A SAT-Based Algorithm for Reparameterization in Symbolic Simulation

Pankaj Chauhan Daniel Kroening Edmund Clarke

December 4, 2003
CMU-CS-03-191

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485 . The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

Keywords: Symbolic Simulation, SAT checkers, Bounded Model Checking, Parametric Representation, Safety Property Checking

Abstract

Parametric representations used for symbolic simulation of circuits usually use BDDs. After a few steps of symbolic simulation, state set representation is converted from one parametric representation to another smaller representation, in a process called reparameterization. For large circuits, the reparametrization step often results in a blowup of BDDs and is expensive due to a large number of quantifications of input variables involved. Efficient SAT solvers have been applied successfully for many verification problems. This paper presents a novel SAT-based reparameterization algorithm that is largely immune to the large number of input variables that need to be quantified. We show experimental results on large industrial circuits and compare our new algorithm to both SAT-based Bounded Model Checking and BDD-based symbolic simulation. We were able to achieve on average 3x improvement in time and space over BMC and able to complete many examples that BDD-based approach could not even finish.

1 Introduction

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 [Bry86] to represent these functions [CM90, Jon99, AJS99, Goe03, GB03, 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, CCK03]. 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.

Outline In section 2, we briefly introduce the basics of Bounded Model Checking and modern SAT algorithms. We also describe how the basic SAT algorithm can be extended in order to obtain the set of satisfying assignments instead of just one satisfying assignment. In section 3, we provide background information about the parametric representation as present in the literature. In section 4, we explain our contribution, i.e., how to implement reparameterization using SAT. In section 6, we present experimental results to demonstrate the effectiveness of the idea. In section 7, we show how it can be used in a number of verification frameworks as a replacement for Bounded Model Checking. We present a proof of correctness of our algorithm in section 8.

Notations and Conventions

We will use the following notations and conventions throughout the paper. 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.

2 SAT

2.1 Introduction

The enabling technique for Bounded Model Checking is satisfiability solving (SAT). 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 GRASP [SS96] and Chaff [MMZ+01], is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics.

