

Split-Case k -Induction for Program Verification

Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer

Oxford University Computing Laboratory, Oxford, UK

Abstract. We present a novel proof rule for verifying software programs using k -induction. The rule takes an unstructured control flow graph (CFG) and a set of disjoint, natural loops occurring in the CFG, and decomposes the CFG into a base case and step case in which all the given loops are eliminated via unwinding. Correctness of the base and step cases guarantees correctness of the whole program. Recursively applying the proof rule yields loop-free base and step cases, which can be checked using SAT/SMT-based techniques. We refer to this verification technique as *split-case k -induction*. The soundness of our new rule is shown by reduction to a more general notion which we term *inductive decomposition*. Because the new rule can be applied to many loops of a program simultaneously, it strictly generalises an existing single-loop proof rule for k -induction; compared with the existing rule, the number of loop-free programs that must be checked is drastically reduced, especially in the presence of nested loops. We describe an implementation of our techniques on top of the CBMC tool. Our experiments, using a range of benchmarks, show that the new rule leads to significantly faster verification than the existing rule, and that additional performance benefits can be obtained by parallelising k -induction analysis on a multicore machine.

1 Introduction

One of the main challenges associated with designing analysis tools for imperative programs is to deal with loops. In this work, we present a novel technique for analyzing imperative programs with loops, based on k -induction.

The k -induction method [18] was introduced in the context of proving safety properties of finite-state transition systems. The method involves proving a base case, establishing that the system is safe during the first k execution steps, and a step case, verifying that whenever the system has safely executed k steps also the $(k + 1)$ st step is safe. The base and step cases can be checked using a SAT solver. If both succeed, the system is correct;¹ if the base case fails, a counterexample to correctness can be derived. If the base case succeeds but the step case fails, the result is inconclusive and a larger value for k can be tried.

In prior work [9] we presented a proof rule for applying k -induction to software programs with loops. Verifying a program using this rule involves choosing an arbitrary outermost loop, L , and proving a base and a step case, which mirror the base and step cases of [18]. The base case checks that all traces from the start of the program executing at most k iterations of L are safe. The step case checks that *any* trace (starting in an

¹ Throughout the report, we use *correct* to refer to *partial correctness*.

arbitrary state) consisting of k successful iterations of L can be extended to include a further successful loop iteration, or successful execution of the loop epilogue.

If L is the *only* loop in a program, the base and step cases generated by the k -induction rule of [9] are guaranteed to be loop-free. They can therefore be checked directly using SAT-/SMT-based methods.

For a program containing multiple loops, there are two obvious strategies:

- (i) **Inward recursive application.** We apply k -induction to one of the outermost loops. At least one of the resulting base and step case programs will contain a loop. The k -induction rule can then be applied (with possibly distinct values of k) recursively to such programs. Eventually, this recursive application will lead to a sequence of loop-free programs which can be checked directly.
- (ii) **Monolithic transformation.** Standard techniques can be used to transform the input program into an equivalent program containing a single, monolithic loop. When k -induction is applied to the transformed program, the resulting base and step case programs are guaranteed to be loop-free and can thus be checked directly.

Strategy (i) takes advantage of the structure of loops in the input program. Verification involves checking many small programs, which can be carried out in parallel. Each time the k -induction rule is applied, a suitable value for k can be chosen based on properties of the loop under consideration. However, the number of cases to be checked may be unmanageable: in the worst case, the number of loop-free programs generated by Strategy (i) is doubly-exponential in the depth of the deepest loop nest in the original program.

Strategy (ii) does not suffer from this case explosion: only two cases must be checked. However, a “one-size-fits-all” value for k must be used when analysing the monolithic loop, dictated by the largest value for k that would be required to show correctness of any particular loop in the original program; furthermore, part of the original structure of the program is lost. As a result, the base and step cases may be very large. With only two cases to check, there is little opportunity for parallel verification.

To overcome the limitations of Strategies (i) and (ii), we present a new k -induction rule which strictly generalises the rule of [9]. Our new rule takes an unstructured, not necessarily reducible, control flow graph (CFG), a set $\mathbf{L}$ of disjoint, natural loops occurring in the CFG, and a set of non-negative *depths*, one corresponding to each loop in $\mathbf{L}$. The rule returns two CFGs: a base case and step case. In each of these CFGs, all the loops of $\mathbf{L}$ are eliminated, each having been unwound according to its given depth.

We prove that our rule is sound: if the base and step case CFGs can be proved correct then the original CFG is correct. Soundness is shown by reduction to a more general notion which we term *inductive decomposition*, formalising the principle of decomposing a program into smaller programs in such a way that every execution path of the original program can be reconstructed from execution paths of the small programs. Both the rule in [9] and our new rule are instances of inductive decomposition.

Applying our new rule recursively, and always taking $\mathbf{L}$ to be a single outermost loop, yields exactly Strategy (i). Strategy (ii) can still be used directly. The rule enables three further general-purpose strategies:

- (iii) **Outward recursive application.** Our new rule can be applied to arbitrary loops, including innermost loops. Thus we can apply the rule recursively as in Strategy (i), each time taking L to be a single, innermost loop.
- (iv) **Simultaneous inward recursive application.** Distinct outermost loops are clearly disjoint. Thus, we can apply the new rule recursively, at each stage taking L to be the set of all outermost loops.
- (v) **Simultaneous outward recursive application.** Similarly, we can apply the rule recursively, at each stage taking L to be the set of all innermost loops.

We show, theoretically, the number of loop-free programs that must be checked with each of Strategies (i)–(v).

We have built a k -induction tool, K-INDUCTOR, for verification of non-recursive C programs. K-INDUCTOR supports each of Strategies (i)–(v), and uses CBMC [6] as a decision procedure for loop-free programs. The tool has a parallel mode, where independent base and step case programs are distributed across multiple processors.

We show experimentally, using a range of benchmarks, that Strategies (iv) and (v) significantly outperform the other strategies in terms of verification time. We also demonstrate the speedups obtained through parallel verification.

In summary, our major contributions are:

- A novel proof rule for k -induction, which can be applied to an arbitrary set of disjoint loops in an input program
- A theoretical and experimental comparison of five strategies for applying the rule
- An experimental evaluation showing that our new rule leads to significantly accelerated verification compared with a previous approach to k -induction

The title of this report refers to our k -induction method as *split-case* k -induction. This is to differentiate it from ongoing work investigating an alternative approach which we call *combined-case* k -induction. In the remainder of this report, we use k -induction to mean split-case k -induction.

2 Overview

We describe the k -induction rule of [9] in more detail, and present an example showing the associated case explosion that occurs in the presence of nested loops (§2.1). We explain how translation to monolithic form can be used as a crude mechanism for overcoming case explosion (§2.2). This is followed by an informal diagrammatic presentation of our new k -induction rule (§2.3). This informal presentation provides intuition for the formal presentation of the rule in §4.

2.1 A simple k -induction rule

In previous work [9], we presented a k -induction rule for dealing with programs of the form shown in Figure 1(a), encoding a *while* loop with prefix α , tail γ , body β , and loop condition c . The rule states that the program of Figure 1(a) is correct if, for some $k \geq 0$, the programs of Figures 1(b) and 1(c) are correct.

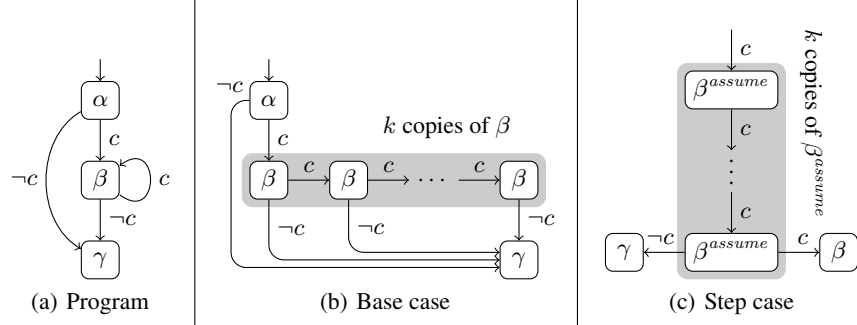


Fig. 1. Original k -induction rule for *while* loops

The *base case* (Figure 1(b)) checks that there are no errors along any execution paths which start at the beginning of the program and execute no more than k loop iterations. The *step case* (Figure 1(c)) checks that any error-free path, starting from an arbitrary state and consisting of k loop iterations, can only be extended (either by a further loop iteration, or by execution of the loop epilogue) in an error-free way. The notation β^{assume} denotes the program fragment β in which every assertion *assert* ϕ is replaced with an assumption *assume* ϕ ; assumptions are partial statements that have no effect if the condition ϕ holds, but that suspend program execution if ϕ is violated (in contrast to assertions, which raise an error if their condition is violated).

If the program fragments α , β and γ comprising the program of Figure 1(a) are all loop-free, then the base and step case are both loop-free programs. Correctness of the base and step cases can be decided using SAT-/SMT-based techniques.

If any of α , β or γ contain loops, at least one of the base and step case will contain a loop. Given a base/step case containing loops, the k -induction rule can be applied to an outermost loop to yield a further base and step case.² Continuing to apply the rule recursively eventually yields a set of loop-free programs, whose correctness implies correctness of the original program.

