Software Verification Using k-Induction*

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

[1] Computer Science Department, Oxford University, 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 state 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* [15], employed by practical program verifiers (including [4, 5, 10, 20], among many others) and a recent k-induction method for program verification [12, 13] which we refer to here 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. We show experimentally that combined-case k-induction frequently allows program verification to succeed using significantly weaker loop invariants than would otherwise be required, reducing annotation overhead.

We start by recapping the inductive invariant and split-case k-induction approaches to verification, and outlining our new combined-case k-induction technique. We then make the following novel contributions:

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

* Supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), the EU FP7 STREP PINCETTE (project ID ICT-257647), EPSRC projects EP/G026254/1 and EP/G051100/1, and a grant from Toyota Motors.

– We state 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, and K-BOOGIE, an extension of the Boogie tool, and experimental results applying these tools to a large set of benchmarks (§6)

Compared with our previous work on k-induction techniques for software [12, 13], which are restricted to programs containing a single *while* loop (supporting multiple loops only via a translation of all program loops to a single, monolithic loop), our novel proof rule handles multiple natural loops in *arbitrary* reducible control-flow graphs.

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. We shall simply use *correctness* to refer to this notion of partial correctness.

2 Overview

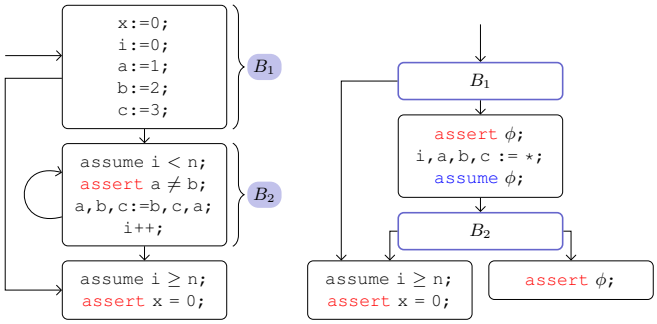
Throughout the paper, we present programs as control flow graphs (CFGs) and use the terms *program* and *CFG* synonymously. We follow the standard approach of modelling control flow using a combination of nondeterministic branches and $assume$ statements. During execution, a statement $assume\ \phi$ 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. The condition for the loop is $i < n$, and is encoded using $assume$ statements at the start of the loop body, and at the start of the node immediately following the loop. 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. Each loop header in the original CFG is prepended in the transformed CFG with a basic block that: asserts the loop invariant, havocs each loop-modified variable,[3] and assumes the loop invariant. Loop entry edges in the original CFG are replaced with edges to these new blocks in the transformed CFG. Each back edge in the original CFG is replaced in the transformed CFG with an edge to a new, childless basic block that asserts the invariant for the associated loop. Otherwise, the CFGs are identical.

We say that a loop is *cut* with respect to 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

[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.



(a) Original CFG (b) CFG after loop cutting

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

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 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.

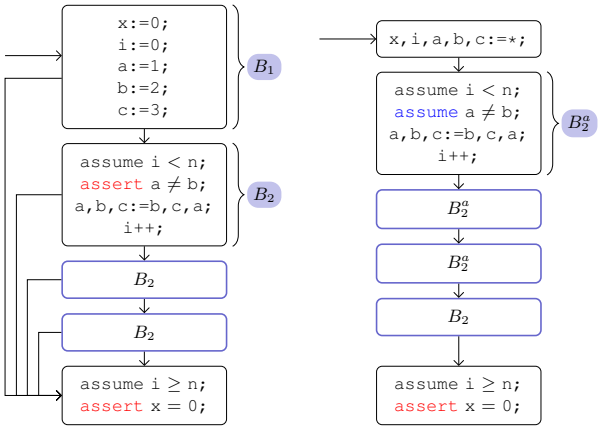
Split-case k-induction. The k-induction method was proposed as a technique for SAT-based verification of finite-state transition systems [21]. Let $\mathbf{I}(s)$ and $\mathbf{T}(s, s')$ be formulae encoding the initial states and transition relation for a system over sets of propositional state variables 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 [12, 13] we investigated a direct lifting of k-induction from transition systems to the level of program loops. We refer to the technique of [12, 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



