IChecker: An Efficient Checker for Inductive Invariants

Feng Lu and K.-T. Cheng
Department of ECE, University of California at Santa Barbara

Abstract

Invariants in sequential circuits could be very useful for sequential optimizations and for speeding up functional verification tasks. However, the lack of efficient and scalable invariant identification tools limits their usage. In this paper, we present a new tool, IChecker, for efficient identification of true invariants for any given initial set of invariant candidates. ICheker uses new circuit simplification techniques to iteratively minimize constrained circuit models, along with a number of heuristics for efficient computation of invariants. Experimental results demonstrate the high efficiency and effectiveness of the proposed approach for identifying sequential invariants.

I. Introduction

Invariants are important for sequential circuit optimization and sequential verification. For example, signals of equivalence invariance can be merged to simplify the sequential circuit. For verification, invariants are the key elements for avoiding state space traversal which is usually impossible for large circuits [1, 2, 3].

A two-timeframe, assume-then-prove circuit model (TTAPCM) was proposed in [1] to calculate equivalence invariants among all signals in a sequential circuit. In the first timeframe of a TTAPCM, candidates of equivalent signals, which are heuristically identified, are assumed equivalent in the first timeframe. The correctness of these equivalences is then verified in the second timeframe. This procedure iterates until a fix-point is reached. The equivalences among signals remaining at the end of this process form the equivalence invariants. SAT-based fix-point calculation was proposed in [2] to compute equivalence invariants on TTAPCM. The authors adopt the Stålmarck method in their SAT engine and incorporated k-induction [7, 8] and unique state induction to achieve complete induction.

Large industrial circuits usually have a large number of invariant candidates which can be derived by random simulation or other techniques. With a large set of candidates, the fix-point calculation to derive invariants would be a time-consuming procedure. These papers did not explain how to perform fix-point calculation efficiently for large designs. In this paper, we describe a highly efficient invariant checker that can work with large designs.

The main contributions of this paper include: (1) proposing new techniques to simplify the constrained circuit models for efficient invariant computation and to avoid adding unnecessary constraints, and (2) heuristics for identifying proper Boolean Satisfiability (SAT) problems for proving invariant candidates, and for determining the order of solving these problems, which can lead to more efficient fix-point calculation.

The rest of the paper is organized as follows: Section II gives the background. In Section III, we first present the techniques to simplify the constrained circuits for computing invariants, and then we discuss and compare several heuristics for proving invariant candidates. Section IV concludes the paper.

II. Background

A synchronous sequential circuit can be modeled by the Huffman circuit model. One timeframe of a sequential circuit is a combinational circuit where the output of each flip-flop is modeled as a pseudo primary input (PPI) and the input of each flip-flop is modeled as a pseudo primary output (PPO). Timeframe expansion is achieved by connecting the PPIs of timeframe $i+1$ to the corresponding PPOs of the previous timeframe i. In this paper, we label the first timeframe as frame 0, the second timeframe as frame 1, and so on . Unless indicated otherwise, the initial state is not applied to the PPIs of frame 0, and these PPIs are treated as primary inputs (PIs).

II-A. Definitions

Given a sequential circuit *SC*, We denote V as the set of its signals.

Given a signal $v \in V$, we use $FI(v)$ to denote the set of its fanins, and $FO(v)$ to denote the set of its fanouts. We levelize the signals in the sequential circuit based on its one-timeframe combinational circuit model. The level of a signal v, denoted as $L(v)$ is computed as follows:
(1). $L(v) = 0$ if $FI(v) = \emptyset$;
(2). $L(v) = 1 + \max((L(i_1), L(i_2), ..., L(i_{|FI(v)|}))$ where i_1, i_2, ..., $i_{|FI(v)|} \in FI(v)$.

For a signal $v \in V$, v^i denotes the corresponding signal of v in timeframe i. We refer to either a signal v or its negation $\overline{v}$ as a literal, v as a positive literal, and $\overline{v}$ as a negative literal. A clause is a logical **OR** of one or more literals.

