are direct dependencies.

Figure 8 shows an unpruned conflict dependency graph with five conflict graphs. A conflict graph B depends on another conflict graph A, if vertex A is reachable from vertex B in the unpruned dependency graph. In Figure 8, conflict graph E depends on conflict graph A. When the SAT algorithm detects unsatisfiability, it terminates with the last conflict graph corresponding to the last conflict. The subgraph of the unpruned conflict dependency graph on which the last conflict graph depends is called the *conflict dependency graph*. Formally,

Definition 6.4 The ***conflict dependency graph*** is a subgraph of the unpruned dependency graph. It includes the last conflict graph and all the conflict graphs on which the last one depends.

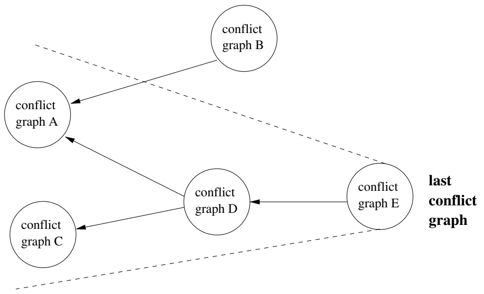

Figure 8: The unpruned dependency graph and the dependency graph (within dotted lines)

In Figure 8, conflict graph E is the last conflict graph, hence the conflict dependency graph includes conflict graphs A, C, D, E. Thus, the conflict dependency graph can be constructed from the unpruned dependency graph by any directed graph traversal algorithm for reachability. Typically, many conflict graphs can be pruned away in this traversal, so that the dependency graph becomes much smaller than the unpruned dependency graph. Intuitively, all SAT decision strategies are based on heuristics. For a given SAT problem, the initial set of decisions/conflicts a SAT solver comes up with may not be related to the final unsatisfiability result. Our dependency analysis helps to remove that irrelevant reasoning.

Generating Conflict Dependency Graph Based on Zchaff

We have implemented the conflict dependency analysis algorithm on top of zchaff [ZMMM01], which has a powerful learning strategy called first UIP (1UIP). Experimental results from [ZMMM01] show that 1UIP is the best known learning strategy. In 1UIP, only one conflict clause is generated from each conflict graph, and it only includes those implications that are closer to the conflict. Refer to [ZMMM01] for the details. We have built our algorithms on top of 1UIP, and we restrict the following discussions to the case that only one conflict clause is generated from a conflict graph. Note here that the algorithms can be easily adapted to other learning strategies.

After SAT terminates with unsatisfiability, our pruning algorithm starts from the last conflict graph. Based on the clauses contained in this conflict graph, the algorithm traverses other conflict graphs that this one depends on. The result of this traversal is the pruned dependency graph.

Identifying Important Variables

The dependency graph records the reasons for unsatisfiability. Therefore, only the variables appearing in the dependency graph are important. Instead of collecting all the variables appearing in any conflict graph, those in the dependency graph are sufficient to disable the spurious counterexample.

Suppose $\bar{s}_{f+1} = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_{f+1} \rangle$ is the shortest prefix of a spurious counterexample that can not be simulated on the concrete machine. Recall that $\hat{s}_f$ is the failure state. During the satisfiability checking of $\bar{s}_{f+1}$, we generate an unpruned conflict dependency graph. When Chaff terminates with unsatisfiability, we collect the clauses from the pruned conflict dependency graph. Some of the literals in these clauses correspond to invisible registers at time frame f. Only those portions of the circuit that correspond to the clauses contained in the pruned conflict dependency graph are necessary for the unsatisfiability. Therefore, the candidates for refinement are the invisible registers that appear at time frame f in the conflict dependency graph.

Refinement Minimization

The set of refinement candidates identified from conflict analysis is usually not minimal, i.e., not all registers in this set are required to invalidate the current spurious abstract counterexample. To remove those that are unnecessary, we have adapted the greedy refinement minimization algorithm in [WHL+01]. The algorithm in [WHL+01] has two phases. The first phase is the addition phase, where a set of invisible registers that it suffices to disable the spurious abstract counterexample is identified. In the second phase, a minimal subset of registers that is necessary to disable the counterexample is identifed. Their algorithm tries to see whether removing a newly added register from the abstract model still disables the abstract counterexample. If that is the case, this register is unnecessary and is no longer considered for refinement. In our case, we only need the second phase of the algorithm. The set of refinement candidates provided by our conflict dependency analysis algorithm already suffices to disable the current spurious abstract counterexample. Since the first phase of their algorithm takes at least as long as the second phase, this should speed up our minimization algorithm considerably.