The efficiency of SAT procedures has made it possible to handle circuits with thousands of state 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
      return SAT;                     // have been assigned
}
```

Figure 1: Basic DPLL backtracking search (used from [MMZ+01] for illustration purposes)

The basic framework for these SAT procedures, shown in Figure 1, 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 assign-

ments to variables does not lead to any satisfying solution. This is a *conflict*, since at least one CNF clause remains unsatisfied. In case of a conflict, new clauses are learned by `analyze_conflict()` that prevent the same unsuccessful search in the future. The conflict analysis also returns a variable for which the other 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 a satisfying assignment has been found and the procedure returns. The strength of various SAT checkers lies in their implementation of constraint propagation, non-chronological backtracking, decision heuristics, and learning. In our algorithm, we use the Chaff SAT checker [MMZ+01], as it has been demonstrated to be one of the most powerful SAT checker on a wide class of problems.

2.2 Obtaining the Set of Satisfying Assignments

Recently, SAT solvers have been used to obtain the set of satisfying assignments or a projection thereof to a subset of the variables for a Boolean formula. This is also used for the proposed reparametrization algorithm.

Obtaining a projection of the set of satisfiable assignments to a subset of the variables corresponds to a Boolean quantification. The algorithm in [McM02] computes a CNF equivalent formula to a Boolean formula $f(X, Y)$ and then universally quantifies the X variables. Mathematically, it computes $\forall X.f(X, Y)$ in clausal form (CNF). Similarly, in our previous paper, we compute $\exists X.f(X, Y)$ in DNF by enumerating various assignments to the X variables. These enumeration procedures have many applications. For example, they are used in image computation [McM02, CCK03, KP03] in Model Checking, or for deciding satisfiability of QBF [PBZ03]. These applications obtain the desired set by enumeration. The SAT solver starts by searching for a satisfying assignment. If a satisfying assignment is found (this corresponds to the `return SAT` case in Figure 1), the partial assignment is recorded and then added as a *blocking clause*. Instead of terminating, the algorithm backtracks and continues to search for the next satisfying assignment until no more assignments are found.

In order to speed up enumeration of satisfying assignments, the algorithms enlarge the partial assignment before adding the blocking clause. There are many ways to do this: In [McM02], the conflict graph is analyzed in order to enlarge the assignment. In [CCK03], symbolic simulation techniques are used. Moreover, storage of the enumerated clauses or cubes is crucial to the efficiency of the algorithm. In [McM02], zBDDs are used for storing enumerated CNF clauses, while we use a hashing scheme in [CCK03]. In [LBC03], the implementation from [CCK03] was compared against BDD-based quantification. The SAT-based technique outperformed the BDD-based technique on most examples.

3 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 $V = \{v_1, v_2, \ldots, v_n\}$. As described above, $\bar{v} = (v_1, v_2, \ldots, v_n)$ 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

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

Example The following example will be used throughout the paper. 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 $\bar{f}(P) = (f_1(P), f_2(P), \ldots, f_n(P))$ over m parameters $P = \{p_1, p_2, \ldots, p_m\}$, then

$$S \quad = \quad \{\bar{v} \in \mathcal{S}_n \mid \exists \bar{p} \in \mathcal{P}_m[v_1 = f_1(\bar{p}) \wedge v_2 = f_2(\bar{p}) \wedge \ldots \wedge v_n = f_n(\bar{p})]\}. \tag{2}$$

Informally, the set of vectors in S is given by the range of the vector of functions $(f_1(\bar{p}), f_2(\bar{p}), \ldots, f_n(\bar{p}))$, 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:

$$\xi(V) = \exists \bar{p}[(v_1 \leftrightarrow f_1(\bar{p})) \wedge (v_2 \leftrightarrow f_2(\bar{p})) \wedge \ldots \wedge (v_n \leftrightarrow f_n(\bar{p}))]. \tag{3}$$

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* [BCL91, 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, $|I^m| = m = p \cdot m + n$. 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 $p \cdot m \gg n$. 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 $h_1(P), h_2(P), \ldots, h_n(P)$ 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.

4 Reparameterization using SAT

4.1 Background

The algorithm computes functions $h_1(P), h_2(P), \ldots, h_n(P)$ 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 $\{p_1, p_2, \ldots, p_i\}$. This will be explicitly denoted by $h_i(p_1, \ldots, p_i)$. We will derive these functions in the order $h_1, h_2, \ldots, h_n$. Intuitively, each new parameter p_i allows for the free choice of the i^{th} state bit v_i. Let $h_i^1(p_1, \ldots, p_{i-1})$ denote the Boolean condition under which the state bit v_i is forced to take value 1, and let $h_i^0(p_1, \ldots, p_{i-1})$ denote the Boolean condition under which the state bit v_i is forced to take value 0, and $h_i^c(p_1, \ldots, p_{i-1})$ 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]:

$$h_i(p_1, \ldots, p_i) \quad = \quad h_i^1(p_1, \ldots, p_{i-1}) \vee (p_i \wedge h_i^c(p_1, \ldots, p_{i-1}))\,. \tag{4}$$

Intuitively, Equation 4 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

$$h_i^c \quad = \quad \neg(h_i^1 \vee h_i^0) = \neg h_i^1 \wedge \neg h_i^0. \tag{5}$$

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.

4.2 Computing h_i^1 and h_i^c

Let us recall the meaning of h_i^1: It denotes the Boolean condition in variables $\{p_1, \ldots, p_{i-1}\}$ 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 $\bar{p} = (p_1, p_2, \ldots, p_{i-1})$ be a specific assignment which makes the Boolean condition $h_i^1(p_1, \ldots, p_{i-1})$ true. Then all input vectors $\bar{\iota} \in \mathcal{W}^m$, for which the functions $f_1, \ldots, f_{i-1}$ evaluate to the same value as $h_1, \ldots, h_{i-1}$, are said to be confirming to the assignment $(p_1, p_2, \ldots, p_{i-1})$. In essence, the evaluation of the new parametric functions and the old parametric functions is the same for these input vectors. The *restriction function* $\rho_i(p_1, \ldots, p_{i-1}, I^m)$ is used to find this set of confirming 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

$$\rho_i(p_1, \ldots, p_{i-1}, I^m) \quad = \quad \bigwedge_{j=1}^{i-1} h_j(p_1, \ldots, p_j) = f_j(I^m). \tag{6}$$

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

$$h_i^1(p_1, \ldots, p_{i-1}) = \forall I^m. \left(\rho_i(p_1, \ldots, p_{i-1}, I^m) \Rightarrow f_i(I^m) = 1\right). \tag{7}$$

Analogously, h_i^0 can be expressed as

$$h_i^0(p_1, \ldots, p_{i-1}) = \forall I^m. \left(\rho_i(p_1, \ldots, p_{i-1}, I^m) \Rightarrow f_i(I^m) = 0\right). \tag{8}$$

Equation 5 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 4 to 8 give us the following high level reparameterization algorithm, that we call OrderedReparam.

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 $\mathcal{X}$.

Theorem 1 *Suppose beginning with the parametric representation* $\mathcal{X} = \{\bar{v} \in \mathcal{S} \mid \exists \bar{\iota} \in \mathcal{W}^m . \bar{v} = \bar{f}(\bar{\iota})\}$, *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* $\mathcal{X} = \mathcal{Y}$.

We prove this theorem in section 8.

Computing h_i^1 and h_i^0 from equations 7 and 8 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

