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 which can be derived from simulation or other techniques. IChecker 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, 3]. Combinational flip-flop mapping (also called latch mapping) [2, 4] is a technique for detecting the equivalence invariants among flip-flops. If proper flip-flop mapping could be derived, a sequential equivalence problem could be transformed into a combinational problem. Flip-flop mapping can be computed efficiently based on a one-timeframe circuit model of a sequential circuit. However, for retimed and sequentially-optimized circuits, we often cannot find enough combinational flip-flop mappings for efficient equivalence checking. Sequential Automatic Test Pattern Generation (ATPG) techniques [6, 7] are used in [5] to identify sequentially equivalent flip-flops, for which more flip-flop mappings can be derived than could be identified by pure combinational methods. In [10], invariants regardless of initial states were detected in performing bounded model checking and used to simplify transition relations.

A two-timeframe, assume-then-prove circuit model (TTAPCM) was proposed in [1] to calculate equivalence invariants among all signals in a sequential circuit. The TTAPCM is built by unrolling the transition relation of the sequential circuit twice. In the first timeframe of a TTAPCM, candidates of equivalent signals, which are heuristically identified, are assumed equivalent, and their equivalences are modeled 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 [3] to compute equivalence invariants on TTAPCM. The authors adopt the Stålmarck method in their SAT engine and incorporated k-induction [11, 12] and unique state induction to achieve complete induction. A tool, named IProver, was developed to identify k-th invariants that hold after k cycles [14].

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 shows the experimental results and Section V 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).

The Huffman circuit model and the circuit models for one timeframe and two timeframes are illustrated in Figure 1.a, Figure 1.b and Figure 1.c respectively.
II-A. Definitions

Given a sequential circuit \( SC \), we denote \( PI, PO, FF, PPI \) and \( PPO \) as the sets of its primary inputs, primary outputs, flip-flops, pseudo primary inputs, and pseudo primary outputs, respectively. In addition, \( INT \) denotes the set of internal signals, and \( V = PI \cup INT \cup PPI \cup PPO \cup PO \) denotes the set of all signals of \( SC \).

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 \( v \in V \), \( v' \) denotes the corresponding signal of \( v \) in timeframe \( i \). Similarly, \( PI' \) represents the set of primary inputs in timeframe \( i \), \( V' \) the set of all signals in timeframe \( i \), etc.

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 \( p(v_1, v_2, ..., v_n) \) 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 \( p(v'_1, v'_2, ..., v'_n) \) where \( v'_k \in V' \) for \( 1 \leq k \leq n \).

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

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

**Corollary 1.** If both \( A \) and \( B \) are IPSs, \( A \cup B \) is an IPS too.

The proof can be derived directly from the definition.

**Corollary 2.** For any finite set \( S \) of properties, there is one and only one maximal subset \( TS \) such that \( TS \) is an IPS.

Proof: Since \( \emptyset \) is an IPS and \( S \) has a finite number of elements, there exists at least one maximal subset that is an IPS. Assume there are two different maximal subsets, \( T \) and \( U \), which are IPSs, according to Corollary 1, \( T \cup U \) is an IPS and is larger than \( T \) and \( U \), which violates the assumption that \( T \) and \( U \) are maximal.

Note that, under the above definitions, inductive 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.

For example, consider a sequential circuit \( S \) with two state variables, \( x \) and \( y \), whose state transition graph is shown in Figure 2. Assume \( u \) and \( v \) are two signal of \( S \) where \( u = (x \text{ AND } y) \) and \( v = (x \text{ OR } y) \). It can be proven that the property set \( \{u = 0, v = 1\} \) is an IPS. This can be easily proved that \( (u = 0) \land (v = 1) \) is true if and only if the state of \( S \) is \( (01) \) or \( (10) \). From Figure 2, it can be seen that when FSM \( S \) is in state \( (01) \) or \( (10) \), all its future reachable states are \( (01) \) and \( (10) \).

---

**II-B. Problem Formulation**

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 \( (\overline{v_1} = \overline{v_2}) \). 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 \( (v_1 = v_2), (v_1 = \overline{v_2}) \) and \( (v_4 = v_5) \) are represented by two equivalence candidate groups \( \{v_1, v_2, v_4, v_5\} \) and \( \{v_3, v_5\} \).

3. Clause invariant candidates (ClsICs): each ClsIC is a clause of one or more literals and is represented by a
invariants is associate with an IPS, and is independent of the initial state. We distinguish these special cases from the general clause invariant candidates because they can be used to simplify the computation model and can be computed more efficiently.

