Chapter 7
Model Checking Embedded C Software Using *k*-Induction and Invariants

Herbert Rocha, Hussama Ismail, Lucas Cordeiro and Raimundo Barreto

7.1 Introduction

The Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) [8] or Satisfiability Modulo Theories (SMT) [2] have been successfully applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [11, 12, 25]. The idea behind the BMC techniques is to check the negation of a given property at a given depth, i.e., given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds the system k times and converts it into a Verification Condition (VC) ψ such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k.

Typically, BMC techniques are only able to falsify properties up to a given depth k; however, they are not able to prove the correctness of the system, unless an upper bound of k is known, i.e., a bound that unfolds all loops and recursive functions to their maximum possible depth. In particular, BMC techniques limit the visited regions of data structures (e.g., arrays) and the number of loop iterations. This limits the state space that needs to be explored during verification, leaving enough that real errors in applications [11, 12, 21, 25] can be found; BMC tools are, however, susceptible to exhaustion of time or memory limits for programs with loops whose bounds are too large or cannot be determined statically.

Consider, for example, the simple program in Listing 22 (left), in which the loop in line 2 runs an unknown number of times, depending on the initial nondeterministic value assigned to `x` in line 1. The assertion in line 3 holds independent of `x`'s initial

H. Rocha (✉)
Federal University of Roraima, Boa Vista, Brazil
e-mail: herberthb12@gmail.com

H. Ismail · L. Cordeiro · R. Barreto
Federal University of Amazonas, Manaus, Brazil
e-mail: hussamaismail@gmail.com

L. Cordeiro
e-mail: lucasccordeiro@gmail.com

R. Barreto
e-mail: xbarretox@gmail.com

© Springer Science+Business Media, LLC 2017

D. Lettnin and M. Winterholer (eds.), *Embedded Software Verification and Debugging*, Embedded Systems, DOI 10.1007/978-1-4614-2266-2_7

value. Unfortunately, BMC tools like CBMC [11], LLBMC [25], or ESBMC [12] typically fail to verify programs that contain such loops. Soundness requires that they insert a so-called *unwinding assertion* (the negated loop bound) at the end of the loop, as in Listing 23 (right), line 5. This *unwinding assertion* causes the BMC tool to fail if k is too small.

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

Listing 22 Unbounded loop

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

Listing 23 Finite unwinding

In mathematics, one usually attacks such unbounded problems using *proof by induction*. A variant called k-induction has been successfully combined with continuously refined invariants [6], to prove that (restricted) C programs do not contain data races [14, 15], or that design-time time constraints are respected [16]. Additionally, k-induction is a well-established technique in hardware verification, where it is easy to apply due to the monolithic transition relation present in hardware designs [16, 18, 32]. This paper contributes a new algorithm to prove correctness of C programs by k-induction in a completely automatic way (i.e., the user does not need to provide the loop invariant).

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

These algorithms were all implemented in the Efficient SMT-based Context-Bounded Model Checker tool (known as ESBMC[1]), which uses BMC techniques and SMT solvers (e.g., [10, 13]) to verify embedded systems written in C/C++ [12]. In Cordeiro et al. [12] the ESBMC tool is presented, which describes how the input program is encoded in SMT; what the strategies for unrolling loops are; what are the transformations/optimizations that are important for performance; what are the

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

benefits of using an SMT solver instead of a SAT solver; and how counterexamples to falsify properties are reconstructed. Here we extend our previous work in Rocha et al. [29] and Ramalho et al. [17] and focus our contribution on the combination of the k-induction algorithm with invariants. First, we describe the details of an accurate translation that extends ESBMC to prove the correctness of a given (safety) property for any depth without manual annotations of loops invariants. Second, we adopt program invariants (using polyhedra) in the k-induction algorithm, to speed up the verification time and to improve the quality of the results by solving more verification tasks in less time. Third, we show that our present implementation is applicable to a broader range of verification tasks, which other existing approaches are unable to support [14, 15, 18].

7.2 Motivating Example

As a motivating example, a program extracted from the benchmarks of the SV-COMP [3] is used as a running example as shown in Listing 24, which already includes invariants using polyhedra.

```
1  int main(int argc, char **argv)
2  {
3   uint64_t i=1, sn = 0;
4   assume( i==1 && sn==0 ); // Invariant
5   uint32_t n;
6   assume(n>=1);
7   while (i<=n) {
8     assume( 1<=i && i<=n ); // Invariant
9     sn = sn+a;
10    i++;
11  }
12  assume( 1<=i && n+1<=i ); // Invariant
13  assert(sn==n*a);
14 }
```

Listing 24 Running example for the k-induction algorithm.

In Listing 24, a is an integer constant and note that variables i and sn are declared with a type larger than the type of the variable n to avoid arithmetic overflow. Mathematically, the code above represents the implementation of the sum given by the following equation:

$$S_n = \sum_{i=1}^{n} a = na, n \geq 1 \tag{7.1}$$

In the code of Listing 24, the invariants produced by PIPS are included as assume statements; the property (represented by the assertion in line 13) must be *true* for any value of n (i.e., for any unfolding of the program). In contrast from our k-induction algorithm, BMC techniques have difficulties in proving the correctness of

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

7.3 Induction-Based Verification of C Programs Using Invariants

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

7.3.1 The Proposed k-Induction Algorithm

Listing 25 shows an overview of the proposed k-induction algorithm. We do not add additional details about the transformations in each step of the algorithm; we keep it simple and describe the details in the next subsections so that one can have a big picture of the proposed method. The input of the algorithm is a C program P together with the safety property ϕ. The algorithm returns *true* (if there is no path that violates the safety property), *false* (if there exists a path that violates the safety property), and *unknown* (if it does not succeed in computing an answer *true* or *false*).