Based on the above notations, we give the following definitions and corollaries.

(a). A property p involving n signals v_1, v_2 ..., v_n is a Boolean function $p(v_1, v_2, \ldots, v_n)$ where $v_k \in V$ for $1 \leq k \leq n$.

(b). A property p in timeframe i, denoted as p^i, is a Boolean function $p(v_1^i, v_2^i, \ldots, v_n^i)$ where $v_k^i \in V^i$ for $1 \leq k \leq n$.

(c). A set S of properties $p_1, p_2, \ldots, p_{|S|}$, is an inductive property set (IPS) if and only if $(p_1^0 = 1 \wedge p_2^0 = 1 \ldots \wedge p_{|S|}^0 = 1) \Rightarrow (p_1^1 = 1 \wedge p_2^1 = 1 \ldots \wedge p_{|S|}^1 = 1)$. Specially, $\emptyset$ is an IPS. Note that, this definition is based on the two-timeframe model, and the initial state is not applied.

(d). If a set S is an IPS, then any property p of S is called a *guarded invariant* of S, or just an *invariant* for brevity.

Note that, under the above definitions, guarded invariants are associated with an IPS and are of interest for finite state machines (FSMs). That is, an IPS defines a guard condition (i.e. the conjunctive of all the properties in the IPS). Once the FSM enters a state that the guard condition is satisfied, any property

of the IPS should hold for all possible subsequent states reachable by the FSM, and thus becomes an invariant.

The invariant checker, IChecker, proposed in this paper addresses the following problem: Given a set S of properties (called invariant candidates), find the maximal subset which is an IPS.

In our implementation, IChecker accepts three types of invariant candidates:

1. Constant invariant candidates (CstICs): each CstIC is an equivalence function of a signal and a binary value, and is represented by a literal. A positive literal v represents $(v = 1)$, while a negative literal $\overline{v}$ represents $(v = 0)$. All such literals are put in one *constant candidate group*.
2. Equivalence invariant candidates (EICs): each EIC is an equivalence function or inverted equivalence function of two signals. e.g. $(v_1 = v_2)$ and $(v_3 = \overline{v_4})$. Instead of explicitly representing EICs by signal pairs, we represent them in *equivalence candidate groups*. Signals involved in an EIC are placed in the same equivalence candidate group. For example, three EICs $(v_1 = v_2)$, $(v_1 = \overline{v_3})$ and $(v_4 = v_5)$ are represented by two equivalence candidate groups $\{v_1, v_2, \overline{v_3}\}$ and $\{v_4, v_5\}$.
3. Clause invariant candidates (ClsICs): each ClsIC is a clause of one or more literals and is represented by a *clause candidate group* whose member(s) is (are) the literal(s) in the clause.

II-B. Fix-point calculation

Given a set S of invariant candidates, the maximum inductive property set can be computed on a TTAPCM by induction. The first timeframe of TTAPCM is for adding constraints derived from the invariant candidates. These constraints enforce the invariant candidates to be held in the first timeframe. The candidates are then checked in the second timeframe and the false candidates are removed from S. The above procedure iterates until a fix-point is reached. The invariant candidates left in S form the maximum inductive property set. Figure 1 gives an example of TTAPCM for equivalence invariant candidate $(v_1 = v_2)$.

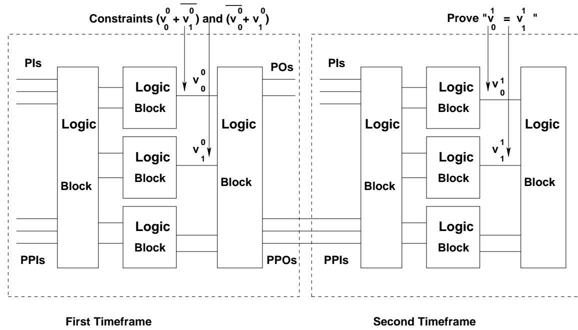


Figure 1: An example of a TTAPCM.

