K-Induction without Unrolling

Arie Gurfinkel
University of Waterloo
http://ece.uwaterloo.ca/~agurfink

Alexander Ivrii
IBM Research
alexi@il.ibm.com

***Abstract*—We present a flexible algorithmic framework `KIC3` that combines `IC3` and k-induction. The key underlying observation is that k-induction can be easily simulated by existing `IC3` implementations by following a slightly different counterexample-queue management strategy.**

I. Introduction

The principle of k-induction is the first successful technique for unbounded SAT-based model checking [1]. It is based on the following generalization of the usual one-step induction principle: a safety property φ is invariant (i.e., holds in all the reachable states of the system) if (a) φ holds in all states reachable in up to k steps, and (b) φ holds for k consecutive steps implies that it holds for $k + 1$-steps. Interestingly, k-induction is complete when restricted to loop-free (or simple) paths. That is, any invariant φ is k-inductive for some k, when only loop-free paths of length k are considered. This gives rise to an unbounded model checking algorithm that repeatedly tries to prove that φ is k-inductive for increasing values of k.

Today, k-induction [1] remains a very important technique for unbounded model checking in both hardware and software domains [2]. A classical implementation of k-induction uses a SAT-solver to check a k-step unrolling of a transition relation, and ensures loop-freedom of counterexamples via additional unique state constraints. However, the scalability of the technique is limited by the depth k of the required unrolling. While combining k-induction with additional invariant synthesis (e.g., [3]) is beneficial, applicability of k-induction remains limited to properties that can be established with a small value of k.

`IC3`/`PDR` [4], [5] is currently the dominant SAT-based unbounded model checking technique. Pioneered by Bradley [4], `IC3` has became the definitive framework for developing SAT- and SMT-based model checking algorithms for both hardware and software verification. Given a safety property φ, `IC3` computes an inductive strengthening F of φ. That is, a formula F such that $\varphi \to F$ and F is inductive. Furthermore, when φ is not an invariant, then `IC3` produces a counterexample. One of the distinguishing features of `IC3` is that it does not explicitly build an unrolling of the transition relation: all reasoning is done over a single step.

As was pointed out in [6], the strengths of k-induction and `IC3` are complementary. Properties that are k-inductive for a small value of k (e.g., 3 or 4) and, therefore, are "easy" for k-induction, are not necessarily easy for `IC3`. More specifically, `IC3` is not guaranteed to terminate after exploring all k-depth counterexamples even when a property is k-inductive. Based on this observation, Jovanovic and Dutertre [6] presented an alternative model checking approach using the insights from both algorithms. However, their approach requires a significant modification of `IC3` and an unrolling-based check for k-induction. In this paper, we explore an alternative solution that tighter and more elegantly integrates k-induction within `IC3`.

The paper makes two contributions. First, we introduce a new algorithm, called `K-Ind`, to decide whether a given safety property φ is k-inductive. The algorithm is based on the insights from `IC3`, and does not explicitly unroll the transition relation. Whenever φ is k-inductive, `K-Ind` returns an inductive strengthening of φ; otherwise, it returns a counterexample to k-induction. Perhaps the most interesting feature of `K-Ind` is that it does not rely on an expensive unique-states constraint to guarantee that only loop-free paths are considered. Furthermore, since it is embedded in the `IC3` framework, it benefits from all the usual `IC3` optimizations such as inductive generalization and generalization of predecessors.

Second, we introduce a framework, called `KIC3`, that combines `IC3` and k-induction in a single `IC3`-like algorithm. Our key insight is that k-induction can be simulated by a specialized counterexample-queue management strategy. This enables `KIC3` to immediately be compatible with all known `IC3`-optimizations and extensions (e.g., [5], [7]–[9]). The algorithm is parameterized by the degree of k-inductive reasoning, where k-induction can be used to simply validate k-inductive conjectures, construct k-inductive strengthening, or recursively block counterexamples to induction.

The rest of the paper is organized as follows. In Section II, we review the necessary background about k-induction and `IC3`. In Section III, we present the `K-Ind` algorithm, and in Section IV, we present the `KIC3` framework. Finally, we conclude the paper with an overview of related work in Section V, an experimental evaluation in Section VI, and conclusion in Section VII.

II. Background

A. Propositional Satisfiability

Let $\mathcal{V}$ be a set of variables. A *literal* is either a variable $b \in \mathcal{V}$ or its negation $\neg b$. A *clause* is a disjunction of literals. A Boolean formula in *Conjunctive Normal Form (CNF)* is a conjunction of clauses. A *cube* is a conjunction of literals. A Boolean formula in *Disjunctive Normal Form (DNF)* is a disjunction of cubes. It is often convenient to treat a clause or a cube as a set of literals, a CNF as a set of clauses, and DNF as a set of cubes. For example, given a CNF formula F, a clause c and a literal ℓ, we write $\ell \in c$ to mean that ℓ occurs in c, and $c \in F$ to mean that c occurs in F.

This work is partially funded by an IBM Faculty Award.

Let $\mathcal{V}$ be a set of variables and $\mathcal{V}' = \{v' \mid v \in \mathcal{V}\}$. A safety verification problem is a tuple $P = (\mathit{Init}, \mathit{Tr}, \mathit{Bad})$, where $\mathit{Init}(\mathcal{V})$ and $\mathit{Bad}(\mathcal{V})$ are formulas with free variables in $\mathcal{V}$ denoting initial and bad states, respectively, and $\mathit{Tr}(\mathcal{V}, \mathcal{V}')$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ denoting the transition relation. Without loss of generality, we assume that Init and Tr are in CNF.