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

7.3.1.1 The Difference to Other k-Induction Algorithms

Our k-induction algorithm is slightly different than those presented by Große et al. [18], Donaldson et al. [15], and Hagen et al. [19]. In Große et al., the forward condition and the inductive step are computed differently from our approach (as

described in Sect. 7.3.1) and the value of k is increased only at the end of the algorithm; in this particular case, computational resources are thus wasted since loops are usually unfolded at least two times. Donaldson et al. [15] and Hagen et al. [19] propose the k-induction with two steps only (i.e., the base case and the inductive step); however, the inductive step of the approach proposed by Donaldson et al. requires annotations in the code to introduce loops invariants. It is worth noting that Donaldson et al. improve the method and reduce the annotation overhead [14]. However, our method is completely automatic as in Hagen et al. [19]. Additionally, as observed in the experimental evaluation (see Sect. 7.4), the use of the forward condition, in our proposed method, improves significantly the quality of the results, because some programs that are hard to be proved by the inductive step can be proved by the forward condition using affine constraints.

```
1  input: program P and safety property φ
2  output: true, false, or unknown
3  k = 1
4  while k <= max_iterations do
5    if base_case(P, φ, k) then
6        show counterexample s[0..k]
7      return false
8    else
9      k=k+1
10     if forward_condition(P, φ, k) then
11       return true
12     else
13       if inductive_step(P, φ, k) then
14         return true
15       end-if
16     end-if
17   end-if
18 end-while
19 return unknown
```

Listing 25 An overview of the k-induction algorithm.

7.3.1.2 Loop-Free Programs

In the k-induction algorithm, the loop unwinding of the program is done incrementally from one to *max_iterations* (cf. Listing 25), where the number of unwindings is measured by counting the number of *backjumps* [27]. In each step of the k-induction algorithm, an instance of the program that contains k copies of the loop body corresponds to checking a loop-free program, which uses only *if*-statements in order to prevent its execution in the case that the loop ends before k iterations.

Definition 7.1 (*Loop-free Program*) A loop-free program is represented by a straight-line program (without loops) by providing an $ite\ (\theta, \rho_1, \rho_2)$ operator, which takes a Boolean formula θ and, depending on its value, selects either the second ρ_1 or the third argument ρ_2, where ρ_1 represents the loop body and ρ_2 represents either

another ite operator, which encodes a k-copy of the loop body, or an assertion/assume statement.

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

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

7.3.1.3 Program Transformations

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

The forward case inserts an unwinding assertion instead of an assumption after the loop, as follows $I \wedge T \Rightarrow \sigma \wedge \phi$. The forward condition, proposed by Große et al. [18], introduces a sequence of commands to check whether there is a path between an initial state and the current state k, while in the algorithm proposed in this paper, an assertion (i.e., the loop invariant) is automatically inserted by our algorithm, without the user's intervention, at the end of the loop to check whether all states are reached in k steps. Our base case and forward condition translations can easily be implemented on top of plain BMC.

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

$$A;\ while(c)\,\{S;\ E;\ U;\}\ R; \tag{7.2}$$

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

Definition 7.2 (*Loop Variable*) A loop variable is a variable $v \subseteq V$, where $V = V_{global} \cup V_{local}$ given that V_{global} is the set of global variables and V_{local} is the set of local variables that occur in the loop of a program.

Definition 7.3 (*Havoc Loop Variable*) A nondeterministic value is assigned to a loop variable v if and only if v is used in the loop termination condition σ, in the loop counter that controls iterations of a loop, or repeatedly modified inside the loop body.

The intuitive interpretation of S, U, and R is that if the current state (after executing E) is different than the previous state (before executing E), then new states are produced in the given loop iteration; otherwise, they are redundant and the code R is then responsible for preventing those redundant states to be included into the states vector. Note further that the code A assigns nondeterministic values to all loop variables so that the model checker can explore all possible states implicitly. In contrast, Große et al. [18] havoc all program variables, which makes it difficult to apply their approach to arbitrary programs since they do not provide enough information to constrain the havocked variables in the program. Similarly, the loop *for* can easily be converted into the loop *while* as follows: $for(B; c; D)\{E;\}$ is rewritten as

$$B;\ while(c)\{E; D;\} \tag{7.3}$$

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

The inductive step is thus represented by $\gamma \wedge \sigma \Rightarrow \phi$, where γ is the transition relation of $\hat{P}'$, which represents a loop-free program (cf. Definition 7.1) after applying transformations (7.2) and (7.3). The intuitive interpretation of the inductive step is to prove that, for any unfolding of the program, there is no assignment of particular values to the program variables that violates the safety property being checked. Finally, the induction hypothesis of the inductive step consists of the conjunction between the postconditions ($Post$) and the termination condition (σ) of the loop.

7.3.1.4 Invariant Generation

To infer program invariants, we adopted the PIPS [24] tool, which is an interprocedural source-to-source compiler framework for C and Fortran programs and relies on a polyhedral abstraction of program behavior. PIPS has been developed for almost 20 years to analyze large size programs automatically [28]. PIPS performs a two-step analysis: (1) each program instruction is associated to an affine transformer, representing its underlying transfer function. This is a bottom-up procedure, starting from elementary instructions, then working on compound statements and up to function definitions; (2) polyhedral invariants are propagated along with instructions, using previously computed transformers.

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

Algorithm 2 shows the method proposed, which receives as inputs the code generated by PIPS (`PIPSCode`) with invariants as comments, and it generates as output a new instance of the analyzed code (`NewCodeInv`) with invariants, adopting the function `assume` (*expr*), where *expr* is a expression supported by the C programming language. The time complexity of this algorithm is $O(n^2)$, where n is code size with invariants generated by PIPS. The algorithm is split into three parts: (1) identify the `#init` structure in the PIPS invariants; (2) generate code to support the translation of the `#init` structure in the invariant; and finally (3) translate mathematical expressions contained in the invariants, which is performed by the invariants transformation in the PIPS format to the C programming language.

Line 5 of Algorithm 2 performs the first part of the invariant translation, which consists of reading each line of the analyzed code with invariants and identifying whether a given comment is an invariant generated by PIPS (line 6). If an invariant is identified and it contains the structure `#init`, then the invariant location (the line number) is stored, as well as, the type and name of the variable, which has the structure prefix `#init` (line 8).

After identifying the `#init` structures in the invariants, the second part of Algorithm 2 performs line 12, which consists of reading again, each line of the analyzed code with invariants (`PIPSCode`), and identifying the beginning of each function in the code. For each identified function, the algorithm checks whether that function has identified some `#init` structure (line 15). If it has been identified, for each variable that has the suffix `#init`, a new line of code is generated at the beginning of the function, with the declaration of an auxiliary variable, which contains the old variable value, i.e., its value at the beginning of the function. The new created variable has the following format `type var_init = var`, where `type` is the identified variable type, and `var` is the identified variable name. During the execution of this algorithm, a new instance of the code (`NewCodeInv`) is generated.

