Experiments with SAT-Based Symbolic Simulation Using Reparameterization in the Abstraction Refinement Framework

Pankaj Chauhan Edmund Clarke Daniel Kroening

May 13, 2004
CMU-CS-04-137

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, Counterexample Guided Abstraction Refinement (CEGAR), Parametric Representation, Reparameterization, SAT Checkers, Bounded Model Checking

Abstract

This paper presents experimental results on the performance effect of using symbolic simulation with SAT-based reparametrization within the Counterexample Guided Abstraction Refinement framework. Abstraction refinement has been applied successfully to prove safety properties of large industrial circuits. However, all existing abstraction refinement frameworks simply use SAT-based Bounded Model Checking (BMC) to refute the property. The model used for the BMC instance is not abstracted, and thus is susceptible to the state space explosion problem. We address this issue by using a symbolic simulator with a SAT-based reparametrization algorithm as a replacement for BMC within the abstraction refinement framework. The reparametrization is performed as soon as the equations maintained by the symbolic simulator become too large. We discuss the quality of the refinement information that is extracted from the symbolic simulator.

1 Introduction

Model checking [CGP00] has become a widely applied technique that produces a major enhancement in circuit design reliability and robustness. However, the effectiveness of model checking of such systems is severely constrained by the state space explosion problem, and much of the research in this area is targeted at reducing the state-space of the model used for verification. One principal method in state space reduction is *Abstraction.* Abstraction techniques reduce the program state space by mapping the set of states of the actual system to an abstract, and smaller, set of states in a way that preserves the behaviors of the system that are of interest.

Many methods define the transition relation of the abstract circuit so that it is guaranteed to be a *conservative* over-approximation of the original circuit, i.e., any safety property that can be established on the abstraction also holds on the original circuit. Thus, if the model checker returns that the property holds on the abstract model, the algorithm terminates and the property holds on the original circuit.

The drawback of the conservative abstraction is that when model checking of the abstraction fails it may produce a counterexample that does not correspond to a counterexample on the original (concrete) circuit. This is usually called a *spurious counterexample.* In order to distinguish spurious from real counterexamples, the counterexample is simulated on the concrete circuit. If the simulation succeeds, the counterexample is real. If not so, it is spurious.

When a spurious counterexample is encountered, *abstraction refinement* is performed by adjusting the abstraction in a way that eliminates this counterexample.

Automated Abstraction Refinement

The abstract-refine process as described above is often performed in an informal, manual manner, and requires considerable expertise. The counterexample guided abstraction refinement framework (CEGAR) automates this approach [CGJ$^+$00,CCS$^+$02]. It has been applied successfully to both hardware and software. The abstraction refinement loop for software was introduced and promoted by the SLAM project at Microsoft [BR00].

First, an initial abstraction is computed. The model checking is then performed on the abstract model. Thus, if the property holds on the abstract model, it also holds on the concrete model, and the algorithm terminates. The abstraction greatly reduces the size of the model, making BDD based model checking feasible.

However, if the property does not hold on the abstract model, the property is not refuted, as the counterexample may result from spurious behavior added by the abstraction process. Thus, the abstract counterexample obtained from the model checker is then simulated on the concrete, unabstracted machine. Only if this simulation run succeeds, the property is refuted. Otherwise, the abstraction has to be refined and the process starts over. The refinement step is automated by using information obtained from the failed simulation.

The existing publications on abstraction refinement for hardware favor SAT-based Bounded Model Checking (BMC) to perform the simulation. In Bounded Model Checking, the transition system is unwound up to a given depth k to form an equation. The equation is satisfiable if and only if an error state is reachable within k steps. In the CEGAR framework, the number of steps k is the length of the abstract counterexample.

Note that the model used by the Bounded Model Checker is the original, full-size model, restricted to an abstract counterexample trace. Thus, the SAT instance will roughly be k times the size of a SAT instance corresponding to the circuit. In case of large industrial circuits, the size of this instance is already prohibitive. Previous experimental results show that the simulation step in the CEGAR framework can be a serious bottleneck [CCS$^+$02].

