Model Checking C Programs with Loops via *k*-Induction and Invariants

Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Raimundo Barreto

Federal University of Amazonas, Brazil

Abstract. We present a novel proof by induction algorithm, which combines k-induction with invariants to model check C programs with bounded and unbounded loops. The k-induction algorithm consists of three cases: in the base case, we aim to find a counterexample with up to k loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property P holds in all states reachable within k unwindings; and in the inductive step, we check that whenever P holds for k unwindings, it also holds after the next unwinding of the system. For each step of the k-induction algorithm, we infer invariants using affine constraints (i.e., polyhedral) to specify pre- and post-conditions. The algorithm was implemented in two different ways, with and without invariants using polyhedral, and the results were compared. Experimental results show that both forms can handle a wide variety of safety properties; however, the k-induction algorithm adopting polyhedral solves more verification tasks, which demonstrate an improvement of the induction algorithm effectiveness.

1 Introduction

The Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) [1] or Satisfiability Module Theories (SMT) [2] are successfully applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [3,4,5]. The idea behind the BMC techniques is to check the negation of a given property at a given depth, i.e., given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k.

Typically, BMC techniques are only able to falsify properties up to a given depth k; however, they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum iteration. In particular, BMC techniques limit the size of data structures (e.g., arrays) and the number of loop iterations to a given bound k. This also limits the state space that needs to be explored in software verification and has allowed BMC tools to find real errors in applications [3,4,5,6], but at the same time it has also made them susceptible to producing time-out or memory-out for programs that contain *unbounded loops* or programs where the number of loop unwindings cannot be determined statically.

Consider for example the simplistic program on the left of Fig. 1 in which the loop in line 2 runs an unknown number of times, depending on the initial value non-deterministically assigned to `x` in line 1. However, the assertion in line 3 holds independent of `x`'s initial value. Unfortunately, BMC tools like CBMC [3], LLBMC [4], or ESBMC [5] typically fail to verify programs that contain such loops. They insert a so-called *unwinding assertion* at the end of the loop, which consists of the negated loop bound. This enforces BMC tools to choose an unwind bound sufficiently large to search deeper in the state space of the program, but with the drawback of exhausting time and memory resources.

```
1 unsigned int x=*;
2 while(x>0) x--;
3 assert(x==0);
```

```
1 unsigned int x=*;
2 if (x>0)   \
3   x--;      } k copies
4 ...        /
5 assert(!(x>0));
6 assert(x==0);
```

Fig. 1: Unbounded loop (left) and finite unwinding (right)

One technique typically used to prove properties, for any given depth, is mathematical induction. The algorithm called k-induction was successfully applied to ensure that (restricted) C programs do not contain data races [7,8] and to respect time constraints specified during the design phase of a system [9]. Additionally, the k-induction is a well-established technique in hardware verification, where it is easier to be applied due to the monolithic transition relation present in hardware designs [9]. This paper contributes with a new algorithm to prove correctness of C programs by mathematical induction in a completely automatic way (i.e., the user does not need to provide the loop invariant).

The main idea of the algorithm is to use an iterative deepening approach and check, for each step k up to a maximum value, three different cases called here as base case, forward condition, and inductive step. Intuitively, in the base case, we intend to find a counterexample of ϕ with up to k iterations of the loop. The forward condition checks whether loops have been fully unrolled and the validity of the property ϕ in all states reachable within k iterations. The inductive step verifies that if ϕ is valid for k iterations, then ϕ will also be valid for the next unfolding of the system. For each step of the algorithm, we infer program invariants using affine constraints to prune the state space exploration and to strength the induction hypothesis.

These algorithms were all implemented in the Efficient SMT-based Context-Bounded Model Checker tool (known as ESBMC[1]), which uses BMC techniques and SMT solvers (e.g., [10,11]) to verify embedded systems written in C/C++ [5]. In Cordeiro et al. [5] the ESBMC tool is presented, which describes how the input program is encoded in SMT; what the strategies for unrolling loops are; what are the transformations/optimizations that are important for performance; what are the benefits of using an SMT solver instead of a SAT solver; and how counterexamples to falsify properties are reconstructed. Here we extend our previous work and focus our contribution on the combination of the k-induction algorithm with invariants. First, we describe the details of an accurate translation that extends ESBMC to prove the correctness of a given (safety)

[1] Available at http://esbmc.org/

property for any depth without manual annotations of loops invariants. Second, we adopt program invariants (using polyhedral) in the k-induction algorithm, to speedup the verification time and to improve the quality of the results by solving more verification tasks in less time. Third, we show that our present implementation is applicable to a broader range of verification tasks, where other existing approaches are unable to support [7,8,14].

