Deeper Bound in BMC by Combining Constant Propagation and Abstraction

Roy Armoni, Limor Fix[1], Ranan Fraer[1], Tamir Heyman[1,3],
Moshe Vardi[2], Yakir Vizel[1], Yael Zbar[1]
[1]Logic and Validation Technology, Intel Corporation, Haifa, Israel
[2]Rich University, Houston, Tx
[3]Carnegie Mellon University, Pittsburgh, PA

Abstract The most successful technologies for automatic verification of large industrial circuits are bounded model checking, abstraction, and iterative refinement. Previous work has demonstrated the ability to verify circuits with thousands of state elements achieving bounds of at most a couple of hundreds. In this paper we present several novel techniques for abstraction-based bounded model checking. Specifically, we introduce a constant-propagation technique to simplify the formulas submitted to the CNF SAT solver; we present a new proof-based iterative abstraction technique for bounded model checking; and we show how the two techniques can be combined. The experimental results demonstrate our ability to handle circuit with several thousands state elements reaching bounds nearing 1,000.

I. Introduction

Since the introduction of *model checking* in the early 1980s [9, 20], its capacity has continued to increase. While early implementations were able to handle designs with only a few thousands of states, later implementations could handle millions of states [5]. *Symbolic model checking* [7], based on Binary Decision Diagrams [6] (BDDs), pushed the capacity to 10^{20} states and more. While such numbers may seem astronomical, in reality they correspond to designs with hundreds state elements [3]. At the same time, design blocks with well-defined functionality typically have thousands of state elements. SAT-based *bounded model checking* (BMC) [3] can typically handle designs with thousands of state elements, but at the cost of limiting the search to counterexamples of bounded length. In practice, SAT-based bounded model checking can rarely reach search bound of 100 or more for designs with thousands of state elements. While many errors can be discovered with bounded search, the small search bound limit confidence in design correctness.

Automated abstraction techniques [17, 8, 18] aim at finding automatically, through a sequence of iterative approximations, a conservative abstraction of the design under verification, and then proving that this abstracted design satisfies the specification using model-checking technology. Such an approach [12, 13] aimed at finding a conservative approximation, but this abstraction is verified, up to a given search bound, using SAT-based bounded model checking. This combination of automated abstraction and bounded model checking, which can be described as *abstracted bounded model checking*, is still in essence a bounded-model-checking technique, but the application of abstraction enables dealing with more complex designs and larger search bounds. While abstracted bounded model checking can handles designs with thousands of state elements, it rarely can reach search bounds beyond a couple of hundreds [12, 13].

Our goal in this paper is to scale abstracted bounded model checking further, aiming to reach search bounds nearing $1,000$ for large designs. We accomplish this by combining two significant algorithmic improvements. Our first key observation is that a bounded model-checking instances are often subject to various constraints (constant values in different time frames) that originate from various initialization and environmental assumptions on the design under verification. While one can conjoin these constraints to the instance submitted to the satisfiability solver by the bounded model checker, we argue that it is more effective to add a pre-processing stage in which these constraints are propagated in order to simplify the formula submitted to the solver. This simplification is typically iterative; as constraints are used to simplify the formula, new constraints are generated, leading to further simplification. We found that constraint propagation leads to significant formula simplification. Our results demonstrate significant improvement in execution time and memory consumption. This leads to an improved bounded model-checking algorithm, which we call BMC-CP. Similar preprocessing simplifications have been shown to be successful [1, 4, 14, 2, 15, 19]. Our contribution is showing how inputs of the design with a known cyclic pattern can be used as constrains in preprocessing simplifications of the BMC formula for a CNF SAT solver.

Our second contribution is an improvement of abstracted bounded model checking. As in [12], our algorithm, ABMC, uses *proof-based abstraction* [18]. Starting with a design M, the algorithm generates a sequence $M'_1, M'_2, \ldots$ of abstracted designs. The property P is checked in each abstracted design up to some large bound t (say, $1,000$). If P holds up to bound t in M'_i, the algorithm stops and concludes that P also holds up to bound t in the original design M. If P does not hold in M'_i, the algorithm proceeds to the next iteration and generates a new abstracted design M'_{i+1}. The construction of M'_{i+1} is based on the unsatisfiability proof generated by the satisfiability solver for the bounded model-checking instance of M. We differ from [12] and [18] in the way we proceed from M'_i to M'_{i+1} and by the use of proof based abstraction for BMC. Our experimental results show that in many cases our technique reaches larger bounds.

