Noname manuscript No.
(will be inserted by the editor)

Verification and Refutation of C Programs based on *k*-Induction and Invariant Inference

Omar M. Alhawi[1] · **Herbert Rocha**[2] · **Mikhail R. Gadelha**[3] · **Lucas Cordeiro**[1] · **Eddie Batista**[4]

Received: date / Accepted: date

Abstract DepthK is a source-to-source transformation tool that employs bounded model checking (BMC) to verify and falsify safety properties in single- and multi-threaded C programs, without manual annotation of loop invariants. Here, we describe and evaluate a proof-by-induction algorithm that combines *k*-induction with invariant inference to prove and refute safety properties. We apply two invariant generators to produce program invariants and feed these into a *k*-induction-based verification algorithm implemented in DepthK, which uses the efficient SMT-based context-bounded model checker (ESBMC) as sequential verification back-end. A set of C benchmarks from the international competition on software verification (SV-COMP) and embedded-system applications extracted from the available literature are used to evaluate the effectiveness of the proposed approach. Experimental results show that *k*-induction with invariants can handle a wide variety of safety properties, in typical programs with loops and embedded software applications from the telecommunications, control systems, and medical domains. The results of our comparative evaluation extend the knowledge about approaches that rely on both BMC and *k*-induction for software verification, in the following ways. (1) The proposed method outperforms the existing implementations that use *k*-induction with an interval-invariant generator (*e.g.*, 2LS and ESBMC), in the category ConcurrencySafety, and overcame, in others categories, such as SoftwareSystems, other software verifiers that use plain BMC (*e.g.*, CBMC). Also, (2) it is more precise than other verifiers based on the property-directed reachability (PDR) algorithm (*i.e.*, SeaHorn, Vvt and CPAchecker-CTIGAR). This way, our methodology demonstrated improvement over existing BMC and *k*-induction-based approaches.

[1]University of Manchester, UK · [2]Federal University of Roraima, Brazil · [3]SIDIA Instituto de Ciência e Tecnologia · [4]TP Vision, Brazil

Keywords Software engineering · Formal methods · Bounded model checking · K-induction · Invariant inference.

1 Introduction

Computer-based systems have been applied to different domains (*e.g.*, industrial, military, education, and wearable), which generally demand high-quality software, in order to meet a target system's requirements. In particular, (critical) embedded systems, such as those in the avionics and medical domains, impose several restrictions (*e.g.*, response time and data accuracy) that must be met and measured, according to users' requirements; otherwise, failures may lead to catastrophic situations. As a result, software testing and verification techniques are essential ingredients for developing systems with high dependability and reliability requirements, where it is necessary to ensure both user requirements and system behaviour.

Bounded model checking (BMC) techniques, either based on boolean satisfiability (SAT) [12] or satisfiability modulo theories (SMT) [5], have been successfully applied to check single and multi-threaded programs and then find subtle bugs in real programs [23, 60, 33, 10, 20]. The idea behind BMC 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 a given 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 can falsify properties up to a given depth k; however, they are not able to prove system correctness, unless an upper bound of k is known, *i.e.*, a bound that unwinds all loops and recursive functions to their maximum possible depths. In summary, BMC techniques limit the visited regions of data structures (*e.g.*, arrays) and

the number of related loop iterations. Thus, BMC restricts the state space that needs to be explored during verification, in such a way that real errors in applications can be found [23, 60, 33, 50]. Nonetheless, BMC tools are susceptible to exhaustion of time or memory limits, when verifying programs with loop bounds that are too large.

For instance, in the simple program illustrated in Fig. 1a, where the star notation indicates non-determinism, the loop in line 4 runs an unknown number of times, depending on the initial non-deterministic value assigned to `N`, in line 1, and the assertions in lines 8 and 9 hold independently of the `N`'s initial value. Unfortunately, BMC tools like the C bounded model checker (CBMC) [23], the low-level bounded model checker (LLBMC) [60], and the efficient SMT-based context-bounded model checker (ESBMC) [33] typically fail to verify this family of programs. That happens because such tools must insert an *unwinding assertion* (the negated loop condition) at the end of the loop, as illustrated in Fig. 1b, line 9, which will fail if k is not set to the maximum value supported by the *unsigned int* data type.

```
1 unsigned int N=*;
2 unsigned int i = 0;
3 long double x=2;
4 while( i < N ){
5   x = ((2*x) - 1);
6   ++i;
7 }
8 assert( i == N );
9 assert(x>0);
```

(a) Simple unbounded loop program.

```
1  unsigned int N=*;
2  unsigned int i = 0;
3  long double x=2;
4  if( i < N ){
5    x = ((2*x) - 1);   } k copies
6    ++i;               }
7  }
8  ...
9  assert( !(i < N) );
10 assert( i == N );
11 assert(x>0);
```

(b) Finite unwinding done by BMC.

Figure 1: Unbounded loop and finite unwinding.

In mathematics, one usually tackles such unbounded problems using *proof by induction*. As a consequence, an approach called k-induction has been successfully combined with continuously-refined invariants [10]. There were also attempts to prove, via k-induction, that (restricted) C programs do not contain data races [26, 25] or that time constraints are respected [30]. Additionally, k-induction is a well-established technique in hardware verification, due to monolithic transition relations present in hardware designs [30, 39, 68]. Finally, regarding the unknown-depth problem mentioned earlier, BMC tools can still be used to prove correctness in those cases, if used as part of k-induction algorithms.

In this respect, some approaches usually require invariants to be manually annotated with their values. For example, Donaldson *et al.* [27, 28], were able to increase the precision of the invariant by manually applying *trace partitioning*[64], a refinement technique for abstract domains that enables inference of disjunctive invariants using non-disjunctive domains, while others resort to static analyses for generating invariants [10, 27] that are later refined, which then strengthen associated induction hypotheses. Nonetheless, a complete automatic methodology for producing strong (inductive) invariants would be beneficial, mainly if consists in direct logical evolution, given that no user interaction or additional refinement is required.

The last paragraphs inspired this work, whose main contribution is a methodology for combining invariant generation and k-induction in order to prove correctness of programs written in the C language. Besides, verification-process automation is also tackled, where users do not need to provide loop invariants, *i.e.*, conditions that hold before a loop, are preserved through each loop iteration, and act as properties that are true at a particular program location. Moreover, the adopted invariant-generation tools were integrated through specific interfacing layers, due to their distinct formats, in order to provide a unified solution based on the proposed approach.

As a consequence, we have added a new module to the ESBMC tool, which employs mathematical induction with invariant inference, in order to prove the correctness of programs containing loops and then evaluate the proposed methodology. Such a module implements an algorithm that executes three steps: base case, forward condition, and inductive step [34]. In the first, the goal is to find a counterexample of size k, while the second one checks whether loops have been fully unrolled, which is achieved by verifying that no *unwinding assertion* fails, and, finally, the third one verifies if a property holds indefinitely, where the mentioned integration with invariants occurs.

The proposed method infers program invariants to prune the state space exploration and to strengthen induction hypotheses. Additionally, to provide a practical and complete implementation of the proposed methodology, two invariant-generation tools were used: paralléliseur interprocédural de programmes scientifiques (PIPS) [62] and path analysis for invariant generation by abstract interpretation (PAGAI) [45]. The proposed method was implemented in a tool called DepthK [65, 66], which rewrites programs using invariants generated by PIPS or PAGAI, *i.e.*, it adds those elements as assumptions and uses ESBMC to verify the resulting program with k-induction [34].

This study is a revised and extended version of previous work [65, 66] and focuses on contributions regarding combi-

nation of k-induction with invariant inference. In particular, the main original contributions of this paper are as follows:

- We describe the original k-induction and our extended version, which combines programs with invariants generated by either PIPS or PAGAI, in order to strengthen its inductive step. In particular, we presented technical details of the combination process for both PIPS and PAGAI. Then, we concluded by presenting an illustrative example in order to demonstrate the effectiveness of our tool. Indeed, the mentioned example was extracted from the International Competition on Software Verification (SV-COMP) 2019 [8], and our extended k-induction was able to prove its correctness, but plain k-induction was not (Section 3).
- We analyze and compare the results of our tool against other existing software verifiers that implement k-induction and property-directed reachability (PDR). In particular, we use ESBMC with k-induction and interval-invariant generator [33], CBMC [53], 2LS [19], SeaHorn [42], Vvt [41], and CPAchecker-CTIGAR [13] (see Section 4). Experimental results showed that k-induction with invariant inference could handle a wide variety of safety properties, in typical programs with loops and embedded software applications from the telecommunications, control systems, and medical domains. Our k-induction proof rule with polyhedral program invariants was able to solve 2223 verification tasks, *i.e.*, it has proved correctness in 602 and found property violations in 1621 benchmarks. These results outperformed other software verifiers, including 2LS, CBMC and ESBMC, in some particular categories (*e.g.*, *SoftwareSystems* and *ConcurrencySafety*), which thus demonstrates improvement over existing BMC and k-induction-based approaches. Besides, our proposed method using PAGAI confirms the hypothesis that DepthK is competitive when compared to the best available PDR-based tool implementations.

Outline. In Section 2.1, we first give a brief introduction to the BMC and k-induction techniques and also compare them with PDR (or incremental construction of inductive clauses for indubitable correctness - IC3) [16, 43]. Section 3 presents our induction-based verification algorithm using polyhedral invariant inference for specifying pre- and post-conditions, which works for C programs. In Section 4, the results of our experiments are described by using several software-model-checking benchmarks extracted from SV-COMP and embedded systems applications. In Section 5, we discuss the related work and, finally, Section 6 presents this work's conclusions.

2 Background

2.1 Bounded Model Checking

BMC based on SAT [12] was initially proposed in the early 2000s to verify hardware designs [12, 11]. Indeed, a group of researchers at Carnegie Mellon University were able to successfully check large digital circuits with approximately 9510 latches, and 9499 inputs, leading to BMC formulae with 4×10^6 variables and 1.2×10^7 clauses to be checked by standard SAT solvers [11]. BMC based on SMT [5], in turn, was initially proposed by Armando *et al.* [3], in order to deal with ever-increasing software-verification complexity.

Generally speaking, BMC techniques aim to check the violation of a given (safety) property at a given system depth. Indeed, given a transition system M, which is derived from the control-flow graph of a program, a property ϕ, which represents program correctness and/or a system's behaviour, and an iteration bound k, which limits loop unrolling, data structures, and context-switches. BMC techniques thus unfold a transition system M k times, in order to convert it into a verification condition ψ, which is expressed in propositional logic or in a decidable-fragment of first-order logic. For example, ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k. The propositional problem associated with SAT-based BMC is formulated as [11]

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg\phi_k, \tag{1}$$

where ϕ_k represents a safety property ϕ at step k, I is the set of initial states of M, and $R(s_i, s_{i+1})$ is the transition relation of M at time steps i and $i+1$. Hence, the equation $\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ means the set of all executions of M with length k and $\neg\phi_k$ represents the condition that ϕ is violated in state k, which is reached by a bounded execution of M with length k. Finally, the resulting (bit-vector) equation is translated into conjunctive normal form in linear time and passed to a SAT solver for checking satisfiability. Eq. (1) can be used to check safety properties [63] (e.g., deadlock freedom), while liveness ones (*e.g.*, starvation freedom) that contain the linear-time temporal Logic (LTL) operator F are verified by encoding $\neg\phi_k$ in a loop within a bounded execution of length at most k, such that ϕ is violated on each state in that loop [61]. This way, Eq. 1 can be rewritten as

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{k} \neg\phi_i \right), \tag{2}$$

where ϕ_i is the propositional variable ϕ at time step i. Thus, this formula can be satisfied if and only if, for some i ($i \leq k$), there exists a reachable state at time step i in which ϕ is violated.