(a) Base case (b) Step case

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

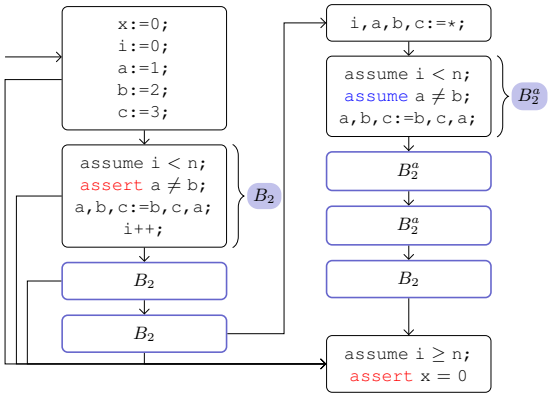


Fig. 3. Combined-case k-induction, with $k = 3$

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.

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.

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 *combined-case* k-induction, the strengths of split-case k-induction and the inductive invariant approach are brought together. 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.

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. 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 $k \geq 3$. As a further difference, note that base and step case are composed *sequentially,* with a transition leading from the last block B_2 of the base case to the first block of the step case. For some programs, this can increase the strength of the induction rule considerably compared to split-case k-induction, since path constraints established in the base case can be helpful for verifying the step case. Unlike with the inductive invariant approach, no external invariant is required.

Of course, combined-case k-induction does not solve the problem of finding invariants. The technique depends on invariants appearing as assertions in loop bodies. For example, if the assertion $a \neq b$ was moved to the exit of the loop in Figure 1(a), the program could not be proved using combined-case k-induction. In practice it may be necessary to strengthen the induction hypothesis by adding manually or automatically derived invariants as assertions in the body of a loop. However, our experimental evalu-

ation in §6 demonstrates that combined-case k-induction frequently makes verification possible with significantly weaker invariants than are otherwise required, thus reducing annotation overhead.

3 Control flow graphs and loops

We present our results in terms of control flow graphs, which are 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 and constrain the effects of nondeterministic assignments, 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.

Definition 2. *The* natural loop *associated with back edge (u, v) is the smallest set $L_{(u,v)} \subseteq V$ satisfying the following conditions:*

- $u, v \in L_{(u,v)}$
- $(u', v') \in E \ \wedge\ v' \in L_{(u,v)} \setminus \{v\} \Rightarrow u' \in L_{(u,v)}$

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

[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.

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

From now on, 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. 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 3. *Let $C = (V, in, E, code)$ be a CFG. The* unwinding *of C is defined as:*

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

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: $S = \{\sigma \mid \sigma : X \to \mathcal{D}\} \cup \{\lightning\}$.

For an expression ϕ and store σ, we write ϕ^σ to denote the value obtained by evaluating ϕ according to the valuation of variables given by σ.

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

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

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

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

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

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

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:

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

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 $traces(C)$. Otherwise C is not correct, and a trace which leads to $\lightning$ is a counterexample to correctness.