In the third (and final part) of Algorithm 2 (line 22), each line of the new code instance (`NewCodeInv`) is read to convert each PIPS invariant into expressions supported by the C programming language. This transformation consists in applying regular expressions (line 27) to add operators (e.g., from $2j$ to $2 * j$) and replacing the structure `#init` to `_init`. For each analyzed PIPS comment/invariant, we generate a new line of code to the new format, where this line is concatenated with the operator `&&` and added to the `__ESBMC_assume` function.

Algorithm 2 Translation algorithm of PIPS invariants

```
1: Input: PIPSCode - C code with PIPS invariants
2: Output: NewCodeInv - New code with invariant supported by C programs
   // dictionary to identify #init
3: dict_variniteloc ← { }
   // list for the new code generated in the translation
4: NewCodeInv ← { }
   // Part 1 - identifying #init in the invariants
5: for all line of the PIPSCode do
6:     if is a PIPS comment in this pattern // P(w,x) {w == 0, x#init > 10} then
7:         if the comment has the pattern ([a-zA-Z0-9_]+)#init then
8:             dict_variniteloc[line] ← the variable suffixed #init
9:         end if
10:    end if
11: end for
   // Part 2 - code generation to support #init structure
12: for all line of PIPSCode do
13:    NewCodeInv ← line
14:    if is the beginning of a function then
15:        if has some line number of this function ∈ dict_variniteloc then
16:            for all variable ∈ dict_variniteloc do
17:                NewCodeInv ← Declare a variable with this pattern type var_init = var
18:            end for
19:        end if
20:    end if
21: end for
   // Part 3 - correct the invariant format
22: for all line of NewCodeInv do
   // list to the translated invariants
23:    listinvpips ← { }
24:    NewCodeInv ← line
25:    if is a PIPS comment in this pattern // P(w,x) {w == 0, x#init > 10} then
26:        for all expression ∈ {w == 0, x#init > 10} do
27:            listinvpips ← Reformulate the expression according to the C programs syntax and
   replace #init by _init
28:        end for
29:        NewCodeInv ← __ESBMC_assume(concatenate the invariants in listinvpips with &&)
30:    end if
31: end for
```

7.3.2 Running Example

In this section, we explain how the k-induction algorithm (see Listing 25) can prove correctness of the C program shown in Listing 24.

7.3.2.1 The Base Case

The base case initializes the limits of the loop's termination condition with nondeterministic values so that the model checker can explore all possible states implicitly. The pre- and postconditions of the loop shown in Listing 24, in static single assignment (SSA) form [27], are as follows:

$$Pre := \begin{bmatrix} n_1 = nondet_uint \wedge n_1 \geq 1 \\ \wedge \; sn_1 = 0 \wedge i_1 = 1 \end{bmatrix}$$

$$Post := \left[i_k \geq 1 \wedge \; i_k > n_1 \Rightarrow sn_k = n_1 \times a \right]$$

where Pre and $Post$ are the pre and postconditions to compute the sum given by Eq. (7.1), respectively, and $nondet_uint$ is a nondeterministic function, which can return any value of type *unsigned int*. In the preconditions, n_1 represents the first assignment to the variable n, which is a nondeterministic value greater than or equal to one. This ensures that the model checker explores all possible unwindings of the program. Additionally, sn_1 represents the first assignment to the variable sn and i_1 is the initial condition of the loop. In the postconditions, sn_k represents the assignment $n + 1$ for the variable sn in Listing 24, which must be *true* if $i_k > n_1$. The code that is not pre or postcondition is represented by the variable Q (i.e., the sequence of commands inside the loop *for*) and it does not undergo any transformation during the base case. The resulting code of the base case transformations can be seen in Listing 26 (cf. Definition 7.1). Note that the *assume* (in line 11), which consists of the termination condition, eliminates all execution paths that do not satisfy the constraint $i > n$. This ensures that the base case finds a counterexample of depth k without reporting any false negative result. Note further that other assume statements, shown in Listing 24, are simply eliminated during the symbolic execution by propagating constants and checking that the resulting expression evaluates to *true* [12].

```
1  int main(int argc, char **argv) {
2    uint64_t i, sn=0;
3    uint32_t n=nondet_uint();
4      assume (n>=1);
5    i=1;
6    if (i<=n) {          }
7      sn = sn + a;       } k copies
8      i++;               }
9    }
10     ...
11   assume(i>n); // unwinding assumption
12   assert(sn==n*a);
13 }
```

Listing 26 Example code for the proof by mathematical induction - during base case.

7.3.2.2 The Forward Condition

In the forward condition, the k-induction algorithm attempts to prove that the loop is sufficiently unfolded and whether the property is valid in all states reachable within k steps. The postconditions of the loop shown in Listing 24, in SSA form, can then be defined as follows:

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

The preconditions of the forward condition are identical to the base case. In the postconditions $Post$, there is an assertion to check whether the loop is sufficiently expanded, represented by the constraint $i_k > n_1$, where i_k represents the value of the variable i at iteration $n + 1$. The resulting code of the forward condition transformations can be seen in Listing 27 (cf. Definition 7.1). The forward condition attempts to prove that the loop is unfolded deep enough (by checking the loop invariant in line 11) and if the property is valid in all states reachable within k iterations (by checking the assertion in line 12). As in the base case, we also eliminate assume expressions by checking whether they evaluate to *true* by propagating constants during symbolic execution.

```
1  int main(int argc, char **argv) {
2    uint64_t i, sn=0;
3    uint32_t n=nondet_uint();
4      assume (n>=1);
5    i=1;
6    if (i<=n) {        ⎫
7      sn = sn + a;     ⎬ k copies
8      i++;             ⎭
9    }
10     ...
11   assert(i>n); // check loop invariant
12   assert(sn==n*a);
13 }
```

Listing 27 Example code for the proof by mathematical induction - during forward condition.

7.3.2.3 The Inductive Step

In the inductive step, the k-induction algorithm attempts to prove that, if the property is valid up to depth k, the same must be valid for the next value of k. Several changes are performed in the original code during this step. First, a structure called *statet* is defined, containing all variables within the loop and the exit condition of that loop. Then, a variable of type *statet* called *cs* (current state) is declared, which is responsible for storing the values of a given variable in a given iteration; in the current implementation, the *cs* data structure does not handle heap-allocated objects. A state

vector of size equal to the number of iterations of the loop is also declared, called sv (state vector) that will store the values of all variables on each iteration of the loop.