One may notice that BMC analyzes only bounded program runs, but generates verification conditions (VCs) that

reflect the exact path in which a statement is executed, the context in which a given function is called, and the bit-accurate representation of expressions. In this context, a verification condition is a logical formula (constructed from a bounded program and desired correctness properties) whose validity implies that a program's behaviour agrees with its specification [18]. Users can specify correctness properties in our context via *assert* statements or automatically generated from a specification language [4]. If all of a bounded program's VCs are valid, then a program complies with its specification up, to the given bound.

BMC tools tend to fail due to memory or time limits if programs with loops whose bounds are too large or cannot be statically determined are verified. In addition, even if a program does not contain a violation up to a given bound k, nothing can be said about k+*1*. Consequently, such limitations has motivated researchers to develop new verification techniques, in order to go deep into a program's search space and, at the same time, prove global correctness. In particular, two possible strategies have been proposed in the literature, in order to achieve that goal: k-induction [30, 68] and IC3 [16, 43], which are briefly described in the following sections.

2.2 Induction-based Verification of C Programs

One approach to achieve completeness in BMC techniques is to prove that an invariant (assertion) is k-inductive using SAT/SMT solvers [30, 68]. The main challenge regarding such a technique relies on computing and strengthening inductive invariants from programs, in order to prove global correctness. In particular, full verification requires, as a crucial step, inference of each loop with a **loop invariant** [32], which is a logical formula that is an abstract specification of a loop. Therefore, loop invariants provide the means to reason about loops and to prove their correctness. According to the Xujie *et al.* [69] inferring loop invariants enables a broad and deep range of correctness and security properties to be proven automatically by a variety of program verification tools spanning type checkers, static analyzers, and theorem provers. Moreover, loop invariants must be inductive in order to check satisfiability for the corresponding VCs, as described by Bradley and Manna [18].

Si *et al.* [69] define loop invariant inference by introducing Hoare logic [46] for proving program-correctness assertions. Let P (pre-condition) and Q (post-condition) denote predicates over program variables and also let S denote a program under evaluation. Based on Hoare rules, such triples can be inductively derived over the structure of S. This way, we can highlight the following one regarding loops:

$$\frac{P \implies I \quad \{I \wedge B\} S \{I\} \quad (I \wedge \neg B) \implies Q}{\{P\} \quad \textbf{while } B \textbf{ do } S \quad \{Q\}} \tag{3}$$

,

where predicate I (the inductive invariant) is called a loop invariant, *i.e.*, an assertion that holds before and after each iteration, as shown in the premise of the rule, and B is a predicate on a program state. Thus, if a loop is equipped with an invariant, proving its correctness means establishing the two following hypotheses [32]:

- The initialization ensures the invariant, which is called initiation property;
- The body preserves the invariant, which is called consecution (or inductiveness) property.

For instance, consider the C-code fragment shown in Figure 1. Suppose that one wants to prove that $P : x > 0$ is invariant. In order to attempt proving the invariant property P, one can apply induction considering that the underlying software-model checker supports IEEE floating-point standard (IEEE 754) [49, 38]:

- In the base case, it holds initially because

$$\underbrace{N = * \wedge i = 0 \wedge x = 2}_{\text{initial condition}} \implies \underbrace{x > 0}_{\text{P}};$$

- In the inductive step, whenever P holds for k loop unwindings, it also holds for $k + 1$ steps, *i.e.*,

$$\underbrace{x > 0}_{\text{P}} \wedge \underbrace{x' = 2 * x - 1 \wedge i' = i + 1}_{\text{transition relation}} \implies \underbrace{x' > 0}_{\text{P'}}.$$

Specifically, if we consider the IEEE 754 standard [49, 38], then the invariant $x > 0$ holds initially and after each iteration and x tends to infinity after 128 iterations, so $x > 0$ is a candidate for a loop invariant. Nonetheless, this invariant is not inductive, given that $x > 0$ before an initial iteration does not ensure that $x > 0$ after each iteration, given that if we initially assign $x = 0.9$, then $x < 0$ after the fourth iteration. As a consequence, even if invariant-generation procedures successfully compute such assertions, which are indeed invariant, those must be inductive, so that k-induction verifiers can automatically prove global correctness. In this specific example, an inductive invariant would be $x > 1$, given that if $x > 1$ holds before the initial iteration, then $x > 1$ also holds after k iterations.

Several invariant-generation algorithms discover linear and polynomial relations among integer and real variables, in order to provide loop invariants and also to discover memory "shapes", in programming languages with pointers, such as those used in PIPs and PAGAI [62, 45]. The current literature regarding that also reports a significant increase in effectiveness and efficiency, while outperforms all previous implementations of k-induction-based verification algorithms for C programs, using invariant generation and strengthening, mostly based on interval analysis [10].

Novel verification algorithms for proving correctness of (a large set of) C programs, by mathematical induction and in a completely automatic way (*i.e.*, users do not need to

provide loop invariants), were recently proposed [34, 10, 19, 65, 25, 66]. Additionally, k-induction based verification was also applied to ensure that (restricted) C programs (1) do not contain violations related to data races [26], considering the Cell BE processor, and (2) do respect time constraints, which are specified during system design phases [30]. Apart from that, k-induction is easily applied, due to the monolithic transition relation present in such designs [30, 68, 39].

Note that k-induction with invariants has the potential to be directly integrated into existing BMC approaches, given that the induction algorithm itself can be seen as an extension after k unwindings. It is possible to generate program invariants with other software modules, which are then translated and instrumented into an input program [65].

2.3 Property-directed Reachability (or IC3)

While BMC is very effective in finding counterexamples, it is indeed incomplete, due to the bound limitation. This weakness motivated the development of IC3 and other complete SAT-based approaches. In particular, Bradley *et al.* [16, 43] introduced IC3, which is also known as PDR. IC3 aims to find an inductive invariant F stronger than P, *i.e.*,

$$\begin{aligned} &INIT \Rightarrow F \\ &F \wedge T \Rightarrow F' \\ &F \Rightarrow P, \end{aligned} \tag{4}$$

where $INIT$ describes the set of initial states and T represents the set of transitions, by learning relatively inductive facts (incrementally) locally. Indeed, that is carried out by iteratively computing an over-approximated reachability sequence $F_0, F_1, ..., F_{l+1}$, such that

$$\begin{aligned} &F_0 = INIT \\ &F_i => F_{i+1} \\ &F_i \wedge T \Rightarrow F'_{i+1} \\ &F_i \Rightarrow P. \end{aligned} \tag{5}$$

In summary, starting from the initial states, every assignment that satisfies the current clause F_i also satisfies the next one (F_{i+1}), every reachable state satisfies the next clause, and a given property is satisfied in every clause, *i.e.*, P is an invariant up to $k + 1$. As a result, if Eq. (5) is performed, F becomes an inductive invariant stronger than P, as shown in Eq. (4).

In fact, IC3 is a procedure for safety verification of systems and some studies have shown that IC3 can scale on specific benchmarks, where k-induction fails to succeed. In particular, the success of IC3 over k-induction procedures is due to the ability of the former to guide a search for inductive instances with counterexamples to inductiveness (CTIs) of a given property. Besides, the previous SAT-based approaches require unrolling the transition relation T (cf. Eqs. 4 and 5), in order to search for an inductive invariant and to strengthen it; however, IC3 performs no unrolling, given that it learns relatively inductive facts locally.

A CTI is a state (more generally, a set of states represented by a cube, *i.e.*, a conjunction of literals) that is a counterexample to consecution [17]. Consider again the C-code fragment shown in Figure 1, where $P : x > 0$ is an invariant, but assume now that x has been initialized with a non-deterministic value, in order to make it harder to infer an invariant, as given by

$$\underbrace{x > 0}_{\text{P}} \wedge \underbrace{x' = 2 * x - 1 \wedge i' = i + 1}_{\text{transition relation}} \Rightarrow \underbrace{x' > 0}_{\text{P'}}.$$

In that specific example, one possible CTI returned by a SAT/SMT solver is $x = 0$. If this particular state is not eliminated from the search space performed by the solver, then the invariant P cannot be established, since it is not inductive, given the initial assignment to x. Indeed, the generated inductive assertion should establish that the CTI $x = 0$ is unreachable and if such an inductive assertion does not exist, then other CTIs can be examined instead (*e.g.*, 0.1, 0.2, . . . , 0.9). As a consequence, the resulting lemmas must be strong enough that consecutively revisiting a finite set of CTIs will eventually lead to an assertion, which is inductive relative to them, thus eliminating the proposed CTI. In the mentioned example, the loop invariant candidate $x > 1$ is inductive and thus eliminates all possible CTIs in our running example.

Recent work has been done to improve the IC3's strengths further, in order to prove safety properties. One notable study was performed by Jovanović *et al.* [52], which presents a reformulation of the IC3 technique by separating reachability checking from inductive reasoning. In particular, those authors further replace the regular induction algorithm by the k-induction proof rule and show that it provides more concise invariants than the original approach proposed by Bradley [16]. Additionally, the mentioned authors implemented that proof rule in the SALLY model checker[1], using the SMT solver Yices2[2], in order to perform the forward search, and MathSAT5[3], to perform backward search. Finally, they showed that their proposed algorithm could solve several real-world benchmarks, at least as fast as other existing approaches.

3 Induction-based Verification of C Programs Using Invariants

In this section, we describe the main contribution of the present paper: a verification methodology for combining k-

[1] https://github.com/SRI-CSL/sally
[2] http://yices.csl.sri.com/
[3] http://mathsat.fbk.eu/

induction and loop invariant generation, which was implemented in a tool named as DepthK[4]. The first step of our methodology consists in generating invariants for a given ANSI-C program, which is performed with external tools to strengthen the associated inductive step. Indeed, PIPS and PAGAI were used for such a purpose, with invariants included as comments in different formats, which led to the development of distinct integration layers for each invariant generator, as described in Sections 3.3 and 3.4. In that sense, future invariant-generation tools would then only need new integration layers, when used along with our verification methodology. Moreover, one may notice that PIPS and PAGAI are suitable for the C language and have the potential to handle a wide variety of safety properties [58].

Although other invariant generators beyond PIPs and PAGAI do exist, such as accelerated symbolic polyhedral invariant computation (ASPIC) and integer set library (ISL), the latter do not handle automatic abstraction and some specific details of the C programming language, such as pointer arithmetic, as mentioned by Maisonneuve *et al.* [58]. Moreover, PIPS and PAGAI present different configuration options, which lead to different results and can also be explored with the goal of pruning state-spaces. Specifically, PIPS [58] is an inter-procedural source-to-source compiler framework for C and Fortran, based on automatic static analysis, which relies on polyhedral abstraction of program behaviour for inferring invariants, while PAGAI [45] is a source code analysis algorithm based on abstract interpretation with linear domains (products of intervals, octagons, and polyhedra) and path focusing, which can generate inductive invariants.

3.1 The *k*-Induction Algorithm

Algorithm 1 shows an overview of the *k*-induction algorithm used in ESBMC [34], which is an extended version of the original *k*-induction initially proposed by Eén and Sörensson [30]. It takes a program *P*, as input, and returns FALSE if a property violation is found, TRUE if it can prove correctness, or UNKNOWN.