The verification problem P is SAT (or UNSAFE) iff there exists a natural number N such that the following formula is SAT:

$$\mathit{Init}(\vec{v}_0) \wedge \left(\bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \wedge \mathit{Bad}(\vec{v}_N) \qquad (1)$$

P is UNSAT (or SAFE) iff there exists a formula $\mathit{Inv}(\mathcal{V})$, called *a safe invariant*, that satisfies the following conditions:

$$\mathit{Init}(\vec{v}) \rightarrow \mathit{Inv}(\vec{v}) \qquad \mathit{Inv}(\vec{v}) \wedge \mathit{Tr}(\vec{v}, \vec{v}') \rightarrow \mathit{Inv}(\vec{v}') \qquad (2)$$

$$\mathit{Inv}(\vec{v}) \rightarrow \neg \mathit{Bad}(\vec{v}) \qquad (3)$$

A formula Inv that satisfies (2) is called an *invariant*, while a formula Inv that satisfies (3) is called *safe*.

B. k-invariants and k-induction

An invariant over-approximates all the reachable states of the transition relation; however, there is no efficient way to check that a formula is an invariant. An inductive invariant is an invariant that is easy to validate.

A formula φ is called a *k-invariant* if it over-approximates all states reachable up to k-steps. That is,

$$\forall 0 \leq N \leq k \cdot \left(\mathit{Init}(\vec{v}_0) \wedge \bigwedge_{i=0}^{N-1} \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \rightarrow \varphi(\vec{v}_N) \quad (4)$$

Note that any formula F that over-approximates the initial states is a 0-invariant. A formula φ is *k-inductive invariant* if it is a k-invariant and

$$\left(\bigwedge_{i=0}^{k} \varphi(\vec{v}_i) \wedge \mathit{Tr}(\vec{v}_i, \vec{v}_{i+1}) \right) \rightarrow \varphi(\vec{v}_{k+1}) \qquad (5)$$

The definition of k-induction naturally extends to k-induction relative to some 0-invariant formula F, by replacing all but the last occurrence of φ in eq. (5) with $(\varphi \wedge F)$.

Neither induction nor k-induction are complete. That is, there are a transition system Tr and an invariant φ such that φ not k-inductive for any k. However, as shown in [1], k-induction is complete when restricted to loop-free (or simple) paths. That is, the antecedent of eq. (5) is strengthened to ensure that the sequence $\vec{v}_0, \ldots, \vec{v}_{k+1}$ is loop free.

C. Description of IC3

We give a brief description of IC3 that highlights some steps, but omits many crucial optimizations. We refer the reader to [10] for an overview of available optimizations and their possible implementations.

IC3 maintains a sequence of sets of clauses $F_0, F_1, \ldots$ called an *inductive trace*. Each set of clauses F_i in a trace is called a *frame*, each clause $c \in F_i$ is called a *lemma*, and the index of a frame is called a *level*. We assume that F_0 is initialized to Init and that $\mathit{Init} \rightarrow \neg \mathit{Bad}$. IC3 maintains the following invariant:

$$F_i \rightarrow \neg \mathit{Bad} \qquad F_{i+1} \subseteq F_i \qquad F_i \wedge \mathit{Tr} \rightarrow F'_{i+1}$$

That is, each element of the trace is safe, the trace is syntactically monotone, and each F_{i+1} is inductive relative to F_i.

```
Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
       $(f_0 - 1)$-inductive
1  Add($Q, \langle s_0, f_0 \rangle$)
2  while $\neg$Empty($Q$) do
3      $\langle s, f \rangle \leftarrow$ Pop($Q$)
4      assert $\neg s$ is $(f-1)$-invariant
5      if $f = 0$ then
6          return CEX
7      if SAT?($\neg s \wedge F_{f-1} \wedge Tr \wedge s'$) then
8          $t \leftarrow$ ExtractPredecessor($s$)
9          Add($Q, \langle t, f-1 \rangle$)
10         Add($Q, \langle s, f \rangle$)
11     else
12         $\langle c, g \rangle \leftarrow$ Generalize($\neg s, f$)
13         AddLemma($c, g$)
14         if $g < f_0$ then
15             Add($Q, \langle s, g+1 \rangle$)
16 return BLOCKED
```

Fig. 1. IC3 Blocking (IC3_Block).

Fig. 1 presents the blocking procedure of IC3. The inputs to IC3_Block are a state s_0 and a level f_0, with $\neg s_0$ already known to be $(f_0 - 1)$-invariant. The procedure either strengthens the inductive trace and returns BLOCKED indicating that $\neg s_0$ is f_0-invariant, or finds a counterexample trace witnessing that s_0 is reachable from Init and returns CEX.

IC3_Block maintains a queue of *proof obligations* (or *CTI's*) of the form $\langle s, f \rangle$ where s is a cube over state variables and f is a *level*. At each point of the execution, it considers a proof obligation $\langle s, f \rangle$ with the smallest level f, and attempts to prove that s is reachable in f steps. If $f = 0$ then there is a real counterexample. Otherwise, it makes a *predecessor* query $\mathit{SAT?}(\neg s \wedge F_{f-1} \wedge \mathit{Tr} \wedge s')$ that checks whether a state in s can be reached from a state in F_{f-1}. If the result is satisfiable, it adds a predecessor of s as a new proof obligation at level $f - 1$. If the result is unsatisfiable, it learns a new lemma c, such that $\mathit{Init} \rightarrow c$, $c \rightarrow \neg s$ and $c \wedge F_{f-1} \wedge \mathit{Tr} \rightarrow c'$, and adds c to F_j, for all $j \leq f$. In other words, the lemma c represents a new over-approximation, and in particular demonstrates why the state s cannot be reached in up to f steps from the initial states. An important optimization is to re-enqueue s at the lowest unknown frame.

Each time that IC3 blocks Bad for one additional level, it enters a propagation phase, in which for each level f and for each lemma $c \in F_f \setminus F_{f+1}$, it executes the following SAT query: $\mathit{SAT?}(c \wedge F_f \wedge \mathit{Tr} \wedge \neg c')$. Whenever this query is unsatisfiable, the lemma c can be added to frame F_{f+1}.