Before starting the loop, all loop variables (cf. Definitions 7.2 and 7.3) are initialized to nondeterministic values and stored in the state vector on the first iteration of the loop so that the model checker can explore all possible states implicitly. Within the loop, after storing the current state and executing the code inside the loop, all state variables are updated with the current values of the current iteration. An *assume* instruction is inserted with the condition that the current state is different from the previous one, to prevent redundant states to be inserted into the state vector; in this case, we compare $sv_j\,[i]$ to cs_j for $0 < j \leq k$ and $0 \leq i < k$. In the example we add constraints as follows:

$$\begin{aligned}
& sv_1\,[0] \neq cs_1 \\
& sv_1\,[0] \neq cs_1 \wedge sv_2\,[1] \neq cs_2 \\
& \ldots \\
& sv_1\,[0] \neq cs_1 \wedge sv_2\,[1] \neq cs_2 \wedge \ldots sv_k\,[i] \neq cs_k
\end{aligned} \tag{7.4}$$

Although we can compare $sv_k\,[i]$ to all cs_k for $i < k$ (since inequalities are not transitive), we found the encoding shown in Eq. (7.4) to be more efficient, leading to fewer timeouts when applied to the SV-COMP benchmarks.

Finally, an *assume* instruction is inserted after the loop, which is similar to that inserted in the base case. The pre- and postconditions of the loop shown in Listing 24, in SSA form, are defined as follows:

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

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

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

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

In the instruction set Q, $sv[i-1]$ is the vector position to save the current state cs_i, S is the actual code inside the loop, and the assignments $cs_i.v_0 = v_{0i} \wedge \ldots \wedge cs_i.v_m = v_{mi}$ represent the value of the variables in iteration i being saved in the current state cs_i. The modified code for the inductive step, using the notation defined in Sect. 7.3.1, can be seen in Listing 28. Note that the *if*-statement (lines 18–26) in Listing 28 is copied k-times according to Definition 7.1. As in the base case, the inductive step also inserts an *assume* instruction, which contains the termination condition. Differently from the base case, the inductive step proves that the property, specified by the assertion, is valid for any value of n.

Lemma 7.1 *If the induction hypothesis* $\{Post \wedge \neg(i \le n)\}$ *holds for* $k+1$ *consecutive iterations, then it also holds for k preceding iterations.*

After the loop *while* is finished, the induction hypothesis $\{Post \wedge \neg(i \le n)\}$ is satisfied on any number of iterations; in particular, the SMT solver can easily verify Lemma 7.1 and conclude that $sn == n * a$ is inductive relative to n. As in previous cases, we also eliminate assume expressions by checking whether they evaluate to *true* by propagating constants during symbolic execution.

```
1  //variables inside the loop
2  typedef struct state {
3   long long int i, sn;
4   unsigned int n;
5  } statet;
6  int main(int argc, char **argv) {
7   uint64_t i, sn=0;
8   uint32_t n=nondet_uint();
9   assume (n>=1);
10  i=1;
11  //declaration of current state
12  //and state vector
13  statet cs, sv[n];
14  //A: assign nondeterministc values
15  cs.i=nondet_uint();
16  cs.sn=nondet_uint();
17  cs.n=n;
18  if (i<=n) { //c: exit condition
19    sv[i-1]=cs;   //S: store current state      }
20    sn = sn + a; //E: code inside the loop      } k copies
21    //U: update variables with local values     }
22    cs.i=i; cs.sn=sn; cs.n=n;
23    //R: remove redundant states
24    assume(sv[i-1]!=cs);
25    i++;
26  }
27  ...
28  assume(i>n); //unwinding assumption
29  assert(sn==n*a);
30 }
```

Listing 28 Example code for the proof by mathematical induction - during inductive step.

7.4 Experimental Evaluation

This section is split into two parts. The setup is described in Sects. 7.4.1, 7.4.2 describes a comparison among DepthK,[2] ESBMC [12], CBMC [23], and CPAchecker (Configurable Software Verification Platform) [7] using a set of C benchmarks from SV-COMP [4] and embedded applications [26, 30, 33].

7.4.1 Experimental Setup

The experimental evaluation is conducted on a computer with Intel Xeon CPU $E5-2670$ CPU, 2.60GHz, 115GB RAM with Linux $3.13.0-35$-generic x86_64. Each verification task is limited to a CPU time of 15 minutes and a memory consumption of 15 GB. Additionally, we defined the *max_iterations* to 100 (cf. Listing 25). To evaluate all tools, we initially adopted: 142 ANSI-C programs of the SV-COMP 2015 benchmarks[3]; in particular, the *Loops* subcategory; and 34 ANSI-C programs used in embedded systems: Powerstone [30] contains a set of C programs for embedded systems (e.g., for automobile control and fax applications); while SNU real time [33] contains a set of C programs for matrix and signal processing functions such as matrix multiplication and decomposition, quadratic equations solving, cyclic redundancy check, fast fourier transform, LMS adaptive signal enhancement, and JPEG encoding; and the WCET [26] contains C programs adopted for worst-case execution time analysis. Additionally, we present a comparison with the tools:

- DepthK v1.0 with k-induction and invariants using polyhedra, the parameters are defined in the wrapper script available in the tool repository;
- ESBMC v1.25.2 adopting k-induction without invariants. We adopted the wrapper script from SV-COMP 2013[4] to execute the tool;
- CBMC v5.0 with k-induction, running the script provided in [5];
- CPAChecker[5] with k-induction and invariants at revision 15596 from its SVN repository. The options to execute the tool are defined in [5]. To improve the presentation, we report only the results of the options that presented the best results. These options are defined in [5] as follows: CPAchecker *cont.-ref. k-Induction* (*k-Ind InvGen*) and CPAchecker *no-inv k-Induction*.

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

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

[4] http://sv-comp.sosy-lab.org/2013/.

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

7.4.2 Experimental Results

In preliminary tests with the DepthK, for programs from the SV-COMP 2015 loops subcategory, we observed that 4.92% of the results are false incorrect. We believe that, in turns, this is due to the inclusion of invariants, which over-approximates the analyzed program, resulting in incorrect exploration of the states sets. We further identify that, in order to improve the approach implemented in DepthK tool, ones needs to apply a rechecking/refinement of the result found by the BMC procedure. Here, we re-check the results using the forward condition and the inductive step of the k-induction algorithm.