6.6 Summary of Experimental Results for SAT Based Abstraction-Refinement

Here, we mention some results that we obtained. For full details, the [CCS+02] paper should be consulted. We have implemented our abstraction refinement framework on top of NuSMV model checker [CCGR99]. We modified the SAT checker Chaff to compute heuristic scores, to produce conflict dependency graphs and to do incremental SAT. The IU-p1 benchmark was verified by conflict analysis based refinement on a SunFire 280R machine with two 750Mhz UltraSparc III CPUs and 8GB of RAM running Solaris. All other experiments were performed on a dual 1.5GHz Athlon machine with 3GB of RAM running Linux.

The set of benchmarks are the same as those used in the reparameterization experiments. It is interesting to note that all benchmarks but IU-p1 and IU-p2 have a valid counterexample.

circuit	# regs	ctrex length	CSMV time	Heuristic Score			Dependency		
				time	iters	# regs	time	iters	# regs
D2	105	15	152	105	10	51	79	11	39
D5	350	32	1,192	29	3	16	38.2	8	10
D6	177	20	45,596	784	24	121	833	48	90
D18	745	28	>4 hrs	12,086	69	346	9,995	142	253
D20	562	14	>7 hrs	1,493	56	281	1,947	74	265
D24	270	10	7,850	14	1	6	8	1	4
IU-p1	4855	true	-	9,138	22	107	3,350*	13	19
IU-p2	4855	true	-	2,820	7	36	712	6	13

Table 3: Comparison between Candence SMV (CSMV), heuristic score based refinement and dependency analysis based refinement for larger circuits. The experiment marked with a * was performed on the SunFire machine with more memory because of a length 72 abstract counterexample encountered.

In Table 3, we compare our methods against the BDD based model checker Cadence SMV (CSMV). We enabled cone of influence reduction and dynamic variable reordering in Cadence SMV. The performance of "vanilla" NuSMV was worse than Cadence SMV, hence we do not report those numbers. We report total running time, number of iterations and the number of registers in the final abstraction. The columns labeled with "Heuristic Score" report the results with our heuristic variable scoring method. We introduce 5 latches at a time in this method. The columns labeled with "Dependency" report the results of our dependency analysis based refinement. This method employs pruning of candidate refinement sets. A "-" in a cell indicates that the model checker ran out of memory.

We can see that our conflict dependency analysis based method outperforms a standard BDD based model checker, the method reported in [CGKS02] and the heuristic score based method. We also conclude that the computational overhead of our dependency analysis based method is well justified by the smaller abstractions that it produces. The variable scoring based method does not enjoy the benefits of reduced candidate refinement sets obtained through dependency analysis. Therefore, it results in a coarser abstraction in general. The heuristic based refinement method adds 5 registers at a time, resulting in some uniformity in the final number of registers. Due to the smaller number of refinement steps it performs, the total time it has to spend in model checking abstract machines may be smaller (as for D5, D6, D20, IU60, IU65, IU70).

6.7 Ongoing and Future Work

Integrating Reparameterization Algorithm in SAT Based Abtraction-Refinement

In abstraction-refinement framework, it is relatively straightforward to use reparameterization alogrithm based symbolic simulation. In [MA03], no counterexample needs to be simulated on concrete machine. In this case, the reparameterization algorithm can be used in BMC as described in previous Section 5. On the other hand, in CEGAR, a counterexample $\bar{s}_m = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_m \rangle$ just assigns Boolean values to a subset of state variables at each step. Suppose along the length of the counterexample, reparameterization is invoked a total of l times at steps $m_1, m_2, \ldots, m_l$, such that $0 < m_1 < m_2 < \ldots \leq m$. Let $(f_1^{m_i}(I^{m_i}), \ldots, f_n^{m_i}(I^{m_i}))$ be the old parametric representation. Let $(h_1^{m_i}(P^{m_i}), \ldots, h_n^{m_i}(P^{m_i}))$ be the new parametric representation at step m_i. Let S^{m_i} be the set of states represented by it. To determine if the counterexample is spurious, we simulate the abstract counterexample by adding $\hat{s}_0$ constraints to the initial state. Then we proceed to add $\hat{s}_1, \hat{s}_2, \ldots$ constraints to the symbolic simulation as described in Section 5.6.2. These constraints are just assignments of values, and hence are easy to add in the symbolic simulation. When we reach the step m_1, the process is repeated considering $\hat{s}_{m_1}$ as constraints on the reparameterized state. Also, at each of the steps m_i, we check to see if the set of states S^{m_i} is empty or not. This can be done by checking if the SAT formula