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 ∈ ℕ;
        result ⟵ ANALYSE(C_k^L);
        if result = CORRECT then                         // k-induction succeeded
            return CORRECT;
        else                                             // k-induction was inconclusive
(**)        either back-track to (*), or return DON'T KNOW;
        end
end
```

4 Proof rule and verification algorithm

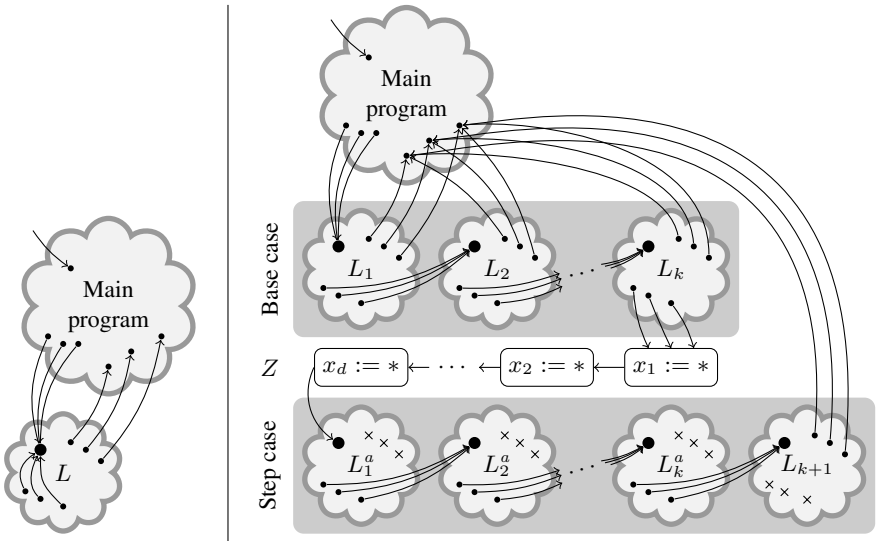
Given a CFG C containing a natural loop L (see Def. 2), and a positive integer k, we shall define a k-induction rule that transforms C into a CFG C_k^L in which loop L is eliminated via k-cutting, such that correctness of C_k^L implies correctness of C. We start by motivating the use of the rule, considering 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. Genuine bugs can only be detected via traces through the k-cut CFG that do not pass through any havoc nodes introduced by the k-induction rule. Alternatively, ANALYSE can simply be executed in parallel with bounded model checking [6].

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 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



(a) Program (b) Unrolled program

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

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 $\mathit{modified}(L)$, 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 $\mathit{assert}\ \phi$ are replaced with $\mathit{assume}\ \phi$, 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 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 that is 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, using the same notation as presented in Figure 4.

Let $C = (V, in, E, code)$ be a CFG and $L \subseteq V$ a loop in C with header h. Assume that $in \notin L$. (This can be trivially enforced by adding an $assume\ tt$ node to C if necessary.) We present a k-induction proof rule for *positive* values of k, under the assumption that $modified(L)$ is non-empty. Extending the definition, and all the results presented in this paper, to allow $k = 0$, and $modified(L) = \emptyset$, 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 $modified(L) = \{x_1, \ldots, x_d\}$ for some $d > 0$. For $1 \leq i \leq k+1$, define $L_i = \{v_i \mid v \in L\}$. Similarly, for $1 \leq i \leq k$, define $L_i^a = \{v_i^a \mid v \in L\}$. Let $Z = \{z_1^h, \ldots, z_d^h\}$. Assume that the sets L_i ($1 \leq i \leq k+1$), L_i^a ($1 \leq i \leq k$) and Z consist of fresh nodes, all distinct from each other and from the nodes in V.

Definition 4. *$C_k^L = (V_k^L, in_k^L, E_k^L, code_k^L)$ is defined as follows:*

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

$in_k^L = in$ *(recall that, by assumption, $in \notin L$)*

$E_k^L =$

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

$$
\begin{aligned}
code_k^L(z_i^h) &= \text{`}x_i := *\text{'} && (1 \le i \le d)\\
code_k^L(v_i^a) &= \begin{cases} assume\ \phi & \text{if } code(v) = assert\ \phi \\ code(v) & \text{otherwise} \end{cases} && (1 \le i \le k)\\
code_k^L(v_i) &= code(v) && (1 \le i \le k+1)\\
code_k^L(v) &= code(v) && \text{for } v \in V_k^L \cap V
\end{aligned}
$$

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

5 Theoretical properties of the k-induction rule

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 isomorphic.

Definition 5. *Let $C = (V, in, E, code)$ and $C' = (V', in', E', code')$ be CFGs. A bijection $\alpha : V \to V'$ is an* isomorphism *between C and C' if $\alpha(in) = in'$ and, for all $u, v \in V$, $code(u) = code'(\alpha(u))$, 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 *and write $C \equiv C'$.*

It is easy to show that $\equiv$ is an equivalence relation on CFGs.

In what follows, C denotes a CFG. The next result follows directly from the definition of a natural loop:

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

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 $(C_{k_L}^L)_{k_M}^M \equiv (C_{k_M}^M)_{k_L}^L$.*