2 Induction-based Verification of C Programs using Invariants

The transformations in each step of the k-induction algorithm take place in the intermediate representation level, after converting the C program into a GOTO-program, which simplifies the representation and handles the unrolling of the loops and the elimination of recursive functions.

2.1 The Proposed k-Induction Algorithm

Figure 2 shows an overview of the proposed k-induction algorithm. We do not add additional details about the transformations on each step of the algorithm; we keep it simple and describe the details in the next subsections so that one can have a big picture of the proposed method. The input of the algorithm is a C program P together with the safety property ϕ. The algorithm returns *true* (if there is no path that violates the safety property), *false* (if there exists a path that violates the safety property), and *unknown* (if it does not succeed in computing an answer *true* or *false*).

In the base case, the algorithm tries to find a counterexample up to a maximum number of iterations k. In the forward condition, global correctness of the loop w.r.t. the property is shown for the case that the loop iterates at most k times; and in the inductive step, the algorithm checks that, if the property is valid in k iterations, then it must be valid for the next iterations. The algorithm runs up to a maximum number of iterations and only increases the value of k if it can not falsify the property during the base case.

Differences to other k-Iduction Algorithms Our k-induction algorithm is slightly different than those presented by Große et al. [14], Donaldson et al. [7], and Hagen et al. [16]. In Große et al., the forward condition and the inductive step are computed differently from our approach (as described in Section 2.1) and the value of k is increased only at the end of the algorithm; in this particular case, computational resources are thus wasted since loops are usually unfolded at least two times. Donaldson et al. [7] and Hagen et al. [16] propose the k-induction with two steps only (i.e., the base case and the inductive step); however, the inductive step of the approach proposed by Donaldson et al. requires annotations in the code to introduce loops invariants, but our method is completely automatic as in Hagen et al [16]. Additionally, as observed in the experimental evaluation (see Section 3), the use of the forward condition, in our proposed method, improves significantly the quality of the results, because some programs that are hard to be proved by the inductive step can be proved by the forward condition using affine constraints.

```
input: program P and safety property $\phi$
output: true, false, or unknown
k = 1
while k <= max_iterations do
  if base_case(P, $\phi$, k) then
    show counterexample s[0..k]
    return false
  else
    k=k+1
    if forward_condition(P, $\phi$, k) then
      return true
    else
      if inductive_step(P, $\phi$, k) then
        return true
      end-if
    end-if
  end-if
end-while
return unknown
```

Fig. 2: An overview of the k-induction algorithm.

Loop-free Programs In the k-induction algorithm, the loop unwinding of the program is done incrementally from one to *max_iterations* (cf. Fig. 2), where the number of unwindings is measured by counting the number of *backjumps* [15]. On each step of the k-induction algorithm, an instance of the program that contains k copies of the loop body corresponds to checking a loop-free program, which uses only *if*-statements in order to prevent its execution in the case that the loop ends before k iterations.

Definition 1 ***(Loop-free Program)*** *A loop-free program is represented by a straight-line program (without loops) by providing an ite* (θ, ρ_1, ρ_2) *operator, which takes a Boolean formula* θ *and, depending on its value, selects either the second* ρ_1 *or the third argument* ρ_2*, where* ρ_1 *represents the loop body and* ρ_2 *represents either another ite operator, which encodes a k-copy of the loop body, or an assertion/assume statement.*

Therefore, each step of our k-induction algorithm transforms a program with loops into a loop-free program, such that correctness of the loop-free program implies correctness of the program with loops.

If the program consists of multiple and possibly nested loops, we simply set the number of loop unwindings globally, that is, for all loops in the program and apply these aforementioned translations recursively. Note, however, that each case of the k-induction algorithm performs different transformations at the end of the loop: either to find bugs (base case) or to prove that enough loop unwindings have been done (forward condition).

Program Translations In terms of program translations, which are all done completely automatic by our proposed method, the base case simply inserts an unwinding assumption, to the respective loop-free program P', consisting of the termination condition σ after the loop, as follows $I \wedge T \wedge \sigma \Rightarrow \phi$, where I is the initial condition, T is the transition relation of P', and ϕ is a safety property to be checked.

The forward case inserts an unwinding assertion instead of an assumption after the loop, as follows $I \wedge T \Rightarrow \sigma \wedge \phi$. The forward condition, proposed by Große et al. [14], introduces a sequence of commands to check whether there is a path between an initial state and the current state k, while in the algorithm proposed in this paper, an assertion (i.e., the loop invariant) is automatically inserted by our algorithm, without the user's intervention, at the end of the loop to check whether all states are reached in k steps. Our base case and forward condition translations can easily be implemented on top of plain BMC.