II-C. 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 3 gives an example of TTAPCM for equivalence invariant candidate $(v_1 = v_2)$.

Figure 3: 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)**

```plaintext
fixpoint ← false
while (!fixpoint)
  Create a two-timeframe model TTC of SC
  (AssumeInvariants(TTC, S)
  fixpoint ← ProveInvariants(TTC, S)
return (S)
```

II-D. Discussion and Remarks

Unlike the traditional invariants which are defined with respect to an initial state [1, 3], our definition of inductive invariants is associate with an IPS, and is independent of the initial state. In fact, a traditional invariant is a property that holds under the reachable-state set of the initial state. On the other hand, an inductive invariant defined here is a property which holds after the guard function of the IPS is satisfied, and may not hold before the guard function is satisfied. Consider the example illustrated in Figure 2. Assuming that the initial state is $(0, 0)$, the property $(u = 0)$ is not a traditional invariant, but is an inductive invariant with respect to the IPS $\{u = 0, v = 1\}$.

Considering the characteristic function $CF$ of the set of reachable states from the initial state, $CF(s) = 1$ if $s$ is a reachable state, otherwise $CF(s) = 0$. It can be seen that the set $\{CF\}$ is an IPS. Also, for any traditional invariant $P$, the set $\{CF, P\}$ is an IPS. Thus inductive invariants contain the traditional invariants. However, $CF$ usually can not be represented explicitly for a large sequential circuit. This results in an interesting problem: how to identify a good set of initial invariant candidates so that the IPS derived from the initial set is an over-approximation of the reachable-state set and the approximation is accurate enough to determine that $P$ is an invariant. This candidate-identification problem itself is interesting and important, but is not the focus of this paper. In this paper, we assume the initial invariant candidates are already given (for example, derived from random simulation).

It can also be shown that if all the properties in a set hold at the $k$-th cycle from the initial state, the properties in the derived IPS would be $k$-th invariants described in [14]. Unlike the traditional invariants that hold for all cycles, $k$-th invariants guarantee to hold only after the $k$-th cycle from the initial state.

In [3], a $k$-step ($k > 1$) induction is employed to detect the traditional invariants. In the $k$-step induction, a property is first checked if it holds in the first $k$ cycles from the initial state. The property is then verified in the $k + 1$-th timeframe while assuming it holds in the first $k$ timeframes (without applying the initial state). For the same initial invariant candidates, the $k$-step induction can detect more invariants than the 1-step induction. On the other hand, the 1-step induction can process more initial invariant candidates and thus can derive a more accurate over-approximation of the reachable-state set. One simple method of combining the 1-step and the $k$-step induction is through circuit modeling. For example, we can build a new circuit which is the $k$-timeframe expansion of the original circuit, and perform the 1-step induction on the new circuit. This approach actually requires an expansion of $2k$ timeframes of the original circuit, in contrast to an expansion of $k + 1$ timeframes in the $k$-step induction. While the computational complexity is increased, this hybrid approach can identify invariants among signals in different timeframes of the original circuit.

III. IChecker: Model 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, 14]. 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 $(v_1^0 + v_2^0)$ and $(v_1^0 + v_2^0)$. However, directly adding such constraints as above
does not help the simplification of the computation model.

The general signal-merge operation $\text{MERGE}(A, B)$ replaces each wire connected to signal $B$ by a wire connected to $A$ and removes the gate producing signal $B$ [13]. Figure 4 illustrates this merge operation. In [10], the merge operations are used for equivalent and constant signals to simplify the verification problem, which can achieve significant performance enhancement. However, applying $\text{MERGE}(\cdot)$ to signals that are “assumed” equivalent or constant in the assume-then-prove circuit model could result in loss of the assumed constraints. Figure 5 shows one such case.

![Figure 4: Illustration of $\text{MERGE}(A, B)$](image)

![Figure 5: Loss of constraints due to $\text{MERGE}(A, B)$](image)

In this case, $(A = B)$ is an equivalence invariant candidate. If $A$ and $B$ are constrained to be equivalent (Figure 5.a), the state space of $(X, Y)$ is constrained to be $(0, 0)$ and $(1, 1)$. Simply merging them by $\text{MERGE}(A, B)$ as shown in Figure 5.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 [15]. 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 $\text{CMERGE}(A, B)$. This operation not only performs the $\text{MERGE}(A, B)$ operation, but also adds constraints among $A$ and fanins of $B$ to ensure that $A$ equals to $B$ under these constraints. Figure 6 illustrates the concept of $\text{CMERGE}(\cdot)$. The specific constraints added by $\text{CMERGE}(\cdot)$ for the example shown in Figure 5 are given in Figure 7.