IC3 terminates if at any point of the execution $F_{f-1} = F_f$ and $F_f \rightarrow \neg \mathit{Bad}$. In this case F_f represents an inductive invariant establishing the correctness of the property.

```
Input: A number $k$, a $k$-invariant $\ell$, a 0-invariant $F_0$
1  $F \leftarrow F_0$
2  Add($Q, \langle \neg\ell, k \rangle$)
3  while $\neg$Empty($Q$) do
4      $\langle s, f \rangle \leftarrow$ Pop($Q$)
5      assert $\neg s$ is $(f-1)$-invariant
6      if $f = 0$ then
7          if $s \cap Init \neq \emptyset$ then return CEX
8          else return K-CTI
9      $F \leftarrow F \wedge \neg s$
10     if SAT?($F \wedge Tr \wedge s'$) then
11         $t \leftarrow$ ExtractPredecessor($s$)
12         assert $t \rightarrow F$
13         Add($Q, \langle t, f-1 \rangle$)
14         Add($Q, \langle s, f \rangle$)
15     else
16         $c \leftarrow$ Generalize($\neg s$)
17         $F \leftarrow F \wedge c$
18         $G \leftarrow G \wedge c$
19 return (BLOCKED, $G$)
```

Fig. 2. k-induction without unrolling (K-Ind).

III. K-Induction without Unrolling

In this section we present K-Ind, an algorithm for deciding whether a given k-invariant formula is k-inductive. Unlike the traditional approach that reduces k-induction to BMC by unrolling the transition relation, our algorithm is based on IC3, and maintains only a single copy of the transition relation. In addition, unlike the traditional approach, K-Ind guarantees loop-free paths without introducing expensive unique-state constraints. In the rest of this section, we present the algorithm, argue for its correctness, and illustrate it on an example.

A. The Algorithm

The pseudo-code of K-Ind is shown in Fig. 2. The inputs to K-Ind are a number k determining the depth of induction, a k-invariant formula ℓ, and a 0-invariant F_0. For simplicity of presentation, we require that ℓ is a clause. The algorithm returns one of three values: CEX to indicate that ℓ is not $(k+1)$-invariant, K-CTI to indicate that ℓ is not k-inductive relative to F_0, and a tuple (BLOCKED, G) to indicate that ℓ is k-inductive relative to F_0, and G is an inductive strengthening of ℓ (i.e., G contains ℓ and is inductive relative to F_0.

K-Ind closely follows the blocking procedure of IC3 (shown in Fig. 1) with several important differences that are highlighted next. First, all SAT-queries are made relative to the single frame F. This can alternatively be explained as IC3_Block in which all frames are the same and do not necessarily form an inductive trace. Second, a lemma learned at any stage of the algorithm holds for all frames. Hence, discharged proof obligations are not re-enqueued to higher levels. In particular, the priority queue Q acts as a LIFO stack. Third, the level of a proof obligation represents how much remaining budget it has rather than the frame on which it should be blocked. Fourth, the assertion on line 5 holds. Fifth, the negation of the predecessor to be blocked is assumed during blocking of all of it descendants (line 9). Lastly, the generalization of predecessors is done with respect to the frame F so that the assertion on line 12 holds.

B. Correctness

In the following, we establish the correctness of K-Ind. Our correctness argument is partitioned into two cases: (1) K-Ind returns CEX or K-CTI, and (2) it returns BLOCKED.

Lemma 1 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be a 0-invariant. If* K-Ind(k, ℓ, F_0) *returns* CEX*, then ℓ is not a $(k+1)$-invariant, and if it returns* K-CTI *then ℓ is not k-inductive relative to F_0.*

Proof: Similarly to IC3_Block, whenever K-Ind reaches line 6, the queue Q contains a loop-free sequence of $k+1$ states consistent with F_0 and satisfying the transition relation. This sequence witnesses that ℓ is not k-inductive. Furthermore, if it intersects with the initial state, then ℓ does not hold after $(k+1)$ steps of the tranistion relation. Hence, ℓ is not $(k+1)$-invariant. ■

Lemma 2 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be a 0-invariant. If* K-Ind(k, ℓ, F_0) *returns a tuple* (BLOCKED, G)*, then G is an inductive strengthening of ℓ relative to F_0.*

Proof: By construction, G contains ℓ and is inductive relative to F. Whenever K-Ind terminates with BLOCKED, every state that has ever been added to Q is blocked. Hence, for every clause c in $F \setminus F_0$ there is a stronger clause d in G. Thus, G is also inductive relative to F_0. ■

Theorem 1 *Let k be a natural number, a clause ℓ be a k-invariant, and F_0 be 0-invariant. Then, assuming that* Generalize($\neg s$) *always returns $\neg s$,* K-Ind(k, ℓ, F_0) *terminates and returns* BLOCKED *iff ℓ is k-inductive relative to F_0,* CEX *iff ℓ is not $(k+1)$-invariant, and* K-CTI *iff ℓ is not k-inductive but $(k+1)$-invariant.*

Proof: We only need to show termination. The rest follows from Lemma 1 and Lemma 2. The number of iterations of the outer loop is bounded by the number of clauses (or cubes). At every iteration of the loop, either a new predecessor is added to the queue Q, or a new clause is added to frame F. No predecessor is added more than once. All the clauses in F are distinct. ■

The assumption that Generalize($\neg s$) always returns $\neg s$ in Theorem 1 is needed only to guarantee that K-CTI is returned whenever ℓ is not k-inductive. Removing this assumption makes the algorithm stronger by allowing it to find an inductive strengthening of ℓ even when ℓ is not k-inductive (of course, this is only possible when ℓ is an invariant).

Interestingly, K-Ind is complete in the sense that if ℓ is an invariant, then there is a k such that K-Ind($k, \ell, \top$) returns BLOCKED. This follows from the fact that K-Ind only considers loop-free counterexamples to k-induction. Note that restriction to loop-free paths follows from assuming the negation of a state to be blocked (line 9). This is a simpler alternative to a traditional approach of encoding loop-freedom of a path via an explicit unique-states constraint.