The major problem with recursively applying the rule of Figure 1 is with respect to nested loops. Consider the example program of Figure 2(a). This matches the *while* loop structure of Figure 1(a), but the loop body β is itself a *while* loop. Assume that β' , the body of the inner loop, is loop-free. Applying the k -induction rule to the outer loop of Figure 2(a) gives the base case shown in Figure 2(b). Unwinding has eliminated the outer loop of Figure 2(a) in Figure 2(b), but the price for this is that the inner loop of Figure 2(a) is duplicated k times in Figure 2(b). Worse still, the inner loop is duplicated $k + 1$ times in the step case, which we do not show.

To prove the base of Figure 2(b), we can pick one of the outermost loops, L say, and apply k_1 -induction (where k_1 is possibly distinct from k). Because the $k - 1$ loops distinct from L form part of either the prefix or tail of L , the base case for k_1 -induction

² The rule of Figure 1 applies to *while* loops; the rule of [9] is a little more general, applying to arbitrarily structured outermost loops. These may arise in base/step derived from *while* loops.

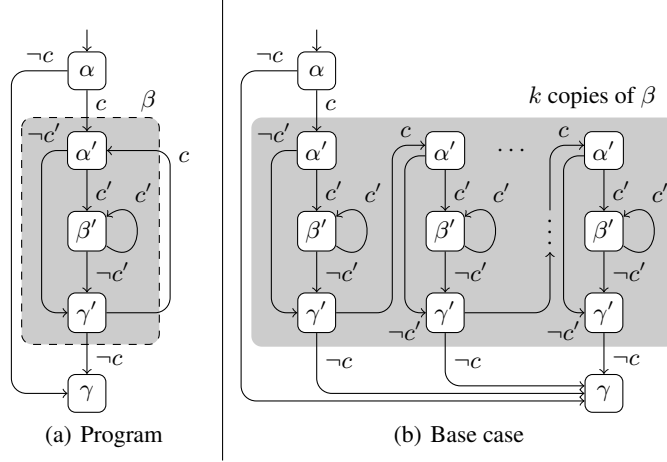


Fig. 2. Case explosion with the original k -induction rule for *while* loops

contains $k - 1$ loops. We can attempt to prove this base case using k_2 -induction, yielding a base case containing $k - 2$ loops, and so on.

If the same value of k is always used when applying k -induction recursively to the program of Figure 2(a), it can be shown that 3×2^k loop-free programs will eventually be obtained. In general, for an arbitrary program, the number of loop-free programs generated by repeated application of the rule of Figure 1 is doubly-exponential in the depth of the deepest loop nest of the input program. We discuss this in more detail in §5.3.

The procedure of recursively applying k -induction has positive properties. Specific values of k can be chosen for each loop under consideration, thus structural properties of the input program can be exploited. The individual loop-free programs generated by the procedure may be rather simple, and because they are independent they can be checked in parallel. However, as our experiments in §6 show, these positive properties do not outweigh the problem of doubly-exponential case explosion.

2.2 Eliminating loops via a monolithic transformation

A crude approach to overcoming the case explosion associated with multiple loops is to transform the input program into an equivalent program containing a single, monolithic loop. It is a well-known “folk-theorem” that a flow graph containing arbitrary cycles can be transformed to a flow graph containing a single loop [13].

Figure 3 shows the result of applying the monolithic transformation to the program of Figure 2(a). A fresh variable t is used to track which portion of the original program should be executed next. The *skip* node is the entry point for the monolithic loop. All edges into or out of a loop in the original program are simulated by setting t to an appropriate value, and jumping to the *skip* node. Edges from the *skip* node lead to previous loop entry and exit points, depending on the value of t .

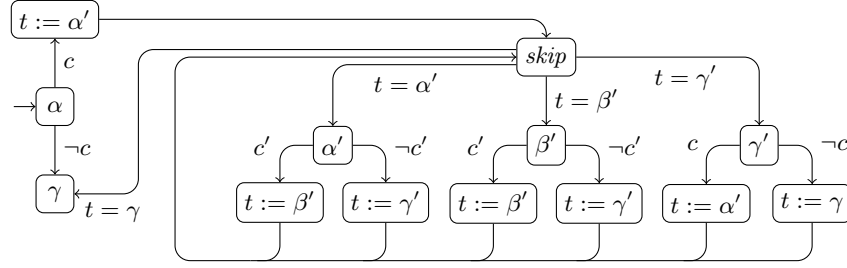


Fig. 3. The multiple loops of Figure 2(a) rewritten as a single, monolithic loop

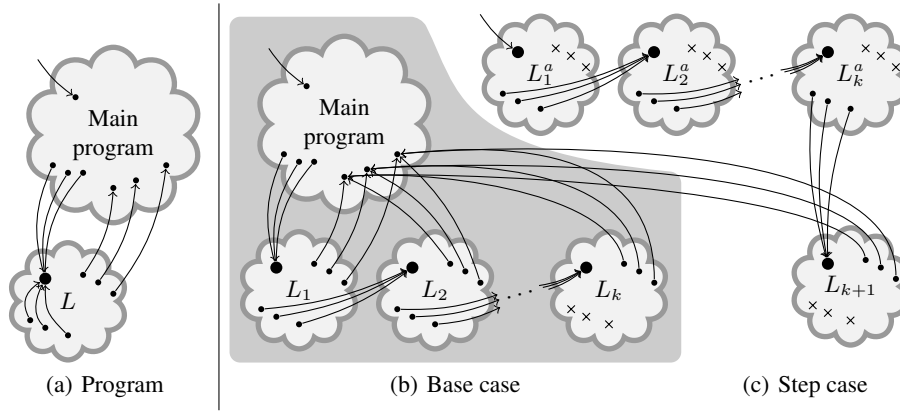


Fig. 4. Schematic overview of the new k -induction rule

The general k -induction rule for outer loops, presented in [9], can be applied directly to the resulting program, yielding a base case and step case that are guaranteed to be loop-free. This completely avoids the case explosion discussed in §2.1. However, the body of the monolithic loop encapsulates *all* loops in the original program, and thus may be sizable. While it may be possible to analyse many loops in the original program using small values for k , the “one-size-fits-all” value of k required to analyse the monolithic loop will be dictated by the largest of these values. As a result, the associated base and step cases, which contain k and $k + 1$ copies of the monolithic loop body respectively, may be very large. Finally, since verification boils down to checking just two cases, there is little opportunity to exploit parallelism.

2.3 A k -induction rule for arbitrary loops

We propose a new rule which allows k -induction to be applied simultaneously to a set of arbitrary, disjoint loops in a CFG. Among the many possible strategies for applying this new rule recursively, we identify four cases: at each stage, we can choose a single outermost loop (this is the approach of §2.1), a single innermost loop, all outermost loops, or all innermost loops. We will see that applying the rule to all innermost or outermost

loops simultaneously dramatically reduces the case explosion associated with applying the rule recursively to single loops. It also improves on the monolithic approach of §2.2 by decomposing the verification task based on the loop structure of the input program.

Rule for a single loop. Our k -induction rule for a single, arbitrary loop is illustrated in Figure 4. Figure 4(a) depicts an arbitrary CFG separated into two parts: a loop L (the smaller cloud), and all nodes outside L (the larger cloud, labelled “Main program”). We require that entry to the CFG, indicated by the edge into “Main program”, is not via L . The program can be trivially 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.

Base case. Figure 4(b) illustrates the base case generated by our new k -induction rule. This consists of the main program as before, but loop L is duplicated k times, indicated by $L_1, \dots, L_k$. In L_i , for $i < k$, back edges in L are replaced by edges to the header node of L_{i+1} . In L_k , such edges are simply removed, indicated by crosses in Figure 4(b). In each L_i ($1 \leq i \leq k$), edges leading to the main program are preserved.

Showing correctness of the base case establishes that any trace of the original program that executes no more than k iterations of L cannot go wrong.

Step case. Figure 4(c) illustrates the associated step case. Note that this step case incorporates the base case of Figure 4(b). Our depiction indicates that the base case nodes are shared between the base and step cases, but in practice the base case nodes are duplicated in the step case. The step case begins with k copies of L , modified so that every statement *assert* ϕ is replaced with *assume* ϕ . These copies are denoted $L_1^a, \dots, L_k^a$ in Figure 4(c) (where a stands for *assume*). In each L_i^a , edges leading to the main program are *removed*, indicated by crosses in Figure 4(c). For $i < k$, back edges in L are replaced in L_i^a by edges to the header node of L_{i+1}^a . Duplicate L_k^a is followed by a further copy of L , denoted L_{k+1} , where assertions are *not* replaced by assumptions, back edges are *removed*, and edges leading to the main program are replaced with edges leading to the *base case*.

Showing correctness of the step case establishes that any trace of the original program that executes k iterations of L without going wrong can be extended to iterate a further iteration of L without going wrong. This corresponds to the sequence $L_1^a \rightarrow \dots \rightarrow L_k^a \rightarrow L_{k+1}$. If execution of this further iteration leaves L (via an edge from L_k to the main program), then the trace can be extended to continue executing the main program without going wrong, as long as such executions are restricted to no more than k iterations of L .

When we present this rule formally in §4, we will show that it is sound: correctness of both base and step case guarantees correctness of the original CFG. Essentially, the rule is sound because any path in the original CFG can be reconstructed by chaining together a path in the base case, followed by a sequence of paths in the step case.

The novel aspect of this rule in comparison to the k -induction rule of [9], or indeed the original k -induction procedure for transition systems [18], is that the guarantee of the base case, that up to k iterations of the given loop can be safely executed, is re-established by the step case.