In the base case (lines 4-6), the algorithm tries to find a counterexample up to k steps. If no property violation is found, then the forward condition (lines 7-8) checks whether the completeness threshold[5] is reached at the current k step. Finally, the inductive step (lines 11-12) checks whether the property ϕ holds indefinitely. If ϕ is valid for k iterations, then it must be valid for the next ones. The algorithm runs up to a certain number of repetitions and only increases the value of k if it cannot falsify the property during the base case. One may notice that k is incremented only at the start of the else branch, on line 10. In our benchmarks,

```
   Input: Program P
   Output: TRUE, FALSE, or UNKNOWN
1  begin
2      k = 1;
3      while k <= max_iterations do
4          if baseCase (P, k) then
5              show the counterexample s[0 . . . k];
6              return FALSE;
7          else if forwardCondition (P, k) then
8              return TRUE;
9          else
10             k=k+1;
11             if inductiveStep (P, k) then
12                 return TRUE;
13             end
14         end
15     end
16     return UNKNOWN;
17 end
```

Algorithm 1: The *k*-induction algorithm.

we also noticed that computational resources are wasted if we start with $k = 1$ in the inductive step since loops are usually unfolded at least two times. The properties checked by each step are generated during their execution, as follows. In the base case and also in the inductive step, in addition to user-defined properties (using `assert` statements), safety properties such as out-of-bounds checks, pointer validity, and division by zero are derived from a program and checked. In the forward condition, only the completeness threshold is checked, which is done in a C program, by using *unwinding assertions*, *i.e.*, it verifies whether all loops were unrolled entirely. There exists no need to check any other properties in the forward condition, as all of them were already checked for the current step k. Those properties are used to assign a non-deterministic value of program variables that participate in a loop.

Although *k*-induction is a successful technique to falsify or prove correctness, the over-approximation employed by the inductive step is unconstrained and might present spurious counterexamples. For example, traces produced by a failed inductive step in *k*-induction are a feasible sequence of k+1 transitions from an arbitrary loop iteration to an error state, where that loop iteration itself may be unreachable. Currently, our main idea as described by Gadelha *et al.* [35], is to search simultaneously forward, from the initial state, and backward, from the error state, whose procedure stops if those two searches meet halfway. This extension aims to convert the *k*-induction algorithm into a bidirectional search approach by using the base case as the forward part and the inductive step as the backward one.

3.2 Extended *k*-Induction Algorithm

Our technique generates invariants (using external tools). It creates a copy of an input program, which is modified by the

[4] `https://github.com/hbgit/depthk`

[5] The completeness threshold defines a bound k such that, if no counterexample of length k or less to a given LTL formula is found, then the formula, in fact, holds over all infinite paths in the model [55]

inductive step of the k-induction algorithm in order to over-approximate its loops. Algorithm 2 describes our extended k-induction algorithm combined with invariants.

```
   Input: Program P, Tool T
   Output: TRUE, FALSE, or UNKNOWN
1  begin
2      Inv = genInvariants (P, T);
3      P' = combine (P, Inv);
4      k = 1;
5      while k <= max_iterations do
6          if baseCase (P', k) then
7              show the counterexample s[0...k];
8              return FALSE;
9          else if forwardCondition (P', k) then
10             return TRUE;
11         else
12             k=k+1;
13             if inductiveStep (P', k) then
14                 return TRUE;
15             end
16         end
17     end
18     return UNKNOWN;
19 end
```

Algorithm 2: Our extended k-induction algorithm.

Similarly to the original k-induction algorithm, our extended version returns FALSE, TRUE, or UNKNOWN, depending on the result of each step. However, it takes three inputs: the original program P, the property ϕ to be checked, and the chosen tool, which is either PIPS or PAGAI.

In the first step of the extended Algorithm 2 (line 2), the chosen tool T is called and it tries to generate invariants for program P. In case of tool failure or no invariant generated, it returns $\varnothing$; otherwise, it returns a set of invariants Inv. The second step is to combine the set of invariants Inv and the original program P, in order to create P' (Line 3). This process is specific to each tool, as invariants are generated in different formats, and, as a consequence, such elements need to be preprocessed before being combined with P. Function `combine` is defined in

$$\texttt{combine} := \left[\, P' := ite(Inv = \varnothing, P, P \wedge Inv) \,\right], \tag{6}$$

where the new program P' is the result of an ite operation. If the set of invariants is $\varnothing$, P' is the original program P; otherwise, $P' = P \wedge Inv$. The technical description of the combination of the original program with invariants generated by PIPS and PAGAI is described in Sections 3.3 and 3.4, respectively.

Finally, the last modification to the base algorithm is to use program P' in every step, during verification, instead of P (lines 6, 9, and 13).

In order to provide reliable results, we have integrated the available witness checkers [8] into DepthK (*i.e.*, CPAchecker and Ultimate Automatizer). After achieving a conclusive result (*true* or *false*), DepthK generates an exstensible markup language (XML) file that contains all program states, *i.e.*, from the initial state to the bad one, which lead to a given property violation. This file is known as the *witness file*. Currently, there exist two state-of-the-art tools able to perform witness validation: CPAchecker [6] and Ultimate Automizer [44]. DepthK currently handles this witness file as follows:

1. It is submitted to CPAchecker [6] for validation and, if the DepthK's result is confirmed, then it is provided to users;
2. If CPAchecker is unable to confirm the result provided by DepthK or if there exists an internal failure in CPAchecker, then it is submitted to Ultimate Automizer [44] and, if the DepthK's result is confirmed, that is provided to users;
3. If both tools are unable to evaluate it, then the result is considered *UNKNOWN*, *i.e*, the witness-validation procedure is unsuccessful in confirming the DepthK's verification result.

Validation of witness files became a rule in SV-COMP, as a way of deeply assessing verification results provided by a given verifier since it is possible to confirm, fully automatically, if values used in state-space exploration, which are available in a counterexample, lead to a correct result.

3.3 Invariant Generation using PIPS

PIPS aims to process large programs by performing a two-step analysis [62] automatically. Firstly, each program instruction is associated with an affine transformer, representing its underlying transfer function. Indeed, that is a bottom-up procedure, which starts from elementary instructions and then goes through compound statements, up to function definitions. Secondly, polyhedral invariants are propagated along with instructions, using previously computed transformers.

PIPS takes a C program as input, generates invariants, and prints a C program with those invariants as comments before each program statement. Then, we process those comments and instrument source code using assume statements.[6]

Each comment must be preprocessed before being added to a C program, as those invariants generated by PIPS contain the suffix `#init` and includes mathematical expressions (*e.g.*, $2j < 5t$). For instance, Figure 2 shows transformers, preconditions, and generated syntax for the program described in [58], using PIPS. One may notice that those mathematical expressions do not contain a multiplication sign between constant and variable names, which does not consist in a valid C syntax.

In Algorithm 3, we describe the process of combining an original C program P with a set of invariants. As previously

[6] ESBMC requires assume statements to be written using `__ESBMC_assume(bool)`

```
1
2  //T() {0==-1}
3  void foo(float x){
4  // T(n){n==0}
5   int n = 0;
6  // T(n){n#init==0}
7      while(1)
8    // T(n){n<=n#init+1}
9          if(x)
10   // T(n){n<=60, n<=n#init+1}
11           if(n<60)
12   // T(n){n==n#init+1, n<=60}
13             n++;
14           else
15   // T(n){n==0, 60<=n#init}
16             n = 0;
17 }
```

Figure 2: Sample with PIPS invariants using the structure #init.

defined, P is the original program, $InvSet$ is the set of invariants, and P' is the combination of the original program P and invariants $InvSet$. The complexity of that algorithm is $O(n^2)$, where n is the code size with invariants generated by PIPS. Algorithm 3 is split into three parts: (1) identification of structures `#init`, (2) generation of code to support translation of strutures `#init` into invariants, and (3) translation of the related mathematical expressions into ANSI-C code.

The first part of Algorithm 3 is performed in Line 7, which consists of reading each line of $InvSet$ and identifying whether a given comment is an invariant generated by PIPS (line 8) or not. If an invariant is identified and it contains a structure `#init`, then its location (*i.e.*, its line number) defined by *Inv.line* is stored, as well as type and name of the associated variable. After identifying structures `#init` in invariants, the second part of this algorithm analyzes each code line in $InvSet$, but now with the goal of identifying the beginning of each programming function (line 15). For each function that is identified, this algorithm checks whether it has structures `#init` (line 18) and, when that is true, a new code line is generated, for each related variable and at the beginning of the same function, with the declaration of an auxiliary variable, which contains a variable's old value, *i.e.*, its initial value. The newly created variable has the format `variable.type var_init = variable.name` , where `variable.type` is its identified type and `variable.name` is its identified (original) name.

Finally, each line containing a PIPS invariant is turned into expressions supported by the ANSI-C standard. Such a transformation consists of applying regular expressions (line 26) to multiplication operators (*e.g.*, from $2j$ to $2*j$) and replacing structures `#init` by `_init`, which indicates that a new auxiliary variable must be generated and its content will be used as initial value for the original one. For each analyzed PIPS comment/invariant in $InvSet$, a new code line is generated in P'. The function `__ESBMC_assume`'s parameter is a conjunction of all invariants generated by PIPS.

```
   Input: Program P, Set of invariants InvSet
   Output: Program P'
1  if Inv is empty then
2  |  return P
3  end
   // dictionary to identify #init
4  dict_varinitloc[] ← { }
   // copy comments of each invariant into array
      pips_comments
5  pips_comments[] ← extract_comments(Inv)
   // list for the new code generated in the
      translation
6  P' ← { }
   // Part 1 - identifying #init in the invariants
7  foreach Inv in InvSet do
8  |  if pips_comments[Inv] is not empty then
9  |  |  if pips_comments[Inv] has the pattern
   |  |  ([a-zA-Z0-9_]+)#init then
10 |  |  |  dict_varinitloc[Inv.line] ← the variable
   |  |  |  suffixed #init
11 |  |  end
12 |  end
13 end
   // list of translated invariants
14 listinvpips ← { }
   // Part 2 - code generation to support #init
      structure and corrections regarding
      invariant format
15 foreach Inv in InvSet do
16 |  P' ← Inv.line
17 |  if P' at Inv.line is a function then
18 |  |  if there exists some line in this function ∈
   |  |  dict_varinitloc then
19 |  |  |  foreach variable ∈ dict_varinitloc do
20 |  |  |  |  P' ← Declare a variable as
   |  |  |  |  variable.type
   |  |  |  |  var_init=variable.name;
21 |  |  |  end
22 |  |  end
23 |  end
   |  // Part 3 - translation of the related
   |     mathematical expressions into ANSI-C
   |     code
24 |  if pips_comments[Inv] is not empty then
25 |  |  foreach expression ∈ pip_comments[Inv] do
26 |  |  |  listinvpips ← Reformulate the expression
   |  |  |  according to the C programs syntax and replace
   |  |  |  #init by _init
27 |  |  end
28 |  |  P' ← __ESBMC_assume(conjunction of all
   |  |  invariants in listinvpips);
29 |  end
30 end
31 return P'
```

Algorithm 3: The combination algorithm for PIPS.