Finally, we combine the constant-propagation technique of BMC-CP with the abstraction technique of ABMC. The combination is far from been trivial. During constant propagation the simplification eliminates many variables from the formula. If the abstraction is applied naively, variables eliminated from the formula for M will not appear in the abstracted design M'_i. This may result in too aggressive abstraction, which leads to too many approximation iterations. We describe here a novel constant-propagation procedure that overcomes this difficulty and enables us to combine constant propagation with abstraction. Overall, this paper presents three improvements to the basic BMC algorithm: (1) BMC-CP - BMC with constant-propagation, (2) ABMC - proof based abstraction and BMC, (3) ABMC-CCP - abstraction and constant-propagation and BMC. Moreover, the exper-

1-4244-0630-7/07/$20.00 ©2007 IEEE.

imental results indicate that ABMC-CCP is the strongest among these three improvements, that is, it reaches the deepest bound on our suite of industrial testcases. Previous paper that combines formula simplification technique with proof based abstractions is [11]. In [11], in contrast to our approach, the simplification are not performed in a preprocessing step, instead additional constrains are added to the SAT formula.

The paper is organized as follows. Section II describes BMC-CP, which combines bounded model checking with constant propagation. Section III describes ABMC, which combines bounded model checking with abstraction. Then, Section IV describes the combination of constant propagation with abstracted bounded model checking. Finally, Section V gives conclusions.

II. Constant Propagation

We present here a short summary of the BMC approach, as described in [3]. Let M be a model with state variables X and input variables U. Let $V \equiv X \cup U$ be the set of all variables. The initial states of M are defined as a set of constraints $I(V)$. The possible transitions of the model are also a set of constraints denoted by a transition relation $TR(V, V')$, where V (V') is the set of current (next) variables in the model before (after) a transition. Let $P(V)$ be a combinational predicate over the current variables V. For a given bound k, a BMC checks whether $P(V)$ holds in all model executions of length k. The transition relation is explicitly unfolded to $k+1$ time frames, $TR(V_{t-1}, V_t), t = 1 \dots k$, where V_t is a copy of the variables V at time frame t. Then, the formula $TR(V_0, V_1) \wedge \dots \wedge TR(V_{k-1}, V_k)$ denotes all executions of the model of length k.

$P(V)$ holds in all model executions of length k that start at initial state satisfying $I(V)$ if and only if the following formula is unsatisfiable.

$$\varphi \equiv I(V_0) \wedge TR(V_0, V_1) \wedge \dots \wedge TR(V_{k-1}, V_k) \wedge \bigvee_{t=0}^{k} \neg P(V_t).$$

A. Algorithmic Framework

The data structure used for circuit-based unfolding is an Expression Graph, to be denoted EG, similar to the And-Inverter Graph (AIG) in [10, 16, 15]. EG is a directed-acyclic graph. The leaves of EG are variables or constants (true, false). The internal nodes are logical operators. To simplify, we restrict the discussion in this paper to the binary AND operator and the unary NOT operator.

The EG of a BMC formula φ is built bottom-up in an iterative fashion. We start by allocating leaf nodes for each variable v_0 that corresponds to a variable v at time frame 0. Then, for each variable v and each time frame t, we add to the graph a node that represents v_t. The expression for x_t is generated by substituting in the next-state function $F_x(V, U')$ the nodes in V_{t-1} and the variables in U_t respectively.

In a typical hardware design several input signals are known to have constant values at different time frames. For instance, the initial state of the execution is usually a state obtained after applying a reset sequence. Many of the state signals are initialized to constant values, and we can take advantage of these constants in the first time frame of the unfolding.