Rule for multiple, disjoint loops. Suppose L and M are disjoint loops in a CFG, and assume that L and M do not target one another: no edge in the CFG goes directly from a node in L to a node in M , or vice-versa. (The latter condition can be trivially enforced by adding dummy nodes to the CFG.) It is clear that the base case and step case of Figure 4 can be extended, so that M is also unwound, up to depth k' (possibly with $k' \neq k$). More generally, the rule naturally extends to be applicable to an arbitrary set of disjoint loops that do not target one another. Unwinding can be performed with respect to a separate value of k for each loop.

3 Control flow graphs

We will now formally define the new k -induction rule as a transformation rule on control flow graphs. For our purposes, CFGs are directed graphs whose vertexes are labelled with statements. To simplify presentation, and in contrast to the schematic graphs shown so far, transition guards are encoded as *assume*-nodes; all definitions could straightforwardly be carried over to a richer control-flow model with labelled transitions and compound basic blocks.

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:

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

Intuitively, a nondeterministic assignment $x := *$ alters the value of x arbitrarily; an assumption *assume* ϕ suspends program execution if ϕ is violated and can be used to encode conditional statements, while an assertion *assert* ϕ raises an error if ϕ is violated. Neither *assume* ϕ nor *assert* ϕ 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 $(V, in, E, code)$, where V is a finite set of nodes, $in \subseteq V$ a set of initial nodes, $E \subseteq V \times V$ a set of edges, and $code : V \rightarrow Stmt$ a mapping from nodes to statements.

3.1 Loops and reducibility

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

Let $C = (V, in, E, code)$ be a CFG. For $u, v \in V$, we say that u *dominates* v if $u = v$, or if every path from a node of 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 $(u', v') \in E \wedge v' \in L_{(u,v)} \Rightarrow u' \in L_{(u,v)}$. For a node v such that there exists a back edge $(u, v) \in E$, the natural loop associated with v is the set $L_v = \bigcup_{(u,v) \text{ is a back edge}} L_{(u,v)}$. Node v is the *header* of loop L_v . For an arbitrary loop L , the *successors* of L is the set $succ(L) = \{v \mid v \notin L \wedge (u, v) \in E \text{ for some } u \in L\}$.

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, C is *irreducible*. It can be shown that every cyclic reducible CFG also contains natural loops, which will guarantee the applicability of our k -induction rule.³

3.2 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 $C = (V, in, E, code)$ be a CFG. The unwinding of C is defined as:

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

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: $S = \{\sigma \mid \sigma : X \rightarrow \mathcal{D}\} \cup \{\bot\}$.

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

$$post(\bot, s) = \{\bot\} \quad (\text{for any statement } s)$$

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

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

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

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

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:

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

³ Strictly, this is only true if all nodes in a CFG are reachable. For ease of presentation, we assume throughout that unreachable parts of CFGs are implicitly pruned away.

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

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

3.3 Inductive decompositions of control flow graphs

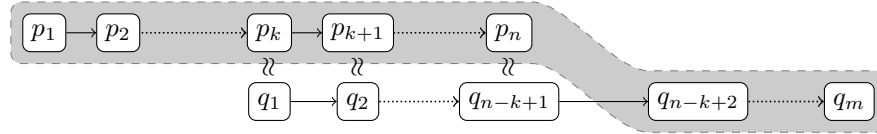
The k -induction rules that we have presented informally up to this point are all instances of a more general principle, in the following called *inductive decomposition*. All of the rules work by decomposing programs or CFGs C into a pair D_1, D_2 of simpler CFGs (the base and the step case), in such a way that the correctness of C follows from the correctness of D_1, D_2 . This construction is justified by proving that every statement sequence in the unwinding of C can be decomposed into sequences from the unwindings of D_1 and D_2 . To this end, we first define how to close a set of statement sequences under self-composition:

Definition 3. Given a set $P \subseteq Stmt^*$ of statement sequences (over a set of variables X), the set $gen(P) \subseteq Stmt^*$ of sequences generated by P is the least set containing the empty sequence (i.e., $\epsilon \in gen(P)$) and satisfying the following implication:

$$\begin{aligned} & \langle p_1, p_2, \dots, p_n \rangle \in gen(P), \langle q_1, q_2, \dots, q_m \rangle \in P, \\ & k \in \{n - m + 2, \dots, n + 1\}, \forall i \in \{0, \dots, n - k\}. q_{i+1} = p_{i+k}^{assume} \\ \implies & \langle p_1, p_2, \dots, p_n, q_{n-k+2}, \dots, q_m \rangle \in gen(P) \end{aligned}$$

where we define $(assert \phi)^{assume} = assume \phi$, and $s^{assume} = s$ for all other statements s .

The overlapping sequences of statements in the definition are illustrated by the following diagram:



Forming the extension of a set of statement sequences preserves correctness: if every sequence in the set P represents a correct program (path), then so does every sequence in $gen(P)$. Intuitively, this is because the concatenation of correct sequences $\langle p_1, p_2, \dots, p_n \rangle$ and $\langle q_{n-k+2}, \dots, q_m \rangle$ again forms a correct sequence, with the prefix $\langle q_1, q_2, \dots, q_{n-k+1} \rangle$ representing a pre-condition that is known to be established by $\langle p_1, p_2, \dots, p_n \rangle$. Vice-versa, if P is incorrect, then so is $gen(P)$, because $P \subseteq gen(P)$. A formal proof of this result is given in the appendix.

This observation gives rise to a general methodology for verifying programs: given a complex CFG C , simpler CFGs $D_1, \dots, D_n$ are constructed such that every statement sequence of C is generated by sequences of $D_1, \dots, D_n$. If it is possible to verify $D_1, \dots, D_n$, also the correctness of C has been established. More formally:

Definition 4. The CFGs $D_1, \dots, D_n$ cover the CFG C if

$$\text{unwinding}(C) \subseteq \text{gen}(\text{unwinding}(D_1) \cup \dots \cup \text{unwinding}(D_n)). \quad (1)$$

In this case, we also call $D_1, \dots, D_n$ an inductive decomposition of C .

Theorem 1. If CFG C is covered by correct CFGs $D_1, \dots, D_n$, then C is correct.

We prove Theorem 1 in Appendix A.

The converse of Theorem 1 does not hold, since the inclusion (1) might be strict; this means that inductive decomposition can introduce spurious and potentially incorrect paths. When using the k -induction rule, the set of spurious paths can be reduced by increasing the parameter k .

4 k -Induction for control flow graphs

We present our new k -induction rule formally as an inductive decomposition, and show that it is sound (§4.1). We then discuss the problem of preserving reducibility: guaranteeing that the base and step case associated with a reducible CFG are in turn reducible (§4.2).

4.1 Formal definition of the k -induction rule

We first present our k -induction rule formally for a single, arbitrary loop.

Let $C = (V, in, E, code)$ be a CFG, $L \subseteq V$ a loop in C with header h , and k a non-negative integer. Assume that $in \cap L = \emptyset$. (As discussed in §2.3, this scenario can always be avoided by adding *assume tt* nodes to C if necessary.)

We describe an inductive decomposition of C into two CFGs, $C_{b,k}^L$ (base case) and $C_{s,k}^L$ (step case), such that the loop L is eliminated in both $C_{b,k}^L$ and $C_{s,k}^L$.

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 .

Definition 5. Base case, cf. Figure 4(b). $C_{b,k}^L = (V_{b,k}^L, in_{b,k}^L, E_{b,k}^L, code_{b,k}^L)$ is defined as follows:

- $V_{b,k}^L = (V \setminus L) \cup \bigcup_{1 \leq i \leq k} L_i$
- $in_{b,k}^L = in$ (recall that, by assumption, in and L are disjoint)
- $E_{b,k}^L = \{ (u, v) \mid (u, v) \in E \wedge u, v \notin L \} \quad \text{Edges in Main program}$
 - $\cup \{ (u, h_1) \mid k > 0 \wedge (u, h) \in E \wedge u \notin L \} \quad \text{Main program} \rightarrow L_1$
 - $\cup \{ (u_i, h_{i+1}) \mid 1 \leq i < k \wedge (u, h) \in E \wedge u \in L \} \quad L_i \rightarrow L_{i+1}$
- $\cup \{ (u_i, v_i) \mid 1 \leq i \leq k \wedge (u, v) \in E \wedge u, v \in L \wedge v \neq h \} \quad \text{Edges in } L_i$
 - $\cup \{ (u_i, v) \mid 1 \leq i \leq k \wedge (u, v) \in E \wedge u \in L \wedge v \notin L \} \quad L_i \rightarrow \text{Main program}$

$$- \text{code}_{b,k}^L(u) = \begin{cases} \text{code}(v) & \text{if } u \text{ has the form } v_i \text{ for some } 1 \leq i \leq k \\ \text{code}(u) & \text{otherwise} \end{cases}$$