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, which often use information obtained from the BMC run. In [CCS$^+$02], the conflict graph maintained by the SAT solver is used to derive a measure of the importance of the variables. The most important variables are used to build the abstract model.

Symbolic Simulation

However, BMC is not the only technique that is applicable for the simulation step in the abstraction refinement loop. *Symbolic simulation* is a widely applied technique for the analysis of synchronous circuits. As in BMC, the transition relation is unwound into equations that represent the set of states that is reachable in exactly k steps. The equations are parameterized in the initial state and the inputs of the circuit. Thus, the set of states is stored in a *parametric representation.*

Most implementations of symbolic simulators use BDDs [Bry86] to represent these equations [CM90,Jon99,AJS99,Goe03,GB03,YS02]. However, 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 once the simulator is about to run out of memory. The new representation can be significantly smaller since it usually requires fewer variables. 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.

In [CCK03a,CCK04], a symbolic simulator with SAT-based reparametrization is presented. The equations are not stored using BDDs, but simply as a syntax tree with sharing. Thus, the equations only grow linearly with the number of simulation steps. Once they become too large, the algorithm performs a reparametrization using a SAT solver. This algorithm outperforms the BDD-based symbolic simulators on large examples. However, proving a safety property correct using a forward symbolic simulator requires unwinding the circuit up to the completeness threshold [KS03]. This is infeasible for large examples. Thus, the symbolic simulator is useful as means of refutation only, as is BMC. However, the symbolic simulator in [CCK04] only allows transition functions, not arbitrary transition relations.

Contribution

This paper presents experimental results on a combination of two already existing techniques:

– We extend the symbolic simulation algorithm presented in [CCK04] in order to handle arbitrary transition relations in order to allow constraining the simulation run with values from the abstract counterexample.
– We compare the performance of the CEGAR framework using BMC and using a SAT-based symbolic simulator. Our new experiments show that the symbolic simulator addresses the capacity problem caused by BMC, and that the overall performance benefits greatly from the reduced simulation time.
– During reparametrization, some information from the earlier transitions is lost, as only the set of reachable states is retained. This lost information is no longer available to compute a refinement in the case the simulation fails. The experiments show that this loss is insignificant for most circuits. However, the new algorithm fails on a few medium-size benchmarks due to insufficient refinement.

Related Work In [CGKS02], various ways of obtaining refinement information are explored. The refutation is done using SAT-based BMC.

In [MA03], the CEGAR 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 [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 cited approaches therefore solely rely on Bounded Model Checking to refute the property. The extensions that are introduced by these publications are used only to improve refinement or to detect the case that the property is true. The related work does not address the simulation bottleneck.

Outline In section 2, we provide background information about counterexample guided abstraction refinement and related techniques. In section 3, we describe how the reparametrization step can be adjusted to take additional constraints on the transition relation into account. In section 4, we report the results of our new experiments. In section 5, we describe how to detect fixed points during the symbolic simulation.

2 SAT-Based Counterexample Guided Abstraction Refinement

We briefly describe the SAT-based CEGAR framework used in [CCS+02] in this section. More details can be found in the referenced paper.

2.1 Localization Reduction

For circuits, a very simple and inexpensive form of abstraction, called *Localization Reduction* [Kur94] has proven to be effective: Latches are replaced by free inputs, and the logic that computes the next value of the latch is removed. The remaining latches are called the *visible* latches. The latches that are removed are called *invisible latches.* The resulting circuit is smaller, and hence easier to verify.

This method defines the transition relation of the abstract circuit so that it is guaranteed to be a *conservative* over-approximation of the original circuit, i.e., any safety property ϕ that can be established on the abstraction also holds on the original circuit. An example of a more general abstraction technique is predicate abstraction. The drawback of any conservative abstraction is that when the verification of the abstract model fails, one may obtain a counterexample that does not correspond to any concrete counterexample. This is usually called a *spurious counterexample.* When a spurious counterexample is encountered, *refinement* is performed by adjusting the set of visible latches in a way that eliminates this counterexample. The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm [Kur94,CGJ+00,DD01], or CEGAR for short.