Algorithm II.1 summaries the process, where *SC* denotes the sequential circuit and S the set of invariant candidates. Function $AssumeInvariants(TTC, S)$ enforces the invariant candidates of S to be held in the first timeframe of TTC, and $ProveInvariants(TTC, S)$ check the validity of the invariant candidates of S in the second timeframe of TTC, and remove the false candidates from S. If any false candidate is found in $ProveInvariants(TTC, S)$, it returns false. Otherwise it returns true.

Algorithm II.1: COMPUTEINVARIANTS(*SC*, *S*)

$fixpoint \leftarrow false$
while $(!fixpoint)$
do
$\begin{cases} \text{Create a two-timeframe model } TTC \text{ of } SC \\ AssumeInvariants(TTC, S) \\ fixpoint \leftarrow ProveInvariants(TTC, S) \end{cases}$
return (S)

III. IChecker: Simplifications and Algorithms

III-A. The "assuming" step of invariant candidates

Given a TTAPCM of a sequential circuit, the previous methods directly add constraints to make the invariant candidates hold in the first timeframe [1, 10]. For example, for a constant invariant candidate $(v_1 = 1)$, these methods add an unique literal clause (v_1^0), and for an equivalence invariant candidate $(v_1 = v_2)$, they add two clauses $(v_1^0 + \overline{v_2^0})$ and $(\overline{v_1^0} + v_2^0)$. However, directly adding such constraints as above does not help the simplification of the computation model.

The general signal-merge operation $MERGE(A, B)$ replaces each wire connected to signal B by a wire connected to A and removes the gate producing signal B [9]. Figure 2.b illustrates this merge operation. In [6], the merge operations are used

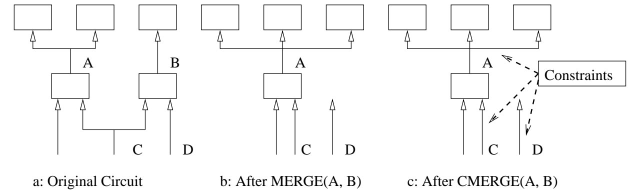


Figure 2: Illustration of MERGE and CMERGE

for equivalent and constant signals to simplify the verification problem, which can achieve significant performance enhancement. However, applying $MERGE()$ to signals that are "assumed" equivalent or constant in the assume-then-prove circuit model could result in loss of the assumed constraints. Figure 3.b shows one such case.

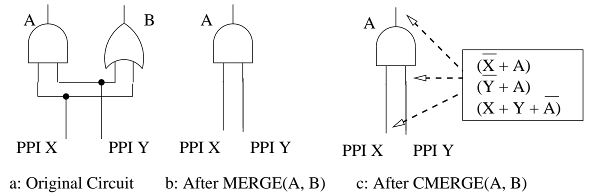


Figure 3: Examples of MERGE and CMERGE

In this case, $(A = B)$ is an equivalence invariant candidate. If A and B are constrained to be equivalent (Figure 3.a), the state space of (X, Y) is constrained to be (0, 0) and (1, 1). Simply merging them by MERGE(A, B) as shown in Figure 3.b will lose the constraints to the state space of (X, Y). This problem has been pointed out and addressed by Mony *et al.* in [11]. In their approach, when signals A and B are constrained to be equivalent, each wire connected to signal B is replaced by a wire connected to A, and constraints are added between A and B to enforce them equivalent. In this paper, to address the problem of loss of constraints, we propose a similar, while improved, approach by introducing a new operation $CMERGE(A, B)$. This operation not only performs the $MERGE(A, B)$ operation, but also adds constraints among A and fanins of B to ensure that A equals to B under these

constraints. Figure 2.c illustrates the concept of $CMERGE()$. The specific constraints added by $CMERGE()$ for the example shown in Figure 3.a are given in Figure 3.c.