C. An Example

In this section, we illustrate K-Ind and highlight the difference with IC3_Block on an example. Consider the following transition system P

$$\begin{aligned}
\mathcal{V} &= \{x, a, b, c\}\\
\mathit{Init} &\equiv x \wedge a \wedge b \wedge c\\
\mathit{Tr} &\equiv (x' = \neg x \vee a \vee b \vee c) \wedge\\
&\qquad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0)\\
\mathit{Bad} &\equiv \neg x
\end{aligned}$$

Note that $a = 1$ and $x = 1$ are invariants of P, while $b = 1$ and $c = 1$ are not. In fact, $c = 1$ only holds on the initial cycle, and $b = 1$ only holds on the first two cycles. Consider the property $\ell \equiv \neg \mathit{Bad} = (x = 1)$. ℓ is not 1-inductive but is 2-inductive.

We begin by illustrating a run of `K-Ind` with inputs: $\ell = \{x = 1\}$, $k = 2$ and $F_0 = \top$. On the first iteration of the loop, F is updated to (x), and the predecessor query for s_0 relative to F yields a SAT query:

$$\begin{aligned}
&(x) \wedge (x' = \neg x \vee a \vee b \vee c) \wedge\\
&\qquad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0) \wedge (x' = 0)
\end{aligned}$$

This query is satisfiable. This, in particular, shows that ℓ is not 1-inductive. The corresponding predecessor is

$$t = \{x = 1, a = 0, b = 0, c = 0\}$$

On the second iteration of the loop, F is updated to $(x) \wedge (\neg x \vee a \vee b \vee c)$, and the predecessor query for t relative to F yields a SAT query

$$\begin{aligned}
&(x) \wedge (\neg x \vee a \vee b \vee c) \wedge (x' = \neg x \vee a \vee b \vee c) \wedge\\
&\quad (a' = a \vee b) \wedge (b' = c) \wedge (c' = 0) \wedge (x' = 1) \wedge\\
&\qquad\qquad (a' = 0) \wedge (b' = 0) \wedge (c' = 0)
\end{aligned}$$

This query is unsatisfiable. Assuming that `Generalize` does not remove any literals, `K-Ind` learns the lemma $(\neg x \vee a \vee b \vee c)$ and adds it to F. With the proof obligation t being blocked, the algorithm re-examines ℓ, and makes the predecessor query

$$\begin{aligned}
&(x) \wedge (\neg x \vee a \vee b \vee c) \wedge\\
&\quad (x' = \neg x \vee a \vee b \vee c) \wedge (a' = a \vee b) \wedge (b' = c) \wedge\\
&\qquad\qquad (c' = 0) \wedge (x' = 0)
\end{aligned}$$

This query is also unsatisfiable. The algorithm outputs BLOCKED, with the constructed 1-inductive strengthening of φ being $\psi = x \wedge (\neg x \vee a \vee b \vee c)$. Finally, we note that generalization relative to F (the procedure `Generalize`) could have also yielded the lemma $(a = 1)$ (resulting in the strengthening $(x \wedge a)$), but could *not* have yielded lemma $(b = 1)$ since $(b = 1)$ is not inductive relative to $(x) \wedge (\neg x \vee a \vee b \vee c)$.

Next, consider `IC3` on the same example. The blocking procedure of `IC3` similarly finds t as a predecessor of s, but makes the next predecessor query relative to $F_0 = \mathit{Init}$. More importantly, it calls `Generalize` on $(\neg x \vee a \vee b \vee c)$ relative to F_0, possibly learning the lemma $(b = 0)$ instead. Then, `IC3` concludes that s_0 is blocked on level 2. However, since $(x) \wedge (b)$ is not an inductive invariant, `IC3` needs to unfold the trace for an additional frame and continue blocking s on frame 3.

This example shows that `K-Ind` guarantees to strengthen a k-inductive property to an inductive one, while `IC3` does not provide any such guarantees.

```
Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
       $(f_0 - 1)$-inductive
1 $res \leftarrow$ UNKNOWN
2 while $res =$ UNKNOWN do
3     $strategy \leftarrow$ AdjustStrategy()
4     $res \leftarrow$ BlockUsingStrategy($s_0, f_0, strategy$)
5 return $res$
```

Fig. 3. `KIC3` Top-level blocking (`KIC3_Block`).

IV. KIC3: K-INDUCTIVE IC3 ALGORITHM

In this section, we describe the `KIC3` framework that unifies `IC3` and k-induction model checking algorithms. The core of `KIC3` is a blocking procedure that integrates `IC3_Block` with a variant of k-induction. This procedure is then incorporated into a flexible approach for blocking proof obligations and for pushing existing lemmas forward.

A. *Top-level blocking in* `KIC3`

A pseudo-code for the top-level blocking procedure, `KIC3_Block` of `KIC3` is shown in Fig. 3. The procedure takes as input a state s_0 and a level f_0 and assumes $\neg s_0$ is an $(f_0 - 1)$-invariant. The procedure outputs either BLOCKED to indicate that $\neg s_0$ is f-invariant, or CEX to indicate that s_0 is reachable from *Init*. As in `IC3`, `KIC3` maintains an inductive trace $F_0, F_1, \ldots$ that is updated throughout the blocking process. Internally, `KIC3_Block` implements a portfolio approach, delegating lower-level blocking to other procedures such as `IC3_Block` (shown in Fig. 1), or `KIC3_Block_Kind` (shown in Fig. 3 and described later in this section). Note that the internal blocking procedure might return UNKNOWN to indicate that it has given up before finding a solution.

B. *k-induction blocking in KIC3*

Fig. 4 presents the k-inductive blocking procedure `KIC3_Block_Kind` of `KIC3`. In addition to a state s_0 and a level f_0, `KIC3_Block_Kind` requires two parameters: k_0 – the induction depth, and m_0 – the maximum number of predecessors to k-induction to be blocked by an external blocking procedure. The output of `KIC3_Block_Kind` is one of BLOCKED, CEX or UNKNOWN. As s_0 is only known to be $(f_0 - 1)$-inductive, the actual induction depth k is set to the smaller of the two values k_0 and f_0 (line 1). Our algorithm is strongly reminiscent of both `IC3_Block` and `K-Ind`, with several important differences that are described below.