Some of the inputs of the design have a known cyclic pattern. For instance an input that toggles in every time frame. If the initial value of this input is known to be constant zero or constant one, then one can compute the value of the input at every time frame. Another typical pattern is input that stays constant for a few time frames and then becomes free.

Our algorithm takes advantage of this information by injecting the constant values into the EG at different time frames according to the input pattern and performing the constant propagation (CP) described later. To illustrate the approach, consider a simple model with two state variables x and y and two inputs c and r. The property to check is

$$P(x, y) \equiv (x = y).$$

The initial states are defined by

$$I(x, y) \equiv (x = 1) \wedge (y = 0).$$

The transition relation is

$$TR(x, y, x', y') \equiv \begin{cases} x' & = (\neg c \wedge c')?(\neg y \wedge r) : x \ \wedge \\ y' & = (c \wedge \neg c')?x : y \end{cases}$$

We use the knowledge that the input c is initialized to 0 and it is toggling on every phase and change the AIG accordingly ($c_0 = 0, c_1 = 1, c_2 = 0$). In addition, x is initialized to 1 (the node x_0 in the AIG gets the value 1) and y is initialized to 0 ($y_0 = 0$). The final BMC formula for bound 2 after CP is: $(y_2 = r_1) \wedge (\neg(0 = y_2))$.

The constant propagation algorithm manipulates the EG, it propagates the values of constant nodes and replaces additional nodes in the graph by constants. The pseudocode for the constant-propagation algorithm presented in Figure 1 refers only to negation and conjunction but can be extended to many types of operators. The algorithm evaluates a subexpression in EG that corresponds to a node e by traversing the subgraph rooted in e, using a DFS post order. In particular, if the values of the operand(s) of e imply a constant value on e the algorithm replaces the entire subgraph rooted in e by the calculated constant value. Each node e in the EG may have no operands, a single operand denoted $e.e1$, or two operands, denoted $e.e1, e.e2$.

```
function  CP(e)
  if (e.visited) return e
  e.visited = true /* new node*/
  if(e is constant) return e
  e.e1= CP(e.e1)
  if (e.operator = negation ∧ (e.e1 = zero ∨ e.e1 = one))
    e = (e.e1 = zero) ?one : zero
  if (e.operator = and)
    e.e2 =  CP(e.e2)
    if (e.e1 = zero ∨ e.e2 = zero ∨ (e.e1 = ¬e.e2))
      e = zero
    else if (e.e1 = one ∧ e.e2 = one)
      e = one
  return e
```

Fig. 1. Pseudocode for the constant propagation

B. Experimental results

Our experiments were conducted using twelve of the largest models from a recent Intel's hardware design. In these models several inputs of the design have a known cyclic pattern. We used a PC with a dual 2.7Ghz Pentium© 4 processor and 4GB memory.

In Table I, we describe experimental results that demonstrate the benefits of constant propagation for bounded model checking. In each example we give execution time and number of clauses at a given bound. If the algorithm succeeds in completing the search at a certain bound, we provide execution time (in seconds). If the algorithm reaches a certain bound but execution time exceeded 10 hours, we mark as *Tout*. If the algorithm reaches a certain bound but failed to complete it because of memory overflow, we mark as *Mout*. On average, the number of clauses sent to the SAT solver has been reduced by a factor of 2.

Consider, for example, the test case $P45$ with $6,219$ state elements (variables). The SAT solver applied to the original BMC formula (without constant propagation) reaches bound 63. When the SAT solver tries to complete bound 63 holding at that point in time with a 10-million-clause formula it requires more than $36,000$ seconds. When the SAT solver was applied to the same BMC problem after simplification by CP - bound 63 was completed after 270 seconds, which is more than hundred times faster and was holding at that point of time only 6 million clauses.

One might argue that if the initial constants are provided as unit clauses to the SAT solver, then the unit clause rule would perform a similar constant propagation with the same effect. We tested this hypothesis, by providing only the initial value $c_0 = 0$ and the toggling constant $c' = \neg c$. It is true that the SAT solver does eventually rediscover the same constant values at different time frames. But it does not have immediate effect of the downstream simplification for the other signals. Not surprisingly, the conclusion was that it still pays off to simplify the BMC formula in advance and not rely on the SAT solver to perform the equivalent of constant propagation.