However, for the inductive step of the algorithm, several transformations are carried out. In particular, the loop $while(c)\,\{E;\}$ is converted into

$$A; while(c)\,\{S; E; U;\}\, R; \tag{1}$$

where A is the code responsible for assigning non-deterministic values to all loops variables, i.e., the state is havocked before the loop, c is the halt condition of the loop *while*, S is the code to store the current state of the program variables before executing the statements of E, E is the actual code inside the loop *while*, U is the code to update all program variables with local values after executing E, and R is the code to remove redundant states.

Definition 2 ***(Loop Variable)*** *A loop variable is a variable $v \subseteq V$, where $V = V_{global} \cup V_{local}$ given that V_{global} is the set of global variables and V_{local} is the set of local variables that occur in the loop of a program.*

Definition 3 ***(Havoc Loop Variable)*** *Nondeterministic value is assigned to a loop variable v if and only if v is used in the loop termination condition σ, in the loop counter that controls iterations of a loop; or repeatedly modified inside the loop body.*

The intuitive interpretation of S, U, and R is that if the current state (after executing E) is different than the previous state (before executing E), then new states are produced in the given loop iteration; otherwise, they are redundant and the code R is then responsible for preventing those redundant states to be included into the states vector. Note further that the code A assigns non-deterministic values to all loops variables so that the model checker can explore all possible states implicitly. Differently, Große et al. [14] havoc all program variables, which makes it difficult to apply their approach to arbitrary programs since they do not provide enough information to constrain the havocked variables in the program. Similarly, the loop *for* can easily be converted into the loop *while* as follows: $for(B; c; D)\,\{E;\}$ is rewritten as

$$B;\ while(c)\,\{E; D;\} \tag{2}$$

where B is the initial condition of the loop, c is the halt condition of the loop, D is the increment of each iteration over B, and E is the actual code inside the loop *for*. No further transformations are applied to the loop *for* during the inductive step. Additionally, the loop *do while* can trivially be converted into the loop *while* with one difference, the code inside the loop must execute at least once before the halt condition is checked.

The inductive step is thus represented by $\gamma \wedge \sigma \Rightarrow \phi$, where γ is the transition relation of $\hat{P}$, which represents a loop-free program (cf. Definition 1) after applying transformations (1) and (2). The intuitive interpretation of the inductive step is to prove that, for any unfolding of the program, there is no assignment of particular values to the program variables that violates the safety property being checked. Finally, the induction hypothesis of the inductive step consists of the conjunction between the postconditions ($Post$) and the termination condition (σ) of the loop.

Invariant Generation To infer program invariants, we adopted the PIPS [18] tool, which is an interprocedural source-to-source compiler framework for C and Fortran programs and relies on a polyhedral abstraction of program behavior. PIPS development has been driven for almost twenty years to automatic analysis of large size programs [19]. PIPS performs a two-step analysis: (1) each program instruction is associated to an affine transformer, representing its underlying transfer function. This is a bottom-up procedure, starting from elementary instructions, then working on compound statements and up to function definitions; (2) polyhedral invariants are propagated along with instructions, using previously computed transformers.

In our proposed method, PIPS receives the analyzed program as input and then it generates invariants that are given as comments surrounding instructions in the output C code. These invariants are translated and instrumented into the program as assume statements. In particular, we adopt the function `assume(`*expr*`)` to limit possible values of the variables that are related to the invariants. This step is needed since PIPS generates invariants that are presented as mathematical expressions (e.g., $2j < 5t$), which are not accepted by C programs syntax and invariants with $\#init$ suffix that is used to distinguish the old value from the new value.

2.2 Running Example

A program extracted from the benchmarks of the SV-COMP [17] is used as a running example as shown in Figure 3, which already includes invariants using polyhedral. In Figure 3, a is an integer constant and note that variables i and sn are declared with a type larger than the type of the variable n to avoid arithmetic overflow. Mathematically, the code above represents the implementation of the sum given by the following equation:

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1 \tag{3}$$

In the code of Figure 3, the invariants produced by PIPS are included as assume statements; the property (represented by the assertion in line 13) must

```
int main(int argc, char **argv)
{
 long long int i = 1, sn = 0;
 assume( i==1 && sn==0 ); // Invariant
 unsigned int n;
 assume(n>=1);
 while (i<=n) {
   assume( 1<=i && i<=n ); // Invariant
   sn = sn+a;
   i++;
 }
 assume( 1<=i && n+1<=i ); // Invariant
 assert(sn==n*a);
}
```

Fig. 3: Running example for the k-induction algorithm.

be *true* for any value of n (i.e., for any unfolding of the program). Differently from our k-induction algorithm, BMC techniques have difficulties in proving the correctness of this (simple) program since the upper limit value of the loop, represented by n, is non-deterministically chosen (i.e., the variable n can assume any value from one to the size of the *unsigned int* type, which varies between different types of computers). Due to this condition, the loop will be unfolded $2^n - 1$ times (in the worst case, $2^{32} - 1$ times on 32 bits integer), which is thus impractical. Basically, the bounded model checker would symbolically execute several times the increment of the variable i and the computation of the variable sn by $4,294,967,295$ times. To solve the problem of unfolding the loop $2^n - 1$ times, the translations previously described are performed.