Definition 6. Step case, cf. Figure 4(c). $C_{s,k}^L = (V_{s,k}^L, in_{s,k}^L, E_{s,k}^L, \text{code}_{s,k}^L)$ is defined as follows:

$$\begin{aligned}
- V_{s,k}^L &= \bigcup_{1 \leq i \leq k} L_i^a \cup L_{k+1} \cup V_{b,k}^L \\
- in_{s,k}^L &= \begin{cases} \{h_1^a\} & \text{if } k > 0 \\ \{h_{k+1}\} & \text{otherwise} \end{cases} \\
- E_{s,k}^L &= \left\{ (u_i^a, v_i^a) \mid 1 \leq i \leq k \wedge (u, v) \in E \wedge u, v \in L \wedge v \neq h \right\} & \text{Edges in } L_i^a \\
&\quad \cup \left\{ (u_i^a, h_{i+1}^a) \mid 1 \leq i < k \wedge (u, h) \in E \wedge u \in L \right\} & L_i^a \rightarrow L_{i+1}^a \\
- &\quad \cup \left\{ (u_k^a, h_{k+1}) \mid k > 0 \wedge (u, h) \in E \wedge u \in L \right\} & L_k^a \rightarrow L_{k+1} \\
&\quad \cup \left\{ (u_{k+1}, v_{k+1}) \mid (u, v) \in E \wedge u, v \in L \wedge v \neq h \right\} & \text{Edges in } L_{k+1} \\
&\quad \cup \left\{ (u_{k+1}, v) \mid (u, v) \in E \wedge u \in L \wedge v \notin L \right\} & L_{k+1} \rightarrow \text{Main program} \\
&\quad \cup E_{b,k}^L & \text{Edges in base case} \\
- \text{code}_{s,k}^L &= \begin{cases} \text{assume } \phi \text{ if } u = v_i^a \text{ for some } 1 \leq i \leq k \text{ and } \text{code}(v) = \text{assert } \phi \\ \text{code}(v) \text{ if } u = v_i^a \text{ for some } 1 \leq i \leq k \text{ and } \text{code}(v) \neq \text{assert } \phi \\ \text{code}(v) \text{ if } u = v_{k+1} \\ \text{code}_{b,k}^L(u) \text{ otherwise} \end{cases}
\end{aligned}$$

The soundness of the k -induction rule follows from the fact that base and step case together form an inductive decomposition of the original program:

Lemma 1. *The CFGs $C_{b,k}^L$ and $C_{s,k}^L$ cover the CFG C .*

We prove Lemma 1 in Appendix B.

Corollary 1. *If $C_{s,k}^L$ and $C_{b,k}^L$ are correct, then C is correct.*

The k -induction rule can be generalised to apply simultaneously to an arbitrary set of disjoint loops. For a CFG $C = (V, in, E, \text{code})$, let $\mathbf{L} = (L_1, \dots, L_d)$ be a vector of loops in C , for some $d > 0$, and assume $L_i \cap L_j = \emptyset$ for $1 \leq i \neq j \leq d$. Let $\mathbf{k} = (k_1, \dots, k_d)$ be a vector of non-negative integers.

Assume that no initial nodes are contained in any of the loops ($L_i \cap in = \emptyset$, $1 \leq i \leq d$) and that the loops do not target one another ($\text{succ}(L_i) \cap L_j = \emptyset$ for all $1 \leq i \neq j \leq d$, where succ is defined as in §3.1). Both conditions can be trivially enforced by adding dummy nodes to C .

Definitions 5 and 6 can be naturally extended to inductively decompose C into two CFGs, $C_{b,\mathbf{k}}^L$ (base case) and $C_{s,\mathbf{k}}^L$ (step case), such that the loops $\mathbf{L}$ are eliminated in both $C_{b,\mathbf{k}}^L$ and $C_{s,\mathbf{k}}^L$ by unwinding loop L_i by depth k_i ($1 \leq i \leq d$). We present the definitions of $C_{b,\mathbf{k}}^L$ and $C_{s,\mathbf{k}}^L$ in Appendix C.

4.2 Preserving reducibility

The approach of §2.2 allows us to apply k -induction to an irreducible CFG, via a translation to reducible, monolithic form. However, this crude translation destroys loop structures in the CFG, limiting the extent to which our k -induction rule can be recursively applied. Thus, we would only like to apply the transformation of §2.2 as a last resort. In particular, we would like to ensure that given a reducible CFG C , the base case $C_{b,k}^L$ and step case $C_{s,k}^L$ generated by our rule are both reducible, for any choice of L and k . This guarantee would allow us to maximally decompose a CFG into cycle-free base and step cases.

It is easy to see that this guarantee holds for the $C_{b,k}^L$. Unfortunately, it is *not* guaranteed for $C_{s,k}^L$. However, we can employ a semantics-preserving transformation to $C_{s,k}^L$ to restore reducibility. Details of this transformation, which is linear in the size of C , are provided in Appendix D. In the remainder of the report, when we write $C_{s,k}^L$, we assume that this transformation has been applied.

5 A verification procedure using k -induction

We now discuss how the k -induction rule for inner loops defined in the previous section can systematically be applied to verify (potentially irreducible) programs with an arbitrary loop structure. There are two primary places where the resulting verification procedure is parameterised: (i) various loop selection strategies can be chosen, *e.g.*, preferring innermost or outermost loops; and (ii) a strategy for choosing the parameter k of the rule has to be defined.

5.1 The general procedure

The overall verification procedure is Algorithm 1, which applies the k -induction rule in order to verify a CFG C . The algorithm assumes functions `DECIDE` for deciding the correctness of a cycle-free program using bounded model checking, and `MONOLITHICISE` for turning an arbitrary CFG into a reducible CFG containing a single loop, as described in §2.2. In order to avoid the transformation of large CFGs into monolithic form (losing a lot of program structure), the function `MONOLITHICISE` is only applied if intermediate CFGs are derived that no longer contain natural loops, but which are not yet acyclic; when applying Algorithm 1 to CFGs that are reducible from the beginning, this never happens.

There are two recursive invocations of `ANALYSE` in Algorithm 1: one to establish correctness of the base case, and one for the step case. The correctness of the original program can only be concluded if both cases have been verified. The recursive calls are arranged to minimise the number of verification attempts:

- if the **base case** is known to be `INCORRECT`, also incorrectness of the original program can be concluded, so that no further verification attempts are made and the procedure returns.

Algorithm 1: ANALYSE

```
Input: CFG  $C$ 
Output: One of {CORRECT, INCORRECT, DON'T KNOW}
if  $C$  is loop-free then
  if  $C$  is reducible then                                // then  $C$  is acyclic
  | return DECIDE( $C$ );
  else
  | return ANALYSE(MONOLITHICISE( $C$ ));                    // first make  $C$  reducible
  end
else                                                        // apply the  $k$ -induction rule
  (*) choose loops  $\mathbf{L} = (L_1, \dots, L_d)$  in  $C$ , and depths  $\mathbf{k} = (k_1, \dots, k_d) \in \mathbb{N}^d$ ;
  |  $\text{base-case-result} \leftarrow \text{ANALYSE}(C_{b,k}^{\mathbf{L}})$ ;
  | if  $\text{base-case-result} = \text{CORRECT}$  then                    // base-case succeeded
  | |  $\text{step-case-result} \leftarrow \text{ANALYSE}(C_{s,k}^{\mathbf{L}})$ ;
  | | if  $\text{step-case-result} = \text{CORRECT}$  then                // step-case succeeded
  | | | return CORRECT;
  | | else
  | | | (**) either back-track to (*), or return DON'T KNOW;
  | | end
  | else                                                    // return INCORRECT or DON'T KNOW
  | | return  $\text{base-case-result}$ ;
  | end
end
```

- if the **base case** call returns DON'T KNOW, nothing can be concluded about the program. It might then be necessary to increase a $\mathbf{k}$ chosen at an earlier recursion level (increasing the $\mathbf{k}$ of the current ANALYSE invocation can only make the base case harder); thus the procedure returns.
- if the base case is CORRECT but the **step case** is not, nothing can be concluded about the program either. It might be necessary to increase some previously chosen $\mathbf{k}$; thus execution either back-tracks to (*) to choose a different $\mathbf{k}$, or the procedure returns.

A successful run of the procedure (with result CORRECT) can be seen as a binary tree whose leaves correspond to correctness checks on acyclic programs, while inner nodes represent applications of the k -induction rule; in the following, we call this tree the *verification tree*.

5.2 Exploiting parallelism

Algorithm 1 can be parallelised in two ways. First, the two recursive calls to ANALYSE can be issued in parallel, so that the step case for C is speculatively analysed before the result for the base case is known. Second, at (*), multiple choices for $(L_1, \dots, L_d)$ and $(k_1, \dots, k_d)$ can be tried in parallel. In this report, we investigate the former opportunity for parallelisation, leaving the latter for future work.

Having invoked ANALYSE on the base and step case CFGs in parallel, if the base case invocation returns a negative result before the step case invocation completes, the step case invocation can be cancelled. If the step case invocation returns a negative result first then there are two choices: (i) wait for the base case result; if the result is negative, the procedure ANALYSE can return with this result; or (ii) cancel the base case invocation and backtrack to (*). The first choice, which is used in our experiments (§6) can lead to faster verification if the program turns out to be correct, but can delay detection of an error otherwise.

5.3 Loop selection strategies

Algorithm 1 is parametric in the loop selection strategy, and in a strategy to choose suitable values of k (lines (*) and (**) in the program). These parameters can have a significant impact on the overall verification performance, both considering the theoretic complexity and empirical results (§6). For our experiments, we consider the following strategies for selecting loops at (*) in Algorithm 1:

- **Inward-1**: always choose a single outermost loop.
- **Outward-1**: always choose a single innermost loop.
- **Inward-all**: always choose *all* outermost loops simultaneously.
- **Outward-all**: always choose *all* innermost loops.

These strategies correspond to strategies (i), (iii), (iv) and (v) discussed in §1, respectively.

In addition, we can apply MONOLITHICISE to the input program before applying Algorithm 1. Since the resulting program contains only a single loop, the above loop selection strategies are irrelevant. We refer to this strategy as **Monolithic**.

