Software Verification Using k-Induction

Extended version including appendix with proofs

Alastair F. Donaldson[1], Leopold Haller[1], Daniel Kroening[1], and Philipp Rümmer[2]

[1] Oxford University Computing Laboratory, Oxford, UK
[2] Uppsala University, Department of Information Technology, Uppsala, Sweden

Abstract. We present *combined-case k-induction*, a novel technique for verifying software programs. This technique draws on the strengths of the classical inductive-invariant method and a recent application of k-induction to program verification. In previous work, correctness of programs was established by separately proving a base case and inductive step. We present a new k-induction rule that takes an unstructured, reducible control flow graph (CFG), a natural loop occurring in the CFG, and a positive integer k, and constructs a *single* CFG in which the given loop is eliminated via an unwinding proportional to k. Recursively applying the proof rule eventually yields a loop-free CFG, which can be checked using SAT-/SMT-based techniques. We prove soundness of the rule, and investigate its theoretical properties. We then present two implementations of our technique: K-INDUCTOR, a verifier for C programs built on top of the CBMC model checker, and K-BOOGIE, an extension of the Boogie tool. Our experiments, using a large set of benchmarks, demonstrate that our k-induction technique frequently allows program verification to succeed using significantly weaker loop invariants than are required with the standard inductive invariant approach.

1 Introduction

We present a novel technique for verifying imperative programs using k-induction [21]. Our method brings together two lines of existing research: the standard approach to program verification using *inductive invariants* [16], employed by practical program verifiers (including [4, 5, 11, 19], among many others) and a recent k-induction method for program verification [13] which we refer to as *split-case k-induction*.

Our method, which we call *combined-case k-induction*, is directly stronger than both the inductive invariant approach and split-case k-induction, potentially allowing a program to be verified using weaker loop invariants than would usually be required. In addition, combined-case k-induction avoids the problem of exponential case explosion associated with split-case k-induction.

We recap the inductive invariant and split-case k-induction approaches to verification, and outline our new combined-case k-induction technique in §2, using an example. After introducing necessary notation (§3), we make the following novel contributions:

– We formally present combined-case k-induction as a proof rule operating on control flow graphs, and prove soundness of the rule (§4)

– We consider theoretical properties of the rule, and prove a confluence theorem, showing that, in a multi-loop program, the order in which our rule is applied to loops does not affect the result of verification (§5)
– We present two implementations of our method: K-INDUCTOR, a verifier for C programs built on top of the CBMC model checker, and K-BOOGIE, an extension of the Boogie tool and experimental results applying these tools to a large set of benchmarks (§6)

Our experiments demonstrate that combined-case k-induction frequently allows program verification to succeed using significantly weaker loop invariants than are required with either the standard inductive invariant approach, or split-case k-induction

Throughout the paper, we are concerned with proving partial correctness with respect to assertions: establishing that whenever a statement $assert\ \phi$ is executed, the expression ϕ evaluates to true. In the rest of the paper we simply use *correctness* to refer to this notion of partial correctness.

2 Overview

Throughout the paper, we present programs as control flow graphs (CFGs). We use the terms *program* and *CFG* synonymously. We follow the standard approach of modelling control flow using a combination of nondeterministic branches and $assume$ statements. During execution, a statement $assume\ \phi$ causes execution to silently (and non-erroneously) halt if the expression ϕ evaluates to false, and does nothing otherwise.

Consider the simple example program of Figure 1(a). The program initialises a, b and c to distinct values, and then repeatedly cycles their values, asserting that a and b never become equal. Variable x is initialised to zero, and after the loop an assertion checks that x has not changed. The program is clearly correct.

The inductive invariant approach. To formally prove a program's correctness using inductive invariants, one first associates a candidate invariant with each loop header in the program. One then shows that a) the candidate invariants are indeed loop invariants, and b) these loop invariants are strong enough to imply that no assertion in the program can fail. A technique for performing these checks in the context of unstructured programs is detailed in [3]. The technique transforms a CFG with loops into a loop-free CFG in which each loop header in the transformed CFG is prepended with a basic block that: asserts the loop invariant, havocs each loop-modified variable,[3] and assumes the loop invariant. Each back edge in the transformed CFG is replaced with an edge to a new, childless basic block that asserts the invariant for the associated loop.

We say that each loop is *cut* with invariant ϕ. This is illustrated in Figure 1(b) for the program of Figure 1(a), where invariant ϕ is left unspecified. Cutting every loop in a CFG leads to a loop-free CFG, for which verification conditions can be computed using weakest preconditions (an efficient method for this step is the main contribution of [3]). These verification conditions can then be discharged to a theorem prover, and if they

[3] A variable is *havocked* if it is assigned a nondeterministic value. A *loop-modified variable* is a variable that is the target of an assignment in the loop under consideration.

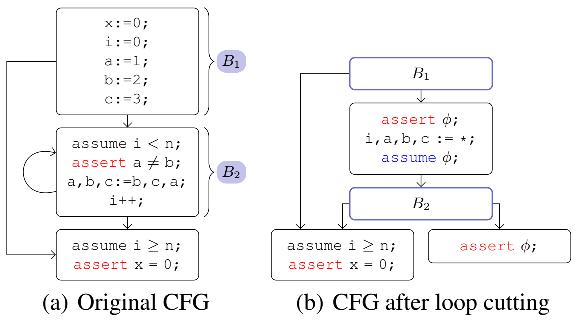


Fig. 1. A simple program, and the CFG obtained using the inductive invariant approach.

are proven, the program is deemed correct. In Figure 1(b), taking ϕ to be $(a \neq b \wedge b \neq c \wedge c \neq a)$ allows a proof of correctness to succeed.

The main problem with the inductive invariant approach is finding the required loop invariants. Despite a wealth of research into automatic invariant generation (see [8] and references therein for a discussion of state-of-the-art techniques), this is by no means a solved problem, and in the worst case loop invariants must still be specified manually.

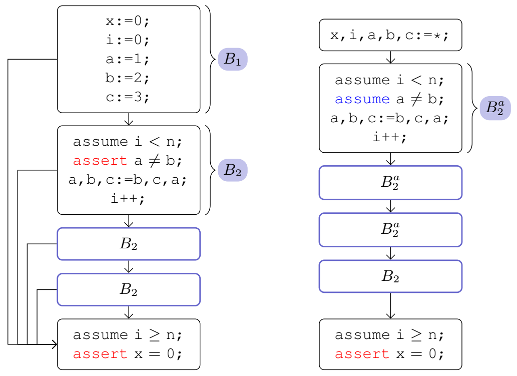
Split-case k-induction. The k-induction method was proposed as a technique for SAT-based verification of finite-state transition systems [21]. Let $\mathbf{I}(s)$ and $\mathbf{T}(s, s')$ be formulae encoding the initial states and transition relation for a system over sets of propositional state variables s and s', $\mathbf{P}(s)$ a formula representing states satisfying a safety property, and k a non-negative integer. To prove $\mathbf{P}$ by k-induction one must first show that $\mathbf{P}$ holds in all states reachable from an initial state within k steps, *i.e.*, that the following formula (the base case) is unsatisfiable:

$$\mathbf{I}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{T}(s_{k-1}, s_k) \wedge (\overline{\mathbf{P}(s_1)} \vee \cdots \vee \overline{\mathbf{P}(s_k)}) \tag{1}$$

Secondly, one must show that whenever $\mathbf{P}$ holds in k consecutive states $s_1, \ldots, s_k$, $\mathbf{P}$ also holds in the next state s_{k+1} of the system. This is established by checking that the following formula (the step case) is unsatisfiable:

$$\mathbf{P}(s_1) \wedge \mathbf{T}(s_1, s_2) \wedge \cdots \wedge \mathbf{P}(s_k) \wedge \mathbf{T}(s_k, s_{k+1}) \wedge \overline{\mathbf{P}(s_{k+1})} \tag{2}$$

In prior work [13] we investigated a direct lifting of k-induction from transition systems to the level of program loops. We refer to the technique of [13] as *split-case* k-induction, as it follows the transition system approach of splitting verification into a base case and step case. Split-case k-induction is applied to a single loop in a program. In the simplest case, no loop invariant is externally provided. Instead, assertions appearing directly in the loop body take the role of an invariant. Given a CFG containing a loop, two programs are derived; we illustrate these for our running example in Figure 2 with $k = 3$. The *base case program* (Figure 2(a)) checks that no assertion can be violated within k loop iterations. This is analogous to Equation 1 above. The *step case program* (Figure 2(b)) is analogous to Equation 2. It checks whether, after executing the loop body successfully k times from an arbitrary state, a further loop iteration can be successfully executed. In this further loop iteration, back edges to the loop header are removed, while edges that exit the loop are preserved. Thus the step case verifies that on loop exit, the rest of the program can be safely executed.


```
x:=0;
i:=0;
a:=1;
b:=2;
c:=3;

assume i < n;
assert a ≠ b;
a,b,c:=b,c,a;
i++;

assume i ≥ n;
assert x = 0;
```

```
x,i,a,b,c:=*;

assume i < n;
assume a ≠ b;
a,b,c:=b,c,a;
i++;

assume i ≥ n;
assert x = 0;
```

(a) Base case (b) Step case

Fig. 2. Split-case k-induction, with $k = 3$

Correctness of both base and step case implies correctness of the whole program. On the other hand, an incorrect base case indicates an error; an incorrect step case might either indicate an error or a failure of k-induction to prove the program correct with the current value of k (which is, in fact, the case for the step case pictured in Figure 2(b)).