Table II presents the maximum bound reach by BMC with and without CP. As expected, constant propagation is enabling much deeper bounds. For example, test case $P45$ has reached maximum bound of 63 without CP and bound 94 with CP.

Circuit	#vars	Bound	BMC		BMC-CP		Ratio
			Time	Size	Time	Size	Size
P8	27,201	12	*Tout*	16M	70	6M	38%
P15	5,946	65	*Mout*	9M	4K	5M	53%
P19	6,907	69	*Mout*	11M	5K	6M	55%
P24	5,954	79	*Mout*	11M	14K	6M	54%
P38	6,028	77	*Mout*	11M	13K	6M	54%
P54	6,028	77	*Mout*	11M	8K	6M	54%
P69	5,938	81	*Mout*	11M	13K	6M	54%
P45	6,219	63	*Tout*	10M	270	6M	56%
P37	7,180	71	*Mout*	13M	3K	7M	58%
Pf	1,585	167	*Tout*	9M	8K	5M	53%
Pbb	1,458	54	*Tout*	2M	21K	1M	51%
Pc	1,648	115	*Tout*	7M	2K	3M	45%
Ave							52%

TABLE I
Comparison of the execution time (in seconds) and the number of clauses (Size) used by BMC with and without CP.

III. Abstract BMC

This section presents an iterative model abstraction algorithm combined with BMC, called ABMC. For a given model M, a sequence $M'_1, M'_2, \ldots,$ of abstract models is generated automatically. The property P is checked in each abstract model up to bound t, the target bound. If P holds up to bound t in the abstract model M'_i, the algorithm stops and concludes that P also holds up to bound t in the original model M. If P does not hold in the abstract model M'_i the algorithm proceeds to the next iteration and generates a new abstract model M'_{i+1}.

Circuit	#vars	BMC	BMC-CP	Ratio
P8	27,201	11	38	345%
P15	5,946	64	96	150%
P19	6,907	68	88	129%
P24	5,954	78	100	128%
P38	6,028	76	102	134%
P54	6,028	76	100	132%
P69	5,938	80	102	128%
P45	6,219	62	94	152%
P37	7,180	70	94	134%
Pf	1,585	166	270	163%
Pbb	1,458	53	55	104%
Pc	1,648	114	206	181%
Ave				157%

TABLE II
Comparison of the maximum bound completed with and without CP under the same time (ten hours) and memory budgets.

An abstract model M'_i is automatically generated by running BMC on the original model M up to bound $k+i$, where $k < t$ is an external parameter to the algorithm. If the original model M is large (small), we start with a small (large) k. In more details, in order to generate M'_i, a BMC formula is created on the original model M up to bound $k+i$ and sent to the SAT solver. If the formula is satisfiable, the algorithm stops and returns the satisfying assignment as the counterexample. If the formula is unsatisfiable, then the SAT solver returns the set of clauses that cause unsatisfiability (unSAT core). The algorithm collects the set V' of model variables referred to in the unSAT core. Clearly, $V' \subseteq V$, where V is the the set of variables in the original model M. Using the set of variables V', a new abstract model, M'_i, is constructed consisting of the next-state-functions of all variables in V'. Note that, if the number of variables in V' is not small enough, that is $\frac{|V'|}{|V|} > \delta$, the model M'_i is skipped and the next model M'_{i+1} is generated. A pseudocode of the algorithm in presented in Figure 2.

```
function  ABMC(M, P, t, δ)
 1.  initialize k
 2.  While k < t
 3.    V = concrete model variables
 4.    φ = Build-BMC-formula (V, k)
 5.    if SAT-solver(φ) = SAT return CEX
 6.    V' = variables in unSAT core
 7.    if |V'|/|V| ≤ δ
 8.      φ = Build-BMC-formula(V', t)
 9.      if SAT-solver(φ) = unsat
10.        return Valid up to t
11.    k = k + 1
12.  return Valid up to t
```

Fig. 2. Pseudocode for ABMC deep BMC algorithm

Lines 8-10 in Figure 2 actually oversimplify. In reality, we apply bounded model checking to the abstract model incrementally, trying to reach bound t. If we are unable to complete bound t, we report the largest bound completed.