This framework is shown below: one starts with a coarse abstraction h, and then one verifies the abstract transition relation $\hat{M}$ induced by h. If the abstract model checking run fails and generates a counterexample, the counterexample is simulated on the concrete model M to see if it is valid or not. If it is not valid, the counterexample is analyzed to infer the refinement h' of the abstraction function. The actual steps of the loop follow the *abstract-verify-refine* paradigm and depend on the abstraction and refinement techniques used.

1. Generate an initial abstraction function h.

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

2.2 Validating the Abstract Counterexample

Given an abstract model $\hat{M}$ and a safety formula ϕ, we run the usual BDD-based symbolic model checking algorithm to determine if $\hat{M} \models \phi$. Suppose that the model checker produces an abstract counterexample path $\langle c_0, c_1, \ldots, c_k \rangle$. Let $t(0), \ldots, t(k)$ be a trace in the concrete machine. In order to check whether this counterexample also exists in the concrete model M or not, we symbolically simulate M beginning with the initial state $I(t(0))$ using a fast SAT checker. At each stage of the symbolic simulation, we constrain the values of the visible variables according to the abstract counterexample. Thus, the equation for BMC-based symbolic simulation is:

$$\begin{aligned} I(t(0)) \wedge c_0(t(0)) \wedge R(t(0), t(1)) \wedge c_1(t(1)) \wedge \ldots \\ \wedge R(t(k-1), t(k)) \wedge c(t(k)) \end{aligned} \tag{1}$$

Each $c(t(i))$ is a predicate that constraints the visible variables in the state $t(i)$. The invisible variables are not constrained. If this propositional formula is satisfiable, we successfully simulated the counterexample on the concrete machine and can conclude that $M \not\models \phi$. As done in BMC, a counterexample trace can be extracted from the satisfiable assignment provided by the SAT solver.

2.3 SAT-Based Refinement

If the counterexample is spurious, then formula 1 is unsatisfiable. Modern SAT checkers can identify the cause of unsatisfiability of a SAT instance (see, e.g., [ZM03]). In [CCS$^+$02], we proposed two methods to determine a small set of variables necessary for the unsatisfiability of the SAT formula. The first method is based on scoring invisible variables during the SAT check. Essentially, a weighted score based on the number of backtracks a variable receives during the SAT check and the number of times the variable appears in a conflict clause is computed. The invisible state variables from all the simulation steps are ranked based on this score, and a small set of the highest scored variables are used for the refinement. In the second method, a conflict dependency graph is built to analyse the relations between various conflicts that occur during unsatisfiable SAT check. From the roots of this directed graph (vertices with no incoming edges), the causes for the unsatisfiability are inferred. The variables corresponding to the roots of the graph are then used as new visible variables.

The set of refinement candidates identified from conflict analysis is usually not minimal, i.e., not all registers in this set are required to invalidate the current spurious abstract counterexample. To remove those that are unnecessary, we have adapted the greedy refinement minimization algorithm in [WHL$^+$01]. This refinement algorithm has two phases. The first phase identifies the registers sufficient to prevent the spurious counterexample. In the second phase, a minimal set of registers necessary to prevent the counterexample is identified. For our experiments we only use the second phase, in which we remove one register at a time to see if the counterexample is removed or not. If not, then the register is not required in the refinement.

Note that data from of the whole counterexample is used to infer refinement information. If reparameterization is used, we lose all the information from the counterexample up to the last time the parametrization was done, and hence only the last segment of the counterexample is analysed to infer refinement.

3 SAT Based Reparameterization in Symbolic Simulation

In [CCK04], we presented an algorithm for SAT-based reparameterization in symbolic simulation for functional circuits. In order to use it to simulate abstract counterexamples in the CEGAR framework, it has to be extended to handle handle general transition relations (for example through SMV style `TRANS` and `INVAR`) statements.

This section describes the parameterization algorithm when $R(\bar{v}, \bar{v}')$ is the transition relation of the system. We assume that the states $\bar{v}$ of the transition system are an assignment to a vector of n state bits. The bit i of the vector v is denoted by v_i.