The Base Case The base case initializes the limits of the loop's termination condition with non-deterministic values so that the model checker can explore all possible states implicitly. The pre- and postconditions of the loop shown in Figure 3, in static single assignment (SSA) form [15], are as follows:

$$Pre := \begin{bmatrix} n_1 = nondet\_uint \land n_1 \geq 1 \\ \land\ sn_1 = 0 \land i_1 = 1 \end{bmatrix}$$

$$Post := \begin{bmatrix} i_k \geq 1 \land\ i_k > n_1 \Rightarrow sn_k = n_1 \times a \end{bmatrix}$$

where *Pre* and *Post* are the pre- and postconditions to compute the sum given by Equation (3), respectively, and *nondet_uint* is a non-deterministic function, which can return any value of type *unsigned int*. In the preconditions, n_1 represents the first assignment to the variable n, which is a non-deterministic value greater than or equal to one. This ensures that the model checker explores all possible unwindings of the program. Additionally, sn_1 represents the first assignment to the variable sn and i_1 is the initial condition of the loop. In the postconditions, sn_k represents the assignment $n + 1$ for the variable sn in Figure 3, which must be *true* if $i_k > n_1$. The code that is not pre- or postcondition is represented by the variable Q (i.e., the sequence of commands inside the loop *for*) and it does not undergo any transformation during the base case. The resulting code of the base case transformations can be seen in Figure 4 (cf. Definition 1). Note that the *assume* (in line 11), which consists of the termination condition, eliminates all execution paths that do not satisfy the constraint $i > n$. This ensures that the base case finds a counterexample of depth k without reporting any false negative result. Note further that other assume statements, shown in Figure 3, are simply eliminated during the symbolic execution by propagating constants and checking that the resulting expression evaluates to *true* [5].

```
1  int main(int argc, char **argv) {
2    long long int i, sn=0;
3    unsigned int n=nondet_uint();
4    assume (n>=1);
5    i=1;
6    if (i<=n) {            }
7      sn = sn + a;         } k copies
8      i++;                 }
9    }
10   ...
11   assume(i>n); // unwinding assumption
12   assert(sn==n*a);
13 }
```

Fig. 4: Example code for the proof by mathematical induction, during base case.

The Forward Condition In the forward condition, the k-induction algorithm attempts to prove that the loop is sufficiently unfolded and whether the property is valid in all states reachable within k steps. The postconditions of the loop shown in Figure 3, in SSA form, can then be defined as follows:

$$Post := \left[i_k > n_1 \land sn_k = n_1 \times a \right]$$

The preconditions of the forward condition are identical to the base case. In the postconditions *Post*, there is an assertion to check whether the loop is sufficiently expanded, represented by the constraint $i_k > n_1$, where i_k represents the value of the variable i at iteration $n+1$. The resulting code of the forward condition transformations can be seen in Figure 5 (cf. Definition 1). The forward condition attempts to prove that the loop is unfolded deep enough (by checking the loop invariant in line 11) and if the property is valid in all states reachable within k iterations (by checking the assertion in line 12). As in the base case, we also eliminate assume expressions by checking whether they evaluate to *true* by propagating constants during symbolic execution.

The Inductive Step In the inductive step, the k-induction algorithm attempts to prove that, if the property is valid up to depth k, the same must be valid for the next value of k. Several changes are performed in the original code during this

```
int main(int argc, char **argv) {
  long long int i, sn=0;
  unsigned int n=nondet_uint();
  assume (n>=1);
  i=1;
  if (i<=n) {      }
    sn = sn + a;   } k copies
    i++;           }
  }
  ...
  assert(i>n); // check loop invariant
  assert(sn==n*a);
}
```

Fig. 5: Example code for the proof by mathematical induction, during forward condition.

step. First, a structure called *statet* is defined, containing all variables within the loop and the halt condition of that loop. Then, a variable of type *statet* called *cs* (current state) is declared, which is responsible for storing the values of a given variable in a given iteration; in the current implementation, the *cs* data structure does not handle heap-allocated objects. A state vector of size equal to the number of iterations of the loop is also declared, called *sv* (state vector) that will store the values of all variables on each iteration of the loop.