Lemma 2 shows that, for disjoint loops, the order in which k-induction is applied to each loop is irrelevant; an isomorphic CFG always results. Thus, for mutually disjoint loops $L_1, \ldots, L_d$ in a CFG C, and positive integers $k_1, \ldots, k_d$, we can write $C_{k_1,\ldots,k_d}^{L_1,\ldots,L_d}$ to denote a 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 $C_{k_M}^M$ 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 $L_1^a, \ldots, L_{k_M}^a$ respectively. Def. 4 ensures that they are all disjoint in $C_{k_M}^M$.

The CFG $C_{k_L}^L$ 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:*

$$(C_{k_L}^L)_{k_M}^{M'} \equiv (C_{k_M}^M)_{k_L,\ldots\ldots\ldots\ldots\ldots\ldots,k_L}^{L_1,\ldots,L_{k+1},L_1^a,\ldots,L_k^a}.$$

We now show that if we repeatedly apply the k-induction rule to obtain a loop-free CFG, as long as a value for k is used consistently for each loop in C 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 $origin(L_{1_1}) = origin(L_1) = origin(L) = L$, $origin(M_1) = origin(M) = M$, and $origin(N) = N$. Also, CFG $C^L_{k_L}$ includes loops $M' \subset N'$ identical to M and N, except that L has been unrolled. We have $origin(M') = M$ and $origin(N') = N$.

Definition 6. *Let* $\mathbf{k} : (\text{loops of } C) \to \mathbb{N}$ *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*

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

Our main confluence theorem states that the result of exhaustively applying the combined-case k-induction rule is independent (up to isomorphism) of the order in which loops are eliminated. The result is stated with respect to Def. 6, and is proved using Lemmas 1–3.

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 isomorphic, and loop-free.*

It should be noted that, although the final CFGs are isomorphic regardless of the order of loop elimination, intermediate CFGs can differ both in size and in the number of remaining loops. Also, the total number of required applications of the k-induction rule depends on this order: eliminating loops starting from innermost loops will altogether need fewer rule applications than elimination starting with outer loops.

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 program parts outside of the eliminated loop L unchanged. By a simple complexity analysis, we find that the size of loop-free programs derived though repeated application of k-induction is (singly) exponential in the depth of the deepest loop nest in the worst case, but only linear in the number of disjoint loops. Thus the size of generated programs is not a bottleneck for combined-case k-induction in practice.

6 Experimental evaluation

We have implemented our techniques in two tools. K-BOOGIE is an extension of the BOOGIE verifier, allowing programs written in the BOOGIE language to be verified using combined-case k-induction, as well as with the inductive invariant approach supported by regular BOOGIE. When combined-case k-induction is selected, our novel k-induction rule is used to eliminate innermost loops first. As BOOGIE is an intermediate language for verification, K-BOOGIE can be applied to programs originating from several different languages, including Spec# [4], Dafny [20], Chalice, VCC, and Havoc. K-INDUCTOR is a k-induction-based verifier for C programs built on top of the CBMC tool [9]. K-INDUCTOR supports both split- and combined-case k-induction. Again, with combined-case k-induction, loops are processed innermost first. With split-case k-induction, all outermost loops are simultaneously eliminated in each application of the k-induction rule; we have found this strategy works best in practice.

We use K-BOOGIE to compare the standard inductive invariant approach to verification with our novel combined-case k-induction method, and K-INDUCTOR to compare combined-case k-induction with split-case k-induction. Both tools, and all our benchmarks, are available online: **http://www.cprover.org/kinduction**

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, annotated with assertions, pre-/post-conditions, and loop invariants, and were not previously known to be amenable to k-induction. Six of the procedures contain multiple loops, three contain (singly) nested loops. Our findings are summarised in Table 1.