```
// Input: Parametric Representation f̄(I^m) = (f_1(I^m), f_2(I^m), ..., f_n(I^m)).
// Output: Parametric Representation h̄(P) = (h_1(P), h_2(P), ..., h_n(P)).
OrderedReparam(f̄(I^m) = (f_1(I^m), f_2(I^m), ..., f_n(I^m))
1   for i = 1 to n
2       ρ_i ← 1
3       for j = 1 to i − 1
4           ρ_i ← ρ_i ∧ (h_j = f_j)
5       endfor
6       h_i^1 ← ∀I^m.(ρ_i ⇒ f_i = 1)
7       h_i^0 ← ∀I^m.(ρ_i ⇒ f_i = 0)
8       h_i^c ← ¬(h_i^1 ∨ h_i^0)
9       h_i ← h_i^1 ∨ (p_i ∧ h_i^c)
10  endfor
11  return (h_1(P), h_2(P), ..., h_n(P))
```

Figure 2: High Level Description of the Reparameterization Algorithm

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 a previous paper [CCK03], we 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. We used the procedure to compute successive images of a set of states to get the set of reachable states. The procedure assumes that the formula is given in conjunctive normal form (CNF). The procedure quantifies a subset of the variables and generates a disjunctive normal form (DNF) clausal representation in terms of the remaining variables. 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

$$\begin{aligned} h_i^\alpha(p_1, \ldots, p_{i-1}) &= \forall I^m.\rho_i(p_1, \ldots, p_{i-1}, I^m) \rightarrow f_i(I^m) = \alpha & (9)\\ &= \neg\exists I^m.\neg\left(\rho_i(p_1, \ldots, p_{i-1}, I^m) \rightarrow f_i(I^m) = \alpha\right) & (10)\\ &= \neg\exists I^m.\rho_i(p_1, \ldots, p_{i-1}, I^m) \wedge f_i(I^m) \neq \alpha & (11) \end{aligned}$$

Thus, the existential quantification can be carried out by our SAT-based procedure to compute $\neg h_i^\alpha$. The formula $\rho_i(p_1, \ldots, p_{i-1}, I^m) \wedge f_i(I^m) \neq \alpha$ 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 $\{p_1, \ldots, p_{i-1}\}$ 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 4. 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.

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

While enumerating cubes in variables $\{p_1, \ldots, p_{i-1}\}$ 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 $\{p_1, \ldots, p_{i-1}\}$ variables for the following formula:

$$\rho_i(p_1, \ldots, p_{i-1}, I^m) \tag{12}$$

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 $\rho_i(p_1, \ldots, p_{i-1}, I^m)$, we ask it to enumerate on

$$t_i(p_1, \ldots, p_{i-1}, I^m) = \rho_i(p_1, \ldots, p_{i-1}, I^m) \wedge (f_i(I^m) = \beta). \tag{13}$$

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 $\{p_1, \ldots, p_{i-1}\}$ 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 5.

4.4 Incremental SAT

The optimized SAT formula for computing $h_i^\alpha, \alpha \in \{0,1\}$ (Equation 12) 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:

$$\begin{aligned} \rho_i(p_1, \ldots, p_{i-1}, I^m) &= \rho_{i-1}(p_1, \ldots, p_{i-2}, I^m) \wedge \\ & \quad (h_{i-1}(p_1, \ldots, p_{i-1}) = f_{i-1}(I^m)) \end{aligned} \tag{14}$$

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 13), 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 Safety Properties and Counterexamples

So far, we have described how to do SAT-based symbolic simulation when the circuit is given in functional form, and the initial state constraint is given in parametric form. Most circuits are in functional form, however, the initial state constraint is frequently given as a predicate on the initial state variables. Safety properties are also given as predicates. We now describe how to handle the initial state and the safety property predicates and how to generate counterexamples.

5.1 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

$$v_1 = h_1(P) \wedge v_2 = h_2(P) \wedge \ldots \wedge v_n = h_n(P) \wedge Bad(V) \tag{15}$$

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

5.2 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.

6 Experimental Results

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 1) 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 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.

ckt	# regs	# PIs	bug len.	Run time			max len.	max time	# reps.
				BMC	[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 1: 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 1, the first column is the name of the circuit. In the "# regs" column, we report the number of latches in the circuit, while in the third column, we record the number of inputs of the circuit. In the column marked "bug len.", we denote 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 "fmcad 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.

7 Extensions

7.1 Proving Safety Properties

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 for a given variable ordering, the fixed point could be detected by comparing the last two parametric representations.

7.1.1 Set Union