The advantage of applying $\text{CMERGE}(\cdot)$ is more than just removing a signal from the computation model. More importantly, similar to the approaches of [13, 15], after applying such operations, a structure-based method [13] can be employed to further simplify the computation model and to avoid adding unnecessary constraints. Figure 8 illustrates this point. In this example, $(A = B)$ and $(C = D)$ are equivalence invariant candidates. After performing the $\text{CMERGE}(A, B)$ operation to enforce the equivalence of $A$ and $B$, as shown in Figure 8.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 $\text{MERGE}(C, D)$ 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.

Note that, in our implementation, the inverters are represented as a wire (edge) property [13]. So the merging of inverted equivalent signals can be done in a similar fashion by modifying the wire property. The $\text{CMERGE}(\cdot)$ 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. Algo-
Algorithm III.1 summarizes the \textit{AssumeInvariants(\(TTC\))} process.

This algorithm first handles the constant and equivalence invariant candidates following the \textit{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 \textit{CMERGE(\(\cdot\))} operation or a structure-based simplification. So we must obtain the literal, \(u\), equivalent to \(v^0\) in \(TTC\), before \textit{CMERGE(\(\cdot\))} can be applied. This literal can be obtained by the function \(\text{GetUnmergedLiteral}(v, n, TTC)\) 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 \(\text{GetMerged}(l)\) returns the literal of \(t\) equivalent to \(l\); otherwise \(\text{GetMerged}(l)\) 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 \textit{CMERGE(\(\cdot\))} is called, structure-based simplifications are performed on \(TTC\).

\begin{algorithm}[h]
\caption{AssumeInvariants(\(TTC\), \(S\))}
\begin{algorithmic}
  \State \textbf{num} $\leftarrow$ the number of elements of \(LA\)
  \For{\(i = 1\) to \textbf{num}}
    \State \(v \leftarrow LA[i]\
    \State u $\leftarrow$ \text{GetUnmergedLiteral}(v, 0, TTC)
    \If{\((v \in \) a constant candidate group)}
      \State \text{CMERGE}(1, u)
      \State \text{Simplify TTC by structure based methods}
    \Else
      \If{\((v \in \) an equivalence candidate group \(EG\))}
        \State \(x \leftarrow \) the first member of \(EG\)
        \State \(v \leftarrow \text{GetUnmergedLiteral}(x, 0, TTC)\)
        \State \text{CMERGE}(v, u)
        \State \text{Simplify TTC by structure based methods}
      \EndIf
    \EndIf
  \EndFor
  \State \text{Process clause candidate groups}
\end{algorithmic}
\end{algorithm}

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 \(TTC\).

\begin{algorithm}[h]
\caption{GetUnmergedLiteral(\(v\), \(n\), \(TTC\))}
\begin{algorithmic}
  \State \textbf{u} $\leftarrow v^0$ of \(TTC\)
  \While{(\text{GetMerged}(\textbf{u}) \neq \text{NULL})}
    \State \text{do} $\leftarrow \text{GetMerged}(\textbf{u})$
  \EndWhile
  \State \text{return} (\textbf{u})
\end{algorithmic}
\end{algorithm}

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 TTPACMs can be greatly simplified.