In a program with multiple loops, applying split-case k-induction to one loop may lead to a base and step case that each contain loops. In this case, the splitting procedure can be applied recursively until loop-free CFGs are obtained, whose verification conditions can be discharged to a prover. This may lead to an exponential number of cases to be checked [14]. Although this offers the potential for parallel verification on a multicore machine, early experiments in this vein do not show promising results [14].

Compared with the inductive invariant approach, split-case k-induction has the advantage that verification may succeed using weaker loop invariants. The assertion $a \neq b$ in Figure 1(a) can be established using split-case k-induction as shown in Figure 2 by taking $k \geq 3$: unlike the inductive invariant approach, no external invariant (like $a \neq b \wedge b \neq c \wedge c \neq a$) is required. However, split-case k-induction has the disadvantage that in the step case (Figure 2(b)), information about the values of variables not occurring in the loop is entirely lost. Although the variable x in the example is not modified in the loop, proving the assertion $x = 0$ after the loop is beyond the reach of split-case k-induction. For split-case k-induction to succeed on this example, an invariant like $x = 0$ must be added to the loop body as an assertion. In contrast, with the inductive invariant approach, the fact that x is assigned to zero before the loop is preserved by the loop cutting process.

Our contribution: combined-case k-induction. In this paper, we present *combined-case* k-induction, which brings together the strengths of split-case k-induction and the inductive invariant approach. Like the inductive invariant approach, combined-case k-induction works by cutting loops in the input CFG one at a time, resulting in a single program that needs to be checked, but like split-case k-induction, no external invariant is required.

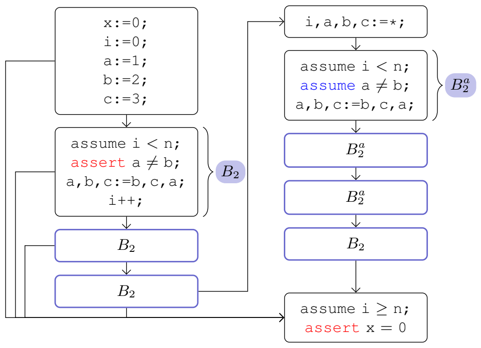


Fig. 3. k-Induction using a single check

A non-negative integer k_L is associated with each loop L in the input CFG. Loop L is then k_L*-cut* by replacing it with: k_L copies of the loop body, statements havocking all loop-modified variables, and k_L copies of the loop body where all assertions are replaced with assumptions and edges exiting the loop are removed. The last of the "assume" copies of the loop body is followed by a regular copy of the loop body, in which back edges to the loop header are removed.

Figure 3 illustrates combined-case k-induction applied to the example CFG of Figure 1(a); the single loop has been 3-cut. Comparing Figure 3 with Figure 2, observe that the base and step cases of Figure 2 are essentially merged in Figure 3. There is one key difference: variable x, which is not modified by the loop of Figure 1(a), is *not* havocked in Figure 3. Thus, unlike with split-case k-induction, we do not lose the information that the variable always retains its original value. Furthermore, because combined-case k-induction does not split verification into multiple cases, the exponential case explosion associated with split-case k-induction is completely avoided.

Summary of overview. Our novel technique, combined-case k-induction, builds on the strengths of the inductive invariant and split-case k-induction approaches to program verification. With combined-case k-induction, the program of Figure 1(a), which is beyond the reach of split-case k-induction, can be directly verified with $k \geq 3$. Unlike with the inductive invariant approach, no external invariant is required. Our experimental results in §6 demonstrate that combined-case k-induction frequently makes verification possible with weaker invariants than are otherwise required.

We are now ready to formally present our contribution.

3 Control flow graphs and loops

We present our results in terms of control flow graphs, which minimal, but general enough to uniformly translate imperative programs where procedure calls are either inlined, or replaced with pre- and post-conditions. In the diagrams of §2 we presented CFGs whose nodes are basic blocks. For ease of formal presentation, from this point on we consider CFGs whose nodes are single statements.

Let X be a set of integer variables, and let Expr be the set of all integer and boolean expressions over X, using standard arithmetic and boolean operations. The set Stmt of statements over X covers nondeterministic assignments, assumptions, and assertions:

$$\mathit{Stmt} = \{x := * \mid x \in X\} \cup \{\mathit{assume}\ \phi \mid \phi \in \mathit{Expr}\} \cup \{\mathit{assert}\ \phi \mid \phi \in \mathit{Expr}\}.$$

Intuitively, a nondeterministic assignment $x := *$ alters the value of x arbitrarily; an assumption $\mathit{assume}\ \phi$ suspends program execution if ϕ is violated and can be used to encode conditional statements, while an assertion $\mathit{assert}\ \phi$ raises an error if ϕ is violated. Neither $\mathit{assume}\ \phi$ nor $\mathit{assert}\ \phi$ have any effect if ϕ holds. We also use $x := e$ as shorthand for ordinary assignments, which can be expressed in the syntax above via a sequence of nondeterministic assignments and assumptions.

Definition 1. *A control flow graph (CFG) is a tuple* $(V, \mathit{in}, E, \mathit{code})$*, where* V *is a finite set of nodes,* $\mathit{in} \in V$ *an initial node,* $E \subseteq V \times V$ *a set of edges, and* $\mathit{code} : V \to \mathit{Stmt}$ *a mapping from nodes to statements.*

Loops and reducibility. We briefly recap notions of dominance, reducibility, and natural loops in CFGs, which are standard in the compilers literature [1].

Let $C = (V, \mathit{in}, E, \mathit{code})$ be a CFG. For $u, v \in V$, we say that u *dominates* v if $u = v$, or if every path from in to v must pass through u. Edge $(u, v) \in E$ is a *back edge* if v dominates u. The *natural loop* associated with back edge (u, v) is the smallest set $L_{(u,v)} \subseteq V$ satisfying $u, v \in L_{(u,v)}$, and $(u', v') \in E \wedge v' \in L_{(u,v)} \setminus \{v\} \Rightarrow u' \in L_{(u,v)}$. For a node v such that there exists a back edge $(u, v) \in E$, the natural loop associated with v is the set $L_v = \bigcup_{u \in V, (u,v) \text{ is a back edge}} L_{(u,v)}$. Node v is the *header* of loop L_v.

For an arbitrary loop L, $\mathit{modified}(L)$ denotes the set of variables that may be modified by nodes in L. Formally, $\mathit{modified}(L) = \{x \in X \mid \exists l \in L \ .\ \mathit{code}(l) = `x := *'\}$.[4]

In a *reducible* CFG, the only edges inducing cycles are back edges. More formally, C is reducible if the CFG $C' = (V, \mathit{in}, \mathit{FE}, \mathit{code})$ is acyclic, where FE is the set $\{(u, v) \in E \mid (u, v) \text{ is not a back edge}\}$ of *forward edges*; otherwise, C is *irreducible*.

In the remainder of the paper, we assume that all CFGs are reducible. This ensures that every cycle in a CFG is part of a loop, and allows our k-induction method to work recursively, unwinding loops one-by-one until a loop-free CFG is obtained. This is not a severe restriction: structured programming techniques guarantee reducibility, and standard (though expensive) techniques exist for transforming irreducible CFGs into reducible ones [1].