In DepthK, the program verification with invariants modifies the k-induction algorithm (Listing 25), as presented in Algorithm 3. In this new k-induction algorithm, we added the following variables: `last_result`, which stores the last result identified in the verification of a given step of the k induction, and `force_basecase`, which is an identifier to apply the rechecking procedure in the base case of the k-induction. The main difference in the execution of Algorithm 3 is to identify whether in the forward condition (line 18) and the inductive step (line 22), the verification result was `TRUE`, i.e., there was no property violation in a new k unwindings.

After running all tools, we obtained the results shown in Table 7.1 for the SV-COMP 2015 benchmark and in Table 7.2 for the embedded systems benchmarks, where each row of these tables means: name of the tool (Tool); total number of programs that satisfy the specification (correctly) identified by the tool (Correct Results); total number of programs that the tool has identified an error for a program that meets the specification, i.e., false alarm or incomplete analysis (False Incorrect); total number of programs that the tool does not identify an error, i.e., bug missing or weak analysis (True Incorrect); Total number of programs that the tool is unable to model check due to lack of resources, tool failure (crash), or the tool exceeded the verification time of 15 min (Unknown and TO); the run time in minutes to verify all programs (Time).

Table 7.1 Experimental results for the SV-COMP'15 loops subcategory

Tool	DepthK	ESBMC + k-induction	CPAchecker no-inv k-Induction	CPAchecker cont.-ref. k-Induction (k-Ind InvGen)	CBMC + k-induction
Correct results	94	70	78	76	64
False incorrect	1	0	0	1	3
True incorrect	0	0	4	7	1
Unknown and TO	47	72	60	58	74
Time (min)	190.38	141.58	742.58	756.01	1141.17

Algorithm 3 The *k*-induction algorithm with a recheck in base case.

```
1: Input: Program P' with invariants and the safety proprities φ
2: Output: TRUE, FALSE, or UNKNOWN
3: k = 1
4: last_result = UNKNOWN
5: force_basecase = 0
6: while k <= max_iterations do
7:     if force_basecase > 0 then
8:         k=k+5
9:     end if
10:    if BASECASE(P', φ, k) then
11:        show the counterexample s[0...k]
12:        return FALSE
13:    else
14:        if force_basecase > 0 then
15:            return last_result
16:        end if
17:        k=k+1
18:        if FORWARDCONDITION(P', φ, k) then
19:            force_basecase = 1
20:            last_result = TRUE
21:        else
22:            if INDUTIVESTEP(P', φ, k) then
23:                force_basecase = 1
24:                last_result = TRUE
25:            end if
26:        end if
27:    end if
28: end while
29: return UNKNOWN
```

Table 7.2 Experimental results for the Powerstone, SNU, and WCET benchmarks

Tools	DepthK	ESBMC + k-induction	CPAchecker no-inv k-Induction	CPAchecker cont.-ref. k-Induction (k-Ind InvGen)	CBMC + k-induction
Correct results	17	18	27	27	15
False incorrect	0	0	0	0	0
True incorrect	0	0	0	0	0
Unknown and TO	17	16	7	7	19
Time (min)	77.68	54.18	1.8	1.95	286.06

We evaluated the experimental results as follows: for each program we identified the verification result and time. We adopted the same scoring scheme that is used in SV-COMP 2015.[6] For every bug found, 1 score is assigned, for every correct

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

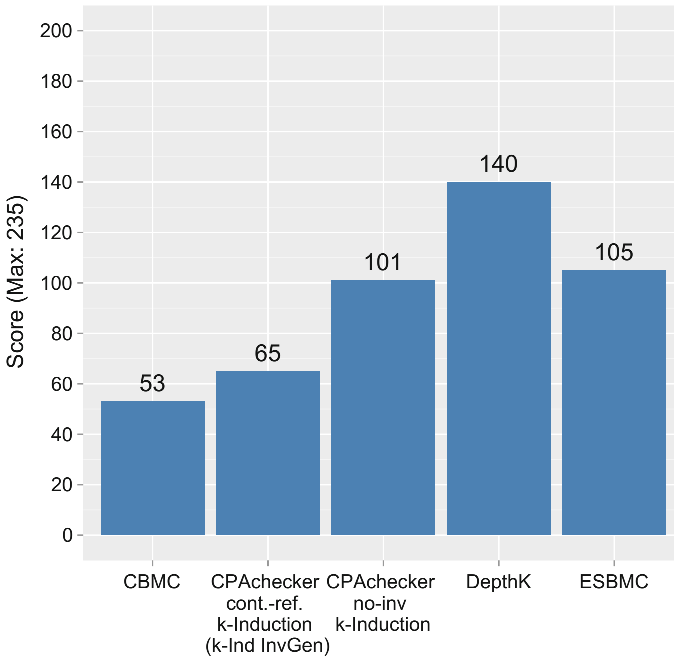

Fig. 7.1 Score to loops subcategory

safety proof, 2 scores are assigned. A score of 6 is subtracted for every wrong alarm (False Incorrect) reported by the tool, and 12 scores are subtracted for every wrong safety proof (True Incorrect). According to [6], this scoring scheme gives much more value in proving properties than finding counterexamples, and significantly punishes wrong answers to give credibility for tools. Figures 7.1 and 7.2 present the comparative results for the SV-COMP and embedded systems benchmarks, respectively. It is noteworthy that for the embedded systems programs, we have used safe programs [12] since we intend to check whether we have produced strong invariants to prove properties.

The experimental results in Fig. 7.1 show that the best scores belong to the DepthK, which combines k-induction with invariants, achieving 140 scores, ESBMC with k-induction without invariants achieved 105 scores, and CPAchecker *no-inv k-induction* achieved 101 scores. In Fig. 7.2, we found that the best scores belong to the CPAchecker *no-inv k-induction* with 54 scores, ESBMC with k-induction without invariants achieved 36 scores, and DepthK combined with k-induction and invariants, achieved 34 scores. We observed that DepthK achieved a lower score in the embedded system benchmarks. However, the DepthK results are still higher than that of CBMC and in the SV-COMP benchmark, DepthK achieved the highest score among all tools. In DepthK, we identified that, in turns, the lower score in the embedded system benchmarks is due to 35.30% of the results identified as `Unknown`, i.e., when