$$\hat{s}^1_{m_i} = v_1 = f_1^{m_i}(I^{m_i}) \wedge \hat{s}^2_{m_i} = v_2 = f_2^{m_i}(I^{m_i}) \ldots \hat{s}^n_{m_i} = v_n = f_n^{m_i}(I^{m_i}) \tag{32}$$

has any satisfiable assignments or not. Here, $\hat{s}^j_{m_i}$ denotes the assignment to the j^{th} state variable by the abstract counterexample in step number m_i. If Equation 32 is satisfiable, then we know that the counterexample is real and we proceed to build the real counterexample as described in the previous section. If not, we need to identify refinement information from the failed SAT instance.

For identifying refinement information, we can use the heuristics of [CCS+02]. However, there is one important difference. Equation 32 does not contain state variables for all information steps. The state variables that appear in the formula are from step m_{i-1} to step m_i only. However, as reported in [CCS+02], just looking at a part of the failed counterexample (often just the failure state) provides useful refinement information. Therefore, I hope that the refinement information extracted from the partial SAT instance will be useful. I want to evaluate the quality of refinement information that we get from such SAT instances by extensive experimentation.

Checking Liveness via Safety Properties

I want to very briefly review the process used to convert simple liveness properties to safety properties. The full description is given in [SB03]. Let us consider liveness properties of the form AFp. The counterexamples to AFp are of the form $EG \not p$, i.e., an infinite path such that all states on the path satisfy $\not p$. For a finite state system, such an infinite path must look like a lasso, i.e., a possibly empty sequence of states $s_0, s_1, \ldots, s_l$ where no state is repeated concatenated with a loop $s_l, s_{l+1}, \ldots, s_m = s_l$. The authors describe methods to translate $\mathbf{AF}\, p$ to a safety property. Each method adds some state variables to the original system model. In each method, a Boolean variable `found` becomes true as soon as p holds on any state. Another Boolean variable `looped` is defined that indicates that a state has already been seen before in the path, thus loop has bee closed. The truth of $\mathbf{AF}\, p$ property is thus given by the truth of the $\mathbf{AG}(\texttt{looped}- > \texttt{found})$ property. The two methods differ in the way the `looped` signal is generated. In the first method, a counter that counts upto the completeness threshold (CT) of $\mathbf{AF}\, p$ property is introduced. Once the counter has gone beyond the CT, then `looped` is set to true. The counter however can be very large. In the second method, a copy of all state variables is made and these new state variables non-deterministically copy the value of the real state variable at some step. Now once we a state equivalent to the saved state, we know that we have looped. The authors describe experiments with both methods. They also extend the method to handle general LTL properties.

It is easy to add this feature to SAT based abstraction-refinement framework. Since the translation is done in the system model, it can be easily incorporated in our framework.

7 Research Timeline

I propose the following timeline for the thesis research.

December 2003-Febraury 2004 I plan to work on improving the SAT enumeration algorithm. I want to explore the non-clausal representation for SAT enumeration and work on using different heuristics for improving the cube enlarement procedure in the enumeration algorithm. The goal is to improve the performance of the algorithm for many reachability benchmarks and predicate abstraction benchmarks. I also want to experiment with iterative refinement in SAT based reachability analysis.

Apart from improvements to the enumeration algorithm, I will also start working on incorporating reparameterization algorithm in abstraction-refinement framework. Based on the initial results, various approaches to refinement may be needed. Incorporating the translation of liveness properties to safety properties in the framework is also a goal of this period.