3.4 Invariant Generation using PAGAI

PAGAI [45] is a static analyzer that uses structures of the low-level virtual machine (LLVM) compiler [56] and computes inductive invariants on numerical variables of an input program. Indeed, it uses a source-code analysis algorithm based on abstract interpretation to infer invariants for each control point in a C/C++ program. PAGAI performs a linear-relation analysis, which obtains invariants as convex polyhedra; however, it also supports other abstract domains, *e.g.*, octagons and products of intervals, which is not true for PIPS. Indeed, this last difference provides some variability for the adopted tools, which can be explored for tuned behaviour in specific scenarios.

In the experimental evaluation presented by Henry *et al.* [45], PAGAI was applied to real-world examples (industrial code and GNU programs). According to those authors, front-ends for many analysis tools place restrictions (*e.g.*, no backward goto instructions and no pointer arithmetic), which may compromise safety-critical embedded programs. At the same time, PAGAI does not suffer from such issues. Nonetheless, it may apply coarse abstractions to some C/C++ programs, which can lead to weak invariants, whose conjunctions with safety properties are not inductive, as later confirmed in the experimental results of that work.

PAGAI takes a program as input, generates invariants, and outputs a new program, with invariants as comments before each statement. Similarly to our approach based on PIPS, those invariants are added to a program as assume statements. Let P denote the original program, $InvSet$ be a set that has each invariant extracted from the PAGAI annotations and its code location (*i.e.*, the line number of the code), in the analyzed program P, and P' be the combination between the original program P and invariants from $InvSet$. Aiming to include the program invariants inferred in $InvSet$, our PAGAI-based translation approach uses assume statements (in our case, `__ESBMC_assume`) to integrate them into P'. In contrast to PIPS, such invariants require minor changes, given that they are already ANSI-C compliant.

3.5 Illustrative Example

As an illustrative example, we describe a C program[7] extracted from SV-COMP 2019 [8] (shown in Figure 3). We chose this particular example because it demonstrates the importance of invariants, when verifying programs: the k-induction algorithm without invariants (using ESBMC) was unable to prove its correctness, during the same competition.

Two properties are being checked here (lines 7 and 8), eight times for each outer-loop iteration. One may notice that the nested loop does not change the properties being verified,

[7] *loop-invgen/id_build_true-unreach-call_true-termination.i*

```
1  int __VERIFIER_nondet_int();
2  int main() {
3    int offset, length, nlen;
4    int i, j;
5    for (i=0; i<nlen; i++) {
6      for (j=0; j<8; j++) {
7        assert(0 <= nlen-1-i);
8        assert(nlen-1-i < nlen);
9      }
10   }
11   return 0;
12 }
```

Figure 3: Verification example with two properties.

but rather repeats checks eight times. Indeed, this is reduced by ESBMC, which checks a property only once per (outer) loop iteration, as follows:

1. `assert(nlen-1-i < nlen)`:
   can be rewritten as `assert(i>=0)`;
2. `assert(0 <= nlen-1-i)`:
   can be rewritten as `assert(nlen >= i+1)`.

This program is safe, so the traditional base case will never find a property violation. In order to prove correctness using the forward condition, a BMC tool will try to unwind the outer loop $2^{31} - 1$ times, while unwinding the inner one 8 times, on each iteration: only when it can unwind to that depth, it will reach all possible states, which is infeasibly expensive in both time and memory.

```
1  int __VERIFIER_nondet_int();
2  int main() {
3    int offset, length, nlen;
4    int i, j;
5    i = 0;     // loop initial condition
6    i = __VERIFIER_nondet_int();          //
         assume nondet
7    __VERIFIER_assume(i<nlen);            //
         assume loop condition
8    assert(0 <= nlen-1-i);                //
         loop body begin
9    assert(nlen-1-i < nlen);  // ....
10   i++;           // loop body end
11   __VERIFIER_assume(!(i<nlen));         //
         assume negated loop cond
12   return 0;
13 }
```

Figure 4: Verification example rewritten during the inductive step.

A plain inductive step is also unable to prove correctness, which is done by rewriting the mentioned program as illustrated in Figure 4. The transformations are: (1) all variables written inside a loop are treated as nondeterministic, as one can see in line 6, (2) it is assumed that the loop is indeed executed, as performed in line 7, and (3) after the loop's body, it

terminates, which happens in line 11. One may notice that the inner loop is not present in this verification. Indeed, although `j` is written inside that loop body (when it is incremented), it is not part of the property's verification.

Indeed, the plain inductive step will easily find a counterexample for this program, as both `i` and `nlen` are unconstrained. ESBMC finds a property violation for `nlen = 4` and `i = -536870913`, and although `nlen = 4` is a value reachable by `nlen` during program execution, `i` will never reach `-536870913`. It is worth noticing that the related counterexample is spurious, due to the overapproximation previously mentioned.

```
1  int __VERIFIER_nondet_int();
2  int main()
3  {
4    int offset, length, nlen =
         __VERIFIER_nondet_int();
5    int i, j;
6    for(i = 0; i <= nlen-1; i += 1) {
7      __ESBMC_assume( 0<=i && i+1<=nlen );
8      for(j = 0; j <= 7; j += 1) {
9        __ESBMC_assume( 0<=i && i+1<=nlen && 0<=j
             && j<=7 );
10       assert(0<=nlen-i-1);
11       __ESBMC_assume( 0<=i && i+1<=nlen && 0<=j
             && j<=7 );
12       assert(nlen-i-1<nlen);
13     }
14   }
15   __ESBMC_assume( 0<=i && nlen<=i );
16   return 0;
17 }
```

(a) Verification example with invariants generated by PIPS.

```
1  int __VERIFIER_nondet_int();
2  int main()
3  {
4    int offset, length, nlen =
         __VERIFIER_nondet_int();
5    int i, j;
6    for (i=0; i<nlen; i++) {
7      __ESBMC_assume( 2147483647-i >= 0 );
8      __ESBMC_assume( i >= 0 );
9      __ESBMC_assume( -1+nlen-i >= 0 );
10     for (j=0; j<8; j++) {
11       assert(0 <= nlen-1-i);
12       assert(nlen-1-i < nlen);
13     }
14   }
15   return 0;
16 }
```

(b) Verification example with invariants generated by PAGAI.

Figure 5: Verification example with invariants.

The programs in Figure 5 show invariants inserted by both PIPS and PAGAI, using the intrinsic function `__ESBMC_assume`. Both tools generate inductive invariants that are able to prove correctness of both properties:

1. PIPS generates `0<=i && i+1<=nlen`
2. PAGAI generates `i >= 0 && -1+nlen-i >= 0`,

Which are precisely the properties under verification. By assuming those two invariants, BMC tools will remove every state that violates those properties, leaving only reachable states. Our extended *k*-induction algorithm, combined with invariants, is a simple but substantial modification to the original *k*-induction and allows us to increase the number of programs that can be proved correct.

4 Experimental Evaluation

In this section, we present an experimental evaluation to establish a baseline for empirical comparisons involving DepthK, CPAchecker-*k*induction, and PDR-based tools, in the context of software verification, and to determine whether we can combine the strengths of *k*-induction with those of loop invariant generation. Our method was implemented in DepthK; we applied it to verify C benchmarks from embedded system applications and also the SV-COMP editions 2018[8] and 2019[9]. Additionally, we have also compared our approach against ESBMC [33], CBMC [53], and 2LS [19][10,11]. DepthK was also evaluated against the available PDR-based verifiers CPAchecker-CTIGAR [13], SeaHorn [42], and Vvt [41][12], which can be applied to actual C programs. SeaHorn is the most prominent software verifier that implements IC3; however, given its last participation in SV-COMP, in 2016, it did not perform well against other existing software verifiers (including DepthK), given a large number of incorrect results as reported by Beyer *et al.* [7]. Therefore, regarding IC3 tools comparison and given their current limited availability for C programs [9], we have considered only the verification tasks where the property to verify is the unreachability of a program location (*i.e.*, ReachSafety and SoftrwareSystems), as presented in Table 3.

4.1 Experimental Setup

In order to evaluate the effectiveness of our proposed method, we use C programs with loops from public repositories,

[8] https://sv-comp.sosy-lab.org/2018/results/results-verified/

[9] https://sv-comp.sosy-lab.org/2019/results/results-verified/

[10] https://sv-comp.sosy-lab.org/2018/results/results-verified/META_ReachSafety_depthk.table.html

[11] https://sv-comp.sosy-lab.org/2018/results/results-verified/META_SoftwareSystems_depthk.table.html

[12] https://www.sosy-lab.org/research/pdr-compare/supplements/results/table.html

which constitute the main reference in the software verification area, such as: the set of C benchmarks from SV-COMP [8] and embedded systems applications [59, 67, 70]. For each benchmark, we check a single property encoded as an assertion or as an error location, i.e., we check whether an assertion is not violated or whether an error label is unreachable in any finite execution of the program.

The SV-COMP's benchmarks used in this experimental evaluation include:

- **ReachSafety**, which contains benchmarks for checking the reachability of an error location;
- **MemSafety**, which presents benchmarks for checking memory safety;
- **ConcurrencySafety**, which provides benchmarks for checking concurrency problems;
- **Overflows**, which is composed of benchmarks for checking if variables of signed-integers type overflow;
- **Termination**, which contains benchmarks for which termination should be decided;
- **SoftwareSystems**, which provides benchmarks from real software systems.

Regarding the PDR-based experimental results presented in Table 3, we consider only verification tasks that explore the strength of the PDR approach, where the property to verify is the unreachability of a program location. From the benchmarks above, we excluded properties for *overflows*, *memory safety*, and *termination*, which are not in the scope of this evaluation, and the categories *ReachSafety-Recursive* and *ConcurrencySafety*, each of which is not supported by at least one of the evaluated implementations. The remaining set of categories consists of 5591 verification tasks.

The embedded-system applications used in this experimental evaluation are classified in 3 categories: Powerstone [67], which is used for automotive-control and fax applications; Real-Time SNU [70], which contains a set of programs for matrix handling and signal processing, such as matrix multiplication and decomposition, second-degree equations, cyclic-redundancy check, Fourier transform, and JPEG encoding; and WCET [59], which is a set of programs for executing worst-case time analysis.

Here, we analyze the number of true and false positives and also the number of true and false negatives. Additionally, we generate a score for each analyzed tool based on the total number of correct and incorrect results. In particular, this experimental evaluation is performed with the following tools:

- DepthK v3.1 with k-induction and invariants, using polyhedra (through PIPS and PAGAI), where ESBMC's parameters are defined in a wrapper script available in the DepthK's repository;
- ESBMC v6.0 along with plain k-induction, which is run with an interval-invariant generator that pre-processes input programs, infers invariants based on intervals, and includes them into programs.
- CBMC v5.11 with a bounded model checker, which, in the absence of additional loop transformations or k-induction, runs the script provided by Beyer, Dangl, and Wendler [10];
- CPAchecker[13] (revision 15596, acquired directly from its SVN repository) [10], which was executed with options k-induction together with invariants, and for k-induction without invariant;
- CPAchecker-CTIGAR (revision 27742) [13], which is an adaptation of PDR to software verification. Our evaluation compares CPAchecker with the implementations of CTIGAR;
- 2LS v0.7.2 along with k-induction and invariants, which is named *kIkI* and is executed with a wrapper script from SV-COMP 2019 [8];
- SeaHorn(F16-0.1.0-rc3) [42], which is a modular verification framework that uses constrained Horn clauses as the intermediate verification language. SeaHorn's verification condition generator is based on IC3;
- VVT [41], which is an implementation of the CTIGAR approach[13] that uses an SMT-based IC3 algorithm [15] incorporating Counterexample Guided Abstraction Refinement (CEGAR) [22]. Our evaluation compares the combination of Vvt-CTIGAR and bounded model checking, which is named as Vvt-Portfolio.