To evaluate the applicability of k-induction, we first split conjunctive loop invariants in the programs into multiple invariants, and then eliminated all invariants that were not necessary to verify assertions and post-conditions even with the normal Boogie induction rule. Since Boogie uses abstract interpretation (primarily with an interval domain) to automatically infer simple invariants, in this step also all those invariants were removed that can be derived with the help of inexpensive abstract interpretation techniques. The elimination of invariants was done in a greedy manner, so that in the end a minimum set of required invariants for each procedure was obtained (though not necessarily a set with the smallest number of invariants).

We then checked, using $0 \leq k \leq 4$, 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. In Table 1, **# removable** shows the number of invariants that could be individually removed, in comparison to the total number of invariants. As second step, we determined maximum sets of invariants that could be removed simultaneously, shown under **# sim. remov.** in Table 1. In both cases, we show the largest value of k required for invariant removal, over all loops (**required** k), which was determined by systematically enumerating all combinations of k. For each procedure, we show the verification time with the normal Boogie loop rule (**time w/o** k**-ind.**), the range of times needed by the various runs with k-induction (**times w/** k**-ind.**, using the smallest k for which verification succeeded), the number of lines of

Procedure	# removable, required k	# sim. remov., required k	time w/o k-ind.	times w/ k-ind.	LOC/ # loops	LOC program
Procedures generated from Dafny programs						
VSI-b1.Add	2/4, 1	2/4, 1	2.5s	[2.6s, 2.9s]	114/2	710
VSI-b2.BinarySearch	0/5, 1		2.5s	2.6s	100/1	595
VSI-b3.Sort	1/16, 1	1/16, 1	3.9s	[6.2s, 6.2s]	186/2	798
VSI-b3.RemoveMin	1/6, 1	1/6, 1	3.0s	[4.5s, 4.5s]	176/2	
VSI-b4.Map.FindIndex	3/4, 2	2/4, 1	3.7s	[3.6s, 4.6s]	84/1	956
VSI-b6.Client.Main	1/3, 1	1/3, 1	3.1s	[3.5s, 3.5s]	139/1	900
VSI-b8.Glossary.Main	4/16, 1	3/16, 1	5.3s	[18.7s, 21.6s]	381/3	1998
VSI-b8.Glossary.readDef	0/1, 1		3.4s	3.6s	71/1	
VSI-b8.Map.FindIndex	0/1, 1		3.3s	3.4s	66/1	
Composite.Adjust	1/3, 2	1/3, 2	5.3s	[44.3s, 44.3s]	80/1	1275
LazyInitArray	1/5, 1	1/5, 1	5.0s	5.0s	165/1	806
SchorrWaite.RecursiveMark	0/6, 1		3.4s	4.2s	98/1	1175
SchorrWaite.IterativeMark	2/17, 1	2/17, 1	4.8s	5.7s	177/1	
SchorrWaite.Main	4/27, 1	3/27, 1	33.3s	[16.9s, 34.5s]	275/1	
SumOfCubes.Lemma0	1/2, 1	1/2, 1	2.6s	[2.6s, 2.6s]	81/1	915
SumOfCubes.Lemma1	1/2, 1	1/2, 1	2.7s	[2.7s, 2.7s]	65/1	
SumOfCubes.Lemma2	1/2, 1	1/2, 1	2.5s	[2.5s, 2.5s]	48/1	
SumOfCubes.Lemma3	1/2, 1	1/2, 1	2.5s	[2.5s, 2.5s]	51/1	
Substitution	0/1, 1		2.7s	2.8s	131/1	846
PriorityQueue.SiftUp	1/2, 2	1/2, 2	2.9s	3.3s	92/1	819
PriorityQueue.SiftDown	1/2, 2	1/2, 2	3.1s	[16.0s, 16.0s]	101/1	
MatrixFun.MirrorImage	2/6, 1	2/6, 1	2.9s	[3.5s, 3.5s]	125/2	922
MatrixFun.Flip	1/3, 1	1/3, 1	2.7s	[2.8s, 2.8s]	103/1	
ListReverse	2/3, 2	2/3, 2	2.4s	[2.4s, 2.4s]	71/1	329
ListCopy	1/4, 1	1/4, 1	2.5s	[2.5s, 2.5s]	141/1	434
ListContents	1/3, 1	1/3, 1	3.4s	[6.1s, 6.1s]	141/1	717
Cubes	3/4, 2	3/4, 4	2.8s	[2.7s, 3.3s]	97/1	339
Celebrity.FindCelebrity1	1/1, 2	1/1, 2	2.5s	[3.0s, 3.0s]	98/1	795
Celebrity.FindCelebrity2	0/1, 1		2.5s	2.7s	99/1	
Celebrity.FindCelebrity3	0/2, 1		2.5s	2.6s	86/1	
VSC-SumMax	1/2, 1	1/2, 1	2.4s	[2.7s, 2.7s]	77/1	458
VSC-Invert	0/1, 1		15.4s	[3.2s, 3.2s]	61/1	568
VSC-FindZero	1/2, 1	1/2, 1	2.7s	[2.7s, 2.7s]	90/1	625
VSC-Queens.CConsistent	0/3, 1		2.6s	2.7s	79/1	825
VSC-Queens.SearchAux	0/1, 1		2.9s	3.2s	139/1	
Native Boogie programs						
StructuredLocking	1/1, 1	1/1, 1	1.8s	[1.8s, 1.8s]	16/1	40
StructuredLockingWithCalls	0/1, 1		1.8s	1.8s	13/1	
Structured.RunOffEnd1	1/1, 1	1/1, 1	1.8s	[1.8s, 1.8s]	12/1	53
BubbleSort	7/14, 1	7/14, 1	2.1s	[3.0s, 3.2s]	33/3	42