Suppose two sets of states S_1 and S_2 are given using the parametric representations $\bar{h}(P) = (h_1(P), \ldots, h_n(P))$ and $\bar{g}(Q) = (g_1(Q), \ldots, g_n(Q))$, respectively. Suppose $P \cap Q = \emptyset$, i.e., the representations do not share parameters. If that is not the case, the parameters can just be renamed to make them disjoint. We define $\uplus$ as an operator for two parametric representations as follows:

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

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. 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. The symbolic simulator is used to disprove the property only. This is described in the next sections.

7.1.2 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 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]. Let the next state function be denoted by $f_i(V)$. Then, for each state variable v_i, a constraint of the form $f_i(V) = c_i(Q)$ is added for every time step. In the SAT formula used for symbolic simulation, each $f_i(V)$ is represented by an intermediate variable. Let vf_i denote this variable. Similarly, $c_i(Q)$ is also represented by another intermediate variable vc_i. Thus, 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. The parameters Q are fresh variables for each time step. Finally, when reparameterization is done, the Q parameters added for the invariants are removed along with the I^m variables.

For the special case of counterexample guided abstraction refinement, an abstract counterexample, which is just an assignment to a subset of the state

variables at each time step, needs to be simulated on the concrete machine. The abstract counterexample contains constraints in parameterized form that are to be applied to the symbolic simulation.

7.1.3 SAT-based Reparameterization in the Abtraction-Refinement Framework

In the abstraction-refinement framework, it is straightforward to use symbolic simulation with the SAT-based reparameterization algorithm. In [MA03], no counterexample needs to be simulated on concrete machine, as only the length of the counterexample is of interest. In this case, the reparameterization algorithm is simply used as replacement for BMC as described in Section 4.

On the other hand, in the counterexample-guided abstraction refinement framework, a counterexample $\bar{s}_m = \langle \hat{s}_0, \hat{s}_1, \ldots, \hat{s}_m \rangle$ is used to assigns values to a subset of the state variables at each step. Suppose that 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 the old parametric representation be denoted by $(f_1^{m_i}(I^{m_i}), \ldots, f_n^{m_i}(I^{m_i}))$, and let $(h_1^{m_i}(P^{m_i}), \ldots, h_n^{m_i}(P^{m_i}))$ be the newly computed parametric representation at step m_i, and let S^{m_i} be the set of states represented by it. In order to determine if the counterexample is spurious, we simulate the abstract counterexample by adding $\hat{s}_0$ as constraints to the initial state. Then we proceed by adding $\hat{s}_1, \hat{s}_2, ...$ as constraints to the symbolic simulation as described in Section 7.1.2. These constraints are just assignments of values to state variables, and hence are easy to add in the symbolic simulation.

When we reach the step m_1, the process is repeated using $\hat{s}_{m_1}$ 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

$$\hat{s}^1_{m_i} = v_1 = f_1^{m_i}(I^{m_i}) \wedge \hat{s}^2_{m_i} = v_2 = f_2^{m_i}(I^{m_i}) \ldots \hat{s}^n_{m_i} = v_n = f_n^{m_i}(I^{m_i}) \quad (16)$$

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 16 is satisfiable, then we know that the counterexample can be concretized and we proceed to build the concrete counterexample as described in the previous section. If not, we need to extract refinement information from the failed SAT instance.

For extracting refinement information, we can use the heuristics described in [CCS+02]. However, there is one important difference: Equation 16 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) already provides useful refinement information. Future work is to evaluate the quality of the refinement information that we get from such SAT instances by extensive experimentation.

7.2 Checking Liveness using Safety Properties

Biere et al. [SB03] propose to reduce a liveness property to a safety property as follows: Consider properties of the form AFp. The counterexamples to AFp properties are of the form $EG\neg p$, i.e., an infinite path such that all states on the path satisfy $\neg p$. 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 two 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 the $\mathbf{AF}\,p$ property is thus given by the truth of the $\mathbf{AG}(\texttt{looped} \longrightarrow \texttt{found})$ property.