To compare the above loop selection strategies theoretically (for the case of reducible CFGs), we examine the maximum number of leaves in resulting verification trees, *i.e.*, the number of programs that have to be verified using bounded model checking. It is easy to see that **Inward-all** and **Outward-all** produce the same number of leaves, namely 2^d , where d is the maximum depth of loop nests in the original program. This is because every application of k -induction to all outermost or all innermost loops decreases the maximum depth d by 1 in both base and step case. It is also clear that the number of leaves associated with the **Monolithic** strategy is always two.

The **Outward-1** strategy produces 2^n leaves, where n is the number of natural loops occurring in the original program, since every application of k -induction to an innermost loop eliminates exactly one loop and generates two new cases.

With the strategy **Inward-1**, the number of leaves depends on the nesting depth and on the number of loops in the original program. Note that the loop structure of a reducible CFG can be represented as a tree:

$$\langle \text{Loop-structure} \rangle ::= B \mid L(\langle \text{Loop-structure} \rangle) \mid \langle \text{Loop-structure} \rangle; \langle \text{Loop-structure} \rangle$$

where B denotes a loop-free piece of code, $L(s)$ denotes the occurrence of a loop with body s , and $s;t$ denotes composition of two disjoint pieces of code. The following recurrence equations describe how to compute the number $C(s)$ of leaves resulting

from the application of **Inward-1** to a program with loop structure s , assuming that the same value of k is used each time k -induction is applied:

$$C(B) = 1 \quad C(s; t) = C(s) \cdot C(t) \quad C(L(s)) = C(s)^k + C(s)^{k+1}$$

By solving the recurrence equations, it can be derived that the number of leaves using the **Inward-1** strategy grows doubly exponential in the maximum nesting depth, and singly exponential in the number of disjoint loops.

The theoretically predicted numbers of leaves are sometimes higher than the numbers actually observed in experiments (§6). This happens because some of the loops in base or step case might be unreachable, and are therefore pruned away in our implementation; such optimisations are not taken into account in the complexity analysis.

5.4 Strategies for selecting the parameter k

Once a vector $\mathbf{L} = (L_1, \dots, L_d)$ of loops has been chosen, it is necessary to select a parameter vector $\mathbf{k} = (k_1, \dots, k_d)$ describing how often each loop is to be unwound. Choosing a $\mathbf{k}$ that is too small can result in an unprovable step case, while large $\mathbf{k}$ lead to increased time and space consumption in subsequent verification steps (although large parameters do in principle not prevent successful verification). Because it is hard to compute optimal parameters $\mathbf{k}$ in general, implementations will practically enumerate different values of $\mathbf{k}$ until verification succeeds (result **CORRECT**), or until the incorrectness of the original program can be deduced from failing attempts to verify base cases (result **INCORRECT**).

It can be expected that heuristics and domain-specific knowledge (which are beyond the scope of this report) can help in practice to quickly find suitable values of $\mathbf{k}$. From a completeness point of view, the most essential property of an enumeration strategy is *fairness*: regardless of the precise enumeration order, it has to be guaranteed that every lower bound $k_0 \in \mathbb{N}$ is eventually exceeded by all values of $\mathbf{k}$ chosen during some program run. Although it has not been formally proven yet, we conjecture that the different loop selection strategies are equivalent with respect to their ability of proving program correctness or incorrectness under a fair enumeration strategy.

6 Experimental evaluation

We have implemented a k -induction tool for verification of non-recursive C programs, which we call **K-INDUCTOR**. **K-INDUCTOR** takes an input program and inlines all function calls, so that the program is represented by a single CFG. **K-INDUCTOR** analyses the CFG associated with the input program using the techniques of this report: the user can select any of the five k -induction strategies discussed in §5.3. The bounded model checker CBMC [6], together with MiniSat 2.0, is used as a decision procedure for cycle-free programs. **K-INDUCTOR**, and all benchmarks used for evaluation, are available online.⁴

⁴ <http://www.cprover.org/kinductor>

K-INDUCTOR applies the following fair enumeration strategy when analysing a vector of loops: k is initialised to $\mathbf{0}$, each component of k is increased by one each time verification of a step case returns DON'T KNOW, and the components of k are limited by upper bound 10. The tool supports a multithreaded mode, where verification is parallelized across multiple cores as discussed in §5.2.

6.1 Benchmarks and platform

We apply K-INDUCTOR to a set of benchmark programs provided with the IBM Cell SDK [15]. These are sequential programs written for the Synergistic Processor Element (SPE) cores of the Cell BE processor [14]. The programs perform basic data processing, using increasingly sophisticated data-movement strategies to transfer data between the main memory of the Cell architecture and private SPE local memory. Data-movement is performed using *direct memory access* (DMA) operations. A program can achieve efficient data movement by issuing multiple asynchronous DMA operations in parallel. However, DMA operations are notoriously hard to program correctly, and DMA *races*, where concurrently executing DMAs operate on a common region of memory, and at least one modifies the memory, lead to nondeterministic bugs that are difficult to diagnose and fix. We use the SCRATCH tool [9] to automatically instrument such programs with assertions to check DMA race freedom, and apply K-INDUCTOR to the resulting programs. Simplified version of these benchmarks, where inner loops were manually abstracted, were studied in [9]. The new techniques in this report for handling multiple loops allow us to avoid this manual abstraction step.

Each benchmark program uses either single, double or triple buffering for data-movement. For each buffering strategy there are two program variants, one where processing is performed in-place on a local buffer, and one where data is copied from an input buffer to a separate output buffer during processing. The latter variants are marked *IO* in our figures. Abstract interpretation is used to infer simple loop invariants, and a single strengthening assertion is added manually to the double and triple buffering examples; after this, all benchmarks can be verified using k -induction. We consider variants of the double and triple buffering examples where value-based trace partitioning [17] is applied manually to the input program to remove the need for manually-supplied assertions. These examples are marked *TP* in our figures.

Finding DMA races in buggy programs can be achieved using bounded model checking. We address the more challenging problem of proving *absence* of DMA races, thus we apply K-INDUCTOR to correct versions of all benchmarks.

We note that these benchmarks cannot be verified using abstract interpretation alone, with standard abstract domains. Furthermore, it has been shown that the benchmarks are not amenable to verification by existing CEGAR-based methods [9].

Figure 5 illustrates the size of each benchmark, showing in each case the number of lines of code after DMA instrumentation, and the number of loops. All benchmarks include at least one nested loop, and no benchmark contains loops beyond one level of nesting. Note that the number of loops increases significantly when value-based trace partitioning, which duplicates parts of the input program, is applied.

Experiments are performed on a 3.2GHz Intel Xeon machine with 48Gb RAM and 8 processing cores, running Ubuntu.

Benchmark	# lines	# loops	Benchmark	# lines	# loops
1-buf	152	2	2-buf-IO-TP	390	5
1-buf-IO	160	2	3-buf	397	4
2-buf	270	3	3-buf-TP	611	8
2-buf-TP	359	5	3-buf-IO	420	4
2-buf-IO	284	3	3-buf-IO-TP	1813	28

Fig. 5. Number of lines and number of loops, after inlining, for each benchmark

6.2 Results

	Monolithic			Inward-1			Outward-1			Inward-all			Outward-all		
Benchmark	time	cases	k	time	cases	k	time	cases	k	time	cases	k	time	cases	k
1-buf	0.64	6	2	1.16	16	1	0.91	12	2	0.71	10	1	0.91	12	2
1-buf-IO	1.43	8	3	1.78	16	1	1.44	12	2	1.10	10	1	1.44	12	2
2-buf	4.13	8	3	5.35	34	1	4.25	26	2	2.00	10	1	2.36	12	2
2-buf-TP	3.89	8	3	28.22	328	1	21.39	202	3	1.36	10	1	1.93	12	2
2-buf-IO	6.29	8	3	8.71	34	1	6.73	26	2	2.95	10	1	3.63	12	2
2-buf-IO-TP	7.91	8	3	98.12	328	1	68.77	202	3	3.42	10	1	5.13	12	2
3-buf	9.01	8	3	18.66	70	1	15.05	54	2	4.25	10	1	5.28	12	2
3-buf-TP	31.88	8	3	2296.97	5722	1	1594.08	2782	4	7.08	10	1	9.73	12	2
3-buf-IO	15.55	8	3	21.15	70	1	17.07	54	2	5.59	10	1	5.35	12	2
3-buf-IO-TP	207.33	8	3	timeout	≥ 5040	≥ 1	timeout	≥ 2368	≥ 6	50.40	10	1	61.07	12	2

Fig. 6. Experimental results for serial versions of each benchmark and strategy. For each benchmark, the fastest verification time is highlighted in bold

Figure 6 shows results obtained by applying K-INDUCTOR to our benchmarks, using the loop selection strategies of §5.3 and the default enumeration strategy described above. For each benchmark/strategy combination, we show the time (in seconds) for verification, the number of cycle-free programs verified by CBMC, and the largest value of k required for any loop. The maximum verification time was set to one hour.

The results clearly show the strength of the **Inward/Outward-all** strategies over the other strategies, with the **Inward-all** winning in almost all cases. We attribute this to the significantly smaller number of cases generated by these strategies in comparison to the **Inward/Outward-1** strategies, and the fact that, compared with the **Monolithic** strategy, the **Inward/Outward-all** strategies result in simpler loop-free programs, because the smallest possible value of k for each loop in the program is used.