The present experimental evaluation was conducted on a computer with Intel Core $i7-4790$ CPU @ 3.60GHz and 32 GB of RAM, running Linux Ubuntu 18.04 LTS x64. Each verification task is limited to a CPU time of 15 minutes and a memory consumption of 15 GB.

4.2 Experimental Results

After running all tools, we obtained the results shown in Tables 1 , 2 and 3. Table 1 shows the results for the embedded system benchmarks, where **Tool** is the name of the Tool used in the experiments, **Correct Results** is the number of correctly proven programs, **Incorrect Results** is the number of programs where the respective Tool found at least one error, although being completely correct, or it does not identify an error, but the program contains a property violation, **Unknown and TO** is the number of programs that the Tool is unable to verify, due to lack of resources, tool failure (crash), or verification timeout (15 min), and **Time** is the runtime, in minutes, to verify the entire benchmark set.

Table 2 shows the results for all evaluated tools, regarding the SV-COMP 2019's benchmark suite, where **Category** is the SV-COMP's category, **Tool** is the name of the tool used in the experiments, **Correct True** is the number of programs where the respective tool did not find a bug, and that is correct, **Correct False** is the number of programs where the respective tool correctly found a bug, **True Incorrect** is

[13] `https://svn.sosy-lab.org/software/cpachecker/trunk`

the number of programs where the respective tool does not identify an error, which is correct, and **False Incorrect** is the number of programs where the respective tool found at least one error, although being completely correct.

Table 3 shows the results for all the 5591 verification tasks and has the same categorisation as Table 2. Nonetheless, it compares the effectiveness and efficiency of our implementation to the only available verifiers that implement a pure PDR approach for software-model checking.

Regarding Table 1, we have the following observations:

- The winning tool using k-induction is 2LS, which was able to give a correct result in all 34 verification tasks from the embedded-system benchmarks (Powerstone, SNU, and WCET): the difference with our approach is the way invariants are produced, i.e., we generate them before verification starts, while 2LS does that on each step, in order to continuously strengthen them.
- The results presented by DepthK using PIPS were lower than that of ESBMC and CPAchecker, but it outperformed CBMC with k-induction: although there exist no failures in our verification process, many inconclusive results were presented (i.e., unknown and TO), the reason being that both (PIPS and PAGAI) are not adequately prepared to handle some C features (e.g., bit-shift operations) used in embedded-system applications.
- The results presented by our approach are directly related to the maturity of each invariant generation tool: it is possible to improve the results presented by PIPS, since it has a wide variety of configurations that can be exploited by users, and PAGAI cannot be configured by users, which is the main difficulty for generating inductive invariants for embedded-system applications.
- Invariant generation configuration may be an improvement task: as already mentioned, PIPS presents a wide variety of configurations, which can be adaptively done for a given scenario or benchmark type.
- ESBMC v6.0 has received significant improvements in its k-induction algorithm, with bug and memory-leak fixes and guard simplification: using only its k-induction technique was enough to outperform our proposed method and its execution time was also shorter, due to implementation of those same improvements and because it does not require additional time for generating invariants;
- CPAchecker using only k-induction was slightly worse than ESBMC with k-induction. There exist two reasons to explain this result. The first one is that CPAchecker's invariant-generation algorithm works in the background, while it verifies a program. Consequently, the cumulative runtime of its two versions might exceed that of ESBMC, because the latter performs everything sequentially, in one single call, i.e., in order to generate a control flow graph, annotate a program with invariants, and finally verify it. The second one is that CPAchecker continuously refines invariants, which might run for longer times and then get strengthened, thus proving program correctness.
- CBMC with k-induction that verifies the absence of violated assertions under a given loop unwinding bound; however, it seems to be still experimental. In particular, CBMC relies on a limited loop unwinding technique and concurrent programs, which is unable to unroll nested loops [25].

In Tables 2, we have the following observations about those experimental results, per category.

- **ReachSafety**:
  - Our proposed method based on PAGAI presented the lowest number of correct answers, if compared with other existing approaches, because the generated invariants increased verification times and memory consumption rapidly, so DepthK did not reach a result promptly or even exceeded the amount of allowed memory;
  - ESBMC is the tool that correctly verified the most significant number of benchmarks, which demonstrated the effectiveness of the new interval-invariant used during verification processes. This improvement influenced results for some categories, such as *ReachSafety* and *SoftwareSystems*; however, ESBMC failed to verify other benchmarks, due to an internal bug in ESBMC, which made it unable to track variables going out of scope [33];
  - 2LS is the tool that correctly verified the second most significant number of benchmarks, which demonstrates its ability to analyze programs requiring combined reasoning about the shape and content of dynamic data structures and instrumentation for memory safety properties. Nonetheless, the reasoning about array contents is still missing, and the 2LS' algorithm *k*I*k*I does not support recursion yet.
- **MemSafety**:
  - ESBMC and CBMC were denoted as the first- and the second-best tools, respectively, due to recent improvements explicitly implemented for this category [53]. However, the number of incorrect results, although relatively low, is the main problem.
  - Our proposed method and 2LS were the tools that solved the lowest numbers of benchmarks. However, 2LS presented a large number of incorrect results, due to the lack of a bit-precise verification engine driven by weak invariants that did not take into account the nature of this category and its respective safety properties.
- **ConcurrencySafety**:
  - The proposed method using PAGAI is the tool that correctly verified the most significant number of benchmarks and was indeed able to increase that figure, due to its invariant inference. Also, our method was able to minimize ESBMC weakness by reducing the num-

Tool	DepthK (PIPS)	DepthK (PAGAI)	ESBMC (k-ind)	CPAchecker (k-ind)	CPAchecker (cont. ref. k-ind.)	CBMC (k-ind)	2LS
Correct Results	16	14	29	27	27	15	34
Incorrect Results	0	0	0	0	0	0	0
Unknown and TO	18	20	5	7	7	19	0
Time(min)	55.51	56.13	54.18	1.8	1.95	286.06	10.6

Table 1: Experimental results for the Powerstone, SNU, and WCET benchmarks.

Category	**Tool**	**Correct True**	**Correct False**	**Total Correct**	**Incorrect True**	**Incorrect False**	**Total Incorrect**
ReachSafety	DepthK (PAGAI)	62	700	762	0	4	4
	ESBMC (k-ind)	1332	893	**2225**	7	3	10
	2LS	1062	453	1515	1	2	3
	CBMC	641	669	1310	0	0	**0**
MemSafety	DepthK (PAGAI)	80	45	125	1	16	17
	ESBMC (k-ind)	130	64	**194**	8	2	10
	2LS	65	66	131	3	68	71
	CBMC	119	71	190	2	6	**8**
ConcurrencySafety	DepthK (PAGAI)	194	608	**802**	16	4	20
	ESBMC (k-ind)	182	600	782	14	7	21
	2LS	-	-	-	-	-	-
	CBMC	165	331	496	0	3	**3**
Overflows	DepthK (PAGAI)	0	167	167	0	0	**0**
	ESBMC (k-ind)	85	149	**234**	0	0	**0**
	2LS	87	140	227	0	0	**0**
	CBMC	31	169	200	0	0	**0**
Termination	DepthK (PAGAI)	266	0	266	14	0	14
	ESBMC (k-ind)	717	0	717	0	0	**0**
	2LS	676	306	**982**	0	3	3
	CBMC	718	0	718	0	0	**0**
SoftwareSystems	DepthK (PAGAI)	0	101	101	0	6	6
	ESBMC (k-ind)	1165	28	**1193**	12	2	14
	2LS	171	0	171	0	0	**0**
	CBMC	28	8	36	1	2	3
Total	DepthK (PAGAI)	602	1621	2223	31	30	61
	ESBMC (k-ind)	3611	1734	**5345**	41	14	55
	2LS	2061	965	3026	4	73	77
	CBMC	1702	1248	2950	3	11	**14**

Table 2: Experimental results for the SV-COMP'19.

Category	**Tool**	**Correct True**	**Correct False**	**Total Correct**	**Incorrect True**	**Incorrect False**	**Total Incorrect**
ReachSafety	DepthK (PAGAI)	395	644	1039	0	7	7
	CPAchecker-CTIGAR	397	214	611	0	1	**1**
	SeaHorn	1010	595	**1605**	6	105	111
	Vvt-Portfolio	528	311	839	9	22	31
SoftwareSystems	DepthK (PAGAI)	393	58	451	0	3	3
	CPAchecker-CTIGAR	435	41	477	0	0	**0**
	SeaHorn	1714	149	**1863**	40	12	52
	Vvt-Portfolio	0	0	0	0	0	0
Total	DepthK (PAGAI)	788	702	1490	0	10	10
	CPAchecker-CTIGAR	832	255	1087	0	1	**1**
	SeaHorn	2724	744	**3468**	46	117	163
	Vvt-Portfolio	528	311	839	9	22	31

Table 3: PDR-based experimental results SV-COMP'18.

ber of incorrect results. One of the main contributions of the present work is the generation of sufficiently inductive invariants that can guide ESBMC to correct results, given that invariants inferred by PAGAI were essential to eliminate states that would typically induce ESBMC to fail.

  - ESBMC has full concurrency support, and its standard context-BMC algorithm can solve a wide range of verification tasks, without the aid of k-induction and external invariants. Nonetheless, this tool produced the most significant number of incorrect results, due to the lack of support for some POSIX Pthreads functions, which are still un-modelled;
  - CBMC was the tool that solved the lowest numbers of benchmarks, which was caused by current limitations in the treatment of pointers, and despite this tool's latest improvements [2];
  - 2LS does not have native support for verifying concurrent programs based on POSIX/Pthreads [8].
- **Overflows**:
  - ESBMC and 2LS were denoted as the first- and the second-best tools, respectively, due to recent improvements regarding inductive invariants and considering programs that require joint reasoning about shape and content of dynamic data structures;
  - CBMC and our proposed method were the tools that solved the lowest numbers of benchmarks due to the lack of a bit-precise verification engine that efficiently handles arithmetic overflow checks.
- **Termination**:
  - 2LS and CBMC were the tools that successfully solved the highest number of benchmarks; however, if a larger number of unwindings is needed, the approach becomes quite inefficient. The strengths of BMC, on the other hand, are its predictable performance and amenability to the full spectrum of categories;
  - ESBMC using k-induction generated better results than our method that infers invariants, which happened because many inconclusive results were presented and that led to a meagre amount of verification tasks successfully analyzed;
- **SoftwareSystems**:
  - ESBMC using k-induction generated better results, when compared with other tools that infer invariants, which happened because of the relational analysis that can keep track of relations between variables;
  - 2LS, which also uses inductive invariants, was slightly better than our method; however, it presented the same issues relating to limitation of states to be verified. That happened because some of the benchmarks, *e.g.* those requiring reasoning about arrays contents, demand invariants stronger than what is inferred by 2LS;
  - DepthK overcame CBMC because it supports structures with pointers and considers variables of this type in the static analysis and also during invariant generation, as with 2LS.