The advantage of applying $CMERGE()$ is more than just removing a signal from the computation model. More importantly, similar to the approaches of [9, 11], after applying such operations, a structure-based method [9] can be employed to further simplify the computation model and to avoid adding unnecessary constraints. Figure 4 illustrates this point. In this example, $(A = B)$ and $(C = D)$ are equivalence invariant candidates. After performing the $CMERGE(A,B)$ operation to enforce the equivalence of A and B, as shown in Figure 4.b, we can use a structure-based method to detect that C and D are equivalent. This means that the constraints added to enforce the equivalence of A and B can also enforce the equivalence of C and D. So C and D can be merged by $MERGE(C,D)$ without adding additional constraints. From this example, we can also see the advantages of processing the invariant candidates following the topological order from the inputs toward the outputs. If invariant candidate $(C = D)$ is processed before $(A = B)$, the constraints among C and the inputs of D need to be added, which become redundant after $(A = B)$ is processed.

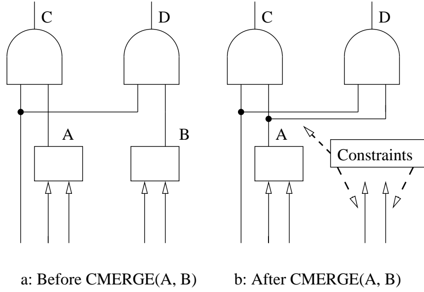


Figure 4: An example: CMERGE(A, B) enables MERGE(C, D)

Note that, in our implementation, the inverters are represented as a wire (edge) property [9]. So the merging of inverted equivalent signals can be done in a similar fashion by modifying the wire property. The $CMERGE()$ operation is implemented to accept **literals** as its parameters, and it can merge signals of inverted equivalence.

To describe the algorithm for enforcing the candidates to be held in the first timeframe of a TTAPCM, we use LA to denote the array of literals of the sequential circuit SC. LA is sorted in ascending order of the signal level, and thus follows the topological order from inputs to outputs in the one-timeframe combinational model of SC. We also assume the literals in the candidate groups are sorted in ascending order by their indexes in LA. TTC is the two timeframe circuit model of SC, and S is the set of invariant candidates. Algorithm III.1 summarizes the $AssumeInvariants()$ process.

This algorithm first handles the constant and equivalence invariant candidates following the **topological order** from inputs to outputs. For a literal v, its corresponding literal in the first timeframe of TTC, v^0, might have been merged by an earlier $CMERGE()$ operation or a structure-based simplification. So we must obtain the literal, u, equivalent to v^0 in TTC, before $CMERGE()$ can be applied. This literal can be obtained by the function $GetUnmergedLiteral(v,n,TTC)$ shown in Algorithm III.2 (Note that in TTC, if a signal s is merged with a signal t, and l is a literal of s, function $GetMerged(l)$ returns the literal of t equivalent to l; otherwise $GetMerged(l)$ returns $NULL$). In this function, n can be either 0 or 1 which indicates the first timeframe or the second timeframe of the TTC respectively, since in each timeframe of the TTC, there is a signal corresponding to v.

Whenever $CMERGE()$ is called, structure-based simplifications are performed on TTC.

Algorithm III.1: ASSUMEINVARIANTS(TTC, S)

$num \leftarrow$ the number of elements of LA
for $i \leftarrow 1$ **to** num
do
- $v \leftarrow LA[i]$
- $u \leftarrow GetUnmergedLiteral(v, 0, TTC)$
- **if** ($v \in$ a constant candidate group)
  - **then** $CMERGE(1,u)$; Simplify TTC by structure based methods
- **else if** ($v \in$ an equivalence candidate group EG)
  - **then** $x \leftarrow$ the first member of EG; $y \leftarrow GetUnmergedLiteral(y, 0, TTC)$; $CMERGE(y,u)$; Simplify TTC by structure based methods

Process clause candidate groups

The process for clause candidate groups is much simpler. For each candidate group, we replace its literals by the corresponding unmerged literals of TTC in the first timeframe, and remove the redundant literals. If the resulting clause does not contain a positive literal and a negative literal of the same signal, the clause is then added to TTC.