Before starting the loop, all loops variables (cf. Definitions 2 and 3) are initialized to non-deterministic values and stored in the state vector on the first iteration of the loop so that the model checker can explore all possible states implicitly. Within the loop, after storing the current state and executing the code inside the loop, all state variables are updated with the current values of the current iteration. An *assume* instruction is inserted with the condition that the current state is different from the previous one, to prevent redundant states to be inserted into the state vector; in this case, we compare $sv_j\,[i]$ to cs_j for $0 < j \leq k$ and $0 \leq i < k$. In the example we add constraints as follows:

$$\begin{aligned} & sv_1\,[0] \neq cs_1 \\ & sv_1\,[0] \neq cs_1 \wedge sv_2\,[1] \neq cs_2 \\ & \ldots \\ & sv_1\,[0] \neq cs_1 \wedge sv_2\,[1] \neq cs_2 \wedge \ldots sv_k\,[i] \neq cs_k \end{aligned} \tag{4}$$

Although, we can compare $sv_k\,[i]$ to all cs_k for $i < k$ (since inequalities are not transitive), the number of constraints can still grow very large quickly, and easily “blow-up” the SMT solver. In the SV-COMP benchmarks, we observed a substantial improvement in performance if we generate and check constraints as described in Equation (4).

Finally, an *assume* instruction is inserted after the loop, which is similar to that inserted in the base case. The pre- and postconditions of the loop shown in

Figure 3, in SSA form, are defined as follows:

$$Pre := \begin{bmatrix} n_1 = nondet\_uint \wedge n_1 \geq 1 \\ \wedge\ sn_1 = 0 \wedge i_1 = 1 \\ \wedge\ cs_1.v_0 = nondet\_uint \\ \wedge\ \ldots \\ \wedge\ cs_1.v_m = nondet\_uint \end{bmatrix}$$

$$Post := \left[i_k > n_1 \Rightarrow sn_k = n_{\times} a \right]$$

In the preconditions Pre, in addition to the initialization of the variables, the value of all variables contained in the current state cs must be assigned with non-deterministic values, where m is the number of (automatic and static) variables that are used in the program. The postconditions do not change, as in the base case; they only contain the property that the algorithm is trying to prove. In the instruction set Q, changes are made in the code to save the value of the variables before and after the current iteration i, as follows:

$$Q := \begin{bmatrix} sv[i-1] = cs_i \wedge S \\ \wedge\ cs_i.v_0 = v_{0i} \\ \wedge\ \ldots \\ \wedge\ cs_i.v_m = v_{mi} \end{bmatrix}$$

In the instruction set Q, $sv[i-1]$ is the vector position to save the current state cs_i, S is the actual code inside the loop, and the assignments $cs_i.v_0 = v_{0i} \wedge \ldots \wedge cs_i.v_m = v_{mi}$ represent the value of the variables in iteration i being saved in the current state cs_i. The modified code for the inductive step, using the notation defined in Section 2.1, can be seen in Figure 6. Note that the *if*-statement (lines 18-26) in Figure 6 is copied k-times according to Definition 1. As in the base case, the inductive step also inserts an *assume* instruction, which contains the termination condition. Differently from base case, the inductive step proves that the property, specified by the assertion, is valid for any value of n.

Lemma 1 *If the induction hypothesis $\{Post \ \wedge \ \neg(i \leq n)\}$ holds for $k+1$ consecutive iterations, then it also holds for k preceding iterations.*

After the loop *while* is finished, the induction hypothesis $\{Post \ \wedge \ \neg(i \leq n)\}$ is satisfied on any number of iterations; in particular, the SMT solver can easily verify Lemma 1 and conclude that $sn == n * a$ is inductive relative to n. As in previous cases, we also eliminate assume expressions by checking whether they evaluate to *true* by propagating constants during symbolic execution.

3 Experimental Evaluation

To evaluate the proposed method, we initially adopted the benchmarks of the SV-COMP 2015[2], in particular the *Loops* subcategory. The k-induction algorithm was implemented in two different ways, with and without invariants using polyhedral. The implementation of the algorithm with invariants is called

[2] http://sv-comp.sosy-lab.org/2015/

```
//variables inside the loop
typedef struct state {
 long long int i, sn;
 unsigned int n;
} statet;
int main(int argc, char **argv) {
 long long int i, sn=0;
 unsigned int n=nondet_uint();
 assume (n>=1);
 i=1;
 //declaration of current state
 //and state vector
 statet cs, sv[n];
 //A: assign non-deterministc values
 cs.i=nondet_uint();
 cs.sn=nondet_uint();
 cs.n=n;
 if (i<=n) { //c: halt condition
  sv[i-1]=cs;  //S: store current state
  sn = sn + a; //E: code inside the loop
  //U: update variables with local values
  cs.i=i; cs.sn=sn; cs.n=n;
  //R: remove redundant states
  assume(sv[i-1]!=cs);
  i++;
 }
 ...
 assume(i>n); //unwinding assumption
 assert(sn==n*a);
}
```

Fig. 6: Example code for the proof by mathematical induction, during inductive step.