Febraury 2004-June 2004 During this period, I will continue improving the reparameterization algorithm and the abstraction-refinement framework. There is a choice of different enumeration algorithms [McM02, CCK03b, KP03, SH03] and I want to evaluate by extensive experimentation the effectiveness of some algorithms for our purpose. I want to work on study the effect of state variable ordering on parameterization algorithm and algorithms to find good ordering. I also plan to work on incorporating invariantconstraints in reparameterization algorithm.

June 2004-August 2004 This is the final period. I will clean up the software, perform experiments on various benchmarks and mostly work on writing thesis. The thesis defense will happen shortly after. Some portion of the time will also go in job search.

8 Summary of the Proposal

This thesis proposal addresses the state explosion problem in model checking large industrial circuits with thousands of latches. I use powerful SAT checkers to solve various problems that arise while formally verifying large systems. First, at the core, I proposed a SAT based image computation algorithm and provide a SAT based enumeration procedure. Building on this enumeration algorithm, I introduced a novel SAT based reparameterization algorithm that increases the capacity of symbolic simulation by large extent. These improvements help in bounded model checking of large systems. Finally, the reparameterization algorithm is incorporated in a SAT based abstraction-refinement framework, thus providing completeness for safety and liveness properties and increasing the applicability of the abstraction-refinement procedure. The problem of liveness property checking is reduced to safety property checking by the translation in [SB03]. I believe the improvements at various levels in symbolic model checking will enable a large capacity increase for verification of industrial circuits.

References

[BBS91] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaulation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1991.

[BCC+99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the Design Automation Conference (DAC'99)*, pages 317–320, 1999.

[BCCZ99] Armin Biere, Alexandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, number 1579 in LNCS, 1999.

[BCL91a] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in Symbolic Model Checking. In *28th ACM/IEEE Design Automation Conference*, 1991.

[BCL91b] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with partitioned transition relations. In A. Halaas and P. B. Denyer, editors, *Proceedings of the International Conference on Very Large Scale Int egration*, Edinburgh, Scotland, August 1991.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CCGR99] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV 1999)*, number 1633 in Lecture Notes in Computer Science, pages 495–499. Springer, July 1999.

[CCJ+01a] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Tom Shiple, Helmut Veith, and Dong Wang. Non-linear quantification scheduling in image computation. In *Proceedings of ICCAD'01*, pages 293–298, November 2001.

[CCJ+01b] Pankaj Chauhan, Edmund M. Clarke, Somesh Jha, Jim Kukula, Helmut Veith, and Dong Wang. Using combinatorial optimization methods for quantification scheduling. In Tiziana Margaria and Tom Melham, editors, *Proceedings of CHARME'01*, volume 2144 of *LNCS*, pages 293–309, September 2001.

[CCK03a] Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. A SAT based algorithm for reparameterization in symbolic simulation. Technical Report CMU-CS-03-191, Carnegie Mellon University, School of Computer Science, 2003.

[CCK03b] Pankaj Chauhan, Edmund M. Clarke, and Daniel Kroening. Using SAT based image computation for reachability analysis. Technical Report CMU-CS-03-151, Carnegie Mellon University, School of Computer Science, 2003.

[CCS+02] Pankaj Chauhan, Edmund M. Clarke, Samir Sapra, , James Kukula, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of FMCAD'02*, volume 2517 of *LNCS*, pages 33–50, November 2002.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, Yorktown Heights, NY, May 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFF+01] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, number 2102 in LNCS, pages 436–453. Springer Verlag, 2001.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169, July 2000.

[CGKS02] Edmund Clarke, Anubhav Gupta, James Kukula, and Ofer Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Conference on Computer Aided Verification (CAV 2002)*, 2002.

[CGP00] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CJLV02] Edmund Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proceedings of the 17^{th} Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, 2002.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Proc. Intl. Conf. on Computer-Aided Design (ICCAD)*, pages 78–82. IEEE Computer Society Press, November 1990.

[GB94] D. Geist and I. Beer. Efficient Model Checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Sixth Conference on Computer Aided Verification (CAV 1994)*, volume 818 of *LNCS*, pages 299–310, Stanford, CA, USA, 1994. Springer-Verlag.

[GB03] Amit Goel and Randal E. Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *Proceedings of Design Automation and Test in Europe (DATE'03)*, pages 10816–10821, 2003.