The two methods differ in the way the `looped` signal is generated. In the first method, a counter that counts up to the completeness threshold (CT) of the $\mathbf{AF}\,p$ property that is introduced. Once the counter has gone beyond the CT, then `looped` is set to true. The counter can be very large, however. In the second method, a copy of all state variables is made and these new state variables nondeterministically copy the value of the real state variable at some step. Once a state equivalent to the saved state is found, a loop is detected. 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.

7.3 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 SAT-based reparameterization algorithm.

It is not clear that the size of the parametric functions is related to the size of the transition functions. We 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), a very high correlation ($> 85\%$) was observed while for other circuits (D2, D5, D18), the correlation was minimal ($< 60\%$). This indicates that static ordering based on the size of the transition function can be useful. 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.

8 Correctness Proof

We provide a proof of Theorem 1 in this section. We prove that the parametric representation that we get from the algorithm OrderedReparam is equivalent to the original representation. As usual, $\bar{v} = (v_1, \ldots, v_n) \in \mathcal{S}_n$ denotes a particular assignment to n variables $V = \{v_1, \ldots, v_n\}$, $\bar{\iota} \in \mathcal{W}^m$ denotes a particular assignment to variables I^m, and $\bar{p} = (p_1, \ldots, p_n) \in \mathcal{P}_n$ denotes a particular assignment to variables $P = \{p_1, \ldots, p_n\}$. We will use $\mathcal{X}_n$ and $\mathcal{Y}_n$ to denote subsets of $\mathcal{S}_n$. We will often drop the subscripts from $\mathcal{S}_n, \mathcal{P}_n, \mathcal{X}_n$, and $\mathcal{Y}_n$ when it is clear that the sets are constructed from n length vectors.

Proof of Theorem 1:

We will prove this theorem in two parts. First, we prove that $\mathcal{X} \subseteq \mathcal{Y}$, and then we prove that $\mathcal{Y} \subseteq \mathcal{X}$.

$\boxed{\mathcal{X} \subseteq \mathcal{Y}}$.

If $\mathcal{X} = \emptyset$, then the relation obviously holds. Otherwise, let $\bar{v}$ be an arbitrary element of $\mathcal{X}$. Then by definition of $\mathcal{X}$ there exists an assignment $\bar{\iota} \in \mathcal{W}^m$ such that $f_1(\bar{\iota}) = v_1 \wedge \ldots \wedge f_n(\bar{\iota}) = v_n$. In order to show that $\bar{v} \in \mathcal{Y}$, we have to provide an assignment $\bar{p} \in \mathcal{P}$ such that $h_1(\bar{p}) = f_1(\bar{\iota}) = v_1 \wedge \ldots \wedge h_n(\bar{p}) = f_n(\bar{\iota}) = v_n$.

We will prove this by induction on n.

Base Case: $n = 1$

By definition of $\rho_1(\bar{\iota})$, it is true that $\rho_1(\bar{\iota})$ is 1 for any input vector $\bar{\iota}$, or formally $\forall \iota \in \mathcal{W}^m . \rho_1(\bar{\iota}) = 1$. Subsequently, $h_1^\alpha = (\forall \bar{\iota} \in \mathcal{W}^m . f_1(\bar{\iota}) = \alpha), \forall \alpha \in \{0, 1\}$. Since h_1^1, h_1^0, and h_1^c are mutually exclusive, only one of them is 1 and the rest are 0. Also recall that $h_1(p_1) = h_1^1 \vee p_1 \cdot h_1^c$.

Now $h_1^{\neg v_1}$ has to be 0, because there is an input vector $\bar{\iota}$ for which $f_1(\bar{\iota}) = v_1$. There are two possibilities left. Either $h_1^{v_1} = 1$ or $h_1^c = 1$. If $h_1^{v_1} = 1$, then $h_1(p_1) = v_1$ for any p_1. On the other hand, if $h_1^c = 1$, then $h_1(p_1) = p_1$, so we choose $p_1 = v_1$. Then $h_1(p_1) = p_1 = v_1$.

Induction Step: $n \rightarrow n + 1$

The induction hypothesis is

$$\forall \bar{v} \in \mathcal{X}_n . \exists \bar{p} \in \mathcal{P}_n . h_1(\bar{p}) = v_1 \wedge \ldots \wedge h_n(\bar{p}) = v_n.$$

Here, $\mathcal{X}_n$ and $\mathcal{P}_n$ are used to emphasize that $\bar{v}$ and $\bar{p}$ are assignments to n variables. We have to prove that

$$\forall \bar{v} \in \mathcal{X}_{n+1} . \exists \bar{p} \in \mathcal{P}_{n+1} . h_1(\bar{p}) = v_1 \wedge \ldots \wedge h_{n+1}(\bar{p}) = v_{n+1}.$$

Let $\bar{v} = (v_1, \ldots, v_{n+1}) \in \mathcal{X}_{n+1}$. Then by definition of $\mathcal{X}_{n+1}$, there exists an $\bar{\iota} \in \mathcal{W}^m$ such that $f_1(\bar{\iota}) = v_1 \wedge \ldots \wedge f_{n+1}(\bar{\iota}) = v_{n+1}$. According to the induction hypothesis, there exists $(p_1, \ldots, p_n)$ such that $h_1(p_1) = v_1 \wedge \ldots \wedge h_n(p_1, p_2, \ldots, p_n) = v_n$. We will extend this assignment by p_{n+1} such that $h_{n+1}(p_1, \ldots, p_{n+1}) = v_{n+1}$. By definition, h_{n+1}^1, h_{n+1}^0 and h_{n+1}^c depend only on $p_1, \ldots, p_n$. So the particular assignment $(p_1, \ldots, p_n)$ assigns specific values to these three functions, and they are mutually exclusive. Note that we have $\rho_{n+1}(p_1, \ldots, p_n, \bar{\iota}) = 1$ by definition of ρ_{n+1} and by the induction hypothesis.

Also from the definition, $h^{\alpha}_{n+1}(p_1, p_2, \ldots, p_n) = 1$ if and only if for all input vectors satisfying $\rho_i(p_1, \ldots, p_n, \bar{\iota}) = 1$, f_{n+1} evaluates to α. If there exists at least one input vector for which ρ_i evaluates to 1 and f_{n+1} evaluates to $\neg\alpha$, then the $h^{\alpha}_{n+1} = 0$.