DepthK [3]. The ESBMC v1.24.1 was adopted in both implementation. This way, we performed a comparison between DepthK (i.e., k-induction and invariants), ESBMC with k-induction, and ESBMC with plain BMC.

3.1 Experimental Setup

The experiments were conducted on a computer with Intel Core i7-2600, 3.40GHz with 24GB of RAM with Ubuntu 14.04.1 LTS 64-bit. Each verification task is limited to a CPU run time of 15 min and a memory consumption of 15 GB. Additionally, we defined the *max_iterations* to 100 (cf. Fig. 2).

Loops subcategory consists of 142 verification tasks, which are organized as follows: 49 benchmarks contain valid properties (i.e., the verification tool must be able to prove correctness) and 93 benchmarks contain invalid properties (i.e., the verification tool must be able to falsify the property).

[3] https://github.com/hbgit/depthk

3.2 Experimental Results

We evaluate the experimental results as follows: we adopt the same score scheme that is used by the SVCOMP rules[4]; in particular, we check the verification result and time presented by each implementation. Figure 7 shows the comparative results between the scores generated by DepthK with k-induction and invariants using polyhedral, ESBMC using k-induction only, and ESBMC using plain BMC. The total scores in the *Loops* subcategory for ESBMC with plain BMC is 66; ESBMC using k-induction only is 115; and DepthK combining k-induction and invariants is 141.

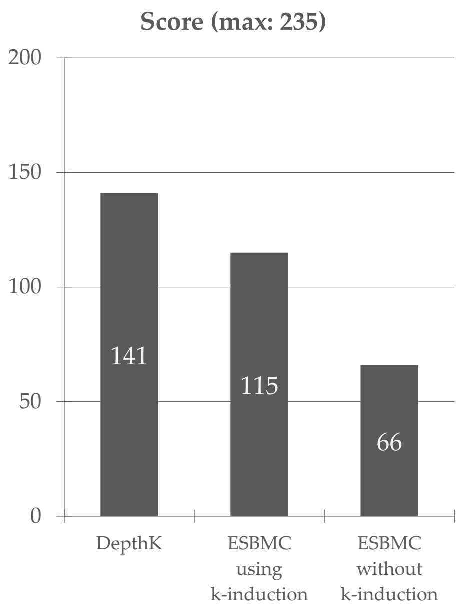

Fig. 7: Verification Scores

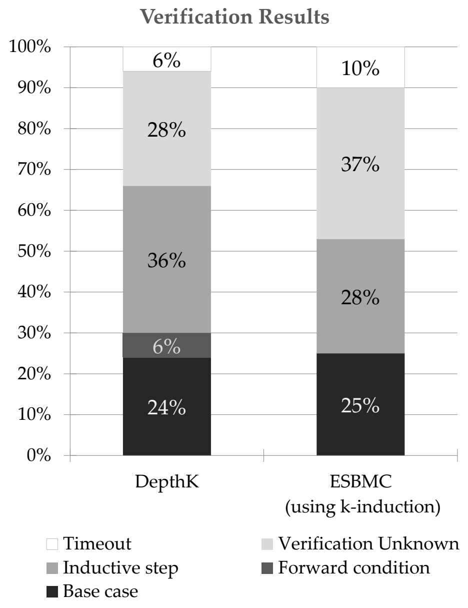

Fig. 8: Verification results

Figure 8 shows the distribution of the result by each step of the k-induction algorithm (i.e., base case, forward condition, and inductive step), including verifications that result in unknown and timeout. If we analyze the distribution of the results, we identified that DepthK was able to prove properties during the forward condition in 6% of the verification tasks, and ESBMC with k-induction proves properties only during the inductive step. As a result, we observe that invariants help prove that the loop is sufficiently unfolded and whether the property is valid, taking into account that this is performed in the forward condition step. DepthK has not found a solution in 28% of the verification tasks; this is explained by the invariants generated from PIPS, which could not generate strong invariants to be k-inductive either due to the transformers or due to the

[4] http://sv-comp.sosy-lab.org/2015/rules.php

invariants are not convex. ESBMC with k-induction does not find a solution in 37% of the verification tasks, therefore providing evidences that invariants can improve the verification results. DepthK and ESBMC with k-induction found a solution in 66% and 53% of the verification tasks, respectively. However, they reported 6% and 10% of timeouts, respectively.