Let $c_0(\bar{v}), c_1(\bar{v}), \ldots, c_k(\bar{v})$ be predicates on concrete states $\bar{v}$. The predicates correspond to the constraints imposed by an abstract counterexample with k steps that we are interested in simulating on the concrete machine.

Let $\tau(t)$ denote a predicate that holds if and only if t is a valid concrete trace of length k in the model M conforming with the counterexample $c_0, \ldots, c_k$, or formally:

$$\tau(t) :\Longleftrightarrow I(t(0)) \wedge c_0(t(0)) \bigwedge_{j=0}^{k-1} (R(t(j), t(j+1)) \wedge c_{j+1}(t(j+1))) \tag{2}$$

We aim at obtaining a small, symbolic representation for the set of all states $\bar{v}$ such that there exists a trace of length k in M that ends in the state $\bar{v}$. We denote the set by $\mathcal{X}$.

$$\mathcal{X} := \{\bar{v} \in \mathcal{S} \,|\, \exists t \in \mathcal{S}^{k+1} : \tau(t) \wedge \bar{v} = t(k)\} \tag{3}$$

The set $\mathcal{X}$ is then used in a new simulation instance instead of the original initial state predicate I. This process can be iterated to explore the model further until the counterexample is either found to be real or spurious. In order to make this process efficient, a small, symbolic representation for $\mathcal{X}$ must be found. We now describe how to compute a parametric representation for $\mathcal{X}$.

For each state bit v_i, a (re-)parametrization algorithm computes a new function $h_i(\bar{p})$. The function maps a parameter vector $\bar{p}$ to the value of the state bit i.

The vector of all such functions is denoted by $\bar{h}(\bar{p})$. The set of states represented by the functions is simply the range of $\bar{h}$, i.e., the set of values of the functions for arbitrary parameters. Thus, the representation is called parametric. Formally, the set of states represented by the functions $\bar{h}(\bar{p}) = (h_1(\bar{p}), h_2(\bar{p}), \ldots, h_n(\bar{p}))$ is denoted by $\mathcal{Y}$:

$$\mathcal{Y} := \{\bar{v} \in S \mid \exists \bar{p} \in \mathcal{P}.\bar{h}(\bar{p}) = \bar{v}\} \tag{4}$$

The parameter $\bar{p}$ is a vector of bits $\{p_1, p_2, \ldots, p_l\}$, where $l \leq n$. We denote the set of all parameter vectors by $\mathcal{P}$. Thus, the number of parameters is at most equal to the number of state variables.

The functions h_i have a specific structure. The function h_i only depends on the parameters p_1 to p_i. The algorithm computes these functions in the order $h_1, h_2, \ldots, h_n$.

Note that a particular assignment to the state variables v_1 to v_i may restrict the possible values any later bit may have. As an example, consider the set of states consisting of the three states $(0,1)$, $(1,0)$ and $(1,1)$. If h_1 maps a particular $\bar{p}$ to 0, then h_2 must map the same $\bar{p}$ to 1. We say that the second state bit is *forced* to 1. In contrast to that, if h_1 maps $\bar{p}$ to 1, the value of h_2 is not restricted. It may either be 0 or 1, i.e., it has free *choice*.

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, 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 example above, suppose we let the first bit be represented by the free parameter p_1. If the first bit is 0, then the second bit is forced to be 1. 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 [GB03]:

$$h_i(p_1, \ldots, p_i) = h_i^1(p_1, \ldots, p_{i-1}) \vee (p_i \wedge h_i^c(p_1, \ldots, p_{i-1}))\,. \tag{5}$$

Intuitively, Equation 5 is interpreted as follows. If h_i^1 holds, the value of bit v_i is 1 regardless of the other two functions, hence the first term in the equation. If h_i^c is true, the choice is free, and the bit is given by the parameter p_i. Otherwise, the bit is forced to zero.

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

$$h_i^c = \neg(h_i^1 \vee h_i^0) = \neg h_i^1 \wedge \neg h_i^0. \tag{6}$$

Continuing our example, we get $h_2(p_1, p_2) = \neg p_1 \vee (p_2 \wedge p_1)$. Thus, to compute h_i, it is sufficient to compute any two of the three functions h_i^1, h_i^0 and h_i^c, which we describe now.