Table 3 presents results for the chosen PDR-based tools, which were evaluated using the SV-COMP 2018's benchmark suite. There exist 5591 verification tasks, with 1457 of them containing bugs, while the remaining 4134 are considered to be safe, when checked with the best configurations of DepthK, CPAchecker, SeaHorn, and Vvt-portfolio. Indeed, such results give an overview of the best configurations used by the three different chosen software verifiers that use PDR. One may notice that SeaHorn achieved the highest numbers of total correct results, but it also presented a significant amount of total incorrect ones. Because SeaHorn is unsound for satisfiability, it can report that some expression-tree satisfies behavioural specification ψ, when in fact no such expression-tree exists. Also, SeaHorn verifier is unreliable for most bit-vector operations, *e.g.*, bit-shifting [47]. Despite considering only verification tasks that explore the strengths of the PDR approach, DepthK with PAGAI came second and overcame the other two PDR-based tools, as a result of our well-engineered implementation that provides invariants based on the idea of k-induction. Additionally and despite the PDR tools' compatibility restrictions, DepthK has widely used SV-benchmarks collection of verification tasks. In conclusion, our proposed method using PAGAI can be regarded as a competitive verification tool, when compared with the chosen PDR-based ones.

Figures 6 and 7 show the scoring system adopted in the SV-COMP's benchmarks, including comparative results for the SV-COMP's loops subcategory and embedded-system benchmarks, respectively. Here, we used only the SV-COMP's loops subcategory in this analysis, since loops are a widely recognized challenge regarding the inference of program invariants. One can notice that for embedded-system benchmarks, safe programs [33] were used, since the main goal here is to check whether strong (inductive) invariants are inferred, i.e., conditions that hold throughout an entire program, in order to prove correctness.

DepthK with PAGAI achieved a lower score than PIPS, in the embedded system benchmarks, due to the fact that PAGAI was unable to produce inductive invariants, with the potential to support ESBMC in reaching a verification result *true* or *false*. Indeed, PAGAI is a relatively new tool that is still under development, mainly regarding invariant prediction, and unlike PIPS that has many configuration options, PAGAI does not allow a combination of static analysis methods to infer invariants, which is the main reason for its low score.

On the one hand, in the loops benchmarks, DepthK (PIPS) achieved the second-highest score, among all tools using invariants, being overcome only by 2LS. On the other hand, DepthK (PAGAI) presented the lowest score in the embedded-system benchmarks, mainly due to 58.82% of results identified as `Unknown`, i.e., when it is not possible to determine an outcome or due to tool failure. There exist also failures related to invariants and code generation, which are given as input to the BMC procedure. As already mentioned, PAGAI is still under development (in a somewhat preliminary state), but one can argue that its results are still promising.

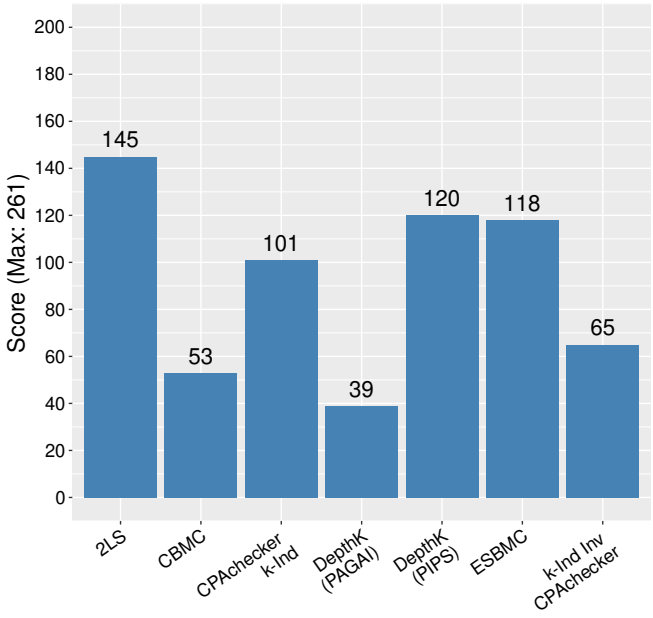

Figure 6: Score regarding loops subcategory.

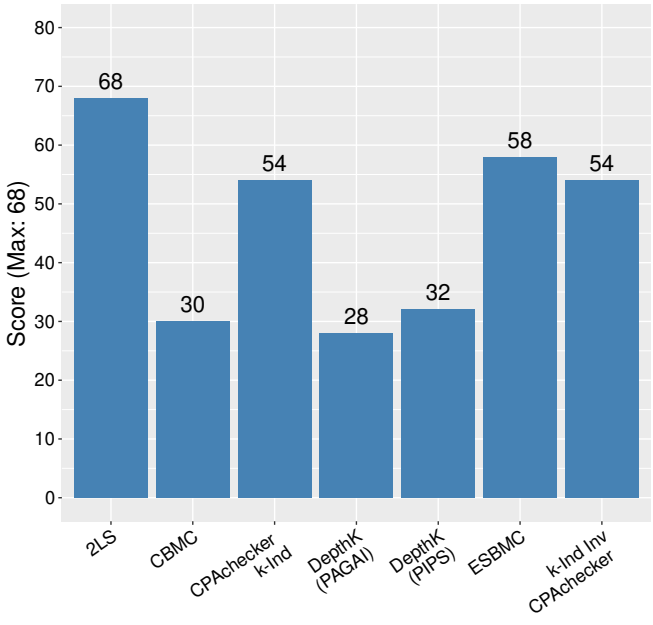

Figure 7: Score regarding embedded systems.

The CPAchecker k-induction is better than DepthK (PAGAI), since it is a more sophisticated tool while getting very close to ESBMC, in the embedded-system benchmarks. The main problem with both approaches is the number of errors that directly impacts their final scores. Nonetheless, CPAchecker k-induction was not better than 2LS, which can be explained by the fact that 2LS implements invariant generation techniques, incremental BMC, and k-induction. The 2LS's result was also the best regarding the embedded systems category, because it generated stronger invariants when compared with DepthK (PIPS)/PAGAI; however, the number of correct results was close to that obtained by ESBMC.

CBMC with k-induction was the tool that generated the highest number of inconclusive results, in the loops subcategory, and the verification time is noticeably superior to all tools used in the presented experimental evaluation; however, one can point out that the number of errors is not so different if compared with other approaches, and CBMC shows that it is as stable as the other tools.

In order to measure the impact of the invariants application to k-induction verification schemes, the distribution regarding DepthK + PIPS/PAGAI and ESBMC results were classified, per verification step: base case, forward condition, and inductive step. In this analysis, only the results related to DepthK and ESBMC were evaluated, given that they are part of the proposed approach and it is not possible to identify the steps of the k-induction algorithm (in standard logs), in other tools. Figure 8 shows the distribution of the results, for each verification step, regarding the SV-COMP's loops subcategory, while Figure 9 presents results for the embedded-system benchmarks.

The distribution of results in Figure 8, during the execution of the k-induction algorithm in the loops subcategory, shows that invariants generated by PIPS helped the k-induction algorithm in DepthK to increase the number of correct results. Here, the default ESBMC presents weaknesses for programs with loops, since it is unable to produce inductive loop invariants, in order to prove correctness.

By analyzing the presented results, we noticed that invariants allowed the k-induction algorithm to prove that loops were sufficiently unwound and whenever a property is valid for k unwindings, it is also valid after the next unwinding of a system. We also identified that the DepthK (PIPS) and DepthK (PAGAI) did not find a solution (leading to `Unknown` and `Timeout`) in 33.09% and 49.29% of the loops subcategory (see Figure 8), respectively. In the embedded system benchmarks, DepthK (PIPS) did not find a solution in 52.94% and DepthK (PAGAI) in 58.82% (see Figure 9). This is explained by the invariants generated from PIPS and PAGAI, which could not produce inductive invariants for the k-induction algorithm, either due to a transformer or invariants that are not convex.

We believe that PIPS' results can be significantly improved by fixing errors related to the tool's implementation since some results generated as `Unknown` are related to failures in our tool execution, which happened due to the algorithm that identifies functions and variables in the analyzed source code. Additionally, we have identified that adjustments are needed in the PIPS's script parameters, in order to generate invariants, since PIPS has a broad set of commands for code transformation and that might lead to a positive impact, regarding invariant generation for specific classes of programs.

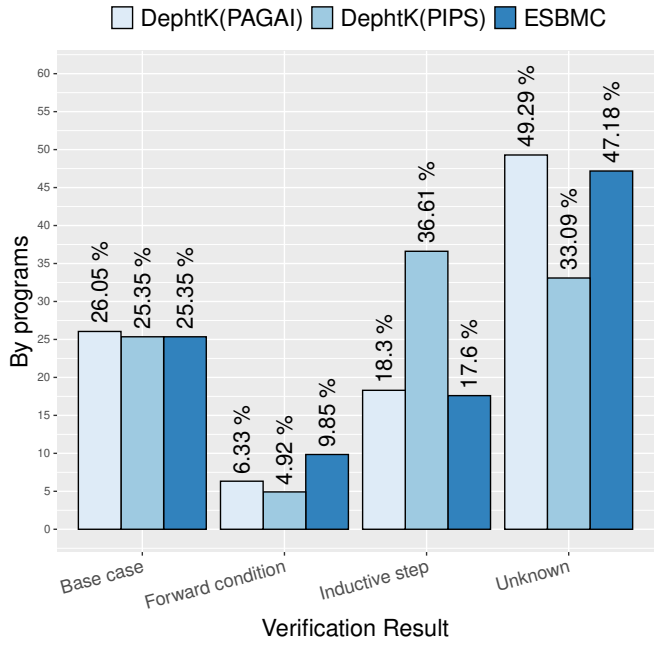

Figure 8: Results for the loops subcategory.

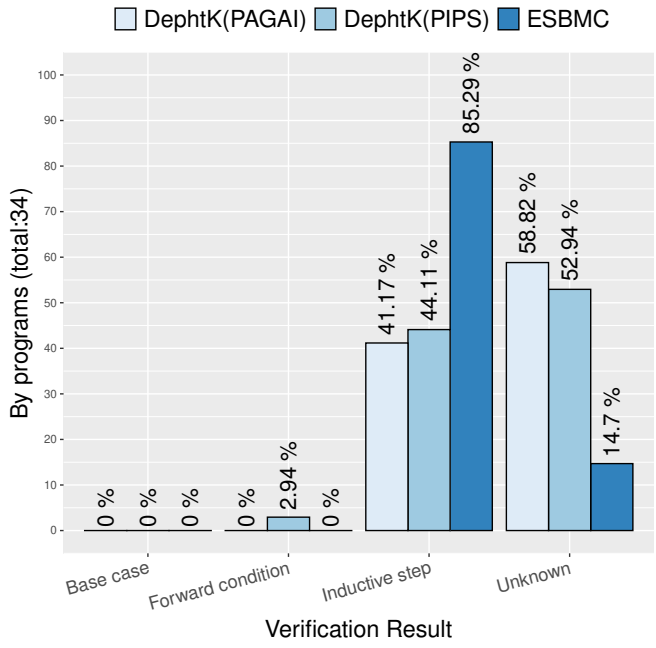

Figure 9: Results for the embedded programs.

Due to the full range of benchmarks used in this experimental evaluation and the fact that PIPS has numerous configuration options, one possible improvement in our approach is to use PIPS, in order to allow us to discard unreachable states to reduce code, in addition to identifying loops to generate invariants to limit their unfolding. Thus, by adopting a new combination of PIPS's option set, results could be improved, and the k-induction algorithm would then speed up with stronger (inductive) invariants, for each benchmark.