First, all SAT queries (line 16) are performed relative to the same frame $F_{f_0 - 1}$. The level f of a proof obligation $\langle s, f \rangle$ has a slightly different interpretation than in `IC3_Block`: s is guaranteed to be unreachable from *Init* in $f - 1$ steps or less (assertion on line 6). The same queue management strategy as in `IC3_Block` is used. That is, proof obligations with lowest levels are chosen first. As in `K-Ind`, all the learned lemmas hold up to level f_0. The discharged proof obligations do not need to be re-enqueued, and the priority queue Q acts as a LIFO stack. However, unlike `K-Ind`, the query on line 16 does not fully implement loop-free paths, and instead a more relaxed condition of `IC3_Block` is used.

```
   Input: A state $s_0$ and a level $f_0$ s.t. $\neg s_0$ is
          $(f_0 - 1)$-inductive; parameters $k_0$ and $m_0$
1  $k \leftarrow \mathtt{min}(k_0, f_0)$
2  $m \leftarrow 0$
3  $\mathtt{Add}(Q, \langle s_0, f_0 \rangle)$
4  while $\neg\mathtt{Empty}(Q)$ do
5      $\langle s, f \rangle \leftarrow \mathtt{Pop}(Q)$
6      assert $\neg s$ is $(f-1)$-invariant
7      if $f = f_0 - k$ then
           // Found cex to $k$-induction
8          if $m < m_0$ then
9              $m \leftarrow m + 1$
10             if $\mathtt{BlockInRange}(s, f_0 - k, f_0) = $ CEX
               then
11                 return CEX
12         else if $(f = 0) \wedge (s \cap Init \neq \emptyset)$ then
13             return CEX
14         else
15             return UNKNOWN
16     if SAT?$(\neg s \wedge F_{f_0-1} \wedge Tr \wedge s')$ then
17         $t \leftarrow \mathtt{ExtractPredecessor}(s)$
18         $\mathtt{Add}(Q, \langle t, f-1 \rangle)$
19         $\mathtt{Add}(Q, \langle s, f \rangle)$
20     else
21         $\langle c, g \rangle \leftarrow \mathtt{Generalize}(\neg s, f_0)$
22         $\mathtt{AddLemma}(c, g)$
23 return BLOCKED
```

Fig. 4. `KIC3` blocking using k-induction (`KIC3_Block_Kind`).

```
   Input: A state $s_0$, levels $f_0$ and $f_1$ s.t. $s_0$ is
          $(f_0 - 1)$-invarant
1  for $f = f_0, \ldots, f_1$ do
2      if $(f = 0) \wedge (s_0 \cap Init \neq \emptyset)$ then
3          return CEX
4      if $(f \neq 0) \wedge (\mathtt{KIC3\_Block}(s_0, f) = $ CEX$)$ then
5          return CEX
6  return BLOCKED
```

Fig. 5. `KIC3` Top-level blocking in a range of frames (`BlockInRange`).

Second, when a proof obligation $\langle s, f \rangle$ at level $f = f_0 - k$ is examined, and consequently a K-CTI is discovered, the algorithm may attempt to recover by blocking this counterexample to k-induction. Since $\neg s$ is $(f_0 - 1)$-invariant, s needs to be blocked in every level in the range $[f_0 - k, f_0]$ (see line 10). The implementation of `BlockInRange` is shown in Fig. 5. As usual, we require that $\neg s_0$ is $(f_0 - 1)$-invariant. The procedure iterates over levels from f_0 to f_1 and calls `KIC3_Block` to block s_0 at the corresponding level. It returns BLOCKED if s_0 is blocked on all levels in $[f_0, f_1]$ (and hence $\neg s_0$ is f_1-invariant), and CEX otherwise.

Third, there is a parameter to limit the number of K-CTIs considered. When the number of K-CTIs reaches the maximum number m_0, `KIC3_Block_Kind` returns UNKNOWN. Note that whenever $m_0 = 0$, the external blocking procedure is not used at all, and `KIC3_Block_Kind` returns UNKNOWN (or possibly CEX) as soon as the first K-CTI is discovered. This limits the algorithm to only learn "high-quality" lemmas that hold up to level f_0, at the risk of eventually returning UNKNOWN sooner. On the other hand, when $m_0 = \infty$, all the K-CTIs are blocked with an external blocking procedure. In this case, `KIC3_Block_Kind` is also guaranteed to return either BLOCKED or CEX (and to never return UNKNOWN).

C. Correctness and Termination

In this section, we argue the correctness of `KIC3_Block_Kind`. First, the assertion on line 6 holds: the top-level proof obligation s_0 satisfies the assertion by assumption, and other proof obligations are added on line 18 and satisfy the assertion due to the following lemma.

Lemma 3 *Let s be a state, and $f \geq 0$ be a natural number, such that $\neg s$ is f-invariant. Let t be a predecessor of s. Then $\neg t$ is $(f-1)$-invariant.*

Proof: By contradiction. Assume $\neg t$ is not $(f-1)$-invariant. Then, there is a counterexample trace π of length at most f that reaches t from the initial states. Since t is a predecessor of s, there is a one-transition extension of π that shows that $\neg s$ is not f-invariant. ∎

Second, whenever `KIC3_Block_Kind` reaches line 13, the queue Q contains a sequence π of $k+1$ states satisfying the transition relation, with the first state in the sequence intersecting the initial states, and the last state intersecting s_0. The path π shows that s_0 is reachable from *Init* in $k+1$ steps.

Third, whenever the algorithm reaches line 23, correctness is argued as in `IC3`: each lemma c added in line 22 satisfies $Init \rightarrow g$ and $g \wedge F_{f_0-1} \wedge Tr \rightarrow g$, and, hence, is inductive relative to F_{f_0-1}.