Algorithm III.2: GETUNMERGEDLITERAL(v, n, TTC)

$u \leftarrow v^n$ of TTC
while $(GetMerged(u)! = NULL)$
  do $u \leftarrow GetMerged(u)$
return (u)

In our algorithm, since enforcing constraints of the ClsICs will not incur the merge of the signals, we process the ClsICs after processing the CstICs and the EICs. If ClsICs are enforced earlier, it might happen that some signals in the ClsICs are merged with other signals later, and thus causes unnecessary overhead to update the enforced constraints. The CstICs and the EICs are enforced to hold following the topological order of the signals involved in them. Thus, through the CMERGE operation and the structure-based simplification, the TTPACM can be greatly simplified.

The experimental results that compare our simplification techniques with the original techniques [1, 10] is in Table I

TABLE I: RESULTS OF THE ENHANCED SIMPLIFICATION TECHNIQUES

Circuit	#ff	#const	#eq	original method		our method	
				#g	#c	#g	#c
case1	128	248	490	10061	888	8668	86
case2	273	1160	1263	14591	2652	8962	297
case3	362	451	1158	5121	1991	2694	407
case4	410	8900	36513	107051	72766	23453	900
case5	414	8870	36833	108471	73378	24471	951
s5378	301	894	1645	6297	3080	2682	242
s13207	553	1428	1506	7191	3658	2297	349
s35932	3777	1784	23265	59069	41888	15431	2473
s38417	2892	8490	11496	46059	27924	10026	496
s38584	2179	2423	12550	35757	18859	15468	1855

In these experiments, the "case" benchmarks are the miter circuits created from industrial examples. In each of the miter circuits, one sub-circuit is a gate-level implementation synthesized from its System-C behavioral-level description, and the other one is a gate-level circuit synthesized from its Verilog register-transfer-level (RTL) description. For some of these industrial cases, bugs are injected in the Verilog RTL description. The "s" benchmarks are the miter circuits created from the ISCAS-89 sequential benchmark circuits. For each of them, one sub-circuit is the original gate-level benchmark circuit; the other one is its retimed version produced by ABC package [12] from Berkeley.

All the miter circuits have been simplified based on both detection of combinational equivalent or constant signals and

flip-flop mapping. Note that, a miter circuits whose output is merged to a constant signal after the simplification is not used in our experiments. All our experiments in this paper were run on a P4 2-GHz Linux machine with 2-GB memory. The initial invariant candidates are generated based on the constant or equivalent signals in the first timeframe of the circuit with respect to the given initial state, and are refined by random simulation. Column "#ff" is the number of flip-flops in the sequential circuit. Column "#const" ("#eq") gives the number of signals in the initial constant (equivalence) invariant candidates. The two columns "#g" and "#c" show the numbers of gates and added constraints in the TTAPCMs of the original method and our method, respectively. The results clearly indicate that the TTAPCM can be greatly simplified in the new method.

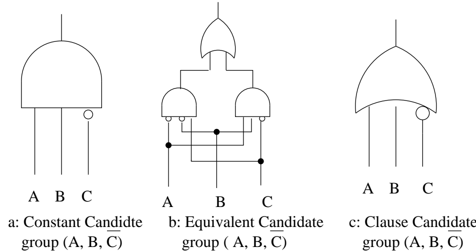
III-B. The "proving" step of invariant candidates

In the previous subsection, we have described the assumption part of the inductive method to derive the invariants. In this section, we discuss and compare two methods for proving the invariant candidates under the assumptions. As described in Section II, the invariant candidates are represented as candidate groups. The main issues to address are (1) how to formulate checking problems from the candidate groups and (2) in what order to solve these problems.