DutchFlag	1/5, 1	1/5, 1	1.9s	[2.0s, 2.0s]	29/1	37

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

Benchmark	LOC	# loops	nesting depth	split-case				combined-case			speedup with combined-case
				min k	max k	time (s)	# invariants	min k	max k	time (s)	
1-buf	151	2	2	1	1	0.32	3	0	1	0.23	1.35
1-buf I/O	178	2	2	1	1	0.39	5	0	1	0.27	1.44
2-buf	254	3	2	1	2	1.18	17	0	2	0.45	2.62
2-buf I/O	304	3	2	1	2	2.06	29	0	2	0.53	3.85
3-buf	282	4	2	1	3	9.56	27	0	3	1.00	9.58
3-buf I/O	364	4	2	1	3	8.47	38	0	3	1.03	8.19
Euler simple	101	3	3	1	2	3.30	10	0	2	2.95	1.12
sync atomic op	91	3	2	1	1	0.40	4	0	1	0.18	2.24
sync mutex	83	2	2	1	1	0.87	2	0	1	0.28	3.08

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

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). Experiments were run on a 2.5GHz Intel Core2 Duo machine with 2 GB RAM and Windows Vista.

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. Since the normal Boogie rule corresponds to 0-induction, augmented with assumptions and assertions encoding the loop invariant, already 1-induction is often able to succeed with a significantly reduced number of invariants. Values of k larger than two proved to be beneficial only for a single procedure (Cubes); additional gains with even larger values of k therefore seem unlikely. On average, 32% of the invariants could be removed (simultaneously) when using k-induction. The verification times with k-induction are only marginally larger than those with the normal Boogie rule, and for some examples even smaller (averaging over all benchmarks, verification time increased by 44%). In cases where the same set of invariants is used, verification times almost coincide. This shows that k-induction, with small values of k, can be useful for general-purpose verification, since the (extremely time-consuming) process of constructing inductive invariants can be shortened.

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 [12, 13] (in which full details can be found). These consist of data processing programs for the Cell BE processor, where data is manipulated using DMA. In [12, 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**), the number of loops processed by k-inductor (**# loops**, this is the number of loops after function inlining, which may cause loop duplication), and the depth of the deepest loop nest (**nesting depth**). All benchmarks involve nested loops. For the split-case and combined-case approaches, we manually determined the smallest values of k required for each loop in order for verification to succeed. In each case, the minimum and maximum values of k required are shown (**min/max** k), as well as the time (in seconds) taken for verification with this combination of k values (**time**). For the split-case approach, we show the number of invariants that had to be added manually for verification to succeed (**# invariants**) – these invariants are *not* required when our novel combined-case method is employed. Finally, we show the speedup obtained by using combined-case k-induction instead of split-case k-induction (**speedup with combined-case**). Experiments are performed on a 3GHz Intel Xeon machine, 40 GB RAM, 64-bit Linux. MiniSat 2 is used as a back-end SAT solver for CBMC. Manually specified invariants are mainly simple facts related to variable ranges; 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 in-

ner loops that are unrelated to DMA processing (and thus do not contain assertions of interest) to be handled using $k = 0$. In such cases, $k = 0$ is sufficient because we do not havoc variables that are not modified by the loop in question. With split-case k-induction, explicit invariant assertions must be added to assert that such variables are invariant under loop execution. This involves a lot of manual effort, and verification with k-induction requires at least $k = 1$ to take advantage of these assertions.

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. Verification using combined-case k-induction never takes longer than three seconds for this benchmark set, suggesting good scalability of the approach.