Analyzing the verification time, we identified that ESBMC with k-induction verified all benchmarks in 15320 seconds; DepthK took approximately 11518 seconds, i.e., a reduction of 25% of the verification time; and ESBMC using plain BMC took about 3100 seconds (but solves less verification tasks).

4 Related Work

The application of the k-induction method is gaining popularity in the software verification community. Recently, Bradley et al. introduce "property-based reachability" (or IC3) procedure for the safety verification of systems [22,23]. The authors have shown that IC3 can scale on certain benchmarks where k-induction fails to succeed. However, we do not compare k-induction against IC3 since it is already done by Bradley [22]; we focus our comparison on related k-induction procedures.

Previous work on the one hand have explored proofs by mathematical induction of hardware and software systems with some limitations, e.g., requiring changes in the code to introduce loop invariants [7,8,14]. This complicates the automation of the verification process, unless other methods are used in combination to automatically compute the loop invariant [20,21]. Similar to the approach proposed by Hagen and Tinelli [16], our method is completely automatic and does not require the user to provide loops invariants as the final assertions after each loop. On the other hand, state-of-the-art BMC tools have been widely used, but as bug-finding tools since they typically analyze bounded program runs [3,4]; completeness can only be ensured if the BMC tools know an upper bound on the depth of the state space, which is not generally the case. This paper closes this gap, providing clear evidence that the k-induction algorithm can be applied to a broader range of C programs without manual intervention.

Große et al. describe a method to prove properties of TLM designs (Transaction Level Modeling) in SystemC [14]. The approach consists of converting a SystemC program into a C program, and then it performs the proof of the properties by mathematical induction using the CBMC tool [3]. The difference to the one described in this paper lies on the transformations carried out in the forward condition. During the forward condition, transformations similar to those inserted during the inductive step in our approach, are introduced in the code to check whether there is a path between an initial state and the current state k; while the algorithm proposed in this paper, an assertion is inserted at the end of the loop to verify that all states are reached in k steps.

Donaldson et al. describe a verification tool called Scratch [7] to detect data races during Direct Memory Access (DMA) in the CELL BE processor from IBM [7]. The approach used to verify C programs is the k-induction technique. The approach was implemented in the Scratch tool that uses two steps, the base case and the inductive step. The tool is able to prove the absence of data races, but it is restricted to verify that specific class of problems for a particular type of hardware. The steps of the algorithm are similar to the one proposed in this paper, but it requires annotations in the code to introduce loops invariants.

Kahsai et al. describe PKIND, a parallel version of the tool KIND, used to verify invariant properties of programs written in Lustre [24]. In order to verify a Lustre program, PKIND starts three processes, one for base case, one for inductive step, and one for invariant generation, note that unlike ESBMC, the k-induction algorithm used by PKIND does not have a forward condition step. The base case starts the verification with $k = 0$, and increments its value until it finds a counterexample or it receives a message from the inductive step process that a solution was found. Similarly, the inductive step increases the value of k until it receives a message from the base case process or a solution is found. The invariant generation process generates a set of candidates invariants from predefined templates and constantly feeds the inductive step process, as done recently by Beyer et al. [25] (we do not compare to Beyer et al. since their technical report appeared only after we submitted our CAV abstract and thus there was no time to further evaluate their work).

5 Conclusions

The main contributions of this work are the design, implementation, and evaluation of the k-induction algorithm adopting invariants using polyhedral in a verification tool, as well as, the use of the technique for the automated verification of reachability properties in heteregenous programs. To the best of our knowledge, this paper marks the first application of the k-induction algorithm to a broader range of C programs with loops. To validate the k-induction algorithm, experiments were performed involving 142 benchmarks of the SV-COMP 2015 *loops* subcategory. The experimental results also show that the k-induction algorithm without invariants was able to verify 52% of the benchmarks in 15320 seconds, and k-induction algorithm with invariants using polyhedral was able to verify 66% of the benchmarks in 11518 seconds, which gives a speedup of roughly 25% faster than the k-induction algorithm without the invariants version. Given a fixed timeout, this speedup can also improve the quality of the results (around 13%), because more programs can be verified if their verification would otherwise be interrupted by the time limit. In addition, both forms were able to prove or falsify a wide variety of safety properties; however, the k-induction algorithm adopting polyhedral solves more verification tasks, which demonstrate an improvement of the induction algorithm effectiveness.