Semantics of control flow graphs. Semantically, a CFG denotes a set of execution traces, which are defined by first unwinding CFGs to prefix-closed sets of statement sequences. Subsequently, statements and statement sequences are interpreted as operations on program states.

Definition 2. *Let* $C = (V, \mathit{in}, E, \mathit{code})$ *be a CFG. The* unwinding *of* C *is defined as:*

$$\mathit{unwinding}(C) = \left\{ \begin{array}{c} \langle \mathit{code}(v_1), \ldots, \mathit{code}(v_n) \rangle \ \mid n > 0 \ \wedge \ v_1 \in \mathit{in} \ \wedge \\ \forall i \in \{1, \ldots, n-1\}.\ (v_i, v_{i+1}) \in E \end{array} \right\} \cup \{\epsilon\} \subseteq \mathit{Stmt}^*$$

[4] In practice, $\mathit{modified}(L)$ could be computed more precisely, *e.g.* disregarding assignments in dead code. For a language with pointers, $\mathit{modified}(L)$ is computed with respect to an alias analysis, in the obvious way.

where ϵ denotes the empty sequence.

A *non-error state* is a store mapping variables to values in some domain $\mathcal{D}$. The set of program states for a CFG over X is the set of all stores, together with a designated error state: $S = \{\sigma \mid \sigma : X \to \mathcal{D}\} \cup \{\lightning\}$.

We give trace semantics to CFGs by first defining the effect of a statement on a program state. This is given by the function $post : S \times \mathit{Stmt} \to 2^S$ defined as follows:

$$post(\lightning, s) \;=\; \{\lightning\} \qquad \text{(for any statement } s\text{)}$$

For non-error states $\sigma \neq \lightning$:

$$\begin{aligned}
post(\sigma, x := *) &= \{\sigma' \mid \sigma'(y) = \sigma(y) \text{ for all } y \neq x\} \\
post(\sigma, \mathit{assume}\ \phi) &= \big(\text{if } \phi^\sigma = \mathit{tt} \text{ then } \{\sigma\}, \text{ otherwise } \emptyset\big) \\
post(\sigma, \mathit{assert}\ \phi) &= \big(\text{if } \phi^\sigma = \mathit{tt} \text{ then } \{\sigma\}, \text{ otherwise } \{\lightning\}\big)
\end{aligned}$$

The function $post$ is lifted to the evaluation function $\mathit{traces} : S \times \mathit{Stmt}^* \to 2^{S^*}$ on statement sequences as follows:

$$\begin{aligned}
\mathit{traces}(\sigma, s) &= \{\langle \sigma, \sigma' \rangle \mid \sigma' \in post(\sigma, s)\} \\
\mathit{traces}(\sigma, \langle s_1, \ldots, s_n \rangle) &= \{\sigma.\tau \mid \exists \sigma'.\ \sigma' \in post(\sigma, s_1) \wedge \tau \in \mathit{traces}(\sigma', \langle s_2, \ldots, s_n \rangle)\}
\end{aligned}$$

Here, for a state $\sigma \in S$ and state tuple $\tau \in S^m$, $\sigma.\tau \in S^{m+1}$ is the concatenation of σ and τ. The set of traces of a CFG C is the union of the traces for any of its paths:

$$\mathit{traces}(C) \;=\; \bigcup \{\mathit{traces}(\sigma, p) \mid \sigma \in S \setminus \{\lightning\} \wedge p \in \mathit{unwinding}(C)\}.$$

Note that there are no traces along which *assume*-statements fail.

We say that CFG C is *correct* if $\lightning$ does not appear on any trace in $\mathit{traces}(C)$. Otherwise C is not correct, and a trace in $\mathit{traces}(C)$ which leads to $\lightning$ is a counterexample to correctness.

4 Proof rule and verification algorithm

Given a CFG C containing a loop L, and a positive integer k, we shall define a k-induction rule that transforms C into a CFG C_k^L in which loop L is eliminated via k-cutting, such that correctness of C_k^L implies correctness of C. We start by motivating the use of the rule for verification by considering the procedure ANALYSE of Algorithm 1.

ANALYSE attempts to prove correctness of C by applying the k-induction rule recursively. At each step, a loop in the CFG, and a corresponding value of k is chosen. The loop is eliminated from the CFG by k-cutting. If the result is a loop-free CFG, correctness is checked by an appropriate decision procedure (*e.g.* an SMT solver). Otherwise, the process continues with the selection of another loop. If a k-cut CFG is not found to be correct (a recursive call to ANALYSE returns DON'T KNOW) then the procedure either returns an inconclusive result, or backtracks and applies k-induction to a different loop, and/or using a different value for k.

Note that ANALYSE cannot be used to determine that a program is incorrect. It could be modified to do so, by explicitly marking those portions of a k-cut CFG in which an error signifies a genuine bug; alternatively, ANALYSE can simply be executed in parallel with bounded model checking [6].

Algorithm 1: ANALYSE

```
Input: Reducible CFG $C = (V, in, E, code)$.
Output: One of $\{$CORRECT, DON'T KNOW$\}$
if $C$ is loop-free then
    if DECIDE($C$) = CORRECT then                // Program is correct
        return CORRECT;
    else                                         // Correctness not determined
        return DON'T KNOW;
    end
else                                             // apply the $k$-induction rule
(*)     choose loop $L$ in $C$ and depth $k \in \mathbb{N}$;
    result $\longleftarrow$ ANALYSE($C_k^L$);
    if result = DON'T KNOW then                  // $k$-induction was inconclusive
(**)        either back-track to (*), or return DON'T KNOW;
    else                                         // $k$-induction was conclusive
        return result;
    end
end
```

4.1 Graphical description of k-induction proof rule

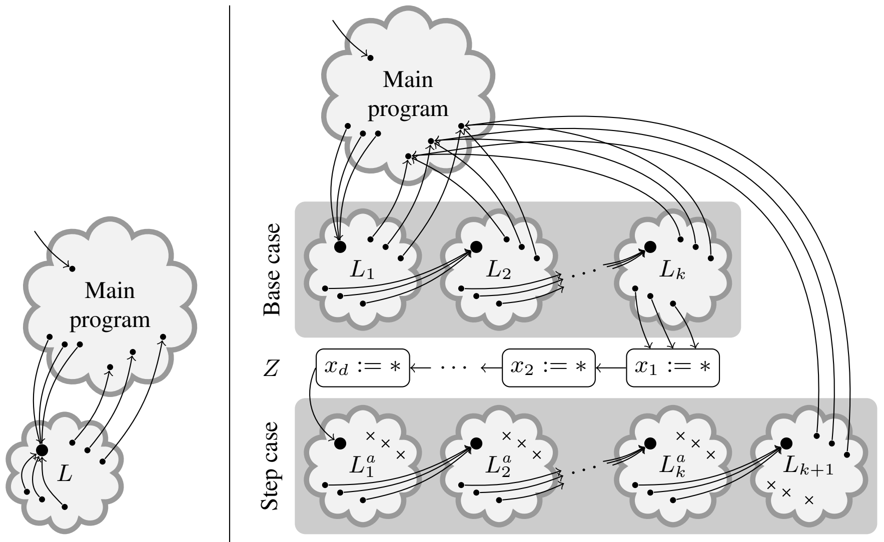
Figure 4(a) depicts an arbitrary CFG C that contains at least one loop, L. The CFG is separated into a loop L (the smaller cloud), and the set of nodes outside L (the cloud labelled "Main program"). The main program may contain further loops, and L may contain nested loops. We assume that entry to the CFG, indicated by the edge into "Main program", is not via L. The program can be re-written to enforce this, if necessary.

Loop L has a single entry point, or *header*, indicated by the large dot in Figure 4(a). There are edges from at least one (and possibly multiple) node(s) in the main program to this header. Inside L, there are back edges from at least one node to the header. In addition, there are zero-or-more edges that exit L, leading back to the main program.

For some unspecified $k > 0$, Figure 4(b) shows the CFG C_k^L generated by our novel k-induction rule, which we present formally in §4.2. The loop L has been k-cut, producing a CFG C_k^L with four components. The nodes outside L are labelled "Main program". Edges from the main program into L in Figure 4(a) are replaced with edges into the first of k copies of the body of L, denoted $L_1, \ldots, L_k$. These are marked "Base case" in Figure 4(b). In each L_i, edges leaving L are preserved, as are edges within L, except for back edges. For $i < k$, a back edge in L is replaced in L_i with an edge to the header node of the next copy of L, namely L_{i+1}. The base case part of C_k^L checks that the first k iterations of L can be successfully executed.