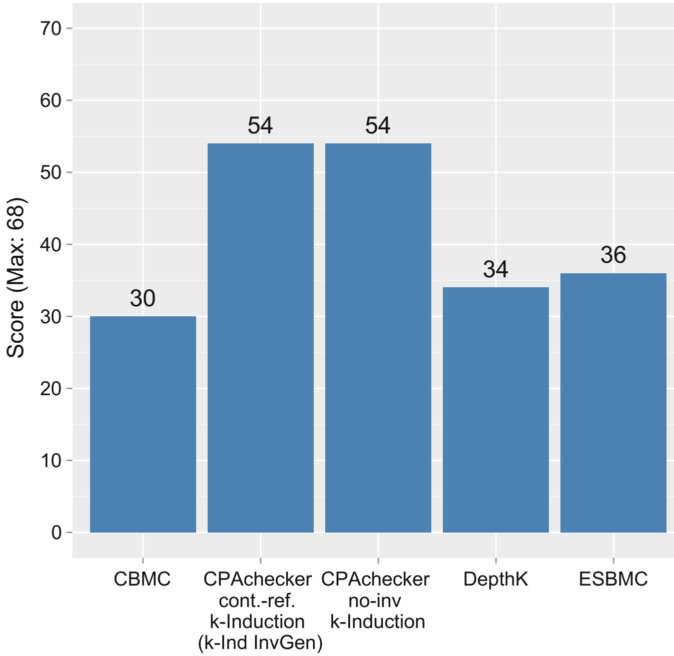

Fig. 7.2 Score to embedded systems

it is not possible to determine an outcome or due to a tool failure. We also identified failures related to invariant generation and code generation that is given as input to the BMC procedure. It is noteworthy that DepthK is still under development (in a somewhat preliminary state), so we argue that the results are promising.

To measure the impact of applying invariants to the k-induction based verification, we classified the distribution of the DepthK and ESBMC results, per verification step, i.e., base case, forward condition, and inductive step. Additionally, we included the verification tasks that result in `unknown` and `timeout` (CPU time exceeded 900 seconds). In this analysis, we evaluate only the results of DepthK and ESBMC, because they are part of our solution, and also because in the other tools, it is not possible to identify the steps of the k-induction in the standard logs generated by each tool. Figure 7.3 shows the distribution of the results, for each verification step, to the SV-COMP loops subcategory, and Fig. 7.4 presents the results to the embedded systems benchmarks.

The distribution of the results in Figs. 7.3 and 7.4 shows that DepthK can prove more than 25.35 and 29.41% of properties, during the inductive step, than ESBMC, respectively. These results lead to the conclusion that invariants helped the k-induction algorithm to prove that loops were sufficiently unwound and whenever the property is valid for k unwindings, it is also valid after the next unwinding of the system. We also identified that DepthK did not find a solution in 33.09% of the

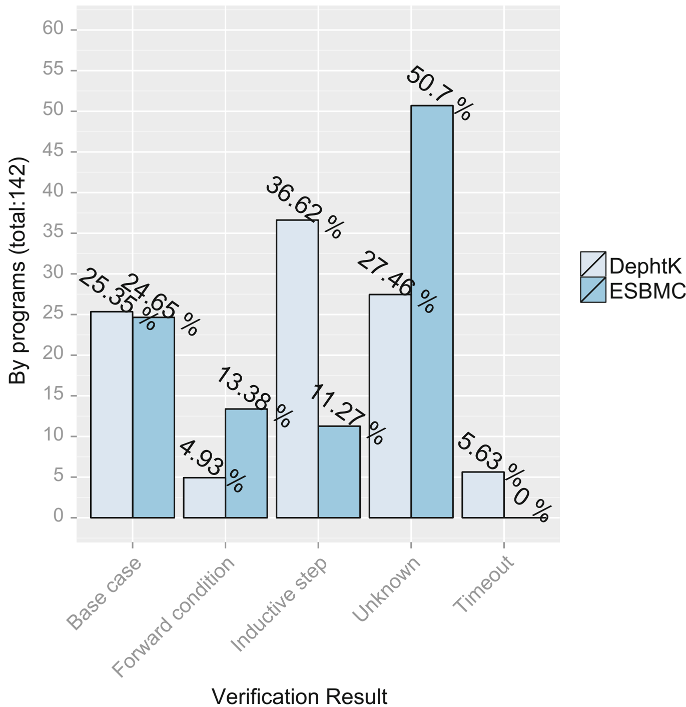

Fig. 7.3 Results for loops

programs in Fig. 7.3, and 50% in Fig. 7.4 (adding `Unknown` and `Timeout`). This is explained by the invariant generated from PIPS, which could not generate invariants strong enough to the verification with the *k*-induction, either due to a transformer or due to the invariants that are not convex; and also due to some errors in the tool implementation. ESBMC with *k*-induction did not find a solution in 50.7% of the programs in Fig. 7.3, i.e., 17.61% more than DepthK (adding `Unknown` and `Timeout`); and in Fig. 7.4, ESBMC did not find a solution in 47.06%, then only 3.64% less than the DepthK, thus providing evidences that the program invariants combined with *k*-induction can improve the verification results.

In Table 7.1, the verification time of DepthK to the loops subcategory is typically faster than the other tools, except for ESBMC, as can be seen in Fig. 7.5. This happens because DepthK has an additional time for the invariants generation. In Table 7.2, we identified that the verification time of DepthK is only faster than CBMC, as shown in Fig. 7.6. However, note that the DepthK verification time is proportional to ESBMC, since the time difference is 23.5 min; we can argue that this time difference is associated to the DepthK invariant generation.

We believe that the DepthK verification time can be significantly improved in two directions: fix some errors in the tool implementation, because some results generated as `Unknown` are related to failures in the tool execution; and adjustments in the PIPS