References

1. A. Biere (2009), "Bounded model checking," in *Handbook of Satisfiability*, pp. 457–481, IOS Press.
2. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli (2009), "Satisfiability modulo theories," in *Handbook of Satisfiability*, pp. 825–885, IOS Press.
3. E. Clarke, D. Kroening, and F. Lerda (2004), "A tool for checking ANSI-C programs," in *TACAS, LNCS 2988*, Barcelona, Spain, March 29 - April 2, pp. 168–176.
4. F. Merz, S. Falke, and C. Sinz (2012), "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *VSTTE*, Philadelphia, PA, USA, January 28-29, pp. 146–161.

5. L. Cordeiro, B. Fischer, and J. Marques-Silva (2012), “SMT-based bounded model checking for embedded ANSI-C software,” *in TSE*, v. 38, 957–974.
6. F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang (2005), “Model checking C programs using F-SOFT,” *ICCD*, San Jose, CA, USA, October 2-5, pp. 297–308.
7. A. Donaldson, D. Kroening, and P. Rummer (2010), “Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors,” in *TACAS, LNCS 6015*, Paphos, Cyprus, March 20-28, pp. 280–295.
8. A. Donaldson, L. Haller, D. Kroening, and P. Rümmer (2011), “Software Verification Using k-Induction,” in *SAS, LNCS 6887*, Venice, Italy, September 14-16, pp. 351–368.
9. N. Eén and N. Sörensson (2003), “Temporal Induction by Incremental SAT Solving,” *Electr. Notes Theor. Comput. Sci.*, v. 89, 543–560.
10. L. M. de Moura and N. Bjørner (2008), “Z3: An efficient SMT solver,” in *TACAS, LNCS 4963*, Budapest, Hungary, March 29-April 6, pp. 337–340.
11. R. Brummayer and A. Biere (2009), “Boolector: An efficient SMT solver for bit-vectors and arrays,” in *TACAS, LNCS 5505*, York, UK, March 22-29, pp. 174–177.
12. J. Morse, L. Cordeiro, D. Nicole, and B. Fischer (2013), “Handling Unbounded Loops with ESBMC 1.20 - (Competition Contribution),” in *TACAS, LNCS 7795*, Rome, Italy, March 16-24, pp. 619–622.
13. M. Ramalho, L. Cordeiro, A. Cavalcante, V. Lucena (2013) “Verificação Baseada em Indução Matemática de Programas C/C++,” In SBESC, Niterói, Rio de Janeiro, November 4-8.
14. D. Große, H. M. Le, and R. Drechsler (2009), “Induction-Based Formal Verification of SystemC TLM Designs,” in *MTV*, Austin, Texas, December 7-9, pp. 101–106.
15. S. Muchnick (1997), *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc.
16. G. Hagen and C. Tinelli (2008), “Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques,” in *FMCAD*, Portland, Oregon, USA, November 17-20, pp. 1–9.
17. D. Beyer (2013), “Second competition on Software Verification - (Summary of SV-COMP 2013),” in *TACAS, LNCS 7795*, Rome, Italy, March 16-24, pp. 594–609.
18. Vivien M., Olivier H. and FranÃ§ois I. (2014), “Computing Invariants with Transformers: Experimental Scalability and Accuracy,” in *NSAD*, pp. 17-31.
19. MINES-ParisTech, PIPS (1989-2009). Available at http://pips4u.org. Open Source under GPLv3.
20. R. Sharma, I. Dillig, T. Dillig, and A. Aiken (2011), “Simplifying loop invariant generation using splitter predicates,” in *CAV, LNCS 6806*, Snowbird, UT, USA, July 14-20, pp. 703–719.
21. C. Ancourt, F. Coelho, and F. Irigoin (2010), “A modular static analysis approach to affine loop invariants detection,” *Electr. Notes Theor. Comput. Sci.*, v. 267, pp. 3–16.
22. A. Bradley (2012), “IC3 and Beyond: Incremental, Inductive Verification, *CAV, LNCS 7358*, Berkeley, CA, USA, July 7-13, pp. 4, Springer.
23. A. Bradley (2013), “Better Generalization in IC3, *FMCAD*, Porland, OR, USA, October 20-23, pp. 157–164.
24. T. Kahsai, C. Tinelli (2011), “PKind: A parallel k-induction based model checker,” in *PDMC*, Snowbird, Utah, USA, July 14, pp. 55–62.
25. D. Beyer, M. Dangl, and P. Wendler, “Combining k-Induction with Continuously-Refined Invariants,” Technical Report, Number MIP-1503, University of Passau, 31st January 2015.