The experimental results that compare our simplification techniques with the original techniques [1, 14] is in Table I. 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 [16] 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.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#ff</th>
<th>#const</th>
<th>#eq</th>
<th>original method</th>
<th>our method</th>
<th>our method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>#g</td>
<td>#c</td>
<td>#g</td>
</tr>
<tr>
<td>case1</td>
<td>128</td>
<td>248</td>
<td>390</td>
<td>10061</td>
<td>888</td>
<td>8668</td>
</tr>
<tr>
<td>case2</td>
<td>273</td>
<td>1160</td>
<td>1263</td>
<td>14591</td>
<td>2652</td>
<td>8962</td>
</tr>
<tr>
<td>case3</td>
<td>179</td>
<td>620</td>
<td>209</td>
<td>4421</td>
<td>864</td>
<td>2235</td>
</tr>
<tr>
<td>case4</td>
<td>116</td>
<td>35</td>
<td>290</td>
<td>5075</td>
<td>333</td>
<td>3858</td>
</tr>
<tr>
<td>case5</td>
<td>362</td>
<td>451</td>
<td>1158</td>
<td>5121</td>
<td>1391</td>
<td>2694</td>
</tr>
<tr>
<td>case6</td>
<td>410</td>
<td>8900</td>
<td>36513</td>
<td>107051</td>
<td>72766</td>
<td>23453</td>
</tr>
<tr>
<td>case7</td>
<td>414</td>
<td>8870</td>
<td>36833</td>
<td>108471</td>
<td>73378</td>
<td>24471</td>
</tr>
<tr>
<td>s5738</td>
<td>301</td>
<td>894</td>
<td>1645</td>
<td>6297</td>
<td>3080</td>
<td>2682</td>
</tr>
<tr>
<td>s13207</td>
<td>553</td>
<td>1428</td>
<td>1506</td>
<td>7191</td>
<td>3658</td>
<td>2297</td>
</tr>
<tr>
<td>s35932</td>
<td>3777</td>
<td>1784</td>
<td>23265</td>
<td>59069</td>
<td>41888</td>
<td>15431</td>
</tr>
<tr>
<td>s38417</td>
<td>2892</td>
<td>8490</td>
<td>11496</td>
<td>46059</td>
<td>27924</td>
<td>10026</td>
</tr>
<tr>
<td>s38584</td>
<td>2179</td>
<td>2423</td>
<td>12550</td>
<td>35757</td>
<td>18859</td>
<td>15468</td>
</tr>
</tbody>
</table>

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 [8] 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 \(g_1\), connect all its inverted literals to logic AND gate \(g_2\), and finally connect the outputs of \(g_1\) and \(g_2\) to an OR gate \(g_3\). 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 9 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.

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: PROVEINVARIANTSJM1(TTC, S)

```plaintext
isfixed ← true
while (GLIST is not empty)
  g ← the first element of GLIST
  Generate SAT problem from g and solve it.
  if (the problem is satisfiable)
    do
      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 [9].

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. TTCC(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.

Algorithm III.4: PROVEINVARIANTSJM2(TTC, S)