Fourth, whenever `KIC3_Block_Kind` calls `BlockInRange` recursively with a state s, by construction `BlockInRange` always calls an internal blocking procedure at the lowest level at which s is not yet blocked. Thus, the pre-conditions of the internal blocking procedure are satisfied.

Finally, the recursion of `BlockInRange` is well founded. Suppose that `KIC3_Block_Kind`(s_0, f_0) calls `BlockInRange` which, in turn, calls `KIC3_Block_Kind`(s_1, f_1). Then either $f_1 < f_0$, or $f_1 = f_0$ and, thus, `BlockInRange` has already blocked s_1 at level $f_0 - 1$ and learned a new lemma. Thus, in both cases the recursion makes a progress and must terminate.

D. Discussion

It is interesting to contrast the blocking strategies in `IC3` and `KIC3`. `IC3_Block` blocks each proof obligation at the lowest level it is yet unknown. Thus, if $\langle s, f \rangle \in Q$ is a proof obligation and t is a predecessor of s, then `IC3_Block` recursively attempts to block t at level $f-1$, and, if successful, blocks t at level f as well. On the other hand, `KIC3_Block_Kind` attempts to directly block t at level f, without blocking it at level $f-1$ first. Thus, from a high-level perspective, `KIC3_Block_Kind` is a variant of `IC3_Block`, with a different counterexample-queue management strategy.

By always making SAT queries relative to the frame F_{f_0-1}, `KIC3` essentially ignores all lemmas not in F_{f_0-1}. On the one hand, this may force it to spend more effort on blocking the top-level proof obligation $\langle s_0, f_0 \rangle$. On the other hand, all learned lemmas automatically hold up to level f_0. Intuitively, since the lemmas are true for more steps of the transition relation they are of a "higher-quality", i.e., more likely to be part of the final inductive invariant.

Note that while KIC3_Block_Kind is similar to K-Ind, it does not fully incorporate the search for loop-free paths. This might be important when proof obligations are not enqueued at their lowest unknown levels. In particular, KIC3_Block_Kind may fail to find an inductive strengthening of $\neg s_0$, even when s_0 is k-inductive relative to F_{f_0-1}. Addressing this deficiency requires an ability to remove lemmas from frames in an IC3 framework. Developing support for this feature is an interesting direction for future work.

Another interesting technical dilemma reflects predecessor generalization on line 17 of KIC3_Block_Kind. More precisely, when the SAT query SAT?$(\neg s \wedge F_{f_0-1} \wedge Tr \wedge s')$ is satisfiable, with $\bar{t}$ being the predecessor of s, it is customary to generalize $\bar{t}$ to a larger set of states t such that any state in t leads to s [5]. In practice, it is not clear whether it is desirable to additionally enforce that $t \rightarrow F_{f_0-1}$ and $t \rightarrow \neg s$. On the one hand, generalizing predecessors with respect to $\neg s \wedge F_{f_0-1}$ avoids spurious K-CTIs. On the other hand, it significantly increases the sizes of proof obligations considered. We do not take these additional constraints into account in our experiments.

E. Portfolio blocking strategies

In this section, we describe the portfolio blocking strategies used in the experimental evaluation. Given a state s_0 to be blocked at level f_0, the $B(k_0, m_0)$-strategy with $1 \leq k_0 \leq \infty$ and $0 \leq m_0 \leq \infty$ is defined as follows:

$B(k_0, m_0)$

1) Block (s_0, f_0) using KIC3_Block_Kind with the induction depth k_0, the maximum number of K-CTIs m_0, and the procedure BlockInRange realized by IC3_Block;
2) If the previous procedure returns UNKNOWN, block (s_0, f_0) using IC3_Block.

There are several important special cases of this strategy. First, when $k = \infty$, KIC3_Block_Kind is called with the largest induction depth (in other words, f_0) allowed for blocking (s_0, f_0). Second, when $m_0 = 0$, KIC3_Block_Kind returns UNKNOWN as soon as the first K-CTI is discovered, in the process only learning lemmas at levels at least f_0. Third, when $m_0 = \infty$, KIC3_Block_Kind blocks all counterexamples to k-induction using IC3_Block, and IC3_Block is never called directly.

F. Pushing in KIC3

Our generic framework for blocking a state at a specific level can also be used in the pushing stage of IC3. The pseudocode of KIC3_Push is shown in Fig. 7. Lines 1–5 implement the traditional pushing procedure: when a lemma $c \in F_f \setminus F_{f+1}$ is inductive relative to F_f, it is also added to the frame F_{f+1}. Interestingly, this process can be reinterpreted as calling KIC3_Block_Kind with parameters $k_0 = 1$ and $m_0 = 0$. On lines 6–7, we propose to additionally push lemmas at the last level of the inductive trace using KIC3_Block_Kind with a larger value of k_0 (and m_0 still equal to 0).

```
1 $N \leftarrow$ Level$((\neg Bad))$
2 for $f = 1, \ldots, N$ do
3     for all lemmas $c \in F_f \setminus F_{f+1}$ do
4         if $F_f \wedge c \wedge Tr \Rightarrow c'$ then
5             $F_{f+1} \leftarrow F_{f+1} \cup \{c\}$
6 for all lemmas $c \in F_N$ do
7     KIC3_Block$(\neg c, f+1)$
```

Fig. 6. KIC3 Pushing (KIC3_Push).

There are many alternative ways to integrate KIC3_Block_Kind into the pushing stage of IC3. However, as in general KIC3_Block_Kind$(\neg c, f+1)$ can add new lemmas to the level $f+1$, additional care must be taken to guarantee termination of the pushing stage.

V. Related and Future Work

There is a large body of work on automating k-induction using SAT-based reasoning and on unbounded model checking using IC3. We focus only on the most closely related work.

It is well-known that k-induction principle is stronger than induction. In fact, k-induction is complete when restricted to loop-free paths while induction is not. Bjørner et al. [11] show that any k-inductive property can be converted into an inductive property by interpolation. Thus, the size of an inductive property is linear in the size of the resolution proof of the corresponding k-inductive property. Our K-Ind algorithm is a constructive proof of this fact. Given a k-inductive property, K-Ind constructs an inductive certificate in CNF.

