A SAT-Based Algorithm for Reparameterization in Symbolic Simulation [*]

Pankaj Chauhan, Edmund M. Clarke, Daniel Kroening
{pchauhan, emc, kroening}@cs.cmu.edu
Computer Science Department, Carnegie Mellon University,
Pittsburgh, PA 15213, USA

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.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Register-Transfer-Level Implementation—*Design Aids*; J.6 [**Computer-Aided Engineering**]: [Computer-Aided Design]

General Terms

Verification, Algorithms

Keywords

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

1. INTRODUCTION

Symbolic simulation is a widely applied technique for the 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 variables and m copies of 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 f_i 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 [6, 7, 1, 12]. 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 Coudert et al. [6] and Aagard et al. [1], the reparameterization algorithm first converts the parametric representation into characteristic function form and then parameterizes this form. In [7], 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

[*]This research was sponsored by the Semiconductor Research Corporation (SRC) under task ID 1027.001, 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 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.

DAC 2004, June 7–11, 2004, San Diego, California, USA.
Copyright 2004 ACM 1-58113-828-8/04/0006 ...$5.00.

[3]. 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 (BMC), which also unrolls the circuit for a finite number of steps and also with 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, three 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.

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. We will always use m for the number of simulation steps and n for the number state variables. 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 $\bar{p} = (p_1, p_2, \ldots, p_k)$. The set of all possible 2^n vectors of n state variables is $\mathcal{S}_n$, the set of all possible 2^k assignments to k parameters is $\mathcal{P}_k$, 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. BMC AND EXTENSIONS

Model checking [5] 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. This problem is partly addressed by *Bounded Model Checking* (BMC) [2]. 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 Chaff [11]. 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 [8]) or when the SAT procedure exceeds its time or memory bounds. BMC has been used successfully to find subtle errors in very large circuits.

The enabling technique for Bounded Model Checking is propositional 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 DPLL based SAT checkers like Chaff [11], is non-chronological backtracking, efficient conflict driven learning of conflict clauses, and improved decision heuristics. A full description of SAT checkers can be found in the references mentioned above. We would just like to make the following observations.

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. 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 [11], as it has been demonstrated to be one of the most powerful SAT checker on a wide class of problems.

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:

In the counterexample guided abstraction refinement framework (CEGAR) (e.g. [4]), 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.

In [10], 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.

In [9], 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.

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

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

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 $\bar{f}(P) = (f_1(P), f_2(P), \dots, f_n(P))$ over m parameters $P = \{p_1, p_2, \dots, p_m\}$, then

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

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

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

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 into a characteristic function requires 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.

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 $p \cdot m \gg n$. The set of reachable states in m steps, as a set of state vectors in V variables, is given by

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

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* [6, 7].

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 [7, 1]. 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 [7, 1].

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 $h_1(P), h_2(P), \dots, h_n(P)$ 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 [7]:

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

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

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

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 $\{p_1, .., p_{i-1}\}$ 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 $\bar{p} = (p_1, p_2, .., p_{i-1})$ 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 $(p_1, p_2, .., p_{i-1})$. In essence, the evaluation of the new parametric functions and the old parametric functions is the same for these input vectors. We introduce the *restriction function* $\rho_i(p_1, .., p_{i-1}, I^m)$ 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

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

Note that $\rho_1 = 1$. Now the condition h_i^1 can be easily expressed as follows: We want a Boolean condition in $\{p_1, .., p_{i-1}\}$ variables under which v_i is forced to take the value 1. So if an assignment $(\bar{p}_1, \bar{p}_2, .., \bar{p}_{i-1})$ 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,

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

Analogously, h_i^0 can be expressed as

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

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.

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

Figure 1: 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 of this theorem is provided in the technical report version of this paper.

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 $\mathcal{X} = \mathcal{Y}$.*

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 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 [3], an efficient procedure to quantify existentially a large number of variables from a Boolean formula is described. The procedure assumes that the formula is given in CNF. The procedure quantifies a subset of the variables and generates a 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 use this 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

$$\begin{aligned}
& h_i^\alpha(p_1, \ldots, p_{i-1}) && (9)\\
= \; & \forall I^m . \rho_i(p_1, \ldots, p_{i-1}, I^m) \rightarrow f_i(I^m) = \alpha && (10)\\
= \; & \neg \exists I^m . \neg\left(\rho_i(p_1, \ldots, p_{i-1}, I^m) \rightarrow f_i(I^m) = \alpha\right) && (11)\\
= \; & \neg \exists I^m . \rho_i(p_1, \ldots, p_{i-1}, I^m) \wedge f_i(I^m) \neq \alpha && (12)
\end{aligned}$$

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

The SAT formulas for computing h_i^1 and h_i^0 only differ in 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 for the following formula:

$$\rho_i(p_1, \ldots, p_{i-1}, I^m) \tag{13}$$

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.

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 13) 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:

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

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 and the conflict clauses inferred from the blocking clauses need to be deleted.

4.5 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.

Suppose 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 [1]. 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 $(h_1(P), \ldots, h_n(P))$ 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.

4.6 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. 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 700 MB. The large industrial circuits we use are taken from various processor designs. Both the circuits were used in [4], 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 ([11]) with the zChaff SAT checker and the abstraction-refinement results in [4]. We invoke reparameterization when the largest function crosses a fixed threshold, which is 10000 nodes in the expression at present. 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, as described in the full version of this paper.

In Table 1, 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. For IUp1 and IUp2, properties are true. 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.

Circuit	# latches	# inputs	bug	Run time			sym max	sym max	# reparame-
			length	BMC[2]	abs-ref [4]	sym	length	time	terizations
$D2^+$	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. Columns marked with "sym" denote the results with our symbolic simulator. Times reported are in seconds. BMC was able to complete just 39 steps and then ran out of memory for IUp1 and IUp2. 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 "sym max length" column denotes the maximum length that we simulate the circuit for in the time given in the column "sym max time". The last column records the number of reparameterizations done for simulating up to the maximum length.

We would like to point out that in [4], 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 and for 183 steps in 3000 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 [7] 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.

6. 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, 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 m steps. Using this representation as new initial state predicate, the algorithm starts over.

The algorithm as given does not prove a property to be true. This is the case with BMC as well, 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.

7. REFERENCES

[1] M. D. Aagaard, R. B. Jones, and C.-J. 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.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the Design Automation Conference (DAC'99)*, pages 317–320, 1999.

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

[4] P. Chauhan, E. M. Clarke, S. Sapra, , J. Kukula, H. Veith, and D. 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.

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

[6] O. Coudert and J. C. 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.

[7] A. Goel and R. 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.

[8] D. Kroening and O. 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.

[9] K. 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.

[10] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In H. Garavel and J. 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.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC'01)*, pages 530–535, 2001.

[12] J. Yang and C.-J. H. Seger. Generalized symbolic trajectory evaluation – abstraction in action. In *Proceedings of FMCAD'02*, volume 2517 of *LNCS*, pages 70–86, 2002.