As there exists an $\bar{\iota}$ for which both $\rho_i(p_1, p_2, \ldots, p_n, \bar{\iota}) = 1$ and $f_{n+1}(\bar{\iota}) = v_{n+1}$ hold, $h^{\neg v_{n+1}}_{n+1}$ cannot be 1. Thus, there are two cases to consider. If $h^{v_{n+1}}_{n+1} = 1$, then $h_{n+1}(p_1, \ldots, p_{n+1}) = v_{n+1}$ for any p_{n+1}. On the other hand, if $h^{c}_{n+1} = 1$, then $h_{n+1}(p_1, \ldots p_{n+1}) = p_{n+1}$. In that case, we choose $p_{n+1} = v_{n+1}$. Then $h_{n+1}(p_{n+1}) = p_{n+1} = v_{n+1}$.

Thus the vector $\bar{p} = (p_1, p_2, \ldots, p_n, p_{n+1})$ has the desired property $h_1(\bar{p}) = v_1 \wedge \ldots \wedge h_{n+1}(\bar{p}) = v_{n+1}$. This holds for $p_1, \ldots, p_n$ by the induction hypothesis, and for p_{n+1} by the arguments above. So the induction step is proved.

We provided an assignment $\bar{p}$ such that $\bar{h}(\bar{p}) = \bar{v}$ for a given $\bar{v} \in \mathcal{X}$. So $\bar{v} \in \mathcal{Y}$, hence $\mathcal{X} \subseteq \mathcal{Y}$.

$\boxed{\mathcal{Y} \subseteq \mathcal{X}}$.

If $\mathcal{Y} = \emptyset$, then the relation obviously holds. Otherwise, suppose $\bar{v} \in \mathcal{Y}$. Then by definition of $\mathcal{Y}$, there exists $\bar{p} \in \mathcal{P}$ such that $h_1(\bar{p}) = v_1 \wedge \ldots \wedge h_n(\bar{p}) = v_n$. In order to show that $\bar{v} \in \mathcal{X}$, we have to provide an assignment $\bar{\iota}$ such that $f_1(\bar{\iota}) = h_1(\bar{p}) = v_1 \wedge \ldots \wedge f_n(\bar{\iota}) = h_n(\bar{p}) = v_n$. However, instead of coming up with just one input vector $\bar{\iota}$, we will come up with the largest set $\mathcal{J}^m \subseteq \mathcal{W}^m$ of input vectors such that any input vector in $\mathcal{J}^m$ will have the desired property. Formally, we will prove the following stronger claim by induction:

$$\forall \bar{v} \in \mathcal{Y}. \exists \mathcal{J}^m \subseteq \mathcal{W}^m. \left[\mathcal{J}^m \neq \emptyset \wedge \mathcal{J}^m = \left\{ \bar{\iota} \in \mathcal{W}^m | \bigwedge_{i=1}^{n} f_i(\bar{\iota}) = \bar{v}_i \right\} \right]$$

Thus, we provide a non empty set of input vectors $\mathcal{J}^m \subseteq \mathcal{W}^m$ such that for every input vector in $\mathcal{J}^m$, the function vector $\bar{f}(\bar{\iota})$ will evaluate to the state vector $\bar{v}$, and for every input vector that is not in $\mathcal{J}^m$, at least one function $f_i(\bar{\iota})$ will not match the value of the bit v_i. Mathematically, we want $\mathcal{J}^m$ to satisfy the following three conditions:

(I) $\mathcal{J}^m \neq \emptyset$

(II) $\forall \bar{\iota} \in \mathcal{J}^m . \bar{f}(\bar{\iota}) = \bar{v}$

(III) $\forall \bar{\iota} \notin \mathcal{J}^m . \bar{f}(\bar{\iota}) \neq \bar{v}$

Any $\bar{\iota}$ from $\mathcal{J}^m$ will suffice for our purpose.

We prove the claim by induction on n.

Base Case: $n = 1$

By definition, we have $\forall \bar{\iota} \in \mathcal{W}^m . \rho_1(\bar{\iota}) = 1$. Subsequently, we conclude $h^{\alpha}_1 = (\forall \iota \in \mathcal{W}^m . f_1(\bar{\iota}) = \alpha), \alpha \in \{0, 1\}$. Since h^1_1, h^0_1 and h^c_1 are mutually exclusive, only one of them is 1 and the rest are 0. We have two cases depending on whether $v_1 = 0$ or $v_1 = 1$.

Case 1: $v_1 = 1$.

Since $v_1 = h_1(p_1) = h_1^1 \vee p_1 \cdot h_1^c$, we have two sub cases to consider: If $h_1^1 = 1$, then $\forall \bar{\iota} \in \mathcal{W}^m.f_1(\bar{\iota}) = 1$, as $\rho_1 = 1$. In this case, $\mathcal{J}^m = \mathcal{W}^m$ and $\mathcal{J}^m$ is obviously non empty. Moreover, conditions (II) and (III) are also clearly satisfied by $\mathcal{J}^m$.

On the other hand, if $p_1 = 1$ and $h_1^c = 1$, then this implies that $h_1^0 = 0$. We choose $\mathcal{J}^m = \{\bar{\iota} \in \mathcal{W}^m \mid f_1(\bar{\iota}) = 1\}$ to satisfy condition (II). $\mathcal{J}^m$ is non empty, since there exist at least one $\bar{\iota}$ such that $f_1(\bar{\iota}) = 1$, due to $h_1^0 = 0$. Moreover, $f_1(\bar{\iota}) \neq 1$ for any $\bar{\iota} \notin \mathcal{J}^m$ by definition. Thus $\mathcal{J}^m$ also satisfies condition (III).

Case 2: $v_1 = 0$.

Since $v_1 = h_1(p_1) = h_1^1 \vee p_1 \cdot h_1^c$, we have $h_1^1 = 0$ and either $h_1^c = 0$ or $p_1 = 0$. So there are two sub cases to consider: If $h_1^c = 0$, then $h_1^0 = 1$. As $\rho_1 = 1$, this implies that $\forall \bar{\iota} \in \mathcal{W}^m.f_1(\bar{\iota}) = 0$. In this case, $\mathcal{J}^m = \mathcal{W}^m$ and $\mathcal{J}^m$ is obviously non empty. Moreover, conditions (II) and (III) are also clearly satisfied by $\mathcal{J}^m$.