In the first method, for each candidate group, a Boolean Satisfiability problem [4] is formulated by adding some extra gates to a TTAPCM. For a constant candidate group, we connect all literals in the group to a logic **AND** gate g, and check whether the output of g can be set to 0. For an equivalence candidate group, we connect all its literals to a logic **AND** gate $g1$, connect all its inverted literals to logic **AND** gate $g2$, and finally connect the outputs of $g1$ and $g2$ to an **OR** gate $g3$. The problem is to check whether the output of $g3$ can be set to 0. For a clause candidate group, we simply connect its literals to an **OR** gate g. The problem is to check whether the output of g can be set to 0.

Figure 5 illustrates the corresponding Boolean Satisfiability problems for the three types of candidate groups.

In the above description of formulating Boolean Satisfiability problems, for the purpose of conciseness, we have referred to the literals in the candidate groups without explicitly mentioning the timeframe. These literals actually correspond to the signals in the second timeframe of the TTAPCM.



a: Constant Candidte group (A, B, $\overline{C}$) b: Equivalent Candidate group (A, B, $\overline{C}$) c: Clause Candidate group (A, B, $\overline{C}$)

Figure 5: Illustration of problem generation for different candidate groups

The advantage of formulating the SAT problems in these ways is considerable. If the SAT problem generated for proving invariant candidates in a candidate group is proved unsatisfiable, all the invariant candidates represented by the group would hold. This applies to both constant candidate groups and equivalence candidate groups. This is particularly powerful for equivalence candidate groups, since an equivalence candidate group with x elements can represent $x*(x-1)/2$ equivalence invariant candidate pairs. If the SAT problem generated for a clause candidate group is satisfiable, the candidate group can be removed. If the SAT problem of a constant or equivalence candidate group is satisfiable, based on the value assignments in its solution, the candidate group can be partitioned into two smaller candidate groups for which new problems can be formulated. Please note that, for a constant candidate group, it can be partitioned into a smaller constant candidate group and an equivalence candidate group of the same or smaller size. For example, in the solution of the SAT problem for a constant candidate group, if all literals in the group are assigned to 0, then none of the signals can be constant invariant, while they still could be equivalent.

The algorithm based on this method is summarized in Algorithm III.3. In this algorithm, TTC is the two timeframe circuit model, S is the set of invariant candidates, and $GLIST$ is the list of candidate groups. If the total number of literals in the initial constant candidate group and the equivalence candidate groups is N, and the number of clause candidate groups is M, the number of generated SAT problems will not be larger than $N+M$. The order of the problems to be solved is determined by the order of their corresponding candidate groups in $GLIST$. The initial candidate groups provided to IChecker are placed in the beginning of $GLIST$ and the new candidate groups obtained by partitioning of the initial groups are appended at the end of $GLIST$ when the groups are generated.

The disadvantages of this method from our experience are related to size and complexity. For large circuits, a constant or equivalence candidate group could have hundreds (even thousands) of literals, for which the generated SAT problems could be difficult to solve. In addition, since the SAT problems for such candidate groups usually involve many signals, it is difficult to determine a good order to solve these SAT problems so that the information learned in solving earlier problems could help make the subsequent problems easier be solved.

Algorithm III.3: PROVEINVARIANTS_IMP1(TTC, S)

```
isfixed ← true
while (GLIST is not empty)
  do { g ← the first of element of GLIST
       Generate SAT problem from g and solve it.
       if (the problem is satisfiable)
         then { isfixed ← false
                if (g is not a clause candidate group)
                  then { Partition g into g1 and g2
                         Add g1 and g2 to GLIST
return (isfixed)
```

Based on the above observations, we propose an improved method to generate SAT problems in a finer granularity. In this improved method, each time a SAT problem is generated for an invariant candidate, instead of an invariant candidate group. The order for the SAT problems to be solved is based on the topological order (from inputs to outputs) of the literals in the corresponding invariant candidates. Invariant candidates can be proved in any order. However, following the topological order could improve the reuse of learned clauses as described in [5].