3.1 Computing h_i^1 and h_i^0

As described above, choosing specific values for the parameters p_1 to p_{i-1} restricts the value the function h_i can have, as the values for the previous bits v_1 to v_{i-1} may force v_i to be either 0 or 1. We formalize this as follows: the predicate ρ_i takes as arguments the parameters p_1 to p_{i-1} and a trace t. It is true if and only if the following two conditions hold:

1. The trace is a valid trace in M, i.e., $\tau(t)$ holds.
2. The first $i-1$ state bits of the last state in the trace match the values given by the functions $h_1(p_1), h_2(p_1, p_2), \ldots, h_{i-1}(p_1, \ldots, p_{i-1})$.

Formally, ρ_i is defined as:

$$\rho_i(p_1, \ldots p_{i-1}, t) := \tau(t) \wedge \bigwedge_{j=1}^{i-1} h_j(p_1, \ldots, p_j) = t(k)_j. \tag{7}$$

Here, $t(k)_j$ denotes the j^{th} state bit of state $t(k)$. Intuitively, $\rho_i(p_1, p_2, \ldots, p_{i-1}, t)$ indicates that a trace t is valid and it conforms to the parameters $p_1, p_2, \ldots, p_{i-1}$. Note that $\rho_1(t) = \tau(t)$, thus, ρ_1 is 1 for any valid trace and $\rho_i(p_1, \ldots, p_{i-1}, t) = 0$ for any invalid trace t.

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. Thus, if an assignment $(p_1, p_2, \ldots, p_{i-1})$ makes $h_i^1(p_1, \ldots, p_{i-1})$ true, then that implies that all traces t that conform with this assignment end in a state $t(k)$ where $t(k)_i$ is 1.

$$h_i^1(p_1, \ldots, p_{i-1}) = \forall t \in \mathcal{S}^{k+1}. \left(\rho_i(p_1, \ldots, p_{i-1}, t) \rightarrow t(k)_i = 1\right) \tag{8}$$

Analogously, h_i^0 can be expressed as

$$h_i^0(p_1, \ldots, p_{i-1}) = \forall t \in \mathcal{S}^{k+1}. \left(\rho_i(p_1, \ldots, p_{i-1}, t) \rightarrow t(k)_i = 0\right). \tag{9}$$

Note that $h_1(p_1) = p_1$, unless the bit v_1 is always 1 or 0, in which case $h_1 = 1$ or $h_1 = 0$. This follows automatically from $\rho_1 = \tau(t)$. The Equations 5 to 9 give us an algorithm for computing a symbolic representation of the set of states reachable in exactly k steps.

As described in [CCK04], we use the procedure described in [CCK03b] to obtain h_i^α by the use of SAT-based enumeration. We also use incremental SAT and a single SAT-enumeration for computing both h_i^0 and h_i^1, as it is done in [CCK04].

4 Experimental Results

We embed the symbolic simulation algorithm with SAT-based reparametrization into the abstraction refinement framework described in [CCS+02]. The symbolic simulation algorithm is used to replace BMC as means of simulating abstract counterexamples. The refinement information is extracted from the full simulation run as in [CCS+02]. In contrast to that, the proposed algorithm with symbolic simulation extracts refinement information only from the last segment of the counterexample simulation. This may result in refinement information of lower quality. Not that both algorithms are just refinement heuristics, and none guarantees the elimination of the spurious counterexample.

Both methods use a BDD-based model checker for the verification of the abstract model. The model checker is based on NuSMV and uses dynamic variable ordering. Apart from deriving refinement information, the initial variable orders for the BDD-based model checker are also derived from the analysis of failed counterexample, as described in [CCS+02]. In the very first iteration of the abstraction refinement loop, no variable orders are provided to NuSMV.

Table 1 lists the circuits that we used for the experiments, and provides some characteristics of the circuits. The circuits are from three different classes. The D and M series circuits are processor benchmarks. The IU circuits are models of the picoJava microprocessor from Synopsys, and the s-series circuits are ISCAS89 sequential benchmarks.