A similar iterative model abstraction algorithm (BDDs-based) was presented in [18], in which the bound k was increased by one or more at each iteration. The new value of k is determined in [18] to be the next bound where P fails in the abstract model M'_i. Our algorithm ABMC makes more iterations towards its target bound t by always

Cir-cuit	Jump				Step			
	Ite	Abs	Bound	Time	Ite	Abs	Bound	Time
P8	7	194	15	*Mout*	15	1,881	19	*Mout*
P15	3	731	260	*Tout*	11	677	280	*Tout*
P19	2	601	164	*Tout*	3	530	152	*Tout*
P24	3	2,571	194	*Tout*	12	683	270	*Tout*
P38	4	2,657	196	*Tout*	14	2,675	186	*Tout*
P54	4	758	240	*Tout*	12	747	244	*Tout*
P69	4	1,065	190	*Tout*	14	2,635	198	*Tout*
P45	4	2,391	328	*Mout*	10	2,203	368	*Tout*
P37	4	2,889	292	*Mout*	9	2,892	354	*Tout*
Pf	7	1,514	182	*Tout*	44	442	714	*Tout*
Pbb	5	935	29	*Tout*	54	935	29	*Tout*
Pc	8	398	66	*Mout*	49	452	70	*Mout*

TABLE III
The table presents the deepest bound completed using two versions of ABMC: In Jump the bound k is incremented based on the counterexample length. In Step the bound k is incremented by one. Abs is the number of variables in the abstract model. *Mout* means memory overflow. *Tout* means timeout (of ten hours).

Cir-cuit	#vars	BMC-CP		ABMC			Ratio
		Bound	Time	Bound	Time	Abs	
P8	27,201	38	*Mout*	19	*Tout*	1,881	50%
P15	5,946	96	*Mout*	280	*Tout*	677	292%
P19	6,907	88	*Mout*	152	*Tout*	530	173%
P24	5,954	100	*Tout*	270	*Tout*	683	270%
P38	6,028	102	*Tout*	186	*Tout*	2,675	182%
P54	6,028	100	*Mout*	244	*Tout*	747	244%
P69	5,938	102	*Mout*	198	*Tout*	2,635	194%
P45	6,219	94	*Mout*	368	*Tout*	2,203	391%
P37	7,180	94	*Tout*	354	*Tout*	2,892	377%
Pf	1,585	270	*Tout*	714	*Tout*	442	264%
Pbb	1,458	55	*Tout*	29	*Tout*	935	53%
Pc	1,670	206	*Tout*	70	*Mout*	452	34%
Ave							210%

TABLE IV
The table compares the deepest bound completed by BMC-CP to ABMC. Abs is the number of variables in the abstract model. *Mout* means memory overflow. *Tout* means timeout (of ten hours).

increasing k by one, compared to the larger increases in [18]. Nevertheless, our experimental results, presented in Table III, show that in most cases our algorithm reaches a deeper bound in a similar time budget.

In this experiment we used a time budget of $36,000$ seconds, the initial value of k was 2 and δ was defined to be 0.9. There are two reasons to the improvement of 30% in the bounds reached. The first one is that by increasing k in a conservative fashion we end up with smaller abstract models at each iteration and thus each iteration consumes less time. The second reason is that increasing k non-conservatively may cause a large jump in the bound, which then cannot be completed by BMC on the concrete model.

For example, for the circuit Pc in in Table III, larger increases of the bound sets k to 34 after 8 iterations. At this stage, the abstraction includes 398 variables and a counterexample. The shortest counterexample in this abstraction is of length 67 and therefore we can conclude that the concrete model does not include a counterexample of length less than 66. Running BMC on the abstract model results in memory overflow at bound 66. With conservative bound increase we set k to 50 in 49 iterations. The abstraction now includes 452 variables and the shortest counterexample has length 71. (BMC on the concrete model terminates at bound 51 due to memory overflow.)