The improved algorithm is shown in Algorithm III.4. We use LA to denote the array of literals of the sequential circuit, and it is sorted by the signal level as in Algorithm III.1. We also assume the literals in the candidate groups are sorted in ascending order by their indexes in LA. $LC(l)$ is the list of clause candidate groups in which literal l is the member with the largest index in LA. TTC is the two timeframe circuit model, and S is the set of invariant candidates.

In this algorithm, when a SAT problem for an invariant candidate is found satisfiable, we perform solution-based random simulation [6] on TTC. That is, for inputs (including primary inputs and pseudo primary inputs) assigned with a binary value in a solution, the assigned values are used, and for unassigned

inputs, random values are assigned to form a vector for simulation. Note that, because all the constraints are considered in the SAT problems for verifying invariant candidates, the value assignments in the solutions must have satisfied all the constraint. Therefore, the vectors used for simulation will not violate the constraints.

Based on the simulation results, invariant candidate groups are refined. A clause candidate group, if the corresponding clause is not satisfied by the simulation results, will be removed. A constant candidate group or an equivalence candidate group, based on the simulation results, will be partitioned into subgroups. In this way, the number of generated SAT problems will be no more than $N+M$, where N is the total number of literals in the initial constant candidate group and the equivalence candidate groups, and M is the number of initial clause candidate groups before the algorithm is applied. Also in this algorithm, TTC is simplified whenever signals are proved to be constant, equivalent, or inverted equivalent.

Algorithm III.4: PROVEINVARIANTS_IMP2(TTC, S)

$isfixed \leftarrow true$
num $\leftarrow$ the number of elements of *LA*
for $i \leftarrow 1$ **to** *num*
do
- $v \leftarrow LA[i]$
- **if** ($v \in$ a constant candidate group)
  - **then**
    - Solving SAT problem $(v=0)$ with all the constraints
    - **if** (the problem is satisfiable)
      - **then**
        - $isfixed \leftarrow false$
        - Perform solution-based simulation on TTC
        - Refine candidate groups by simulation results
    - **else** Simplify TTC based on the constant signal
- **else if** ($v \in$ an equivalence candidate group EG)
  - **then**
    - $u \leftarrow$ the first member of EG
    - Solving SAT problem $(v \neq u)$ with all the constraints
    - **if** (the problem is satisfiable)
      - **then**
        - $isfixed \leftarrow false$
        - Perform solution based simulation on TTC
        - Refine candidate groups by simulation results
      - **else** {Simplify TTC based on the equivalent signals
- **for each** Clause candidate group*cg* of $LC(v)$
  - **do**
    - Generate the clause c of cg
    - Solving SAT problem $(c=0)$ with all the constraints
    - **if** (the problem is satisfiable)
      - **then**
        - $isfixed \leftarrow false$
        - Perform solution based simulation on TTC
        - Refine candidate groups by simulation results

return ($isfixed$)

Note that this algorithm is based on the explicit learning heuristic used in C-SAT [5], and is similar to the SAT-sweeping algorithm in [6]. The difference is that in SAT-sweeping, the checking of equivalent candidate pairs may not follow the topological order as in our algorithm. Given two equivalent invariant candidates $(s_1 = t_1)$ and $(s_2 = t_2)$, assume $IND(s_1) > IND(t_1)$ and $IND(s_2) > IND(t_2)$, where $IND(v)$ denotes the index of literal v in *LA*. It can be shown from our algorithm that $(s_1 = t_1)$ will be checked prior to $(s_2 = t_2)$ if $(IND(s_1) < IND(s_2)) \vee ((IND(s_1) = IND(s_2)) \wedge (IND(t_1) < IND(t_2)))$.

The experimental results of comparing the two methods are given in Table II. In our implementation, IChecker is based on a circuit-SAT solver C-SAT [5]. The benchmarks and the initial invariant candidates used in these experiments are the same as those listed in Table I. All our experiments were run on P4 2GHz Linux machines with 2 GB memory. Column "#ff" ("#g") is the number of flip-flops (gates) of the sequential circuit. Column "#const" ("#eq") gives the number of signals in the final constant (equivalence) invariant. Column "#ite" shows the number of iterations needed to reach the fix-point. The CPU times (in seconds) of the two methods are shown in Column "III.3" and Column "III.4" respectively. Column "status" shows if the output of the miter circuit is a constant invariant.