It is not surprising that verification of the *3-buf-IO-TP* benchmark did not complete within one hour using the **Inward/Outward-1** strategies. As this benchmark contains 28 loops, our theoretical discussion of case explosion in §5.3 tells us that, given perfect values for k , verification using the **Outward-1** strategy would involve considering up to 2^{28} loop-free cases, while the **Inward-1** would lead to an even larger case explosion.

Results for parallelisation. Figure 7 shows the speedups over serial performance obtained by parallelising each benchmark across all eight cores of our Intel Xeon platform. A dash indicates that no comparison was possible due to verification of both serial and parallel benchmarks taking longer than one hour.

We see that for the **Inward/Outward-all** and **Monolithic** strategies, parallelism has little effect, resulting either a modest speedup or slow-down. This is because these

Benchmark	Monolithic	Inward-1	Outward-1	Inward-all	Outward-all
1-buf	0.97	1.58	0.93	1.29	0.94
1-buf-IO	1.00	1.64	0.97	1.37	0.97
2-buf	1.00	1.79	2.09	1.19	1.09
2-buf-TP	0.88	3.29	6.92	0.84	0.89
2-buf-IO	1.05	2.01	2.43	1.27	1.03
2-buf-IO-TP	0.62	4.96	8.82	1.07	1.17
3-buf	0.91	2.48	2.82	1.25	0.55
3-buf-TP	1.02	25.13	29.87	0.88	1.01
3-buf-IO	1.17	2.65	3.02	1.38	0.08
3-buf-IO-TP	0.58	-	-	0.90	0.89

Fig. 7. Speedups obtained via parallel execution on an 8-core system

strategies lead to a relatively small number of cases. Furthermore, we found that among these cases, SAT solving time for loop-free programs tends to be dominated by one or two large instances, limiting the parallel speedup that can be obtained without parallelising the SAT solver.

Excellent speedups are obtained for the **Inward/Outward-1** strategies when applied to the larger benchmarks. Super-linear speedups, of almost $30\times$ for the *3-buf-IO* benchmark with the **Outward-1** strategy, can be attributed to portions of the verification tree being pruned away by parallel threads: when a thread determines that a given base or step case fails, the thread immediately cancels threads working on verification tasks which are now redundant, launching new threads in their place. However, even with such super-linear speedups, verification of our benchmarks using parallel versions of the **Inward/Outward-1** strategies is invariantly slower than verification using serial versions of the **Inward/Outward-all** strategies. Gains due to parallelism do not ameliorate case explosion.

7 Related work

The concept of k -induction was first published in [18, 5], targeting the verification of hardware designs represented by transition relations (although the basic idea had already been used in earlier implementations [16] and a version of one-induction used for BDD-based model checking [7]). 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 [19], improving performance via incremental SAT solving [10], and supporting verification of temporal properties [2].

Besides hardware verification, k -induction has been used to analyse synchronous programs [12, 11]. 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 [9], from which we also draw the benchmarks used in this report; the k -induction rule of [9] is discussed in §2.1. A combination of the k -induction rule of [9], abstract interpretation, and domain-specific invariant strengthening techniques for DMA race analysis is the topic of [8].

Our general formulation of k -induction has similarities with techniques to generate verification conditions for unstructured code [3], which are used in systems like Spec# [4]. The main differences between our rule and [3] are that our rule does not use explicit loop invariants (but exploits assertions present in code), can work with multiple unwindings of loops ($k > 1$), and transforms programs into a base and a step case rather than a single acyclic CFG. In ongoing work, we are investigating extensions of the approach of [3] using k -induction, which we call *combined-case* k -induction. This allows the k -induction rule to exploit knowledge about program variables unchanged in a loop.

8 Summary, and future work

We have presented a general approach for applying k -induction to arbitrary sets of disjoint loops in non-recursive programs. We have investigated five different general-purpose strategies for applying the technique, and evaluated these strategies empirically via our implementation, K-INDUCTOR, using a range of benchmarks from the domain of DMA race analysis. The results clearly demonstrate that exploiting the structure of loops when applying k -induction is superior to applying a monolithic transform, as long as the k -induction rule is applied simultaneously to multiple loops.

It should be possible, in principle, to extend our approach to analyse programs with recursive procedure calls, reasoning inductively on recursion depth. We plan to refine our rules to take advantage of knowledge about program variables that are not modified by particular loops. We also plan to further investigate the potential for parallel verification, as outlined in §5.2.

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: Ernst, M.D., Jensen, T.P. (eds.) PASTE. pp. 82–87. ACM (2005)
4. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: CASSIS, Marseille, France. LNCS, vol. 3362. Springer (2005)
5. Bjesse, P., Claessen, K.: SAT-based verification without state space traversal. In: FMCAD. LNCS, vol. 1954, pp. 372–389. Springer (2000)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176. Springer (2004)
7. Déharbe, D., Moreira, A.M.: Using induction and BDDs to model check invariants. In: CHARME. IFIP Conference Proceedings, vol. 105, pp. 203–213. Chapman & Hall (1997)
8. Donaldson, A.F., Haller, L., Kroening, D.: Strengthening induction-based race checking with lightweight static analysis. In: VMCAI. LNCS, Springer (2011), to appear
9. 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)

10. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.* 89(4) (2003)
11. Franzén, A.: Using satisfiability modulo theories for inductive verification of Lustre programs. *Electr. Notes Theor. Comput. Sci.* 144(1), 19–33 (2006)
12. Hagen, G., Tinelli, C.: Scaling up the formal verification of Lustre programs with SMT-based techniques. In: *FMCAD*. pp. 109–117. IEEE (2008)
13. Harel, D.: On folk theorems. *Commun. ACM* 23(7), 379–389 (1980)
14. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: *HPCA*. pp. 258–262. IEEE Computer Society (2005)
15. IBM: Cell BE resource center (October 2009),
<http://www.ibm.com/developerworks/power/cell/>
16. Lillieroth, C.J., Singh, S.: Formal verification of FPGA cores. *Nord. J. Comput.* 6(3), 299–319 (1999)
17. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29(5) (2007)
18. 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)
19. Vimjam, V.C., Hsiao, M.S.: Explicit safety property strengthening in SAT-based induction. In: *VLSID*. pp. 63–68. IEEE (2007)

Appendix

A Proof of Theorem 1

We prove the stronger property that

$$\begin{aligned} &C_1, \dots, C_n \text{ are correct} \\ &\text{if and only if} \\ &gen(unwinding(C_1) \cup \dots \cup unwinding(C_n)) \text{ is correct} \end{aligned}$$

whereby we say that a set of statement sequences is correct if no trace of any sequence contains $\frac{1}{2}$.

In the whole proof, we use the abbreviation

$$P = unwinding(C_1) \cup \dots \cup unwinding(C_n).$$

“ $\Leftarrow$ ” holds because $P \subseteq gen(P)$.

“ $\Rightarrow$ ” this is shown via induction. We use the minimality principle and assume that $\langle r_1, r_2, \dots, r_l \rangle \in gen(P)$ is an incorrect statement sequence of minimal length; *i.e.*, $\langle r_1, r_2, \dots, r_l \rangle$ has a trace on which the error state $\frac{1}{2}$ occurs, and all sequences in $gen(P)$ of length $< l$ are correct.

Because ϵ trivially is a correct sequence of statements, we know that $l \geq 1$. Because $\langle r_1, r_2, \dots, r_l \rangle \in gen(P)$, the sequence has to be generated by the step condition of Def. 3, which means that there are $\langle p_1, p_2, \dots, p_n \rangle \in gen(P)$, $\langle q_1, q_2, \dots, q_m \rangle \in P$ and $k \in \{n - m + 2, \dots, n + 1\}$ such that $\forall i \in \{0, \dots, n - k\}. q_{i+1} = p_{i+k}^{assume}$ and

$$\langle r_1, r_2, \dots, r_l \rangle = \langle p_1, p_2, \dots, p_n, q_{n-k+2}, \dots, q_m \rangle.$$

Because of $k \geq n - m + 2$, we also know $m \geq n - k + 2$, and thus $n < k$. Due to the minimality of the incorrect sequence $\langle r_1, r_2, \dots, r_l \rangle$, this implies the correctness of $\langle p_1, p_2, \dots, p_n \rangle \in gen(P)$. We furthermore know that $\langle q_1, q_2, \dots, q_m \rangle \in P$ is correct by assumption.

Let $\langle \sigma_0, \sigma_2, \dots, \sigma_l \rangle \in traces(\langle r_1, r_2, \dots, r_l \rangle)$ be a trace on which $\frac{1}{2}$ occurs; say, $\sigma_o = \frac{1}{2}$. Because $\langle \sigma_0, \sigma_2, \dots, \sigma_n \rangle$ is also a trace of the correct sequence $\langle p_1, p_2, \dots, p_n \rangle$, the error state $\frac{1}{2}$ cannot occur in $\langle \sigma_0, \sigma_2, \dots, \sigma_n \rangle$, which means $o > n$. Because no errors occur on $\langle \sigma_0, \sigma_2, \dots, \sigma_n \rangle$, we know that $\langle \sigma_{k-1}, \sigma_2, \dots, \sigma_n \rangle$ is also a trace of $\langle q_1, \dots, q_{n-k+1} \rangle$, and thus that $\langle \sigma_{k-1}, \sigma_2, \dots, \sigma_l \rangle$ is a trace of $\langle q_1, q_2, \dots, q_m \rangle$. This implies that $\langle q_1, q_2, \dots, q_m \rangle$ is incorrect, however, contradicting the assumption that P is correct.