5 Related Work

The k-induction algorithm has already been implemented and further extended by the software verification community, in many studies, which led to comparisons between our approach and other similar k-induction algorithms. Recently, Bradley et al. introduced the "property directed reachability" (or IC3) procedure for safety verification of systems [16, 43] and showed that IC3 could scale on certain benchmarks, where k-induction fails to succeed. We do not compare k-induction with IC3, as Bradley [16] already performed this, and focus on related k-induction procedures.

Donaldson et al. described a verification tool called Scratch [26], which can detect data races during direct memory access (DMA) in CELL BE processors from IBM [26], using k-induction. Properties (in the form of assertions) are automatically inserted to model behaviour of memory control-flow. The mentioned algorithm tries to find violations on those properties or prove that they hold indefinitely, by using a base case step and an inductive one, respectively, without checking the completeness threshold. That method also requires source code to be manually annotated with loop invariants, whereas our approach automatically generates and adds them to a given program. Finally, it can prove the absence of data races in several benchmarks, but it is restricted to verify a specific class of problems for a particular type of hardware. At the same time, our approach is evaluated over a more general group of programs, through (traditional) benchmarks from SV-COMP.

In another related work, Donaldson et al. described two tools for proving program correctness: K-Boogie and K-Inductor [25]. The former is an extension of the Boogie language, which aims to prove correctness (using k-induction) of programs written in some languages (Boogie, Spec, Dafny, Chalice, VCC, and Havoc), while the latter is a bounded model checker for C programs, which is built on top of CBMC [23]. Both use the k-induction algorithm, which consists of a base case and an inductive step and, like previous work, the completeness threshold is not separately checked and relies only on the inductive step, in order to prove correctness. Their proposed k-induction has a preprocessing step; however, differently from ours, in which we introduce invariants, their preprocessing removes all nested loops and leaves only non-nested ones. Those authors compared the results of K-Inductor with Scratch. They showed that their new approach maintained the same coverage (in terms of correctly verified programs) while being faster. However, similarly to previous work, programs needed to be manually changed, in order to insert loop invariants, while our approach does it automatically. Madhukar et al. [57] described a method to accelerate the generation of program invariants,

without k-induction, by adopting analysis of source code regarding loops. The basic idea was to identify invariants and their deviations in order to accelerate verification processes. Their article compared some model checkers (e.g., UFO [1], CPAchecker, CBMC, and IMPARA [14]), without using k-induction and regarding invariant generation, in order to speed up verification of programs with loops. In summary, the technique proposed by Madhukar et al. [57] is based on under-approximating loops for fast counterexample detection. It works as a pre-processor for replacing loops and improving verification processes, which reduces verification times and false results, in order to improve its confidence. In contrast to Madhukar et al., the proposed method does not modify existing loops, but instead includes assumptions based on invariants, in order to guide the k-induction algorithm.

Beyer et al. [10] introduced a different approach regarding invariant generation. In particular, they proposed a k-induction algorithm, which can generate invariants separately from the verification algorithm itself, named as Continuously Refined Invariants. The latter is an assistant for CPAchecker with k-induction and runs in parallel with verification tasks. It starts with weak invariants (i.e., without an invariant generator as PAGAI or PIPS), by using an abstract domain based on expressions over intervals. This method continuously adjusts and refines invariant precision, during verification processes, and creates inductive invariants [10]. This way, verification tasks can take advantage of previously generated values so that the k-induction verification algorithm can strengthen the induction hypotheses. In another work Beyer et al. [9] implemented a standalone PDR algorithm. In particular, they designed an invariant generator based on the ideas of PDR, and also evaluated the PDR invariant-generation module on an extensive set of C verification tasks. This method further extends the knowledge about PDR for software verification and outperforms an existing implementation of PDR. In summary, the PDR-based approach proposed here represents an effective and efficient technique for computing invariants, which are difficult to obtain with automated verification tools. However, the approach proposed by Beyer et al. solves less verification tasks and takes longer than other verifiers, including DepthK. Additionally, our approach is evaluated over a more general group of programs, through the same set of benchmarks extracted from SV-COMP.

Brain et al. [19] proposed an incremental verification method called kIkI, which combines state-of-the-art verification approaches from literature (e.g., plain BMC, k-induction, and abstract interpretation), instead of using only k-induction algorithms. In particular, kIkI firstly applies the BMC technique to refute properties and then finds a counterexample: if it is not possible, a new verification procedure using k-induction (composed of the base case, forward condition, and inductive step) and invariants are applied, in order to prove that a program is safe. If the k-induction algorithm does not prove properties or does not generate a counterexample, an abstract interpretation technique, based on polyhedra, is applied, to generate invariants for the next state-space unrolling. In contrast to Brain et al. [19], the present work is based only on the application of k-induction, considering invariants in the polyhedral domain.

In another related work, Garg et al. [36] introduced the ICE-learning framework for synthesizing numerical invariants. According to Garg et al. [36], there exist many advantages in the (machine) learning approach. For instance, it typically concentrates on finding the simplest concept, which satisfies the constraints implicitly by providing a tactic for generalization. At the same time, white-box techniques (e.g., interpolation [54]) need to build in tactics to generalize. ICE-learning framework uses examples (test runs of the program on random inputs), counterexamples, and implications to refute the learner's conjectures. The ICE-algorithm iterates over all possible template formulas, thus growing in complexity, until it finds an appropriate formula, and adopts template-based synthesis techniques, which use constraint solvers. Garg et al. [36] use octagonal domain and present an empirical evaluation on benchmarks from SV-COMP loops category and programs from literature (e.g., [40]). In contrast to Garg et al. [36], we adopted the polyhedral domain and presented an extensive evaluation over different categories from SV-COMP benchmarks. For future work, we also plan to use a machine learning approach to infer invariants.

Ezudheen et al. [31] extended the ICE learning model for synthesizing invariants using Horn implication counterexamples [37]. According to Ezudheen et al. [31], their main contribution is to devise a decision tree-based Horn-ICE algorithm. The goal of the learning algorithm is to synthesize predicates, which are arbitrary Boolean combinations of the Boolean predicates and atomic predicates of the form $n \leq c$, where n denotes a numerical function, and where c is arbitrary. The implementation of the proposed method uses a predicate template of the form $x \pm y \leq c$, called octagonal constraints, where x, y are numeric program variables or nonlinear expressions over numeric program variables and c is a constant determined by the decision tree learner. Ezudheen et al. [31] present an evaluation using 109 programs, which 52 programs are from SV-COMP'18 recursive category. In comparison to Ezudheen et al. [31], we have used the polyhedral domain. However, our approach uses PAGAI to infer program invariants, where the abstract domains are provided by the APRON library [51], which include convex polyhedral, octagon, and products of intervals. We also, extend the experimental evaluation by adopting 6 categories from SV-COMP and embedded-system applications to validate the program invariant quality. Additionally, our approach does not need to produce samples or counterexamples to infer program invariants as Garg et al. [36] and Ezudheen et al. [31]. PIPS uses an interprocedural analysis, where each program instruction is associated with an affine transformer, representing its underlying transfer functions. For future work, we

intend to investigate the strategy proposed in Ezudheen et al. [31], which chooses a template from a class of templates based on extracting features from a simple static analysis of the program and using priors gained from the experience of verifying similar programs in the past.

Champion et al. [21] proposed combining refinement types with the machine-learning-based for invariant discovery in ICE framework [36] suitable for higher-order program verification. Champion et al. [21] show the implementation of the proposed approach, which consists of two parts: (i) RType is a frontend (written in OCaml [71]) generating Horn clauses from programs written in a subset of OCaml; and (ii) HoIce, written in Rust [14], is one such Horn clause solver and implements the modified ICE framework. According to Champion et al. [21], RType supports a subset of OCaml including (mutually) recursive functions and integers, without algebraic data types. Champion et al. [21] argued that only considered programs that are safe since RType is not refutation-sound. Additionally, aiming to compare their Horn clause solver HoIce to other solvers, Champion et al. [21] show a comparison on the SV-COMP with Spacer (implemented in Z3). Where HoIce timeouts on a significant part of the benchmarks. Champion et al. [21] noted that are unsatisfiable; the ICE framework is not made to be efficient at proving unsatisfiability. In contrast to Champion et al. [21], our approach to infer invariants adopting PAGAI and PIPS that not apply machine-learning techniques. Related to the solver, the PAGAI uses Yices [29] or Z3 [24] through their C API, and PIPS is based on discrete differentiation and integration that is different from the usual abstract interpretation fixed-point computation based on widening. Champion et al. [21] show, in their experimental evaluation, that 11 programs fail because inherent limitations of the proposed approach, where two of them require an invariant of the form $x + y \geq z$. We argue that our proposed approach can handle with that form invariant since we adopt a polyhedral form such as $a.x + b.y \leq c$.

6 Conclusions

We described and evaluated a verification approach based on the k-induction proof rule, in which polyhedral abstraction of program behaviour is used to infer (inductive) invariants. The proposed method, which was implemented in a tool named as DepthK, was used to verify reachability properties using benchmarks from SV-COMP and embedded-systems automatically. In particular, 10522 verification tasks from SV-COMP 2019, 5591 verification tasks from SV-COMP 2018 and 34 ANSI-C programs from real-world embedded-system applications were evaluated. Also, a comparison among DepthK (using PIPS and PAGAI, as invariant generation tools), CPAchecker, ESBMC, CBMC, and 2LS, the latter with k-induction and invariants, was performed.

[14] https://www.rust-lang.org/

The DepthK's k-induction proof rule, together with invariants generated by PIPS, was able to provide verification results as accurate as those obtained with ESBMC k-induction, without invariant inference. We argue that the proposed method, in comparison to other existing software verifiers, shows promising results, which indicates that it can be sufficient to verify real programs. In particular, in benchmarks from SV-COMP 2019, DepthK was able to solve 2223 verification tasks and overcame other verifiers (*e.g.*, 2LS, CBMC and ESBMC) that use either BMC or k-induction proof rule in *ConcurrencySafety* category. Besides, DepthK was able to solve 1490 tasks and overcame other verifiers (*e.g.*, CPAchecker-CTIGAR and Vvt) that use PDR-based techniques.

Additionally, the combination of k-induction with invariants inferred by PAGAI led to fewer accurate results than those obtained with PIPS and also standard ESBMC k-induction. The configurations used in PIPS and PAGAI for benchmarks from embedded systems and SV-COMP indicate that our approach does not cover all possible categories of benchmarks. In particular, improvements in invariant generation must be made, depending on the program that is being verified. Those improvements range from refining PIPS and PAGAI configurations to deal with distinct verification conditions to the identification of the most suitable approach to be used for a given benchmark. As a result, DepthK would be able to deal with a series of verification tasks, through specific strategies both in the invariant generation and decision regarding the use of k-induction with invariant inference. Toward that, machine learning techniques could be used, which would be able to fine-tune configurations and guess the best approach to be employed [48].

For future work, we will investigate a hybrid approach to infer program invariants, which combines both PIPS and PAGAI, i.e., a strategy that merges invariants produced by both tools. We also aim to learn from counterexamples, in order to create stronger (inductive) invariants, and as a consequence increase effectiveness from bug detection perspective using k-induction proof combined with invariants.