K-Ind is built on top of the blocking procedure of IC3 and shares many similarities with it. When the property φ under consideration is k-inductive, both K-Ind (instantiated with the induction depth at least k) and IC3 are guaranteed to terminate and to discover a suitable inductive strengthening of φ. However, K-Ind is guaranteed to only learn k-inductive lemmas, while IC3 provides no such guarantees. In particular, the convergence depth of IC3 might be significantly larger than k. We believe that this is an important theoretical advantage of K-Ind over IC3. Unfortunately, our KIC3_Block_Kind in the KIC3 framework does not guarantee to converge in k steps, making it closer to IC3_Block than to K-Ind. Addressing this deficiency is an interesting topic for future work.

In Quip [9], a variant of IC3, a maximal inductive subset of all the lemmas is explicitly computed and maintained in a separate frame. This guarantees that Quip converges as soon as the trace contains an inductive subset. It is interesting to extend KIC3 to guarantee convergence as soon as the trace contains a k-inductive subset. Using KIC3_Block_Kind for pushing, as suggested in Section IV-F, is a step in that direction.

The PD-Kind algorithm of Jovanovic and Dutertre [6] is closest to ours, and has inspired our work. The main difference is that we have tried to integrate k-induction into IC3 with the fewest modifications of the IC3 framework. For example, PD-Kind requires unrolling the transition relation for validating k-induction queries, while KIC3 does not. Unfortunately, a direct experimental comparison of KIC3 and PD-Kind is difficult, as PD-Kind is implemented at the level

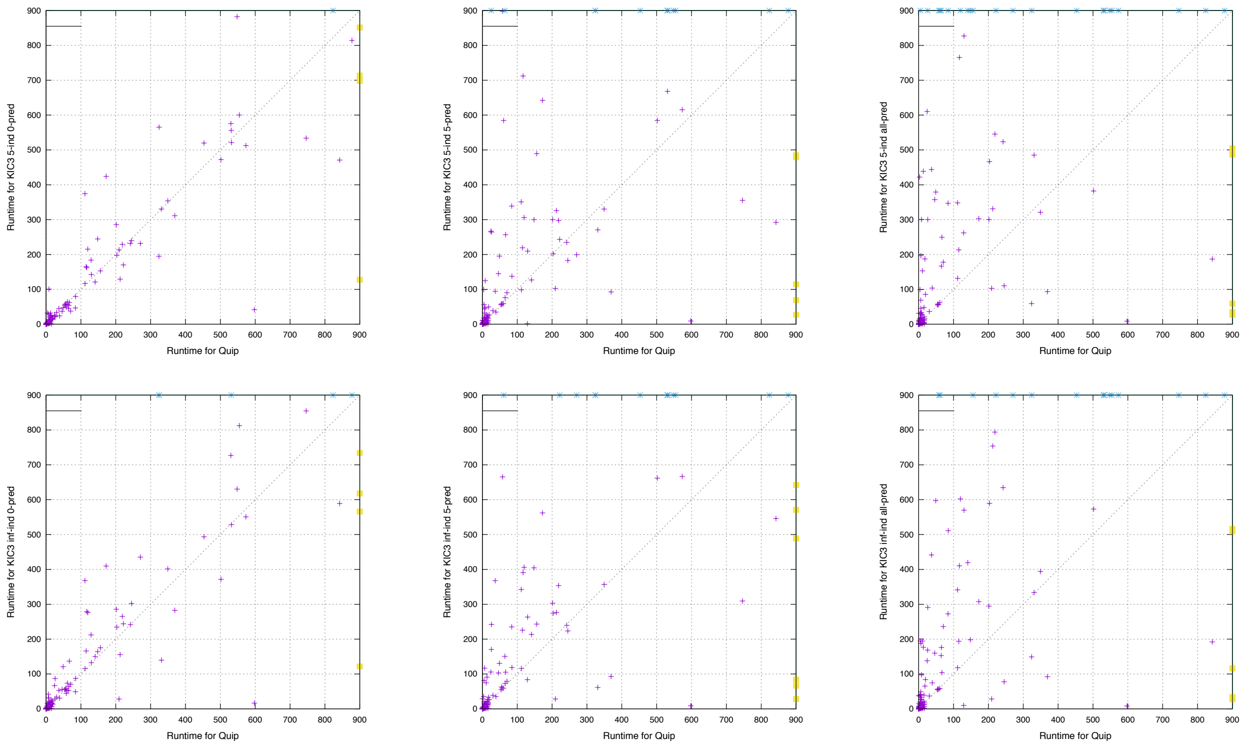

Fig. 7. Summary of experimental results comparing `KIC3` (y-axis) with `Quip` (x-axis) on benchmarks from HWMCC'15.

of SMT and does not target hardware benchmarks. Further, in our experience, a direct implementation of PD-KIND for hardware does not scale, as unrolling the transition relation has a huge negative effect on many benchmarks. At the same time, using k-induction for pushing clauses on the last frame, as we suggest in `KIC3_Push` in Section IV-F, while blocking all predecessors to k-induction using `IC3_Block`, is closely related to the `PUSH` procedure in `PD-Kind`. Overall, adapting the ideas of `PD-Kind` to hardware is a non-trivial task, and to some extent our algorithm can be viewed as a step in this direction.

Recall that `KIC3_Block_Kind` can be seen as `IC3_Block` with a specialized counterexample-queue management. Alternatively, `KIC3_Block_Kind` can also be seen as a form of *abstraction*. Whenever a proof obligation $\langle s, f\rangle$ should be blocked, traditionally, we check whether $\neg s$ is inductive relative to F_{f-1}. However, any abstraction of F_{f-1} can be used as well. For example, using only lemmas that are also in F_f as the abstraction closely corresponds to `KIC3`.