On the other hand, if $h_1^c = 1$, this implies that $p_1 = 0$, we choose $\mathcal{J}^m = \{\bar{\iota} \in \mathcal{W}^m \mid f_1(\bar{\iota}) = 0\}$ to satisfy condition (II). $\mathcal{J}^m$ is non empty, since there exist at least one $\bar{\iota}$ such that $f_1(\bar{\iota}) = 0$, due to $h_1^1 = 0$. Moreover, $f_1(\bar{\iota}) \neq 0$ for any $\bar{\iota} \notin \mathcal{J}^m$ by definition. Thus $\mathcal{J}^m$ also satisfies condition (III).

Thus in both cases, we have found a $\mathcal{J}^m$ with the desired properties.

Induction Step : $n \rightarrow n+1$

The induction hypothesis is that

$$\forall \bar{v} \in \mathcal{Y}_n.\exists \mathcal{J}^m \subseteq \mathcal{W}^m.\left[\mathcal{J}^m \neq \emptyset \wedge \mathcal{J}^m = \left\{\bar{\iota} \in \mathcal{W}^m | \bigwedge_{i=1}^{n} f_i(\bar{\iota}) = \bar{v}_i\right\}\right]$$

We need to prove this for $n+1$, i.e.,

$$\forall \bar{v} \in \mathcal{Y}_{n+1}.\exists \mathcal{K}^m \subseteq \mathcal{W}^m.\left[\mathcal{K}^m \neq \emptyset \wedge \mathcal{K}^m = \left\{\bar{\iota} \in \mathcal{W}^m | \bigwedge_{i=1}^{n+1} f_i(\bar{\iota}) = \bar{v}_i\right\}\right]$$

For clarity, we use $\mathcal{J}^m$ for the induction hypothesis and $\mathcal{K}^m$ for the claim. Suppose we are given $\bar{v} = (v_1, v_2, \ldots, v_{n+1}) \in \mathcal{Y}_{n+1}$. By induction hypothesis, there exists a non empty $\mathcal{J}^m$ such that $\forall \bar{\iota} \in \mathcal{J}^m.f_1(\bar{\iota}) = v_1 \wedge \ldots \wedge f_n(\bar{\iota}) = v_n$. We provide a non empty $\mathcal{K}^m \subseteq \mathcal{J}^m$ as shown below such that $\forall \bar{\iota} \in \mathcal{K}^m.f_{n+1}(\bar{\iota}) = v_{n+1}$ and $\forall \bar{\iota} \in \mathcal{J}^m \setminus \mathcal{K}^m.f_{n+1}(\bar{\iota}) \neq v_{n+1}$. Then, since $\mathcal{K}^m \subseteq \mathcal{J}^m$, we already have $\forall \bar{\iota} \in \mathcal{K}^m.f_1(\bar{\iota}) = v_1 \wedge \ldots \wedge f_n(\bar{\iota}) = v_n$. So $\mathcal{K}^m$ satisfies condition (II). If $\bar{\iota} \notin \mathcal{K}^m$, then there are two cases depending on whether $\bar{\iota}$ is in $\mathcal{J}^m$ or not. If $\bar{\iota}$ is in $\mathcal{J}^m$, then the function f_{n+1} disagrees with v_{n+1}. Otherwise, induction hypothesis gives us that at least one of $f_i(\bar{\iota})$ disagrees with v_i for $i \leq n$. Thus $\mathcal{K}^m$ satisfies condition (III) as well.

Now let us find such a $\mathcal{K}^m$.

Since we have $\bar{v} \in \mathcal{Y}_{n+1}$, there is an assignment $\bar{p} = (p_1, p_2, \ldots, p_{n+1})$ such that $h_1(\bar{p}) = v_1 \wedge \ldots h_{n+1}(\bar{p}) = v_{n+1}$. Note that for all input vectors $\bar{\iota} \in \mathcal{J}^m$, $\rho_{n+1}(p_1, p_2, \ldots, p_n, \bar{\iota}) = 1$ by definition of ρ_{n+1} and by induction hypothesis. Moreover, $\forall \bar{\iota} \notin \mathcal{J}^m.\rho_{n+1}(p_1, p_2, \ldots, p_n, \bar{\iota}) = 0$.

Before commencing our inductive proof, we need to establish the following:

$$\begin{aligned} h^{\alpha}_{n+1} &= \forall \bar{\iota} \in \mathcal{W}^m.\,(\rho_{n+1} \Rightarrow f_{n+1}(\bar{\iota}) = \alpha) \\ &= (\forall \bar{\iota} \in \mathcal{J}^m.\,(\rho_{n+1} \Rightarrow f_{n+1}(\bar{\iota}) = \alpha)) \bigwedge \\ &\quad (\forall \bar{\iota} \notin \mathcal{J}^m.\,(\rho_{n+1} \Rightarrow f_{n+1}(\bar{\iota}) = \alpha)) \\ &= (\forall \bar{\iota} \in \mathcal{J}^m.\,(1 \Rightarrow f_{n+1}(\bar{\iota}) = \alpha)) \bigwedge \\ &\quad (\forall \bar{\iota} \notin \mathcal{J}^m.\,(0 \Rightarrow f_{n+1}(\bar{\iota}) = \alpha)) \\ &= (\forall \bar{\iota} \in \mathcal{J}^m.\,(f_{n+1}(\bar{\iota}) = \alpha)) \bigwedge 1 \\ \text{Thus, } h^{\alpha}_{n+1} &= \forall \bar{\iota} \in \mathcal{J}^m.f_{n+1}(\bar{\iota}) = \alpha. \end{aligned}$$

Since h^1_{n+1}, h^0_{n+1} and h^c_{n+1} are mutually exclusive, only one of them is 1 and the rest are 0. We have two cases depending on whether $v_{n+1} = 0$ or $v_{n+1} = 1$.

Case 1: $v_{n+1} = 1$.

Since $v_{n+1} = h_{n+1}(\bar{p}) = h^1_{n+1} \vee p_{n+1} \cdot h^c_{n+1}$, we have two sub cases to consider. If $h^1_{n+1} = 1$, then $\forall \bar{\iota} \in \mathcal{J}^m.f_{n+1}(\bar{\iota}) = 1$, as $\rho_{n+1} = 1$. In this case, $\mathcal{K}^m = \mathcal{J}^m$ and $\mathcal{K}^m$ is non empty since $\mathcal{J}^m$ is. $\mathcal{K}^m$ also satisfies condition (II) as the first n functions match the first n bits by induction and the last function matches the last bit 1 by the argument above. If $\bar{\iota} \notin \mathcal{K}^m$, then $\bar{\iota} \notin \mathcal{J}^m$, and at least one of $f_1, f_2, \ldots, f_n$ doesn't match the value of the corresponding bit. Thus condition (III) is also satisfied by $\mathcal{K}^m$.

On the other hand, if $h^c_{n+1} = 1$ and $p_{n+1} = 1$, then this implies that $h^0_{n+1} = 0$. We choose $\mathcal{K}^m = \{\bar{\iota} \in \mathcal{J}^m \mid f_{n+1}(\bar{\iota}) = 1\}$ to satisfy condition (II). $\mathcal{K}^m$ is non empty, since there exist at least one $\bar{\iota} \in \mathcal{J}^m$ such that $f_{n+1}(\bar{\iota}) = 1$, due to $h^0_{n+1} = 0$. For $\bar{\iota} \notin \mathcal{K}^m$, if $\bar{\iota} \in \mathcal{J}^m$, then $f_{n+1}(\bar{\iota}) \neq 1$. Otherwise, $\bar{\iota} \notin \mathcal{J}^m$, and inductive hypothesis gives us at least one $f_i, 1 \leq i \leq n$ such that $f_i(\bar{\iota}) \neq v_i$. Thus $\mathcal{K}^m$ also satisfies condition (III).