7 Related work and conclusions

The concept of k-induction was first published in [21, 7], targeting the verification of hardware designs and transition relations. 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. Several optimisations and extensions to the technique have been proposed, including property strengthening to reduce induction depth [22], improving performance via incremental SAT solving [14], and verification of temporal properties [2].

Besides hardware verification, k-induction has been used to analyse synchronous programs [18, 16] and, recently, SystemC designs [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 [12, 13], from which we also draw some of the benchmarks used in this paper. A combination of the k-induction rule of [12, 13], abstract interpretation, and domain-specific invariant strengthening techniques for DMA race analysis is the topic of [11]. The main new contributions of this paper over our previous work are that we present a k-induction technique that can be applied in a structured manner to multiple, *arbitrary* loops in a reducible control-flow graph (prior work was restricted to single-loop programs, with multiple loops handled via translation to a single, monolithic loop), and we present the novel idea of combined-case k-induction, where base and step case are combined into a single program. We have demonstrated experimentally that combined-case k-induction 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.

Combined-case k-induction depends on analysis of those variables which are not modified by a given loop. This can be viewed as a simple kind of loop summary. We plan to investigate whether k-induction can be strengthened using more sophisticated loop summarisation analyses [19]. In addition, we intend to study automatic techniques for selecting and exploring values of k for programs with multiple loops.

Finally, our experiments show that the effectiveness of k-induction varies significantly from example to example. It would be interesting and useful to characterise, ideally formally but at least intuitively, classes of programs for which k-induction is

likely to be beneficial for verification. Our initial efforts in this direction indicate that it is a challenging problem.

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. pp. 82–87. ACM (2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: CASSIS. LNCS, vol. 3362, pp. 49–69. Springer (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. LNCS, vol. 1954, pp. 372–389. Springer (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. LNCS, vol. 2988, pp. 168–176. Springer (2004)
10. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA. pp. 213–226. ACM (2008)
11. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: VMCAI. LNCS, vol. 6538, pp. 169–183. Springer (2011)
12. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: TACAS. LNCS, vol. 6015, pp. 280–295. Springer (2010)
13. Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design (2011), to appear, DOI: 10.1007/s10703-011-0124-2
14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4) (2003)
15. Floyd, R.: Assigning meaning to programs. In: Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. AMS (1967)
16. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. Electr. Notes Theor. Comput. Sci. 144(1), 19–33 (2006)
17. Große, D., Le, H.M., Drechsler, R.: Proving transaction and system-level properties of untimed SystemC TLM designs. In: MEMOCODE. pp. 113–122. IEEE (2010)
18. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: FMCAD. pp. 109–117. IEEE (2008)
19. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: ATVA. LNCS, vol. 5311, pp. 111–125 (2008)
20. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: LPAR (Dakar). LNCS, vol. 6355, pp. 348–370. Springer (2010)
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD. LNCS, vol. 1954, pp. 108–125. Springer (2000)
22. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: VLSID. pp. 63–68. IEEE (2007)