In the final copy of L appearing in the base case, L_k, back edges are replaced with edges to the sequence of nodes marked Z in Figure 4(b). Z has the effect of havocking the variables $x_1, \ldots, x_d$ that comprise $modified(L)$, the loop-modified variables for L.

The final node of Z is followed by k copies of the body of L in which all statements of the form $assert\ \phi$ are replaced with $assume\ \phi$, and all edges leaving L are removed. These modified copies of the body of L are denoted $L_1^a, \ldots, L_k^a$ (where a



(a) Program (b) Unrolled program

Fig. 4. Schematic overview of the new k-induction rule. $\mathit{modified}(L) = \{x_1, \ldots, x_d\}$ is the set of variables modified in loop L.

denotes assume), and back-edges in L are replaced in L_i^a with edges to to the header of L_{i+1}^a, for $i < k$. In L_k^a, back edges are replaced with edges to L_{k+1}. This is a final copy of the body of L, where assertions are left intact, edges leaving L are preserved, and back-edges are removed. The fragments $L_1^a, \ldots, L_k^a$ and L_{k+1} are denoted "Step case" in Figure 4(b). Together with the Z nodes, they check that, from an arbitrary loop entry state, assuming that k iterations of L have succeeded, a further iteration, followed by execution of the main program, will succeed.

It may be instructive to compare the abstract program of Figure 4(a), and corresponding k-cut program of Figure 4(b), with the program of Figure 1(a) and 3-cut program of Figure 3. Loop L of Figure 4(a) corresponds to B_2 in Figure 1(a). Components $L_1, \ldots, L_k$ in Figure 4(b) correspond to the three copies of B_2 on the left of Figure 3, $L_1^a, \ldots, L_k^a$ to the three copies of B_2^a on the right of Figure 3, and L_{k+1} to the additional copy of B_2 on the right of Figure 3. Finally, the Z nodes of Figure 4(b) are reflected by the statement $i, a, b, c := *$ in Figure 3.

4.2 Formal definition of k-induction proof rule

We now formally define our novel k-induction rule as a transformation rule on control flow graphs. To aid understanding, our formal definition uses the same notation as presented in Figure 4,

Let $C = (V, \mathit{in}, E, \mathit{code})$ be a CFG and $L \subseteq V$ a loop in C with header h. Assume that $\mathit{in} \notin L$. (This can be trivially enforced by adding an $\mathit{assume}\ \mathit{tt}$ node to C if necessary.) We present a k-induction proof rule for *positive* values of k, under the assumption that $\mathit{modified}(L)$, the set of variables that may be modified by loop L, is non-empty. Extending the definition, and all the results presented in this paper, to allow

$k = 0$, and $\mathit{modified}(L) = \emptyset$, is trivial, and the implementations we describe in §6 incorporate such extensions. However, a full presentation involves considering pedantic corner cases which make the essential concepts harder to follow without providing further insights into our work.

Thus, let $k > 0$, and suppose $\mathit{modified}(L) = \{x_1, \dots, x_d\}$ for some $d > 0$. For $1 \leq i \leq k+1$, define $L_i = \{v_i \mid v \in L\}$. Thus L_i is a duplicate of L where each node is subscripted by i. Similarly, for $1 \leq i \leq k$, define $L_i^a = \{v_i^a \mid v \in L\}$. Thus L_i^a is a duplicate of L where each node is subscripted by i and superscripted by a. Let $Z = \{z_1^h, \dots, z_d^h\}$ be a fresh set of nodes, distinct from L, L_i $(1 \leq i \leq k+1)$ and L_i^a $(1 \leq i \leq k)$.

Definition 3. *$C_k^L = (V_k^L, \mathit{in}_k^L, E_k^L, \mathit{code}_k^L)$ is defined as follows:*

$V_k^L = (V \setminus L) \cup \bigcup_{i=1}^{k+1} L_i \cup \bigcup_{i=1}^{k} L_i^a \cup Z$

$\mathit{in}_k^L = \mathit{in}$ *(recall that, by assumption, $\mathit{in} \notin L$)*

$E_k^L =$

$\{\ (u,v)$	$\mid (u,v) \in E \wedge u,v \notin L\ \}$	*Edges in* Main program
$\cup\ \{\ (u,h_1)$	$\mid (u,h) \in E \wedge u \notin L\ \}$	Main program $\rightarrow L_1$
$\cup\ \{\ (u_i,v_i)$	$\mid 1 \leq i \leq k+1 \wedge (u,v) \in E \wedge u,v \in L \wedge v \neq h\ \}$	*Edges in* L_i
$\cup\ \{\ (u_i^a,v_i^a)$	$\mid 1 \leq i \leq k \wedge (u,v) \in E \wedge u,v \in L \wedge v \neq h\ \}$	*Edges in* L_i^a
$\cup\ \{\ (u_i,h_{i+1})$	$\mid 1 \leq i < k \wedge (u,h) \in E \wedge u \in L\ \}$	$L_i \rightarrow L_{i+1}$ $(i < k)$
$\cup\ \{\ (u_i^a,h_{i+1}^a)$	$\mid 1 \leq i < k \wedge (u,h) \in E \wedge u \in L\ \}$	$L_i^a \rightarrow L_{i+1}^a$ $(i < k)$
$\cup\ \{\ (u_i,v)$	$\mid 1 \leq i \leq k+1 \wedge (u,v) \in E \wedge u \in L \wedge v \notin L\ \}$	$L_i \rightarrow$ Main program
$\cup\ \{\ (u_k,z_1^h)$	$\mid (u,h) \in E \wedge u \in L\ \}$	$L_k \rightarrow Z$
$\cup\ \{\ (z_i^h,z_{i+1}^h)$	$\mid 1 \leq i < d\ \}$	*Edges in* Z
$\cup\ \{\ (z_d^h,h_1^a)\ \}$		$Z \rightarrow L_1^a$

$$\mathit{code}_k^L(z_i^h) = \text{‘}x_i := *\text{’} \qquad (1 \leq i \leq d)$$

$$\mathit{code}_k^L(v_i^a) = \begin{cases} \mathit{assume}\ \phi & \text{if } \mathit{code}(v) = \mathit{assert}\ \phi \\ \mathit{code}(v) & \text{otherwise} \end{cases} \qquad (1 \leq i \leq k)$$

$$\mathit{code}_k^L(v_i) = \mathit{code}(v) \qquad (1 \leq i \leq k+1)$$

$$\mathit{code}_k^L(v) = \mathit{code}(v) \qquad \text{for } v \in V_k^L \cap V$$

Theorem 1 (Soundness). *If C_k^L is correct then C is correct.*

The proof of Theorem 1 is presented in the appendix.

5 Theoretical properties of the k-induction rule

5.1 Confluence

We now turn to the question of confluence: for fixed values of k, does it matter in which order the loops of a CFG are processed when recursively applying the k-induction rule?

First, we define what it means for CFGs to be equivalent.

Definition 4. *Let $C = (V, \mathit{in}, E, \mathit{code})$ and $C' = (V', \mathit{in}', E', \mathit{code}')$ be CFGs. A bijection $\alpha : V \rightarrow V'$ is an* isomorphism *between C and C' if $\alpha(\mathit{in}) = \mathit{in}'$ and, for all $u, v \in V$, $\mathit{code}(u) = \mathit{code}'(\alpha(u))$, and $(u,v) \in E \Leftrightarrow (\alpha(u), \alpha(v)) \in E'$. If there exists an isomorphism between C and C', we say that C and C' are* isomorphic.

We say that CFGs C and C' are *equivalent*, and write $C \equiv C'$, if they are isomorphic. It is easy to show that $\equiv$ is indeed an equivalence relation.

The following result follows directly from the definition of a natural loop:

Lemma 1. *Let L and M be distinct loops in CFG C. Then either $L \cap M = \emptyset$, $L \subset M$ or $M \subset L$.*

In what follows, C denotes a CFG.