B Proof of Lemma 1

We assume that the k -induction rule is applied to the loop $L \subseteq V$ with loop header $h \in L$. As in the definitions in §4.1, we denote the original CFG, the base case, and the step

case by:

$$\begin{aligned} C &= (V, in, E, code) \\ C_{b,k}^L &= (V_{b,k}^L, in_{b,k}^L, E_{b,k}^L, code_{b,k}^L) \\ C_{s,k}^L &= (V_{s,k}^L, in_{s,k}^L, E_{s,k}^L, code_{s,k}^L). \end{aligned}$$

In $C_{s,k}^L$ we refer to nodes $L_1^a, \dots, L_k^a$ as the *induction hypothesis*, nodes L_{k+1} as the *continuation*, and the nodes of $C_{b,k}^L$ which form part of $C_{s,k}^L$ as the *rest of the program*.

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

Lemma 1 is proven by showing the inclusion

$$unwinding(C) \subseteq gen(unwinding(C_{b,k}^L) \cup unwinding(C_{s,k}^L)).$$

The empty sequence $\epsilon \in unwinding(C)$ is a generated sequence by definition. Thus, let $\langle v_1, v_2, \dots, v_n \rangle$ with $n > 0$, $v_1 = in$, and $\forall i \in \{1, \dots, n-1\}. (v_i, v_{i+1}) \in E$ denote an arbitrary non-empty path of C .

We infer a sequence $t_1, t_2, \dots, t_n \in \mathbb{N}$ counting the current number of loop iterations on $\langle v_1, v_2, \dots, v_n \rangle$:

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

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

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

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

$$\langle code(v_1), code(v_2), \dots, code(v_{p_a-1}) \rangle \in gen(unwinding(C_{b,k}^L) \cup unwinding(C_{s,k}^L)).$$

Base case ($a = 1$): consider the path $\langle v_1, v_2, \dots, v_{p_1-1} \rangle$. From the definition of p_1 and l_i , we know that $l_i \leq k$ for all $i \in \{1, \dots, p_1-1\}$. We translate $\langle v_1, v_2, \dots, v_{p_1-1} \rangle$ to the $C_{b,k}^L$ -path $\langle u_1, u_2, \dots, u_{p_1-1} \rangle$ as follows (recall that $w_i \in L_i$ are nodes of the i th copy of loop L):

$$u_i = \begin{cases} (v_i)_{l_i} & \text{if } v_i \in L \\ v_i & \text{otherwise} \end{cases}$$

Checking the definition of $C_{b,k}^L$ (Def. 5), it can be observed that $(u_i, u_{i+1}) \in E_{b,k}^L$ for all $i \in \{1, \dots, p_1-2\}$, which means that $\langle u_1, u_2, \dots, u_{p_1-1} \rangle$ is a genuine path of $C_{b,k}^L$. It follows that

$$\begin{aligned} code(u_1), code(u_2), \dots, code(u_{p_1-1}) &\in unwinding(C_{b,k}^L) \\ &\subseteq gen(unwinding(C_{b,k}^L) \cup unwinding(C_{s,k}^L)). \end{aligned}$$

Because of $code(u_i) = code(v_i)$, this proves the base case.

Step case ($a \rightarrow a + 1$): We assume, for an arbitrary $a \in \{1, \dots, m - 1\}$

$$\langle \text{code}(v_1), \text{code}(v_2), \dots, \text{code}(v_{p_a-1}) \rangle \in \text{gen}(\text{unwinding}(C_{b,k}^L) \cup \text{unwinding}(C_{s,k}^L)).$$

and show that the sequence $\langle \text{code}(v_1), \text{code}(v_2), \dots, \text{code}(v_{p_a-1}) \rangle$ can be extended using the step condition of Def. 3, concatenating with the suffix

$$\langle \text{code}(v_{p_a}), \text{code}(v_{p_a+1}), \dots, \text{code}(v_{p_{a+1}-1}) \rangle.$$

By definition of p_a , we know that $v_{p_a} = h$ and $l_{p_a} > k$. Let the index $s \in \mathbb{N}$ be the least number such that $\forall i \in \{s, \dots, p_a\}. l_i \geq l_{p_a} - k$; this implies that $v_s = h_l$, $l_s = l_{p_a} - k$, and that $\forall i \in \{s, \dots, p_a - 1\}. l_i \in [l_{p_a} - k, l_{p_a})$. Path $\langle v_s, v_{s+1}, \dots, v_{p_a-1} \rangle$ will constitute the *induction hypothesis* of the k -induction step case.

Similarly, let $t \in \{p_a, \dots, p_{a+1}\}$ be the maximum index such that $\{v_{p_a}, \dots, v_{t-1}\} \subseteq L$. By definition of l_i and p_i , it follows that $\forall i \in \{p_a, \dots, t-1\}. l_i = l_{p_a}$, as well as $h_l \notin \{v_{p_a+1}, \dots, v_{t-1}\}$. The path segment $\langle v_{p_a}, \dots, v_{t-1} \rangle$ will constitute the *continuation* of the k -induction step case, while the suffix $\langle v_t, \dots, v_{p_{a+1}-1} \rangle$ belongs to the *rest of the program*.

We then consider the path $\langle u_s, u_{s+1}, \dots, u_{p_{a+1}-1} \rangle$ defined by

$$u_i = \begin{cases} (v_i)_{l_i - l_s + 1}^a & \text{if } i < p_a \\ (v_i)_{k+1} & \text{if } p_a \leq i < t \\ (v_i)_{l_i} & \text{if } t \leq i \text{ and } v_i \in L \\ v_i & \text{otherwise} \end{cases}$$

Checking Def. 5 and Def. 6, we can observe that $(u_i, u_{i+1}) \in E_{s,k}^L$ holds for all $i \in \{s, \dots, p_{a+1} - 2\}$; this implies that $\langle u_s, u_{s+1}, \dots, u_{p_{a+1}-1} \rangle$ is a genuine path of the step case $C_{s,k}^L$. Furthermore, comparing the sequences of statements

$$\begin{array}{ccccccc} \text{code}(v_{p_a}), & \text{code}(v_{p_a+1}), & \dots, & \text{code}(v_{p_{a+1}-1}) \\ \text{code}_{s,k}^L(u_{p_a}), & \text{code}_{s,k}^L(u_{p_a+1}), & \dots, & \text{code}_{s,k}^L(u_{p_{a+1}-1}) \end{array}$$

we can observe that $\text{code}_{s,k}^L(u_i) = \text{code}(v_i)^{\text{assume}}$ for all $i \in \{p_a, \dots, p_{a+1} - 1\}$. This means that all premises of the step condition in Def. 3 are satisfied, and thus

$$\begin{aligned} & \langle \text{code}(v_1), \dots, \text{code}(v_{p_a-1}), \text{code}_{s,k}^L(u_{p_a}), \dots, \text{code}_{s,k}^L(u_{p_{a+1}-1}) \rangle \\ &= \langle \text{code}(v_1), \text{code}(v_2), \dots, \text{code}(v_{p_{a+1}-1}) \rangle \\ &\in \text{gen}(\text{unwinding}(C_{b,k}^L) \cup \text{unwinding}(C_{s,k}^L)) \end{aligned}$$

which concludes the step case.

C Formal details of k -induction rule for multiple loops

For a CFG $C = (V, in, E, \text{code})$, let $\mathbf{L} = (L_1, \dots, L_d)$ be a vector of loops in C , for some $d > 0$, and assume $L_i \cap L_j = \emptyset$ for $1 \leq i \neq j \leq d$. Let $\mathbf{k} = (k_1, \dots, k_d)$ be a vector of non-negative integers. Let h^i denote the header for loop L_i .

Assume that no initial nodes are contained in any of the loops ($L_j \cap in = \emptyset$, $1 \leq j \leq d$) and that the loops do not target one another ($succ(L_i) \cap L_j = \emptyset$ for all $1 \leq i \neq j \leq d$).

For $1 \leq j \leq d$ and $1 \leq i \leq k_j + 1$, let $L_{j,i} = \{v \mid v_i \in L_j\}$. Thus $L_{j,i}$ is a duplicate of L_j where each node is subscripted by i . Similarly, for $1 \leq j \leq d$ and $1 \leq i \leq k_j$, let $L_{j,i}^a = \{v_i^a \mid v \in L_j\}$. Thus $L_{j,i}^a$ is a duplicate of L_j where each node is subscripted by i and superscripted by a . Also define $L^* = \cup_{1 \leq j \leq d} L_j$.