The results demonstrate that IChecker can efficiently compute invariants for large circuits. For the second method, all equivalence invariants and constant invariants in these circuits can be computed within one hundred seconds.

TABLE II: EXPERIMENTAL RESULTS TO COMPARE THE TWO METHODS

Circuit	#ff	#g	#const	#eq	#ite	III.3 (s)	III.4 (s)	status
case1	128	5032	74	434	19	56	2	NO
case2	273	7297	145	1182	90	544	17	NO
case3	362	2596	64	205	18	9	2	NO
case4	410	53528	8895	36429	4	2504	20	NO
case5	414	54237	8865	36749	4	2808	24	NO
s5378	301	3150	470	1977	17	48	2	YES
s13207	553	3597	249	2587	50	108	5	YES
s35932	3777	29536	1279	20760	24	2437	30	YES
s38417	2892	23031	211	19325	30	2741	28	YES
s38584	2179	17880	572	13902	19	1086	19	YES

IV. Conclusions

In this paper, we have provided efficient algorithms to derive invariants from given invariant candidates. In the algorithms, new rules are applied to simplify the circuit models when constraints are enforced, and to avoid adding unnecessary constraints. We also propose several heuristics to formulate proper SAT problems for efficient and effective verification of invariant candidates. The experimental results clearly demonstrate the high efficiency of our invariant checker.

References

[1] C.A.J. van Eijk, Sequential Equivalence Checking Based on Structural Similarities. *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 814-819, Jul. 2000.

[2] P. Bjesse and K. Claessen, SAT-Based Verification without State Space Traversal. *FMCAD 2000, LNCS 1954*, pp. 372-389, 2000.

[3] S.-Y. Huang, K.-T. Cheng and K.-C. Chen, AQUILA: An Equivalence Checking System for Large Sequential Designs. *IEEE Trans. Computers*, vol. 49, no. 5, pp. 443-463, May 2000.

[4] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: Engineering an efficient SAT solver. *Proc. of ACM/IEEE Design Automation Conf.*, pp. 530-535, June 2001.

[5] F. Lu, L-C. Wang, K.-T. Cheng, and Ric C-Y Huang, A circuit SAT solver with signal correlation guided learning. *Proc. European Design and Test Conf.* pp. 892-897, Mar. 2003.

[6] A. Kuehlmann, Dynamic Transition Relation Simplification for Bounded Property Checking. *Proc. Int. Conf. Computer-Aided Design*, pp. 50-57, Nov. 2004.

[7] M. Sheeran, S. Singh and G. Stålmarck, Checking Safety Properties Using Induction and a SAT-Solver. *Proc. of Int. Conf. Formal Methods in Computer-Aided Design*, pp. 108-125, 2000.

[8] L. D. Moura, H. Rueβ, M. Sorea, Bounded Model Checking and Induction: From Refutation to Verification *Proc. of Int. Conf. Computer-Aided Verifi cation*, Vol 2725, LNSC, pp. 14-26, 2003.

[9] A. Kuehlmann, M. Ganai, and V. Paruthi, Circuit-based Boolean Reasoning. *Proc. of ACM/IEEE Design Automation Conf.*, pp. 232-237 June 2001.

[10] F. Lu, and K.-T. Cheng, Sequential Equivalence Checking Based on K-th Invariants and Circuit SAT Solving. *Proc. of IEEE HLDVT Workshop*, pp. 45-52 Nov. 2005.

[11] H. Mony, J. Baumgartner and A. Aziz, Exploiting Constraints in Transformation-Based Verification. *Proc. of CHARME*, pp. 269-284, 2005.

[12] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/∼alanmi/abc/, Feb, 2006.