The D, M and IU series benchmarks already come with properties. In contrast to that, there are no properties available for the ISCAS89 circuits. We used random simulation to infer reasonable properties for these circuits. The property verified for the s3271 circuit is $\mathbf{AG}\,\mathbf{AF}(\bigvee_{i=0}^{6} ManFinal_i)$, for s13207 the property is $\mathbf{AG}\,\neg(g12 \wedge g1229 \wedge g1325 \wedge 1391 \wedge g1431 \wedge g972 \wedge g182)$, for s15850 the property is $\mathbf{AG}\,\neg(g109 \wedge g878 \wedge g901)$, and for s38417, the property is $\mathbf{AG}\,\neg(g222 \wedge g342)$. We also experimented with other ISCAS89 circuits, however, the length of the longest counterexample to simulate on these circuits was either too short to be of interest, or the time taken by the SAT-based simulation was too small a fraction of the total time.

circuit	# latches	# inputs	bug. length
D6	161	16	20
D18	498	247	28
D19	285	49	32
D20	532	30	14
M3	334	155	true
M4	744	95	true
M5	316	104	true
IUp1	4494	361	true
IUp2	4494	361	true
IUp3	4494	361	true
s3271	116	26	true
s13207	669	31	true
s15850	597	14	true
s38417	1636	28	true

Table 1. Circuits used for abstraction-refinement experiment.

We performed our experiments on a machine with dual AMD Athlon MP 1800+ processors and 3GB memory. The reparameterization is done as soon as the size of the SAT instance for the simulation exceeds 700MB. The total amount of memory was limited to 2.5GB.

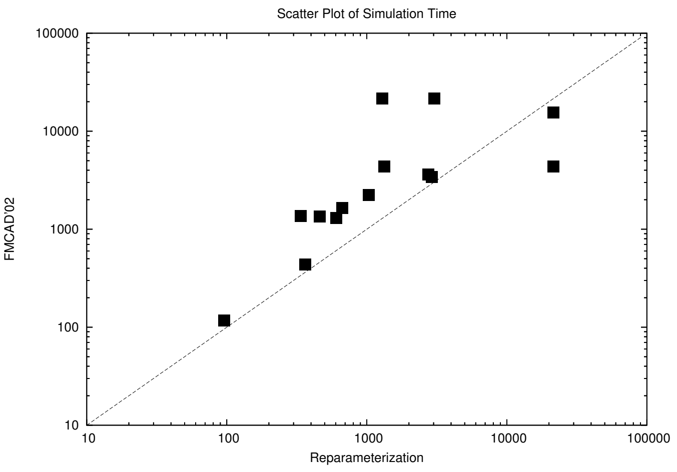
Table 2 describes the comparative experiment of the new technique with the results as described in [CCS+02]. The refinement technique used and all other parameters were the same in both sets of experiments. The only difference is the algorithm used for simulation.

The columns marked "sym" are for the new algorithm, while the columns marked "fmcad" are for the old algorithm. The set marked "# refn" compares the number of refinement iterations required, the set marked "|reg|" compares the number of latches in the final abstract model, the set marked "max |CE|" compares the length of the longest counterexample encountered, the set marked "sim. time" compares the time spent in the simulation of abstract counterexamples over all refinement iterations, and the set marked "total time" compares the total time to prove the property or to disprove it. The last column marked "# rep" lists the total number of reparameterizations done across various simulations for the circuit. Verification was not complete for circuits when the numbers are in bold typeface with an accompanying symbol. The run times are given in seconds.