[Goe03] Amit Goel. *A Unified Framework for Symbolic Simulation and Model Checking*. PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, 2003.

[GYA+01] Aarti Gupta, Zijian Yang, Pranav Ashar, Lintao Zhang, and Sharad Malik. Partition-based decision heuristics for image computation using SAT and BDDs. In *Proc. of the Intl. Conf. on Computer-Aided Design (ICCAD)*, 2001.

[GYAG00] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of FMCAD'00*, volume 1954 of *LNCS*, pages 354–371, November 2000.

[Jon99] Robert B. Jones. *Applications of symbolic simulation to the formal verification of microprocessors*. PhD thesis, Dept. of Electrical Engineering, Stanford University, August 1999.

[KP03] Hyeong-Ju Kang and In-Cheol Park. SAT-based unbounded symbolic model checking. In *Proceedings of Design Automation Conference (DAC'03)*, pages 840–843, 2003.

[KS03] Daniel Kroening and Ofer Strichman. Efficient computation of recurrence diameters. In L. Zuck, P. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *4th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 298–309. Springer Verlag, January 2003.

[LBC03] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Proceedings of the International Conference on Computer-Aided Verification (CAV 2003)*, 2003.

[Lon93] David E. Long. *Model checking, abstraction and compositional verification*. PhD thesis, Carnegie Mellon University, 1993. CMU-CS-93-178.

[MA01] M. Oliver Möler and Rajeev Alur. Heuristics for hierarchical partitioning with application to model checking. In *Proceedings of CHARME*, 2001.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[McM02] Ken McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In F. Somenzi and W. Hunt, editors, *Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, July 2003.

[MKRS00] In-Ho Moon, James H. Kukula, Kavita Ravi, and Fabio Somenzi. To split or to conjoin: The question in image computation. In *Proceedings of the 37th Design Automation Conference (DAC'00)*, pages 26–28, Los Angeles, June 2000.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[MS00] In-Ho Moon and Fabio Somenzi. Border-block triangular form and conjunction schedule in image computation. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Proceedings of the Formal Methods in Computer Aided Design (FMCAD'00)*, volume 1954 of *LNCS*, pages 73–90, November 2000.

[MS03] Maher Mneimneh and Karem Sakallah. SAT-based sequential depth computation. In *Proceedings of ASP-DAC*, January 2003.

[RAP+95] R.K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R.K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM International Workshop on Logic Synthesis*, Lake Tahoe, 1995. IEEE/ACM.

[Rud] Richard Rudell. Espresso. Online. Available: http://www-cad.eecs.berkeley.edu/Software/software.html.

[SB95] C.-J. H. Seger and R. E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[SB03] Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. In *Proceedings of Software Tools for Technology Transfer*, 2003. Submitted for publication.

[SH03] Shuo Sheng and Michael Hsiao. Efficient preimage computation using a novel success-driven atpg. In *Proceedings of Design Automation and Test in Europe (DATE'03)*, 2003.

[Sht00] O. Shtrichman. Tuning SAT checkers for bounded model checking. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, LNCS. Springer Verlag, 2000.

[SS96] J. P. Marques Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. Technical Report CSE-TR-292-96, Computer Science and Engineering Division, Department of EECS, Univ. of Michigan, April 1996.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDDs. In *Proceedings of the IEEE international Conference on Computer Aided Design (ICCAD)*, pages 130–133, November 1990.

[WHL+01] Dong Wang, Pei-Hsin Ho, Jiang Long, James Kukula, Yunshan Zhu, Tony Ma, and Robert Damiano. Formal property verification by abstraction refinement with formal, simulation and hybrid engines. In *Proceedings of the DAC*, pages 35–40, 2001.

[Yan99] Bwolen Yang. *Optimizing Model Checking Based on BDD Characterization*. PhD thesis, Carnegie Mellon University, Computer Science Department, May 1999.

[YS00] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation. Technical report, Strategic Cad Labs, Intel Corporation, November 2000.

[YS02] Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation – abstraction in action. In *Proceedings of FMCAD'02*, volume 2517 of *LNCS*, pages 70–86, November 2002.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Proceedings of the Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.

[ZMMM01] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD'01*, November 2001.