Case 2: $v_{n+1} = 0$.

Since $v_{n+1} = h_{n+1}(\bar{p}) = h^1_{n+1} \vee p_{n+1} \cdot h^c_{n+1}$, we have $h^1_{n+1} = 0$ and either $h^c_{n+1} = 0$ or $p_{n+1} = 0$. Thus, we have two sub cases to consider. If $h^c_{n+1} = 0$, then $h^0_{n+1} = 1$, so $\forall \bar{\iota} \in \mathcal{J}^m.f_{n+1}(\bar{\iota}) = 0$. In this case, $\mathcal{K}^m = \mathcal{J}^m$ and $\mathcal{K}^m$ is non empty since $\mathcal{J}^m$ is. $\mathcal{K}^m$ also satisfies condition (II) as the first n functions match the first n bits by induction and the last function matches the last bit 1 by the argument above. If $\bar{\iota} \notin \mathcal{K}^m$, then $\bar{\iota} \notin \mathcal{J}^m$, and at least one of $f_1, f_2, \ldots, f_n$ doesn't match the value of the corresponding bit. Thus condition (III) is also satisfied by $\mathcal{K}^m$.

On the other hand, if $h^c_{n+1} = 1$, then $p_{n+1} = 0$ and we choose $\mathcal{K}^m = \{\bar{\iota} \in \mathcal{J}^m \mid f_{n+1}(\bar{\iota}) = 0\}$ to satisfy condition (II). $\mathcal{K}^m$ is non empty, since there exist at least one $\bar{\iota} \in \mathcal{J}^m$ such that $f_{n+1}(\bar{\iota}) = 0$, due to $h^1_{n+1} = 0$. For $\bar{\iota} \notin \mathcal{K}^m$, if $\bar{\iota} \in \mathcal{J}^m$, then $f_{n+1}(\bar{\iota}) \neq 0$. Otherwise, $\bar{\iota} \notin \mathcal{J}^m$, and inductive hypothesis gives us at least one $f_i, 1 \leq i \leq n$ such that $f_i(\bar{\iota}) \neq v_i$. Thus $\mathcal{K}^m$ also satisfies condition (III).

Therefore, in both cases, we have found a $\mathcal{J}^m$ with the desired properties.

Hence, by the induction principle, we can always provide a $\mathcal{J}^m$ such that

$\forall \bar{\iota} \in \mathcal{J}^m.\bar{f}(\bar{\iota}) = \bar{v}$. We can choose any $\bar{\iota}$ from $\mathcal{J}^m$. Therefore, $\bar{v} \in \mathcal{X}$, hence $\mathcal{Y} \subseteq \mathcal{X}$. **QED**.

9 Conclusion and Future Work

The paper presents a SAT-based reparameterization algorithm, which allows to perform symbolic simulation much faster than using BDDs. The method uses an unwinding of the transition relation and thus is comparable to BMC. However, the reparameterization step, which is done when the equation becomes too big, allows makes it possible to go much deeper into the transition system than what BMC without reparameterization can do. The reparameterization algorithm captures a small, symbolic representation of the states that are reachable with exactly k steps. Using this representation as new initial state predicate, the algorithm starts over.

The algorithm is incomplete in that it is unable to prove the property to be correct. However, so is BMC, and the presented algorithm can be used as a replacement for BMC within most methods that make BMC complete, such as counterexample guided abstraction refinement.

In the future, we want to evaluate the performance improvements obtainable by using the algorithm as replacement for BMC in this setting. In particular, we would like to investigate how to extract proofs of unsatisfiability or interpolation-based proofs.

References

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of boolean constraints. In *Proceedings of Design Automation Conference (DAC'99)*, pages 402–407. ACM Press, June 1999.

[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.

[BCL91] 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.

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

[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.

[CCK03] 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.

[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.

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

[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.

[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.

[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] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt and F. Somenzi, editors, *Conference on Computer Aided Verification (CAV 2003)*, number 2725 in LNCS, pages 141–153. Springer-Verlag, July 2003.

[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.

[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.

[PBZ03] D. Plaisted, A. Biere, and Y. Zhu. A satisfiability tester for quantified boolean formulae. *Journal of Discrete Applied Mathematics (DAM)*, 2003. In press, available online.

[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.

[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.

[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.

[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.