ckt	# refn		|reg|		max |CE|		sim. time		total time		# rep
	fmcad	sym	fmcad	sym	fmcad	sym	fmcad	sym	fmcad	sym	
D6	48	48	39	39	20	20	438	362	845	718	23
D18	142	127	253	253	28	28	3598	2740	9873	8349	56
D19	37	49	103	112	32	32	4348	1329	14528	12087	95
D20	74	74	265	265	14	14	1359	338	2794	2192	23
M3	58	**42†**	128	**87†**	54	**54†**	4378	**2088†**	15306	>**21600†**	3
M4	173	**94†**	336	**184†**	44	**39†**	15540	**4776†**	20327	>**21600†**	21
M5	7	11	30	30	6	10	3427	2902	8653	10312	3
IUp1	**8‡**	13	**12‡**	19	**72‡**	72	**3390‡**	1295	**4877‡**	4063	117
IUp2	6	6	13	13	22	22	1298	605	2498	1335	16
IUp3	**17⋆**	32	**19⋆**	41	**52⋆**	67	> **21600⋆**	3022	> **21600⋆**	5836	325
s3271	32	32	38	38	48	48	117	96	198	174	3
s13207	15	15	23	23	43	43	2231	1035	4066	2454	13
s15850	8	8	18	18	56	36	1643	669	2998	2108	8
s38417	19	19	29	29	53	53	1347	462	1655	1077	14

Table 2. Comparison of SAT based reparameterization symbolic simulation against plain SAT based simulation as in [CCS+02]. †: Model checking of abstract model timed out, ‡: Simulation of counterexample failed, and ⋆: Simulation of counterexample timed out.

In Figure 1, we show the scatter plots of the simulation time and the total model checking time for both techniques. The horizontal axis is for the new



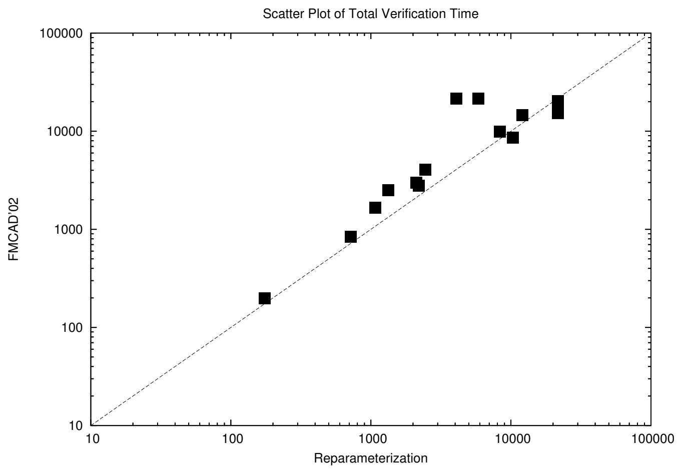


Fig. 1. Scatter plots of simulation time and total time.

simulation algorithm, while the old algorithm is represented by the vertical axis. For the failed instances, we used the time value 21600 in the scatter plots.

The new simulation algorithm yields useful refinement information in most experiments, and the improvement in run-time is due to the faster simulation. The large circuits IUp1 and IUp3 fail to verify with the original simulation algorithm, but can be verified with the new technique. The simulation using SAT-based BMC exceeds the memory bound for IUp1 and the time bound for IUp3. The difference between IUp1 and IUp3 is due to the fact that there is only one very long counterexample for IUp1, while for IUp3 there are multiple long counterexamples. The sum of the time required to simulate all the counterexamples exceeds the time bound.

However, the medium-sized circuits M3 and M4 show negative results. These circuits fail to verify within the time limit of 6 hours because the BDD-based model checking of abstract model times out. We examined the failure of the new algorithm for the circuits M3 and M4. For the M4 circuit, the new set of latches obtained from the truncated simulation using the new technique was different from that obtained by the original algorithm. Thus, the failure is caused by the low quality of the refinement information.

For the M3 circuit the set of latches computed by the new algorithm is the exact same as computed by the BMC-based algorithm. However, we analyze the failed counterexample simulation to derive variable orders for the BDDs used for verifying the abstract model. The BDD variable orders obtained by the new method were different than those obtained by the old method, and cause the BDD-based model checker to fail. When we used the variable orders derived by the old method, the abstract model checking in the new method was successful for 6 more refinement iterations, after which the model checking of abstract model checking failed due to a different set of latches.