Lemma 2 (Confluence of k-induction rule for disjoint loops). *Let L and M be disjoint loops in C, and let k_L and k_M be positive integers. Then $(C^L_{k_L})^M_{k_M} = (C^M_{k_M})^L_{k_L}$.*

Lemma 2 shows that, for disjoint loops, the order in which k-induction is applied to each loop is irrelevant; an identical CFG always results. (Note that the CFGs are truly identical, not merely equivalent.) Thus, for mutually disjoint loops $L_1, \ldots, L_d$ in a CFG C, and positive integers $k_1, \ldots, k_d$, we can unambiguously write $C^{L_1,\ldots,L_d}_{k_1,\ldots,k_d}$ to denote the CFG obtained by applying the k-induction rule d times, on each application eliminating one of the loops L_i according to k_i.

Now consider loops $L \subset M$ of C, and positive integers k_L and k_M.

The CFG $C^M_{k_M}$ contains $k_M + 1$ direct copies of L, and k_M copies of L in which all assertions are replaced with assumptions. This is because L forms part of the body of M. Let us denote these copies of L by $L_1, \ldots, L_{k_M+1}$ and $L^a_1, \ldots, L^a_{k_M}$ respectively. Def. 3 ensures that they are all disjoint in $C^M_{k_M}$.

The CFG $C^L_{k_L}$ contains a loop M' identical to M, except that L has been eliminated from the body of M', and replaced with an unwinding of L proportional to k_L.

Lemma 3 (Confluence of k-induction rule for nested loops). *Let $L \subset M$ be loops of C, and k_L and k_M positive integers. Using the above notation, we have:*

$$(C^L_{k_L})^{M'}_{k_M} \equiv (C^M_{k_M})^{L_1,\ldots,L_{k+1},L^a_1,\ldots,L^a_k}_{k_L,\ldots\ldots\ldots\ldots\ldots\ldots,k_L}.$$

We now show that if we repeatedly apply the k-induction rule to obtain a loop-free CFG, then as long as a value for k is used consistently for each loop in C, the the order in which the k-induction rule is applied to loops is irrelevant.

We assume a map, *origin* which, given any CFG D derived from C by zero-or-more applications of the k-induction rule, and a loop L of D, tells us the original loop in C to which L corresponds. For example, given loops $L \subset M \subset N$ in C and positive integers k_L, k_M, k_N, CFG $C^N_{k_N}$ contains many duplicates of L and M, including loops $L_1 \subset M_1$. In turn, CFG $(C^N_{k_N})^{M_1}_{k_M}$ contains many duplicates of L_1, including L_{1_1}. We have $\mathit{origin}(L_{1_1}) = \mathit{origin}(L_1) = \mathit{origin}(L) = L$, $\mathit{origin}(M_1) = \mathit{origin}(M) = M$, and $\mathit{origin}(N) = N$. Also, CFG $C^L_{k_L}$ includes loops $M' \subset N'$ identical to M and N, except that L has been unrolled. We have $\mathit{origin}(M') = M$ and $\mathit{origin}(N') = N$.

Definition 5. *Let* $\mathbf{k}$: (loops of C) $\rightarrow \mathbb{N}$ *associate a positive integer with each loop of C. For $i \geq 0$, let P_i be the set of all CFGs that can be derived from C by* exactly *i applications of the k-induction rule, together with all loop-free CFGs that can be derived from C by* up to *i applications of the k-induction rule. In all applications of the rule, k is chosen according to the mapping* $\mathbf{k}$.

The sequence (P_i) is defined by $P_0 = \{C\}$ and

$$\begin{aligned} P_i = \{&D^L_{\mathbf{k}(origin(L))} \mid D \in P_{i-1} \wedge L \textit{ is a loop of } D\} \cup \qquad (\textit{for } i > 0) \\ \{&D \in P_{i-1} \quad \mid D \textit{ is loop free }\} \ . \end{aligned}$$

Theorem 2 (Global confluence). *There is an integer n such that $P_m = P_n$ for all $m \geq n$. All the CFGs in P_n are equivalent, and loop-free.*

To prove this theorem, we first consider the special case of *outward* application of the k-induction rule, which means that only loops are unrolled that no longer contain inner loops. Since this strategy guarantees that the total number of loops decreases by exactly 1 each time k-induction is applied, a CFG C containing l loops is reduced to a loop-free CFG by l applications of k-induction, regardless of which (innermost) loop is chosen in each step. In fact, all such application sequences lead to the same CFG:

Lemma 4. *Suppose the sequence (Q_i) is defined by $Q_0 = \{C\}$ and*

$$Q_i = \{D^L_{\mathbf{k}(origin(L))} \mid D \in Q_{i-1}, L \subset D \textit{ a loop without inner loops}\}$$

for $i \in \{1, \ldots, l\}$. Then Q_l is a singleton set containing a loop-free CFG.

A proof is given in the appendix. Theorem 2 can then be proven as follows:

Proof (Theorem 2). Suppose an arbitrary sequence of applications of the k-induction rule to the CFG C to the loops $L_1, L_2, \ldots, L_m$, finally resulting in a loop-free CFG. Let furthermore L_u be the last loop in the sequence containing an inner loop M; this means that the suffix $L_{u+1}, \ldots, L_m$ represents outward application of k-induction. We can then first reorder the suffix with the help of Lemma 4 so that all occurrences of M are eliminated in the beginning of the suffix, and then apply Lemma 3 to swap the k-induction applications to L_u and to M. Iterating this procedure will eventually reduce $L_1, L_2, \ldots, L_m$ to an outward application sequence of k-induction, so that finally Lemma 4 applies. □

5.2 Fair enumeration of values of *k*

We observe that larger values of k can only make C^L_k simpler to verify, in the sense that if C^L_k is correct, then $C^L_{k'}$ is also correct for all $k' > k$. A similar observation applies to programs with multiple loops. A natural strategy to choose k at $(*)$ in Algorithm 1 is therefore to start with small values of k, and to iteratively increment k as long as DON'T KNOW occurs at $(**)$. We call such an enumeration strategy *fair* if, for any $n \in \mathbb{N}$, eventually a run of ANALYSE occurs in which a $k \geq n$ is chosen for each loop of the input program. Due to Theorem 2, it can be concluded that if *any* run of ANALYSE exists that yields the result CORRECT, then also ANALYSE driven by a fair enumeration strategy will eventually return CORRECT.

5.3 Size of loop-free programs produced by *k*-induction

Since the program C^L_k obtained via a single application of the k-induction rule contains $2k + 1$ copies of the loop L, repeated application can increase the size of a program exponentially. Such exponential growth can only occur in the presence of nested loops, however, because k-induction leaves programs parts outside of the eliminated loop L unchanged. By a simple complexity analysis, we can derive that the size of loop-free programs derived though repeated application of k-induction is (singly) exponential in the depth of deepest loop nest in the worst case, but only linear in the number of disjoint loops. This implies that the size of generated programs, in practical cases, is not a bottleneck of the combined-case k-induction method.

6 Experimental evaluation

We have implemented our techniques in two tools. K-BOOGIE is an extension of the BOOGIE verifier, and allows programs written in the BOOGIE language to be verified using k-induction. As BOOGIE is an intermediate language for verification, K-BOOGIE can be applied to programs originating from several different languages, including Spec# [4], Dafny [19], Chalice, VCC, and Havoc. K-INDUCTOR is a k-induction-based verifier for C programs, built on top of the bounded model checker CBMC [9].[5]

Experiments with K-BOOGIE. We apply K-BOOGIE to a set of 26 Boogie programs, the majority of which were machine-generated from (hand-written) Dafny programs included in the Boogie distribution. Most of the programs verify functional correctness of standard algorithms, including sophisticated procedures such as the Schorr-Waite graph marking algorithm. The Boogie programs contain altogether 40 procedures with loops annotated with (possibly multiple) loop invariants, and were not previously known to be amenable to k-induction. Our findings are summarised in Table 1.

To evaluate the applicability of k-induction, we first eliminated loop invariants from the programs that were not necessary even with the normal Boogie induction rule. We then checked, using $k \in \{0, 1, 2, 3\}$, which of the loop invariants were unnecessary with combined-case k-induction. This was done by first trying to remove invariants individually, keeping all other invariants of a procedure (**# removable**, where the number of removable invariants, the total number of invariants). As second step, we determined maximum sets of invariants that could be removed simultaneously (**# sim. removable**). In both cases, we show largest value of k required for invariant removal, over all loops (**required** k). For each procedure, we show the number of lines of executable code (**LOC**) and the number of loops (**#loops**). We also show the total number of lines for each program (**LOC program**), including all procedures and additional definitions (which can be quite considerable). For all but 11 of the procedures, spread over 22 of the 26 programs, we find that, with 1- or 2-induction, we are able to remove invariants that are necessary for the normal Boogie loop rule ($k = 3$ did not allow us to remove further invariants for any of the programs). This illustrates that k-induction, with small values of k, can be useful for general-purpose verification. As the required verification times with k-induction did not differ significantly from those with the normal Boogie rule for most procedures, we do not report detailed times.