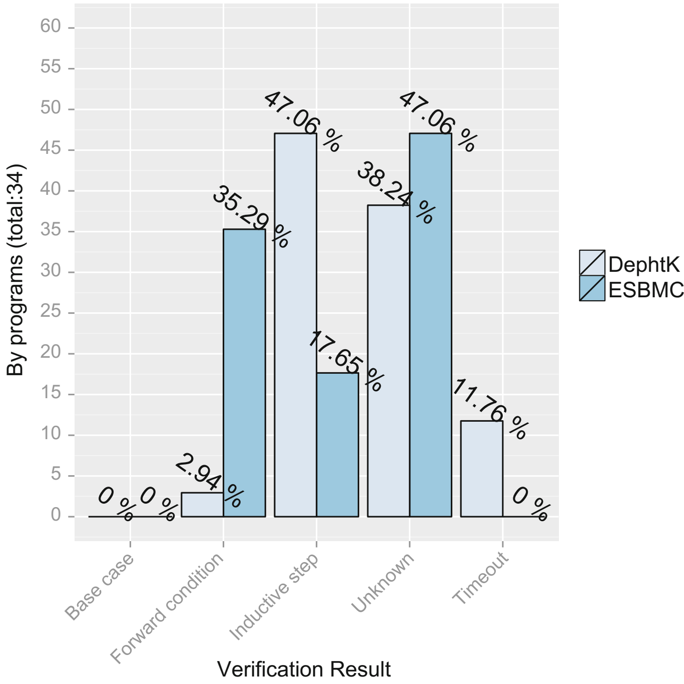

Fig. 7.4 Results for embedded programs

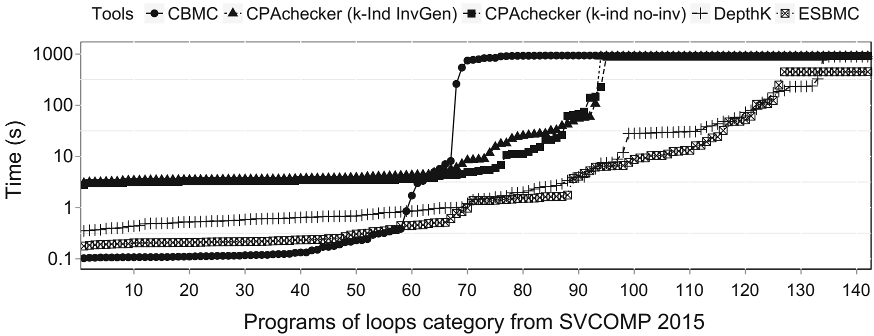

Fig. 7.5 Verification time to the loops subcategory

script parameters to generate invariants, since PIPS has a broad set of commands for code transformation, which might have a positive impact in the invariant generation for specific class of programs.

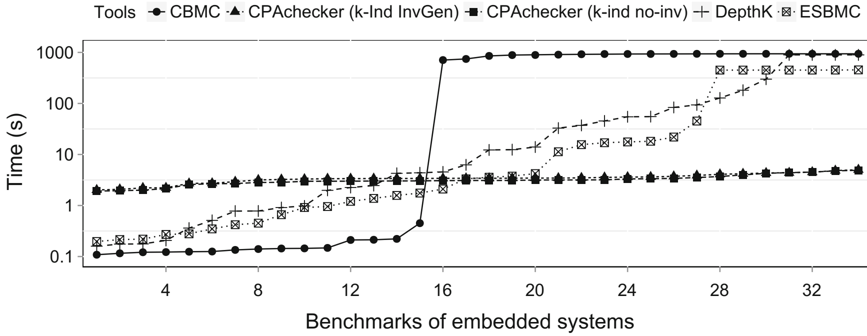

Fig. 7.6 Verification time to the embedded systems programs

7.5 Related Work

The application of the k-induction method is gaining popularity in the software verification community. Recently, Bradley et al. introduce "property-based reachability" (or IC3) procedure for the safety verification of systems [9, 20]. The authors have shown that IC3 can scale on certain benchmarks where k-induction fails to succeed. However, we do not compare k-induction against IC3 since it is already done by Bradley [9]; we focus our comparison on related k-induction procedures.

Previous work on the one hand have explored proofs by mathematical induction of hardware and software systems with some limitations, e.g., requiring changes in the code to introduce loop invariants [14, 15, 18]. This complicates the automation of the verification process, unless other methods are used in combination to automatically compute the loop invariant [1, 31]. Similar to the approach proposed by Hagen and Tinelli [19], our method is completely automatic and does not require the user to provide loops invariants as the final assertions after each loop. On the other hand, state-of-the-art BMC tools have been widely used, but as bug-finding tools since they typically analyze bounded program runs [11, 25]; completeness can only be ensured if the BMC tools know an upper bound on the depth of the state space, which is not generally the case. This paper closes this gap, providing clear evidence that the k-induction algorithm can be applied to a broader range of C programs without manual intervention.

Große et al. describe a method to prove properties of TLM designs (Transaction Level Modeling) in SystemC [18]. The approach consists of converting a SystemC program into a C program, and then it performs the proof of the properties by mathematical induction using the CBMC tool [11]. The difference to the one described in this paper lies on the transformations carried out in the forward condition. During the forward condition, transformations similar to those inserted during the inductive step in our approach, are introduced in the code to check whether there is a path between an initial state and the current state k; while the algorithm proposed in this paper, an

assertion is inserted at the end of the loop to verify that all states are reached in k steps.

Donaldson et al. describe a verification tool called Scratch [15] to detect data races during Direct Memory Access (DMA) in the CELL BE processor from IBM [15]. The approach used to verify C programs is the k-induction technique. The approach was implemented in the Scratch tool that uses two steps, the base case and the inductive step. The tool is able to prove the absence of data races, but it is restricted to verify that specific class of problems for a particular type of hardware. The steps of the algorithm are similar to the one proposed in this paper, but it requires annotations in the code to introduce loops invariants.

Kahsai et al. describe PKIND, a parallel version of the tool KIND, used to verify invariant properties of programs written in Lustre [22]. In order to verify a Lustre program, PKIND starts three processes, one for base case, one for inductive step, and one for invariant generation, note that unlike ESBMC, the k-induction algorithm used by PKIND does not have a forward condition step. This because of PKIND is for Lustre programs that do not terminate. Hence, there is no need for checking whether loops have been unrolled completely. The base case starts the verification with $k = 0$, and increments its value until it finds a counterexample or it receives a message from the inductive step process that a solution was found. Similarly, the inductive step increases the value of k until it receives a message from the base case process or a solution is found. The invariant generation process generates a set of candidates invariants from predefined templates and constantly feeds the inductive step process, as done recently by Beyer et al. [6].

7.6 Conclusions

The main contributions of this work are the design, implementation, and evaluation of the k-induction algorithm, adopting invariants using polyhedra in a verification tool, as well as, the use of the technique for the automated verification of reachability properties in embedded systems programs. To the best of our knowledge, this paper marks the first application of the k-induction algorithm to a broader range of embedded C programs. To validate the k-induction algorithm, experiments were performed involving 142 benchmarks of the SV-COMP 2015 *loops* subcategory, and 34 ANSI-C programs from the embedded systems benchmarks. Additionally, we presented a comparison to the ESBMC with k-induction, CBMC with k-induction, and CPAChecker with k-induction and invariants.