In Table IV, the deepest bound completed by BMC-CP is compared with the one completed by ABMC. The final abstract model in ABMC is $2-10$ times smaller than the original model. On average the bounds of ABMC are 210% deeper. In examples $P8$, Pbb and Pc, BMC-CP's performance was better than that of ABMC.

Both algorithms BMC-CP and ABMC have been proven to be very successful in reaching deeper bounds than BMC. Yet, the bound reached is not always satisfying, for instance, in the example of the Pc. This provide a strong motivation to combine abstraction and constant propagation, as described in the next section.

IV. Abstraction and Constant Propagation

In this section we present an algorithm, ABMC-CCP, which combines abstraction with constant propagation. We demonstrate that these two approaches can be combined to yield an algorithm that is superior to both BMC-CP and ABMC. We need to overcome the following problem. When the CP algorithm is applied, several variables from the original model completely disappear from the formula submitted to the SAT solver. Therefore, these variables will never be selected to be part of the abstract model M'. Recall, that the variables in M' are defined to be the variables that appear in unSAT core.

We demonstrate this problem with the following example. Assume a model M, with two variables x and c, defined by the following next state functions:

$$M \equiv c' = \neg c; \quad x' = (\neg c \vee \neg x) \wedge (c \vee x).$$

For illustration we build a simple BMC formula with bound 2 and no property:

$$\varphi \equiv \begin{array}{llll} (c_1 & = \neg c_0) & \wedge & (x_1 = (\neg c_0 \vee \neg x_0) \wedge (c_0 \vee x_0)) \quad \wedge \\ (c_2 & = \neg c_1) & \wedge & (x_2 = (\neg c_1 \vee \neg x_1) \wedge (c_1 \vee x_1)) \end{array}$$

If we apply CP using the constant "$c_0 = 0$" on the initial value of c, we end up with:

$$\varphi_{opt} \equiv (x_1 = x_0) \wedge (x_2 = \neg x_1).$$

The abstract model M' is extracted from the unSAT core, and c cannot be part of it. In case a clause from the expression $x_2 = \neg x_1$ is part of the unSAT core, M' will contain the variable x and be defined as: $M' \equiv x' = (\neg c \vee \neg x) \wedge (c \vee x)$. Note that in M', the variable c becomes a free variable and its toggling nature is abstracted away. In other words, since the CP algorithm had eliminated the signal c from φ_{opt} and replaced it by a constant, c may only appear in M' as a free and unconstrained signal. We noticed that such abstractions are often too drastic and cause false negatives. Therefore, we identified the need to generate a conservative abstractions of M.

Definition 1 *[Conservative Abstraction] Assume model M satisfies property P till bound k. A model M' is a conservative abstraction of model M w.r.t. k and P if M' satisfies P till bound k.*

As pointed above CP generates abstractions that are too aggressive. We propose to combined ABMC with a more conservative constant-propagation algorithm, CCP, described below. Figure 3 presents the combined algorithm, ABMC-CCP.

This flow is similar to the one presented in Figure 2 (Section III), except for lines 5 and 10, in which two constant propagation stages are used. In line 10, constant propagation is as in Figure 1 (Section II), while in line 5, we use more conservative constant propagation, described next.

The conservative constant propagation algorithm is presented in Figure 4. The pseudocode refers only to negation and conjunction

```
Function ABMC-CCP(M, P, t, δ)
 1.  initialize k
 2.  while k < t
 3.   V = concrete model variables
 4.   φ = Build-BMC-formula (V, k)
 5.   φ_opt =  CCP(φ)
 6.   if SAT-solver(φ_opt) = SAT return CEX
 7.   V' = variables in unSAT core
 8.   if |V'|/|V| ≤ δ
 9.     φ = Build-BMC-formula(V', t)
10.     φ_opt =  CP(φ)
11.     if SAT-solver(φ_opt) = unsat
12.       return Valid up to bound t
13.   k = k + 1
14.  return Valid up to bound t
```