Experiments with K-INDUCTOR. We apply K-INDUCTOR to a set of benchmarks from the domain of direct memory access (DMA) race checking, studied in [13] (in

[5] Both tools, and all our benchmarks, are available online: **http://www.cprover.org/kinduction**.

Procedure	# removable, required k	# sim. removable, required k	LOC/ #loops	LOC program
Procedures generated from Dafny programs				
VSI-b1.Add	2/4, 1	2/4, 1	114/2	710
VSI-b2.BinarySearch	0/5, 1		100/1	595
VSI-b3.Sort	1/16, 1	1/16, 1	186/2	798
VSI-b3.RemoveMin	1/6, 1	1/6, 1	176/2	
VSI-b4.Map.FindIndex	3/4, 2	2/4, 1	84/1	956
VSI-b6.Client.Main	1/3, 1	1/3, 1	139/1	900
VSI-b8.Glossary.Main	4/16, 1	3/16, 1	381/3	1998
VSI-b8.Glossary.readDef	0/1, 1		71/1	
VSI-b8.Map.FindIndex	0/1, 1		66/1	
Composite.Adjust	1/3, 2	1/3, 2	80/1	1275
LazyInitArray	1/5, 1	1/5, 1	165/1	806
SchorrWaite.RecursiveMark	0/6, 1		98/1	1175
SchorrWaite.IterativeMark	2/17, 1	2/17, 1	177/1	
SchorrWaite.Main	4/27, 1	3/27, 1	275/1	
SumOfCubes.Lemma0	1/2, 1	1/2, 1	81/1	915
SumOfCubes.Lemma1	1/2, 1	1/2, 1	65/1	
SumOfCubes.Lemma2	1/2, 1	1/2, 1	48/1	
SumOfCubes.Lemma3	1/2, 1	1/2, 1	51/1	
Substitution	0/1, 1		131/1	846
PriorityQueue.SiftUp	1/2, 2	1/2, 2	92/1	819
PriorityQueue.SiftDown	1/2, 2	1/2, 2	101/1	
MatrixFun.MirrorImage	2/6, 1	2/6, 1	125/2	922
MatrixFun.Flip	1/3, 1	1/3, 1	103/1	
ListReverse	2/3, 2	2/3, 2	71/1	329
ListCopy	1/4, 1	1/4, 1	141/1	434
ListContents	1/3, 1	1/3, 1	141/1	717
Cubes	3/4, 2	2/4, 2	97/1	339
Celebrity.FindCelebrity1	1/1, 2	1/1, 2	98/1	795
Celebrity.FindCelebrity2	0/1, 1		99/1	
Celebrity.FindCelebrity3	0/2, 1		86/1	
VSC-SumMax	1/2, 1	1/2, 1	77/1	458
VSC-Invert	0/1, 1		61/1	568
VSC-FindZero	1/2, 1	1/2, 1	90/1	625
VSC-Queens.CConsistent	0/3, 1		79/1	825
VSC-Queens.SearchAux	0/1, 1		139/1	
Native Boogie programs				
StructuredLocking	1/1, 1	1/1, 1	16/1	40
StructuredLockingWithCalls	0/1, 1		13/1	
Structured.RunOffEnd1	1/1, 1	1/1, 1	12/1	53
BubbleSort	7/14, 1	7/14, 1	33/3	42
DutchFlag	1/5, 1	1/5, 1	29/1	37

Table 1. Experimental results applying K-BOOGIE to Dafny and Boogie benchmarks included in the Boogie distribution.

Benchmark	LOC/#loops	min/max k split	min/max k combined	# invariants split	speedup
1-buf	151/2	1/1	0/1	3	1.4
1-buf I/O	178/2	1/1	0/1	5	1.5
2-buf	254/3	1/2	0/2	17	2.6
2-buf I/O	304/3	1/2	0/2	29	3.9
3-buf	282/4	1/3	0/3	27	9.4
3-buf I/O	364/4	1/3	0/3	38	8.3
Euler simple	101/3	1/2	0/2	10	1.1
sync atomic op	91/3	1/1	0/1	4	2.3
sync mutex	83/2	1/1	0/1	2	3.1

Table 2. Experimental results applying K-INDUCTOR to DMA processing benchmarks.

which full details can be found). These consist of data processing programs for the Cell BE processor, where data is manipulated using DMA. In [13], split-case k-induction is applied to these benchmarks, under the simplifying assumption that in many cases inner loops unrelated to DMA are manually sliced away, leaving single-loop programs. We find that combined-case k-induction allows us to handle inner loops in these benchmarks directly. With split-case k-induction, handling inner loops requires the addition of numerous invariants, as assertions in the program text.

For each DMA processing benchmark, Table 2 shows the number of lines of code (**LOC**) and the number of loops processed by k-inductor (**#loops**, this is the number of loops after function inlining, which may cause loop duplication). We then show the minimum and maximum values of k required for induction to succeed using the split-case and combined-case approaches (**min/max** k **split/combined**), the number of invariants that had to be added manually for split-case k-induction to work (**#invariants split**), and the speedup obtained by using combined-case k-induction over split-case k-induction (**speedup**). Experiments are performed on a 3GHz Intel Xeon machine with 40 GB RAM, running 64-bit Linux. MiniSat 2 is used as a back-end SAT solver for CBMC. Manually specified invariants are mainly simple facts related to the ranges of variables; many could be inferred automatically using abstract interpretation.

The results show that combined-case k-induction avoids the need for a significant number of additional invariants when verifying these examples. This allows many inner loops that are unrelated to DMA processing (and thus do not contain assertions of interest) to be handled using $k = 0$. We also find that combined-case k-induction is uniformly, and sometimes significantly faster than split-case k-induction. We attribute this to the multiple loop-free programs that must be solved with split-case k-induction, compared with the single loop-free program associated with combined-case k-induction.

7 Related work and conclusions

The concept of k-induction was first published in [21, 7], targeting the verification of hardware designs represented by transition relations (although the basic idea had already been used in earlier implementations [20] and a version of one-induction used for BDD-based model checking [10]). A major emphasis of these two papers is on the restriction to loop-free or shortest paths, which is so far not considered in our k-induction rule due to the size of state vectors and the high degree of determinism in software programs. Several optimisations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [22], improving performance via incremental SAT solving [15], and verification of temporal properties [2].

Besides hardware verification, k-induction has been used to analyse synchronous programs [18, 17]. To the best of our knowledge, the first application of k-induction to imperative software programs was done in the context of DMA race checking [13], from which we also draw some of the benchmarks used in this paper. A combination of the k-induction rule of [13], abstract interpretation, and domain-specific invariant strengthening techniques for DMA race analysis is the topic of [12].

We have presented combined-case k-induction, demonstrated experimentally that it can allow verification to succeed using weaker loop invariants than are required with either split-case k-induction or the inductive invariant approach, and that it can significantly out-perform split-case k-induction. As future work, we plan to extend this comparison by implementing split-case k-induction in Boogie. We also plan to investigate techniques for applying K-INDUCTOR to the verification of heap-manipulating C programs.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2006)
2. Armoni, R., Fix, L., Fraer, R., Huddleston, S., Piterman, N., Vardi, M.Y.: SAT-based induction for temporal safety properties. Electr. Notes Theor. Comput. Sci. 119(2), 3–16 (2005)
3. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In *PASTE* (2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In *CASSIS* (2005)
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach, LNCS, vol. 4334. Springer (2007)
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
7. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In *FMCAD* (2000)
8. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. 20(4-5), 379–405 (2008)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: *TACAS* (2004)
10. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In *CHARME* (1997)
11. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In *OOPSLA* (2008)
12. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In *VMCAI* (2011)
13. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS* (2010)
14. Donaldson, A.F., Rümmer, P., Kroening, D.: Split-case k-induction for program verification. Tech. rep. (2011), available: www.allydonaldson.co.uk/papers/DonaldsonKR_TR2011.html
15. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
16. Floyd, R.: Assigning meaning to programs. In: Proceedings of Symposium on Applied Mathematics. pp. 19– 32 (1967)
17. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. 144(1), 19–33 (2006)
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In *FMCAD* (2008)
19. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In *LPAR* (2010)
20. Lillieroth, C.J., Singh, S.: Formal verification of FPGA cores. Nord. J. Comput. 6(3), 299–319 (1999)
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In *FMCAD* (2000)
22. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In *VLSID* (2007)

Appendix

A Correctness of k-induction

A.1 Inductive decomposition of CFGs

The k-induction rule that we have presented is an instance of a more general principle, in the following called *inductive decomposition*. The rule works by transforming CFGs C into a CFG C_k^L that contains fewer (or less deeply nested) loops, in such a way that the correctness of C follows from the correctness of C_k^L. This construction is justified by proving that every statement sequence in the unwinding of C can be decomposed into sequences from the unwinding of C_k^L.