VI. Experiments

The techniques presented in this paper are implemented on top of `Quip` (a variant of `IC3` presented in [9]) in the IBM formal verification tool *IBM RuleBase SixthSense Edition* [12], [13]. All experiments were performed on a 2.13Ghz Linux-based machine with Intel Xeon E7-4830 processor and 16GB of RAM. We have used all single property designs from the HWMCC'15 benchmark set. Each design is initially simplified using standard logic synthesis techniques (similar to the `&dc2` command in ABC [14]). We used a timeout of 900 seconds.

TABLE I. Summary of experimental results.

Technique	Solved	Time (seconds)
`Quip`	230	52,776
$B(5,0)$	**233**	**51,695**
$B(5,5)$	224	57,864
$B(5,\infty)$	212	68,012
$B(\infty,0)$	229	53,757
$B(\infty,5)$	223	57,551
$B(\infty,\infty)$	219	62,397

We omit a direct experimental comparison to the original k-induction algorithm, as k-induction solves tremendously fewer properties than any of the `IC3`-based techniques. On the other hand, on most of the (few) properties that k-induction is able to solve, the value of k is small. In these cases, k-induction (based on unrolling the transition relation) usually outperforms `IC3`-based techniques, in the same manner as `BMC` usually outperforms `IC3` when searching for counterexamples.

We focus the experimental evaluation on the comparison of `Quip` and `KIC3` with different blocking strategies. The experiments are presented for 238 designs – which are all of the designs that remain after removing all instances solved by logic synthesis alone and all instances not solved by any of the techniques. Recall that the blocking strategy $B(k_0, m_0)$ means that `KIC3_Block_Kind` is called with induction depth k_0 and at most m_0 K-CTIs. We say that $k_0 = \infty$ whenever a proof obligation is always blocked with the largest possible induction depth. In what follows, we report the results for 6 different configurations of `KIC3`, obtained by setting k_0 to either 5 or ∞, and setting m_0 to either 0, 5, or ∞. A summary of the overall results is shown in Table I. The columns **Solved** and **Time** represent the total number of instances solved and the cumulative time in seconds, respectively. A detailed comparison between `Quip` and each `KIC3` variant is shown in the scatter plots in Fig. 7. For each of the plots the horizontal axis measures the runtime of `Quip`, while the vertical axis measures the runtime of `KIC3` with the corresponding k-induction blocking strategy. Thus, points below the diagonal represent wins for the pure `Quip` approach, and vice versa.

According to the experiments, the blocking strategy $B(5, 0)$ performs the best, slightly outperforming `Quip` by solving 3 more instances in less time. Furthermore, from the plot on the top-left, we can see that the total runtimes of `Quip` and `KIC3` with $B(5, 0)$ are fairly well correlated, as most points are in the vicinity of the diagonal. However, a more detailed analysis shows that about 30% of the total time to block a proof obligation is spent in the `KIC3_Block_Kind` part of the procedure, so the actual profiles of the two algorithms are significantly different.

We have also experimented with other blocking strategies $B(k_0, 0)$, and in general all the results are highly consistent. For example, the "extreme" configuration $B(\infty, 0)$ solves 4 instance less than $B(5, 0)$ (and 1 instance less than `Quip`), and the plot on the bottom-left still shows high correlation with `Quip`.

At the same time, increasing the number m_0 of K-CTIs has a clear negative effect on the algorithm's performance for almost every k_0. The three top plots demonstrate this for $k_0 = 5$, while the three bottom plots demonstrate this for $k_0 = \infty$. However, we can also note that `Quip` and `KIC3` become less correlated as m_0 is increased, while some instances get solved faster than before.

VII. Conclusions

In this work, we present an algorithm to decide whether a given safety property is k-inductive. This algorithm is based on the insights from `IC3`, and does not explicitly unroll the transition relation or add unique-state constraints to guarantee simple paths. In addition, we show how k-induction can be integrated into `IC3` with minor modifications of the `IC3`-framework. On the practical side, a preliminary experimental evaluation shows a potential benefit of the suggested methods.

References

[1] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, ser. Lecture Notes in Computer Science, W. A. H. Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 108–125. [Online]. Available: http://dx.doi.org/10.1007/3-540-40922-X_8

[2] M. Brain, S. Joshi, D. Kroening, and P. Schrammel, "Safety verification and refutation by k-invariants and k-induction," in *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, 2015, pp. 145–161. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48288-9_9

[3] T. Kahsai and C. Tinelli, "PKind: A parallel k-induction based model checker," in *Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011.*, 2011, pp. 55–62. [Online]. Available: https://doi.org/10.4204/EPTCS.72.6

[4] A. R. Bradley, "SAT-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Jhala and D. A. Schmidt, Eds., vol. 6538. Springer, 2011, pp. 70–87. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18275-4_7

[5] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 125–134. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157675

[6] D. Jovanovic and B. Dutertre, "Property-directed k-induction," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 85–92. [Online]. Available: http://dx.doi.org/10.1109/FMCAD.2016.7886665

[7] A. Cimatti and A. Griggio, "Software model checking via IC3," in *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. A. Seshia, Eds., vol. 7358. Springer, 2012, pp. 277–293. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31424-7_23

[8] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, 2013, pp. 157–164. [Online]. Available: http://ieeexplore.ieee.org/document/6679405/

[9] A. Gurfinkel and A. Ivrii, "Pushing to the top," in *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015.*, 2015, pp. 65–72. [Online]. Available: http://ieeexplore.ieee.org/document/7542254/

[10] A. Griggio and M. Roveri, "Comparing different variants of the IC3 algorithm for hardware model checking," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 1026–1039, 2016. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2015.2481869

[11] N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko, "Horn Clause Solvers for Program Verification," in *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, 2015, pp. 24–51. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-23534-9_2

[12] "RuleBase SixthSense Edition," https://www.research.ibm.com/haifa/projects/verification/Formal_Methods-Home/.

[13] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 159–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30494-4_12

[14] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174. Springer, 2010, pp. 24–40. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_5