5 Computing Fixed Points by Introducing Self Loops

The symbolic simulation computes the set of states reachable in exactly k steps. In order to find fixed points, we need to compute the set of states reachable in k steps or less and we also need a method to compare two representations. In [CCK03a], a method to compute the union of the sets of states in parametric form is presented. However, the method is too expensive to be of any practical use. The majority of the cost is in invoking reparameterization after each simulation step. However, the following method can be used to compute the union of the set of states. The idea is to modify the transition relation such that it also allows self-loops back to each state. Thus, if the original transition relation is $R(v, v')$, we change it to $R(v, v') \vee (v = v')$. For functional circuit descriptions, this can be achieved by driving each latch input from a multiplexer controlled by a free input. The multiplexer selects either the original latch input or the latch state. This is a well known approach for nondeterministically "stalling" the state machine.[1] When simulating using this modified transition relation for k steps, we get the set of states reachable in k or less steps.

In order to detect whether we have reached fixed point or not, we need to compare two state set descriptions for equality. Since our reparameterization algorithm produces canonical representations (provided the order of the state variables is the same), we only need to compare the two parametric representations on a function by function basis. Note that we do not need to invoke reparameterization after each step of the simulation. We just need to compare the last two parametric representations for equality. Suppose $H^k(P) = (h_1^k(P), \ldots, h_n^k(P))$ and $H^{k+\delta}(P) = (h_1^{k+\delta}(P), \ldots, h_n^{k+\delta}(P))$ are the last two parametric representation. Note that δ can be and is usually greater than 1. In order to compare these two representations, we need to compare each function $h_i^k(P)$ with $h_i^{k+\delta}(P)$. Since we represent these functions by Boolean expressions and not by some canonical data structure such as a BDD, a method for checking equality is required. The simplest method is to check $h_i^k(P) \oplus h_i^{k+\delta}(P)$ for satisfiability. If the formula is satisfiable for any i, then the two representations are not equal, and the fixed point is not yet reached. We can also use state of the art combinational equivalence checkers to accomplish this task.

For the circuits we experimented with, the diameter is far too large to actually reach the fixed point. Within the time bound of 6 hours, we were able to simulate the circuit D24 for 8744 steps without reaching a fixed point, the circuit M4 was simulated for 238 steps without reaching the fixed point and the circuit IUp1 was simulated for 936 steps without reaching the fixed point. Even though the the algorithm was not able to reach fixed point for the circuits, the extension of adding self loops to compute the unions of the sets of states at least theoretically allows one to use the reparameterization based algorithm for general property checking. To the best of our knowledge, there is no other algorithm available that is able to reach these depths in a fixed point iteration on such large circuits.

6 Conclusion and Future Work

Using experiments on large industrial circuits, we show that the use of symbolic simulation with SAT-based reparametrization within the Counterexample Guided Abstraction Refinement framework can yield significant performance improvements and enables the verification of larger circuits.

However, the results also show that there are a few circuits for which the SAT-based reparametrization provides insufficient refinement information, and thus, performs worse than BMC. The new technique is therefore not clearly dominant over the old technique, and the user should be given a choice of both techniques.

Both CEGAR and symbolic simulation with SAT-based reparametrization are known already; the contribution of this paper is the quantification of the performance of the combination.

[1] The authors thank Armin Biere for suggesting this.

Future research will investigate criteria that can predict the success of either simulation technique and automated ways to decide which technique should be used. We will also investigate the performance impact using different refinement algorithms.

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.

[BR00] T. Ball and S.K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

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

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

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

[CCK04] Pankaj Chauhan, Edmund Clarke, and Daniel Kroening. A SAT-based algorithm for reparameterization in symbolic simulation. In *Proceedings of DAC 2004*, 2004. To appear.

[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.

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

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

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

[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.

[DD01] Satyaki Das and David Dill. Successive approximation of abstract transition relations. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS'01)*, 2001.

[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, 1999.

[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 *LNCS*, pages 298–309. Springer Verlag, January 2003.

[Kur94] R. Kurshan. *Computer-Aided Verification of Co-ordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[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 *LNCS*, pages 2–17. Springer, 2003.

[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 *LNCS*, pages 1–13. Springer, July 2003.

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

[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.

[ZM03] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In S. Margherita Ligure, editor, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.