Definition 7. Base case $C_{b,k}^L = (V_{b,k}^L, in_{b,k}^L, E_{b,k}^L, code_{b,k}^L)$ is defined as follows:

- $V_{b,k}^L = (V \setminus L^*) \cup \bigcup_{\substack{1 \leq j \leq d \\ 1 \leq i \leq k_j}} L_{j,i}$
- $in_{b,k}^L = in$ (recall that, by assumption, in and L^* are disjoint)
- $E_{b,k}^L = \{ (u, v) \mid (u, v) \in E \wedge u, v \notin L^* \}$
 $\cup \{ (u, h_1^j) \mid 1 \leq j \leq d \wedge k_j > 0 \wedge (u, h^j) \in E \wedge u \notin L_j \}$
 $\cup \{ (u_i, h_{i+1}^j) \mid 1 \leq j \leq d \wedge 1 \leq i < k_j \wedge (u, h^j) \in E \wedge u \in L_j \}$
- $\cup \{ (u_i, v_i) \mid 1 \leq j \leq d \wedge 1 \leq i \leq k_j \wedge (u, v) \in E \wedge u, v \in L_j \wedge v \neq h^j \}$
 $\cup \{ (u_i, v) \mid 1 \leq j \leq d \wedge 1 \leq i \leq k_j \wedge (u, v) \in E \wedge u \in L_j \wedge v \notin L_j \}$
- $code_{b,k}^L(u) = \begin{cases} code(v) & \text{if } u \text{ has the form } v_i \text{ for some } 1 \leq j \leq d \wedge 1 \leq i \leq k_j \\ code(u) & \text{otherwise} \end{cases}$

Definition 8. Step case $C_{s,k}^L = (V_{s,k}^L, in_{s,k}^L, E_{s,k}^L, code_{s,k}^L)$ is defined as follows:

- $V_{s,k}^L = \bigcup_{\substack{1 \leq j \leq d \\ 1 \leq i \leq k_j}} L_{j,i}^a \cup \bigcup_{1 \leq j \leq d} L_{j,k_j+1} \cup V_{b,k}^L$
- $in_{s,k}^L = \{(h_1^j)^a \mid 1 \leq j \leq d \wedge k_j > 0\} \cup \{h_{k_j+1}^j \mid 1 \leq j \leq d \wedge k_j = 0\}$
- $E_{s,k}^L = \{ (u_i^a, v_i^a) \mid 1 \leq j \leq d \wedge 1 \leq i \leq k_j \wedge (u, v) \in E \wedge u, v \in L_j \wedge v \neq h^j \}$
 $\cup \{ (u_i^a, h_{i+1}^j)^a \mid 1 \leq j \leq d \wedge 1 \leq i < k_j \wedge (u, h^j) \in E \wedge u \in L_j \}$
- $\cup \{ (u_{k_j}^a, h_{k_j+1}^j) \mid 1 \leq j \leq d \wedge k_j > 0 \wedge (u, h^j) \in E \wedge u \in L_j \}$
 $\cup \{ (u_{k_j+1}, v_{k_j+1}) \mid 1 \leq j \leq d \wedge (u, v) \in E \wedge u, v \in L_j \wedge v \neq h^j \}$
 $\cup \{ (u_{k_j+1}, v) \mid 1 \leq j \leq d \wedge (u, v) \in E \wedge u \in L_j \wedge v \notin L_j \}$
 $\cup E_{b,k}^L$
- $code_{s,k}^L = \begin{cases} \text{assume } \phi \text{ if } u = v_i^a \text{ for some } 1 \leq j \leq d \wedge 1 \leq i \leq k_j \text{ and} \\ \quad code(v) = \text{assert } \phi \\ code(v) \text{ if } u = v_i^a \text{ for some } 1 \leq j \leq d \wedge 1 \leq i \leq k_j \text{ and} \\ \quad code(v) \neq \text{assert } \phi \\ code(v) \text{ if } u = v_{k_j+1} \text{ for some } 1 \leq j \leq d \\ code_{b,k}^L(u) \text{ otherwise} \end{cases}$

As in §4.1, the soundness of this rule follows from the fact that base and step case together form an inductive decomposition of the original program:

Lemma 2. *The CFGs $C_{b,k}^L$ and $C_{s,k}^L$ cover the CFG C .*

The proof of Lemma 2 is analogous to the proof of Lemma 2, presented in Appendix B.

Corollary 2. *If $C_{s,k}^L$ and $C_{b,k}^L$ are correct, then C is correct.*

D Details of the reducibility-restoring transformation discussed in §4.2

To see how an irreducible step case can be generated from a reducible CFG, consider the step case illustrated graphically in Figure 4(c). Consider an edge (u, v) , where u belongs to loop copy L_{k+1} and v to the base case CFG (such an edge goes from Figure 4(c) to Figure 4(b)). Suppose that v belongs to some loop M that remains in the base case, with header h . If $v \neq h$ then adding the edge (u, v) means that M is no longer a natural loop: it is a cycle with two entry points, h and v .

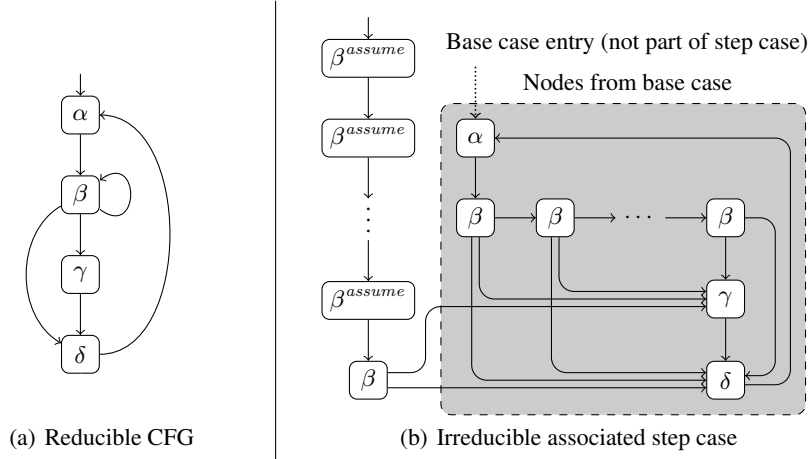


Fig. 8. Applying k -induction to a reducible CFG may yield an irreducible step case

As a concrete example, consider the CFG of Figure 8(a). The inner loop has two successors, γ and δ . The step case obtained by applying k -induction to the inner loop is shown in Figure 8(b). The step case consists of k copies of loop body β , with assertions replaced by assumptions and all edges to nodes outside the loop removed. This is followed by a further copy of β where edges leaving the loop are represented by edges targetting the base case CFG. The base case CFG is duplicated in the step case as indicated in Figure 8(b). The base case CFG contains a cycle, due to the outer loop in Figure 8(a). In the step case, this cycle has multiple entry points due to two edges coming from the continuation. Thus the step case CFG is not reducible.

Restoring reducibility. We avoid generating an irreducible step case when applying k -induction to a single loop L in a CFG C as follows. (The technique generalises directly to the multiple loop case.)

We associate a positive integer i_v with each node $v \in \text{succ}(L)$. We also introduce a single fresh temporary variable t . Let $v \in \text{succ}(L)$, and suppose v is contained in some loop of the original program. When building $C_{s,k}^L$, instead of adding an edge (u, v) from L_{k+1} to the base case CFG $C_{b,k}^L$, we add an edge (u, v') , where v' is a new node that sets t to i_v . We then add an edge from v' to the header for the outermost loop containing v . Let us use D to denote the intermediate CFG where this transformation has been applied. If C is reducible then clearly so is D : all offending edges have been replaced with edges to existing loop headers. We must now modify the D to ensure semantic equivalence with $C_{s,k}^L$.

This is achieved by considering in turn each loop M in D . We replace the header h_M of M with a dummy node d_M , replacing all edges in D of the form (u, h_M) with edges of the form (u, d_M) . We then consider each $v \in \text{succ}(L)$ such that $v \in M$. There are two possibilities. Either M *immediately* contains v , or v is contained in some sub-loop M' of M . In the first case, edges (d_M, v'') and (v'', v) are added to D , where edge (d_M, v'') is guarded by $t = i_v$. Here, v'' is a fresh node sets t to zero, then jumps to v . In the second case, an edge $(d_M, d_{M'})$, guarded by $t = i_v$, is added to D . Finally, we add an edge (d_M, h_M) from the new header for M to the old header. This edge is guarded by $t = 0$.

The second stage of this transformation clearly preserves reducibility. It also ensures that if t has value i_v , indicating that control has just left portion L_{k+1} of the step case, then execution will ripple through dummy loop headers from the outermost loop containing v to the loop that immediately contains v . At this point, t will be set to zero, and execution will reach v as desired. Because t is now zero, and can never be set to another value, further execution will skip over dummy loop headers, executing the base case part of the step case as usual. This matches the semantics of $C_{s,k}^L$.

The number of nodes and edges added to $C_{s,k}^L$ by this transformation is clearly linear in the size of C .

Figure 9 illustrates the result of applying this procedure to the irreducible step case of Figure 8, making it reducible.

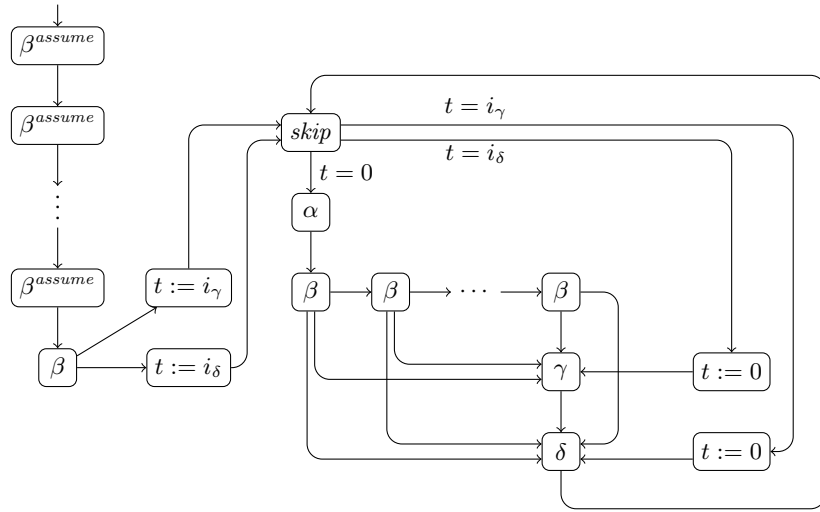


Fig. 9. Making the step case of Figure 8(b) reducible