Fig. 3. Pseudocode for the ABMC-CCP algorithm

but the implementation includes disjunction, exclusive-disjunction, equality, negation and conjunction. A node e in the EG graph includes two additional bits, $e.bit0$ and $e.bit1$. When $e.bit0$ ($e.bit1$) is high it indicates that the subexpression associated with e is evaluated to the constant 0 (1). This is in contrast to the CP algorithm, in which whenever a node in EG is evaluated to a constant we always replace the node e with this constant. Instead here, we only annotate e with the computed constant. Only in two cases (see lines 12 and 13), we perform the actual structural transformation on the EG. That is, if node e is an AND node and if in addition one of its operands is evaluated to 0, for example $e1.bit0$ holds, then we replace e by $e1$.

```
function CCP(e)
 1.  if e.visited return e
 2.  e.visited = true
 3.  if e is a leaf
 4.   if e = 0 (e = 1) set e.bit0 (e.bit1)
 5.   return e
 6.  e.e1 = CCP(e.e1)
 7.  if (e.operator = negation)
 8.   if e1.bit0 set e.bit1, return e
 9.   if e1.bit1 set e.bit0, return e
10.  if (e.operator = and)
11.   e.e2 = CCP(e.e2)
12.   if e1.bit0 return e1
13.   if e2.bit0 return e2
14.   if e1.bit1 ∧ e2.bit1 set e.bit1, return e
15.   if e1 = negation(e2) set e.bit0, return e
```

Fig. 4. Pseudocode for the CCP algorithm

In ABMC-CCP the generated abstract model M'_i is conservative abstraction of M. Intuitively, any sub expression pruned from the formula by CCP can be proved to be unnecessary for ensuring that the abstraction is conservative. For example, if a node e is the conjunction of nodes z and w and in addition w is constantly 0, then the node e can be replaced by the node representing w (becomes allies to w). If e will be included in the unSAT core then e and w will be included in the abstract model. Moreover, in the abstract model, variable z will be a free input, however, due to w being equal to 0 the value of e (recall $e = z \wedge w$) will not be influenced by pruning the logic that drives z.

A detailed proof of the ABMC-CCP algorithm's soundness and completeness is omitted because of space limitation. Intuitively, the soundness and completeness argument is as follows: if we remove line 5 and lines 7-12 (in Figure 2) the algorithm is sound and complete since it performs the classic iterative BMC algorithm in which the bound k is increased by one every iteration and the BMC formula is build for the concrete model M. Including line 5 in the algorithm preserves soundness and completeness since the CCP transformation that generates φ_{opt} from φ guarantees that for every assignment, A, to the variables of φ, "A satisfies φ if and only if A also satisfies φ_{opt}". In other words, CCP only performs a formula rewrite that does not change the course of the BMC algorithm. Including lines 7-12 in the algorithm preserves soundness and completeness since the algorithm terminates in line 12 only if there exists an over approximation of the original model M that satisfies the given property P till bound t.

In Table V, the performance of the new ABMC-CCP algorithm is compared to the performance of ABMC. In most of the cases deeper bounds are achieved. In particular, the length of the bound grow on average by 218%, reaching bound $1,000$ in test case $P45$.

Circuit	#var	ABMC		ABMC-CCP		Ratio
		bound	time	bound	time	
P8	27,201	20	*Mout*	48	*Mout*	240%
P15	5,946	281	*Tout*	512	*Tout*	182%
P19	6,907	153	*Tout*	296	*Tout*	193%
P24	5,954	271	*Tout*	312	*Tout*	115%
P38	6,028	187	*Tout*	300	*Tout*	160%
P54	6,028	245	*Tout*	296	*Tout*	121%
P69	5,938	199	*Tout*	276	*Tout*	139%
P45	6,219	369	*Tout*	1,000	7,869	271%
P37	7,180	355	*Tout*	694	*Tout*	195%
Pf	1,585	715	*Tout*	686	*Tout*	96%
Pbb	1,458	30	*Tout*	61	*Tout*	203%
Pc	1,648	71	*Mout*	498	*Tout*	701%
Ave						218%

TABLE V
Comparison of deepest bound by the ABMC and the ABMC-CCP.