As an important part of the argument, we observe that the statement sequence $\forall x \in \mathit{modified}(L).\ x := *$ inserted in C_k^L can be expanded to arbitrary statements assigning on to variables in $\mathit{modified}(L)$. To define this more formally, we introduce the notion of *rewriting relations* on statement sequences, which are binary relations $\rightarrow \subseteq \mathcal{P}(\mathit{Stmt}^* \times \mathit{Stmt}^*)$. We will later consider in particular those relations $\rightarrow$ that relate statement sequences s with sequences t that have "less behavior" than s.

We define how to close a set of statement sequences under self-composition:

Definition 6. *Given a set $P \subseteq \mathit{Stmt}^*$ of statement sequences (over a set of variables X) and a rewriting relation $\rightarrow$, the set $\mathit{gen}_\rightarrow(P) \subseteq \mathit{Stmt}^*$ of sequences* generated by P modulo $\rightarrow$ *is the least prefix-closed set with the following properties:*

1. *$P \subseteq \mathit{gen}_\rightarrow(P)$*
2. *Closure under rewriting:*

$$\begin{array}{rl} & p.q.r \in \mathit{gen}_\rightarrow(P),\ q' \in \mathit{gen}_\rightarrow(P),\ q \rightarrow q' \\ \Longrightarrow & p.q'.r \in \mathit{gen}_\rightarrow(P) \end{array}$$

3. *Closure under composition:*

$$\begin{array}{rl} & \langle p_1, p_2, \ldots, p_n \rangle \in \mathit{gen}_\rightarrow(P),\ \langle q_1, q_2, \ldots, q_m \rangle \in \mathit{gen}_\rightarrow(P), \\ & k \in \{n - m + 2, \ldots, n + 1\}, \\ & \forall i \in \{0, \ldots, n - k\}.\ q_{i+1} \in \{p_{i+k}, p_{i+k}^{\mathit{assume}}\} \\ \Longrightarrow & \langle p_1, p_2, \ldots, p_n, q_{n-k+2}, \ldots, q_m \rangle \in \mathit{gen}_\rightarrow(P) \end{array}$$

We say that a rewriting relation $\rightarrow$ is *monotonic* if for all sequences $s \rightarrow t$ and all non-error states $\sigma \in S \setminus \{\lightning\}$ the following inclusion holds:

$$\{\sigma' \mid \langle \sigma, \ldots, \sigma' \rangle \in \mathit{traces}(\sigma, t)\} \setminus \{\lightning\} \quad \subseteq \quad \{\sigma' \mid \langle \sigma, \ldots, \sigma' \rangle \in \mathit{traces}(\sigma, s)\} \setminus \{\lightning\}$$

Lemma 5. *If $P \subseteq \mathit{Stmt}^*$ is a set of correct statement sequences and $\rightarrow$ is a monotonic rewriting relation, then $\mathit{gen}_\rightarrow(P)$ is a set of correct statement sequences.*

Proof. Since $\mathit{gen}_\rightarrow(P)$ is inductively defined as the least prefix-closed set satisfying the properties 1, 2, and 3, it is enough to show that the statement sequences generated by each of the implications are correct (observe that all prefixes of a correct statement sequence are trivially correct).

1. Since all sequences in P are correct, this rule only adds correct sequences to $gen_{\rightarrow}(P)$.
2. Suppose $p.q.r \in gen_{\rightarrow}(P)$ and $q' \in gen_{\rightarrow}(P)$ are correct, and $q \rightarrow q'$. Let $\langle \sigma_0, \sigma_1, \ldots, \sigma_l \rangle \in traces(p.q'.r)$ be a trace of $p.q'.r$; this trace can be split into parts

$$\begin{aligned} \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle &\in traces(p), \\ \langle \sigma_n, \sigma_{n+1}, \ldots, \sigma_m \rangle &\in traces(q'), \\ \langle \sigma_m, \sigma_{m+1}, \ldots, \sigma_l \rangle &\in traces(r). \end{aligned}$$

   Since q' is correct, $\sigma_m \neq \lightning$. Because also $q \rightarrow q'$, there is a corresponding trace

$$\langle \sigma_n, \sigma'_{n+1}, \sigma'_{n+2}, \ldots, \sigma'_{m'-1}, \sigma_m \rangle \in traces(q)$$

   which can be composed with the other traces to a trace of $p.q.r$:

$$\langle \sigma_0, \sigma_1, \ldots, \sigma_n, \sigma'_{n+1}, \sigma'_{n+2}, \ldots, \sigma'_{m'-1}, \sigma_m, \sigma_{m+1}, \ldots, \sigma_l \rangle \in traces(p.q.r)\ .$$

   Since $p.q.r$ is correct, this implies that $\sigma_l \neq \lightning$, and thus $\lightning$ cannot occur on the trace $\langle \sigma_0, \sigma_1, \ldots, \sigma_l \rangle$.
3. Suppose $\langle p_1, p_2, \ldots, p_n \rangle \in gen_{\rightarrow}(P)$ and $\langle q_1, q_2, \ldots, q_m \rangle \in gen_{\rightarrow}(P)$ are correct sequences of statements, $k \in \{n - m + 2, \ldots, n + 1\}$, and

$$\forall i \in \{0, \ldots, n-k\}.\ q_{i+1} \in \{p_{i+k}, p_{i+k}^{assume}\}\ .$$

   Further, let $\langle \sigma_0, \sigma_1, \ldots, \sigma_l \rangle \in traces(\langle p_1, p_2, \ldots, p_n, q_{n-k+2}, \ldots, q_m \rangle)$ be a trace of the generated statement sequence. Because the prefix $\langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ is also a trace of the correct sequence $\langle p_1, p_2, \ldots, p_n \rangle$, the error state $\lightning$ cannot occur on this prefix. Thus we know that $\langle \sigma_{k-1}, \sigma_k, \ldots, \sigma_n \rangle$ is also a trace of $\langle q_1, \ldots, q_{n-k+1} \rangle$, and therefore $\langle \sigma_{k-1}, \sigma_k, \ldots, \sigma_l \rangle$ is a trace of $\langle q_1, q_2, \ldots, q_m \rangle$. Since $\langle q_1, q_2, \ldots, q_m \rangle$ is correct, this implies that $\sigma_l \neq \lightning$, and therefore $\lightning$ cannot occur on the trace $\langle \sigma_0, \sigma_1, \ldots, \sigma_l \rangle$.

A.2 *k*-Induction by inductive decomposition

We will use inductive decompositions to prove the soundness of k-induction. To this end, we consider rewriting relations $\rightarrow_h$ generated by the following rules:

$$\frac{}{p \rightarrow_h p} \qquad \frac{p \rightarrow_h p' \quad q \rightarrow_h q'}{p.q \rightarrow_h p'.q'}$$

$$\frac{modified(\{p_1, \ldots, p_n, p'_1, \ldots, p'_{n'}\}) \subseteq \{x_1, \ldots, x_m\}}{\langle p_1, \ldots, p_n, x_1 := *, \ldots, x_m := * \rangle \rightarrow_h \langle p'_1, \ldots, p'_{n'} \rangle}$$

It can be observed that $\rightarrow_h$ is a monotonic relation, so that Lemma 5 applies. Furthermore, $\rightarrow_h$ corresponds to the transformation of the combined-case k-induction rule:

Lemma 6. *Let C and C_k^L be CFGs as defined in Def. 3. Then*

$$\mathit{unwinding}(C) \subseteq \mathit{gen}_{\rightarrow_h}(\mathit{unwinding}(C_k^L)) \ .$$

Lemma 5 and Lemma 6 together directly imply Theorem 1.