References

1. Albarghouthi A, Gurfinkel A, Li Y, Chaki S, Chechik M (2013) UFO: Verification with interpolants and abstract interpretation. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol 7795, pp 637–640
2. Alglave J, Kroening D, Tautschnig M (2013) Partial orders for efficient bounded model checking of concurrent software. In: Computer Aided Verification, LNCS, vol 8044, pp 141–157
3. Armando A, Mantovani J, Platania L (2009) Bounded model checking of software using SMT solvers instead

of SAT solvers. Software Tools for Technology Transfer 11(1):69–83
4. Ball T, Rajamani S (2002) SLIC: A specification language for interface checking (of C). Tech. rep., Microsoft Research
5. Barrett C, Sebastiani R, Seshia S, Tinelli C (2009) Handbook of Satisfiability, IOS Press, chap Satisfiability Modulo Theories, pp 825–885
6. Beyer D (2015) Software verification and verifiable witnesses - (report on SV-COMP 2015). In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 9035, pp 401–416
7. Beyer D (2016) Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 9636, pp 887–904
8. Beyer D (2019) Automatic verification of c and java programs: Sv-comp 2019. In: Beyer D, Huisman M, Kordon F, Steffen B (eds) Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, pp 133–155
9. Beyer D, Dangl M (2019) Software verification with PDR: implementation and empirical evaluation of the state of the art. CoRR abs/1908.06271, URL `http://arxiv.org/abs/1908.06271`, `1908.06271`
10. Beyer D, Dangl M, Wendler P (2015) Boosting k-induction with continuously-refined invariants. In: Computer-Aided Verification, LNCS, vol 9206, pp 622–640
11. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 1633, pp 193–207
12. Biere A, Heule M, van Maaren H, Walsh T (2009) Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications, vol 185. IOS Press
13. Birgmeier J, Bradley AR, Weissenbacher G (2014) Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Computer Aided Verification, LNCS, vol 8559, pp 831–848
14. Björn Wachter DK, Ouaknine J (2013) Verifying multithreaded software with impact. In: Formal Methods in Computer Aided Design, pp 210–217
15. Bradley AR (2011) Sat-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation, Springer, pp 70–87
16. Bradley AR (2012) IC3 and beyond: Incremental, inductive verification. In: Computer Aided Verification, LNCS, vol 7358, p 4
17. Bradley AR (2012) Understanding IC3. In: Theory and Applications of Satisfiability Testing, LNCS, vol 7317, pp 1–14
18. Bradley AR, Manna Z (2007) The Calculus Of Computation: Decision Procedures With Applications To Verification, 1st edn. Springer-Verlag New York, Inc.
19. Brain M, Joshi S, Kroening D, Schrammel P (2015) Safety verification and refutation by k-invariants and k-induction. In: Static Analysis Symposium, LNCS, vol 9291, pp 145–161
20. Carter M, He S, Whitaker J, Rakamarić Z, Emmi M (2016) SMACK software verification toolchain. In: International Conference on Software Engineering, pp 589–592
21. Champion A, Chiba T, Kobayashi N, Sato R (2018) Ice-based refinement type discovery for higher-order functional programs. In: 24th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS), pp 365–384, DOI 10.1007/978-3-319-89960-2_20
22. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Computer-Aided Verification, LNCS, vol 1855, pp 154–169
23. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 2988, pp 168–176
24. De Moura L, Bjørner N (2008) Z3: An efficient SMT solver. In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 4963, pp 337–340
25. Donaldson A, Haller L, Kroening D, Rümmer P (2011) Software verification using k-induction. In: Static Analysis Symposium, LNCS, vol 6887, pp 351–368
26. Donaldson A, Kroening D, Rümmer P (2011) SCRATCH: A tool for automatic analysis of DMA races. In: Symposium On Principles And Practice Of Parallel Programming, pp 311–312
27. Donaldson AF, Haller L, Kroening D (2011) Strengthening induction-based race checking with lightweight static analysis. In: Verification, model checking, and abstract interpretation, LNCS, vol 6538, pp 169–183
28. Donaldson AF, Kroening D, Ruemmer P (2011) Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design 39(1):83–113
29. Dutertre B (2014) Yices 2.2. In: Computer-Aided Verification, LNCS, vol 8559, pp 737–744
30. Eén N, Sörensson N (2003) Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4):543–560
31. Ezudheen P, Neider D, D'Souza D, Garg P, Madhusudan P (2018) Horn-ice learning for synthesizing invariants and contracts. Proc ACM Program Lang 2(OOPSLA):131:1–131:25, DOI 10.1145/3276501

32. Furia CA, Meyer B, Velder S (2014) Loop invariants: Analysis, classification, and examples. ACM Comput Surv 46(3), DOI 10.1145/2506375, URL https://doi.org/10.1145/2506375
33. Gadelha MR, Monteiro F, Cordeiro L, Nicole D (2019) Esbmc v6.0: Verifying c programs using k-induction and invariant inference. In: Beyer D, Huisman M, Kordon F, Steffen B (eds) Tools and Algorithms for the Construction and Analysis of Systems, Springer International Publishing, Cham, pp 209–213
34. Gadelha MYR, Ismail HI, Cordeiro LC (2017) Handling loops in bounded model checking of C programs via k-induction. Software Tools for Technology Transfer 19(1):97–114
35. Gadelha MYR, Monteiro FR, Cordeiro LC, Nicole DA (2018) Towards counterexample-guided k-induction for fast bug detection. In: ACM Joint European Software Engineering Conference And The Foundations Of Software Engineering, pp 765–769
36. Garg P, Löding C, Madhusudan P, Neider D (2014) ICE: A robust framework for learning invariants. In: 26th International Conference Computer Aided Verification (CAV), pp 69–87, DOI 10.1007/978-3-319-08867-9_5
37. Garg P, Neider D, Madhusudan P, Roth D (2016) Learning invariants using decision trees and implication counterexamples. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp 499–512, DOI 10.1145/2837614.2837664
38. Goldberg D (1991) What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1):5–48
39. Große D, Le H, Drechsler R (2009) Induction-based formal verification of systemC TLM designs. In: Workshop On Microprocessor Test And Verification, pp 101–106
40. Gulavani BS, Henzinger TA, Kannan Y, Nori AV, Rajamani SK (2006) Synergy: A new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, SIGSOFT '06/FSE-14, pp 117–127, DOI 10.1145/1181775.1181790
41. Günther H, Laarman A, Weissenbacher G (2016) Vienna verification tool: IC3 for parallel software. In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 9636, pp 954–957
42. Gurfinkel A, Kahsai T, Komuravelli A, Navas JA (2015) The seahorn verification framework. In: Computer-Aided Verification, LNCS, vol 9206, pp 343–361
43. Hassan Z, Bradley AR, Somenzi F (2013) Better generalization in IC3. In: Formal Methods In Computer-Aided Design, pp 157–164
44. Heizmann M, Christ J, Dietsch D, Ermis E, Hoenicke J, Lindenmann M, Nutz A, Schilling C, Podelski A (2013) Ultimate automizer with smtinterpol. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol 7795, pp 641–643
45. Henry J, Monniaux D, Moy M (2012) PAGAI: A path sensitive static analyser. Electronic Notes in Theoretical Computer Science 289:15–25
46. Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12(10):576â580, DOI 10.1145/363235.363259
47. Hu Q, Breck J, Cyphert J, D'Antoni L, Reps T (2019) Proving unrealizability for syntax-guided synthesis. In: Dillig I, Tasiran S (eds) Computer Aided Verification, Springer International Publishing, Cham, pp 335–352
48. Hutter F, Babic D, Hoos HH, Hu AJ (2007) Boosting verification by automatic tuning of decision procedures. In: Formal Methods in Computer-Aided Design, pp 27–34
49. IEEE (2008) IEEE Standard For Floating-Point Arithmetic. IEEE 754-2008
50. Ivančić F, Shlyakhter I, Gupta A, Ganai MK (2005) Model checking C programs using F-SOFT. Computer Design pp 297–308
51. Jeannet B, Miné A (2009) Apron: A library of numerical abstract domains for static analysis. In: Proceedings of the 21st International Conference on Computer Aided Verification, Springer-Verlag, Berlin, Heidelberg, CAV'09, pp 661–667, DOI 10.1007/978-3-642-02658-4_52
52. Jovanović D, Dutertre B (2016) Property-directed k-induction. In: Formal Methods In Computer-Aided Design, pp 85–92
53. Kroening D, Tautschnig M (2014) CBMC - C bounded model checker. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol 8413, pp 389–391
54. Kroening D, Weissenbacher G (2011) Interpolation-based software verification with wolverine. In: 23rd International Conference Computer Aided Verification (CAV), pp 573–578, DOI 10.1007/978-3-642-22110-1_45
55. Kroening D, Ouaknine J, Strichman O, Wahl T, Worrell J (2011) Linear completeness thresholds for bounded model checking. In: Computer-Aided Verification, LNCS, vol 6806, pp 557–572
56. Lattner C, Adve V (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In: Symposium On Code Generation And Optimization, pp 75–96
57. Madhukar K, Wachter B, Kroening D, Lewis M, Srivas MK (2015) Accelerating invariant generation. In: Formal Methods in Computer-Aided Design, pp 105–111
58. Maisonneuve V, Hermant O, Irigoin F (2014) Computing invariants with transformers: Experimental scalability and accuracy. In: Numerical and Symbolic Abstract Domains, pp 17–31
59. Mälardalen WCET Research Group (2012) WCET benchmarks. http://www.mrtc.mdh.se/projects/

wcet/benchmarks.html, [Online; accessed August-2019]
60. Merz F, Falke S, Sinz C (2012) LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Verified Software: Theories, Tools, And Experiments, LNCS, vol 7152, pp 146–161
61. Morse J, Cordeiro LC, Nicole D, Fischer B (2015) Model checking LTL properties over ANSI-C programs with bounded traces. Software and System Modeling 14(1):65–81
62. ParisTech (2013) PIPS: Automatic parallelizer and code transformation framework. https://pips4u.org/, [Online; accessed August-2019]
63. Prasad MR, Biere A, Gupta A (2005) A survey of recent advances in SAT-based formal verification. Software Tools for Technology Transfer 7(2):156–173
64. Rival X, Mauborgne L (2007) The trace partitioning abstract domain. ACM Trans Program Lang Syst 29:26
65. Rocha H, Ismail H, Cordeiro LC, Barreto RS (2015) Model checking embedded C software using k-induction and invariants. In: Brazilian Symposium on Computing Systems Engineering, pp 90–95
66. Rocha W, Rocha H, Ismail H, Cordeiro LC, Fischer B (2017) Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution). In: Tools And Algorithms For The Construction And Analysis Of Systems, LNCS, vol 10206, pp 360–364
67. Scott J, Lee LH, Arends J, Moyer B (1998) Designing the low-power m*CORE architecture. In: Power Driven Microarchitecture Workshop, pp 145–150
68. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Formal Methods In Computer-Aided Design, LNCS, vol 1954, pp 108–125
69. Si X, Dai H, Raghothaman M, Naik M, Song L (2018) Learning loop invariants for program verification. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, Curran Associates Inc., NIPSâ18, p 7762â7773
70. SNU (2012) Real-time benchmarks. http://www.cprover.org/goto-cc/examples/snu.html, [Online; accessed August-2019]
71. Wright A, Felleisen M (1994) A syntactic approach to type soundness. Inf Comput 115(1):38–94, DOI 10.1006/inco.1994.1093