We next look in more details in analyzing ABMC-CCP. Figure 5 compares the number of clauses generated from the BMC formula of the three algorithms. BMC does not use optimizations. ABMC-CPP uses CCP to optimized the BMC formula. Finally, BMC-CP uses CP to optimize the BMC formula. For each bound the number of clauses by each algorithm is given.

BMC-CP generates fewer clauses than BMC-CPP, for all bounds. However, ABMC-CCP can be used for finding abstract models that are much smaller than the concrete models. ABMC-CCP generates fewer clauses than BMC, for all bounds. Therefore, ABMC-CCP is more effective in finding abstract models than BMC without any optimization.

V. Conclusions

Model checking is desired in hardware verification since it provides confident in the correctness of the circuit. However, in many cases model checking is impossible due to complexity and thus bounded model checking is applied. In this paper, we effectively used constrains on inputs of the design that have a known cyclic pattern. We demonstrate the ability to reach bounds nearing 1,000 using proof based abstraction, and in most circuits such a bound already provides

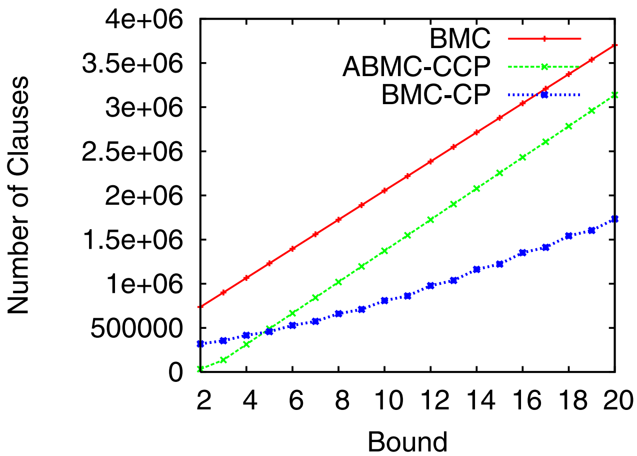

Fig. 5. Number of clauses for each bound in test example $P45$. BMC runs without any optimizations. ABMC-CCP is using CCP. BMC-CP is using CP.

high confidence in the correctness of the circuit due to the fact that circuit diameters are usually smaller than 1,000.

Acknowledgements

We are grateful to Tim Leonard and Abdel Mokkedem for suggesting the constant propagation algorithm.

References

[1] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In *CHARME'05*.

[2] J. Baumgartner, T. Heyman, V. Singhal, and A. Aziz. An Abstraction Algorithm for the Verification of Level-Sensitive Latch-Based Netlists. *FMSD'03*.

[3] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th DAC*, 1999.

[4] P. Bjesse and J. Kukula. Automatic generalized phase abstraction for formal verification. In *ICCAD*, 2005.

[5] M. Browne, E. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. on Computers*, C-35:1035–1044, 1986.

[6] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 1986.

[7] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–171, June 1992.

[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[9] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, Jan. 1983.

[10] M. K. Ganai and A. Aziz. Improved SAT-based Bounded Reachability Analysis. In *ASP-DAC '02*.

[11] A. Gupta, M. Ganai, and P. Ashar. Lazy Constraints and SAT Heuristics for Proof-Based Abstraction. In *VLSI Design*, 2005.

[12] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. Iterative Abstraction using SAT-based BMC with Proof Analysis. In *ICCAD '03*.

[13] A. Gupta and O. Strichman. Abstraction Refinement for Bounded Model Checking. In *CAV*, 2005.

[14] G. Hasteer, A. Mathur, and P. Bannerjee. A framework for equivalence checking of multi-phase FSMs. In *Proc. High Level Design Validation and Test Symp.*, 1997.

[15] A. Kuehlmann. Dynamic Transition Relation Simplification for Bounded Property Checking. In *ICCAD '04*.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust Boolean Reasoning For Equivalence Checking and Functional Property Verification. *TCAD'02*, 21(12).

[17] B. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Series in Computer Science, 1994.

[18] K. L. McMillan and N. Amla. Automatic Abstraction without Counterexamples. In *TACAS*, pages 2–17, 2003.

[19] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. Exploiting suspected redundancy without proving it. In *DAC '05*.

[20] J. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.