The experimental results are promising; the proposed method adopting k-induction with invariants (implemented in DepthK tool) determined 11.27% more accurate results than that obtained by CPAChecker, which had the second best result in the SV-COMP 2015 loops subcategory. The experimental results also show that the k-induction algorithm without invariants was able to verify 49.29% of the programs in the SV-COMP benchmarks in 141.58 min, and k-induction with invariants using polyhedra (i.e., DepthK) was able to verify 66.19% of the benchmarks in 190.38

min. Therefore, we identified that k-induction with invariants determined 17% more accurate results than the k-induction algorithm without invariants.

For embedded systems benchmarks, we identified some improvements in the DepthK tool, related to defects in the tool execution, and possible adjustments to invariant generation with PIPS. This is because the results were inferior compared to the other tools for the embedded systems benchmarks, where DepthK only obtained better results than CBMC tool. However, we argued that the proposed method, in comparison to other state-of-the-art tools, showed promising results indicating its effectiveness. In addition, both forms of the proposed method were able to prove or falsify a wide variety of safety properties; however, the k-induction algorithm, adopting polyhedral solves more verification tasks, which demonstrate an improvement of the induction k-algorithm effectiveness.

References

1. Ancourt C, Coelho F, Irigoin F (2010) A modular static analysis approach to affine loop invariants detection. In: Electronic notes in theoretical computer science (ENTCS). Elsevier Science Publishers B. V, pp 3–16
2. Barrett CW, Sebastiani R, Seshia SA, Tinelli C (2009) Satisfiability modulo theories. In: Handbook of satisfiability. IOS Press, pp 825–885
3. Beyer D (2013) Second competition on software verification—(Summary of SV-COMP 2013). In: Conference on tools and algorithms for the construction and analysis of systems (TACAS). Springer, pp 594–609
4. Beyer D (2015) Software verification and verifiable witnesses—(Report on SV-COMP 2015). In: Conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 401–416
5. Beyer D, Dangl M, Wendler P (2015) Boosting k-Induction with continuously-refined invariants. http://www.sosy-lab.org/~dbeyer/cpa-k-induction/
6. Beyer D, Dangl M, Wendler P (2015) Combining k-Induction with continuously-refined invariants. CoRR abs/1502.00096. http://arxiv.org/abs/1502.00096
7. Beyer D, Keremoglu ME (2011) CPAchecker: a tool for configurable software verification. In: Conference on computer-aided verification (CAV), pp 184–190
8. Biere A (2009) Bounded model checking. In: Handbook of satisfiability. IOS Press, pp 457–481
9. Bradley AR (2012) IC3 and beyond: incremental, inductive verification. In: Computer aided verification (CAV). Springer, p 4
10. Brummayer R, Biere A (2009) Boolector: an efficient SMT solver for bit-vectors and arrays. In: Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems: held as part of the joint European conferences on theory and practice of software (ETAPS). Springer, pp 174–177
11. 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 (TACAS). Springer, pp 168–176
12. Cordeiro L, Fischer B, Marques-Silva J (2012) SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans Softw Eng (TSE):957–974
13. De Moura L, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and algorithms for the construction and analysis of systems (TACAS). Springer, pp 337–340
14. Donaldson AF, Haller L, Kroening D, Rümmer P (2011) Software verification using k-induction. In: Proceedings of the 18th international static analysis symposium (SAS). Springer, pp 351–368

15. Donaldson AF, Kroening D, Ruemmer P (2010) Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In: Proceedings of the 16th international conference on tools and algorithms for the construction and analysis of systems (TACAS). Springer, pp 280–295
16. Eén N, Sörensson N (2003) Temporal induction by incremental SAT solving. Electronic notes in theoretical computer science (ENTCS), pp 543–560
17. Gadelha M, Ismail H, Cordeiro L (2015) andling loops in bounded model checking of c programs via k-induction. Int J Softw Tools Technol Transf (to appear) (2015)
18. Große D, Le HM, Drechsler R (2009) Induction-based formal verification of SystemC TLM designs. In: 10th International workshop on microprocessor test and verification (MTV), pp 101–106
19. Hagen G, Tinelli C (2008) Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Proceedings of the 8th international conference on formal methods in computer-aided design (FMCAD). IEEE, pp 109–117
20. Hassan Z, Bradley AR, Somenzi F (2013) Better generalization in IC3. In: Formal methods in computer-aided design (FMCAD). IEEE, pp 157–164
21. Ivancic F, Shlyakhter I, Gupta A, Ganai MK (2005) Model checking C programs using F-SOFT. In: 23rd international conference on computer design (ICCD). IEEE Computer Society, pp 297–308
22. Kahsai T, Tinelli C (2011) Pkind: A parallel k-induction based model checker. In: Proceedings 10th international workshop on parallel and distributed methods in verification (PDMC), pp 55–62
23. Kroening D, Tautschnig M (2014) CBMC—C Bounded model checker—(Competition Contribution). In: Conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 389–391
24. Maisonneuve V, Hermant O, Irigoin F (2014) Computing invariants with transformers: experimental scalability and accuracy. In: 5th International workshop on numerical and symbolic abstract domains (NSAD). Electronic notes in theoretical computer science (ENTCS). Elsevier, pp 17–31
25. Merz F, Falke S, Sinz C (2012) LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: Proceedings of the 4th international conference on verified software: theories, tools, experiments (VSTTE). Springer, pp 146–161
26. MRTC: WCET Benchmarks (2012) Mälardalen Real-Time Research Center. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
27. Muchnick SS (1997) Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
28. ParisTech M (2013) PIPS: Automatic parallelizer and code transformation framework. http://pips4u.org
29. 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 (SBESC). IEEE, pp 90–95
30. Scott J, Lee LH, Arends J, Moyer B (1998) Designing the Low-Power M*CORE Architecture. In: Power driven microarchitecture workshop, pp 145–150
31. Sharma R, Dillig I, Dillig T, Aiken A (2011) Simplifying loop invariant generation using splitter predicates. In: Proceedings of the 23rd international conference on computer aided verification (CAV). Springer, pp 703–719
32. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Formal methods in computer-aided design (FMCAD), pp 108–125
33. SNU (2012) SNU real-time benchmarks. http://www.cprover.org/goto-cc/examples/snu.html