```plaintext
isfixed ← true
num ← the number of elements of LA
for i ← 1 to num
  v ← LA[i]
  if (v ∈ a constant candidate group)
    if (the problem is satisfiable)
      then (isfixed ← false
      then Perform solution-based simulation on TTC
      Refine candidate groups by simulation results
    else Simplify TTC based on the constant signal
  else if (v ∈ an equivalence candidate group EG)
    then (isfixed ← false
    then 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)
    Generate the clause c of cg
    if (the problem is satisfiable)
      then (isfixed ← false
      then Perform solution-based simulation on TTC
      Refine candidate groups by simulation results
    else Perform solution-based simulation on TTC
  return (isfixed)
```
In this algorithm, when a SAT problem for an invariant candidate is found satisfiable, we perform solution-based random simulation [10] 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. Note that without the simulation-based refinement described above, for an equivalence candidate group with $x$ members, the number of generated SAT problems for verifying all equivalence invariant candidate pairs in the group could reach $x \times (x - 1)/2$ in the worst case. Also in this algorithm, TTC is simplified whenever signals are proved to be constant, equivalent, or inverted equivalent.

Note that this algorithm is based on the explicit learning heuristic used in C-SAT [9], and is similar to the SAT-sweeping algorithm in [10]. 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 $(IND(s_1) < IND(s_2)) \lor ((IND(s_1) = IND(s_2)) \land (IND(t_1) < IND(t_2)))$.

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 [9]. 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 “impl1” and Column “impl2” 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>
<thead>
<tr>
<th>Circuit</th>
<th>#ff</th>
<th>#g</th>
<th>#const</th>
<th>#eq</th>
<th>#ite</th>
<th>impl1 (s)</th>
<th>impl2 (s)</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>case1</td>
<td>128</td>
<td>5032</td>
<td>74</td>
<td>434</td>
<td>19</td>
<td>56</td>
<td>2</td>
<td>NO</td>
</tr>
<tr>
<td>case2</td>
<td>273</td>
<td>7297</td>
<td>145</td>
<td>1182</td>
<td>90</td>
<td>544</td>
<td>17</td>
<td>NO</td>
</tr>
<tr>
<td>case3</td>
<td>179</td>
<td>2216</td>
<td>50</td>
<td>368</td>
<td>90</td>
<td>41</td>
<td>5</td>
<td>NO</td>
</tr>
<tr>
<td>case4</td>
<td>116</td>
<td>2539</td>
<td>35</td>
<td>286</td>
<td>2</td>
<td>3417</td>
<td>8</td>
<td>YES</td>
</tr>
<tr>
<td>case5</td>
<td>362</td>
<td>2596</td>
<td>64</td>
<td>205</td>
<td>18</td>
<td>9</td>
<td>2</td>
<td>NO</td>
</tr>
<tr>
<td>case6</td>
<td>410</td>
<td>53528</td>
<td>8895</td>
<td>36429</td>
<td>4</td>
<td>2504</td>
<td>20</td>
<td>NO</td>
</tr>
<tr>
<td>case7</td>
<td>3578</td>
<td>3150</td>
<td>470</td>
<td>1977</td>
<td>17</td>
<td>48</td>
<td>2</td>
<td>YES</td>
</tr>
<tr>
<td>case8</td>
<td>533</td>
<td>553</td>
<td>249</td>
<td>2587</td>
<td>50</td>
<td>108</td>
<td>5</td>
<td>YES</td>
</tr>
<tr>
<td>case9</td>
<td>35932</td>
<td>3777</td>
<td>1279</td>
<td>20760</td>
<td>24</td>
<td>2437</td>
<td>30</td>
<td>YES</td>
</tr>
<tr>
<td>case10</td>
<td>286</td>
<td>24</td>
<td>2</td>
<td>NO</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IV. Experimental Results

In this section, we demonstrate the stability of our invariant checker IChecker, and compare its performance with that of IProver [14]. IProver is an invariant checker proposed to compute $k$-th invariants [14]. It can also be used to compute inductive invariants if the initial invariant candidates contain only equivalence or constant invariant candidates (i.e. without clause invariant candidates). However, IProver does not have the optimization techniques discussed in sub-section III-A in the assuming step, and the algorithm in the proving step of IProver is somewhat similar to Algorithm III.3.

In the following experiments, Algorithm III.4 is adopted in IChecker, and the benchmarks are the same as those in the above experiments. Table III shows how the performance of IChecker varies with two different sets of initial invariant candidates. In the column “without refinement”, the initial invariant candidates used are generated based on the constant or equivalent signals in the first timeframe of the circuit with respect to the given initial state. In the column “with refinement”, the initial invariant candidates used are the ones used in Column “without refinement” followed by further refinement using random simulation. The first candidate set is less accurate than the second set as there are more false inductive invariants in the first set. For both sets of initial invariant candidates, Columns “#const”, “#eq”, “#ite” and “cpu(s)” show the number of signals in the initial constant invariant candidates, the number of signals in the initial equivalence invariant candidates, the number of iterations in the fix-point calculation, and the CPU time (in seconds) spent in the fix-point calculation.

The experimental results indicate that if the initial invariant candidates are less accurate, it will take more iterations and longer CPU time for fix-point calculation. However, the difference in CPU time is relatively small.

Table IV compares the performance of IChecker and IProver [14]. The initial invariant candidates used for both tools are the same as those in Table I. Columns “IProver (s)” and “IChecker (s)” report the CPU times (in seconds) of IProver and IChecker spent in performing the fix-point calculation respectively. The speedup achieved by IChecker is shown in “ratio”.

The results indicate that IChecker achieves very significant performance improvement (varying from 5X to 525X) over IProver for these cases.
V. Conclusions and Future Work

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.

Observing from experimental results, we notice that low accuracy of the initial invariant candidates would not degrade IChecker's performance much. However, further investigation is needed on how to identify and provide a proper set of initial invariant candidates so that useful invariants for a specific application could be derived. For example, consider the sequential equivalent checking problem. For a miter circuit with output "o", we are usually interested in proving "o = 0" to be an invariant with respect to the set of reachable states for a given initial state. The characteristic function of this set of states is a Boolean function of states that returns true if a state is a reachable state, and returns false otherwise. The one-element set formed by the characteristic function is an IPS. The problem is that how to identify a good set of initial invariant candidates so that if "o = 0" is indeed true, the IPS derived from the initial set is an over-approximation of the reachable set and the approximation is accurate enough to determine "o = 0" is an invariant. For most cases, using only equivalence and constant invariant candidates cannot achieve this goal. We need to provide some relevant clause invariant candidates that could help IChecker to identify such invariants. We are currently investigating this problem.

References