Proof (Lemma 6). We assume that the k-induction rule is applied to the loop $L \subseteq V$ with loop header $h \in L$. As in Def. 3, we denote the original CFG and the CFG resulting from k-induction by:

$$C \ = \ (V, \mathit{in}, E, \mathit{code}), \qquad C_k^L \ = \ (V_k^L, \mathit{in}_k^L, E_k^L, \mathit{code}_k^L) \ .$$

Because C is reducible, by assumption, we know that every transition into the loop ($(u, v) \in E$ with $u \notin L$ and $v \in L$) targets the loop header ($v = h$). Also, by assumption we know that $h \neq \mathit{in}$.

Let $\langle v_1, v_2, \ldots, v_n\rangle$ with $n > 0$, $v_1 = \mathit{in}$, and $\forall i \in \{1, \ldots, n-1\}.\ (v_i, v_{i+1}) \in E$ denote an arbitrary non-empty path of C. We infer a sequence $l(1), l(2), \ldots, l(n) \in \mathbb{N}$ counting the current number of loop iterations on $\langle v_1, v_2, \ldots, v_n\rangle$:

$$l(i) \ = \ \begin{cases} 0 & \text{if } v_i \notin L \\ l(i-1) & \text{if } v_i \in L \setminus \{h\} \\ l(i-1)+1 & \text{if } v_i = h \end{cases}$$

From this, we can derive a maximum set $\{p_1, p_2, \ldots, p_m\} \subseteq \mathbb{N}$ of "fusion points," which are the indexes $1 < p_1 < p_2 < \cdots < p_m = n+1$ satisfying the condition

$$\forall i \in \{1, \ldots, m-1\}.\ \big(v_{p_i} = h \wedge l(p_i) > k\big).$$

We can now prove by induction that for all $a \in \{1, \ldots, m\}$ it is the case that

$$\langle \mathit{code}(v_1), \mathit{code}(v_2), \ldots, \mathit{code}(v_{p_a - 1})\rangle \ \in \ \mathit{gen}_{\rightarrow_h}(\mathit{unwinding}(C_k^L)) \ .$$

In the base case, this follows because the path $\langle v_1, v_2, \ldots, v_{p_a-1}\rangle$ directly corresponds to a path in C_k^L.

For the step case, observe that any sequence of statements in C_k^L through the base case part $\bigcup_{i=1}^{k} L_i$ and Z part can be rewritten using $\rightarrow_h$ to an arbitrary sequence of statements through the loop L. The sequence $\langle \mathit{code}(v_1), \mathit{code}(v_2), \ldots, \mathit{code}(v_{p_a-1})\rangle$ therefore corresponds to a sequence in $\mathit{unwinding}(C_k^L)$ modulo the rewriting relation $\rightarrow_h$. This can be used to extend $\langle \mathit{code}(v_1), \mathit{code}(v_2), \ldots, \mathit{code}(v_{p_a-1})\rangle$ to the sequence $\langle \mathit{code}(v_1), \mathit{code}(v_2), \ldots, \mathit{code}(v_{p_{a+1}-1})\rangle$.

B Proof of Lemma 4

Consider two sequences of k-induction applications to loops in C (without inner loops), unrolling the loops $\bar{L} = L_1, L_2, \ldots, L_l$ and $\bar{M} = M_1, M_2, \ldots, M_l$ of loops. We have to show that both sequences yield the same CFG. Let $m \in \{1, \ldots, l\}$ be minimal such that $L_m \neq M_m$; if no such m exists, we are already finished. We show how to enlarge the common prefix $L_1, L_2, \ldots, L_{m-1}$ of $\bar{L}$ and $\bar{M}$ by repeatedly applying Lemma 2 to $\bar{M}$, without changing the resulting CFGs.

Because every loop in C is eventually unrolled in $\bar{M}$, there is a $u \in \{m+1, \ldots, l\}$ such that $\mathit{origin}(L_m) = \mathit{origin}(M_u)$. Similarly, there has to be a $v \in \{m+1, \ldots, l\}$ such that $\mathit{origin}(L_v) = \mathit{origin}(M_{u-1})$:

$$\begin{tikzcd}
L_1 \arrow[d, equal] & L_2 \arrow[d, equal] & \cdots & L_{m-1} \arrow[d, equal] & L_m \arrow[d, phantom, "\nparallel"] \arrow[rrr, dash, dashed, to=2-8] & \cdots & & L_v \arrow[dl, dash, dashed] & \cdots & L_l \\
M_1 & M_2 & \cdots & M_{m-1} & M_m & \cdots & M_{u-1} & M_u & \cdots & M_l
\end{tikzcd}$$

Because L_v is unwound after L_m, the loop $\mathit{origin}(L_v)$ cannot be an inner loop of $\mathit{origin}(L_m)$, since also L_m would otherwise contain an inner loop. Similarly, loop $\mathit{origin}(M_u)$ cannot be an inner loop of $\mathit{origin}(M_{u-1})$. By Lemma 1, $\mathit{origin}(M_u)$ and $\mathit{origin}(M_{u-1})$ are then disjoint, and so are M_u and M_{u-1}. This implies that unrolling of M_u and M_{u-1} can be transposed without changing the resulting CFG, by Lemma 2, producing the sequence

$$M_1, M_2, \ldots, M_m, \ldots, M_u, M_{u-1}, \ldots, M_l$$

By iterating such transpositions, M_u can eventually be put in the place of M_m, enlarging the common prefix of $\bar{L}$ and $\bar{M}$ to $L_1, L_2, \ldots, L_m$. Iterating this process will eventually make $\bar{L}$ and $\bar{M}$ identical. $\square$

C Detailed proof of Theorem 2

Suppose an arbitrary sequence of applications of the k-induction rule to the CFG C to the loops $L_1, L_2, \ldots, L_m$, finally resulting in a loop-free CFG. We show that this sequence can be transformed, with the help of Lemma 3 and Lemma 4, to a sequence in which k-induction is only applied to loops without inner loops, without changing the final CFG (up to equivalence). By Lemma 4, this implies that all CFGs in P_n are equivalent.

Let $u \in \{1, \ldots, m\}$ be maximal such that L_u contains inner loops; if no such u exists, we are already finished. Let $M \subset L_u$ be an innermost loop of L_u. Since the loops $L_{u+1}, \ldots, L_m$ do not contain inner loops, there are exactly $n = 2\mathbf{k}(\mathit{origin}(L_u)) + 1$ loops in the suffix $L_{u+1}, \ldots, L_m$ corresponding to M: let $L_{v_1}, L_{v_2}, \ldots, L_{v_n}$ be such that $\mathit{origin}(L_{v_i}) = \mathit{origin}(M)$ for all $i \in \{1, \ldots, n\}$. Since the loops $L_{u+1}, \ldots, L_m$ do not contain inner loops, and since $M \subset L_u$ is an innermost loop, the suffix can be reordered with the help of Lemma 4 to the sequence $L'_{u+1}, \ldots, L'_m$, such that $\mathit{origin}(L'_{u+i}) = \mathit{origin}(L_{v_i})$ for all $i \in \{1, \ldots, n\}$; this does not change the resulting CFG.

We can then apply Lemma 3 to reorder the sequence

$$L_1, L_2, \ldots, L_{u-1}, \quad L_u, \quad L'_{u+1}, L'_{u+2}, \ldots, L'_{u+n}, \quad L'_{u+n+1}, \ldots, L'_m$$

to the sequence

$$L_1, L_2, \ldots, L_{u-1}, \quad M, \quad L''_u, \quad L''_{u+n+1}, \ldots, L''_m$$

where L''_u is the loop obtained by applying k-induction to $M \subset L_u$, and $L''_{u+n+1}, \ldots, L''_m$ are derived from $L'_{u+n+1}, \ldots, L'_m$ by relabeling nodes appropriately (recall that Lemma 3 only guarantees equivalent, not identical CFGs).

Iterating this procedure will eventually reduce $L_1, L_2, \ldots, L_m$ to an outward application sequence of k-induction, so that finally Lemma 4 applies.