Model Checking Embedded C Software using *k*-Induction and Invariants

Herbert Rocha*, Hussama Ismail†, Lucas Cordeiro†, and Raimundo Barreto†
*Federal University of Roraima, †Federal University of Amazonas
E-mail: herbert.rocha@ufrr.br, hussamaismail@gmail.com,
lucascordeiro@ufam.edu.br, rbarreto@icomp.ufam.edu.br

***Abstract*—We present a proof by induction algorithm, which combines *k*-induction with invariants to model check embedded C software with bounded and unbounded loops. The *k*-induction algorithm consists of three cases: in the base case, we aim to find a counterexample with up to *k* loop unwindings; in the forward condition, we check whether loops have been fully unrolled and that the safety property ϕ holds in all states reachable within k unwindings; and in the inductive step, we check that whenever ϕ holds for k unwindings, it also holds after the next unwinding of the system. For each step of the *k*-induction algorithm, we infer invariants using affine constraints (*i.e.*, polyhedral) to specify pre- and post-conditions. Experimental results show that our approach can handle a wide variety of safety properties in typical embedded software applications from telecommunications, control systems, and medical devices; we demonstrate an improvement of the induction algorithm effectiveness if compared to other approaches.**

I. Introduction

The Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) have been applied to verify single- and multi-threaded programs and to find subtle bugs in real programs [1], [2], [3]. 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*; 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 [1], [2], [3], [4] 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.

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 [5], to prove that (restricted) C programs do not contain data races [6], [7], or that design-time time constraints are respected [8]. 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 [8], [9], [10]. This paper contributes a new algorithm to prove correctness of C programs by combining *k*-induction with invariants in a completely automatic way.

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, we infer 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 (ESBMC) tool, which uses BMC techniques and SMT solvers to verify embedded systems written in C/C++ [3], [11]. In Cordeiro et al. [3], [11] the ESBMC tool is presented, which describes how the input program is encoded in SMT; what the strategies for unrolling loops are; what are the transformations/optimizations that are important for performance; what are the benefits of using an SMT solver instead of a SAT solver; and how counterexamples to falsify properties are reconstructed.

Here we extend our previous work and focus our contribution on the combination of the *k*-induction algorithm with invariants. First, we describe the details of an accurate translation that extends ESBMC to prove the correctness of a given (safety) property for any depth without manual annotations of loops invariants. Second, we adopt program invariants (using polyhedra) in the *k*-induction algorithm, to improve the quality of the results by solving more verification tasks. Third, we show that our implementation is applicable to a broader range of verification tasks; in particular embedded systems, where existing approaches do not to support [6], [7], [9].

II. 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.

Fig. 1 shows an overview of the proposed *k*-induction algorithm. The input of the algorithm is a C program P with invariants 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

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

Fig. 1: An overview of the *k*-induction algorithm.

condition, global correctness of the loop w.r.t. the property is shown for the case that the loop iterates at most k times; and in the inductive step, the algorithm checks that, if the property is valid in *k* iterations, then it must be valid for the next iterations. The algorithm runs up to a maximum number of iterations and increases the value of k if it cannot falsify the property. In Fig. 1, the algorithm also performs a rechecking/refinement of the result (using the flag `force_basecase`) by the BMC procedure. In particular, we re-check the results in the forward condition and the inductive step (adopting an increment of the actual k). This re-checking procedure is needed due to the inclusion of invariants, which over-approximates the analyzed program; otherwise, the invariants could result in incorrect exploration of the states sets.

A. Loop-free Programs

In the *k*-induction algorithm, the loop unwinding of the program is done incrementally from one to *max_iterations* (cf. Fig. 1), where the number of unwindings is measured by counting the number of *backjumps* [12]. In each step, an instance of the program that contains k copies of the loop body corresponds to checking a loop-free program, which uses only *if*-statements in order to prevent its execution in the case that the loop ends before k iterations.

Definition 1: **(Loop-free Program)** *A loop-free program is represented by a straight-line program (without loops) by providing an* $ite\,(\theta, \rho_1, \rho_2)$ *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.*

Each step of our *k*-induction algorithm (except for the base case) 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 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).

B. 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.

The forward case inserts an unwinding assertion instead of an assumption after the loop, as follows $I \wedge T \Rightarrow \sigma \wedge \phi$. Our base case and forward condition translations are implemented on top of plain BMC. However, for the inductive step of the algorithm, several transformations are carried out. In particular, the loop $while(c)\,\{E;\}$ is converted into

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

where A is the code responsible for assigning non-deterministic values to all 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 2: **(Loop Variable)** *A loop variable is a variable* $v \subseteq V$*, where* $V = V_{global} \cup V_{local}$ *given that* V_{global} *is the set of global variables and* V_{local} *is the set of local variables that occur in the loop of a program.*

Definition 3: **(Havoc Loop Variable)** *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 modified inside the loop body.*

The intuitive interpretation of S, U, and R is that if the current state (after executing E) is different than the previous state (before executing E), then new states are produced in the given loop iteration; otherwise, they are redundant and the code R is then responsible for preventing those redundant states to be included into the states vector. Note further that the code A assigns non-deterministic values to all loop variables so that the model checker can explore all possible states implicitly. Similarly, the loop *for* can easily be converted into the loop *while* as follows: $for(B; c; D)\,\{E;\}$ is rewritten as

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

where B is the initial condition of the loop, c is the 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 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 1) after applying transformations

(1) and (2). The intuitive interpretation of the inductive step is to prove that, for any unfolding of the program, there is no assignment of particular values to the program variables that violates the safety property being checked. Finally, the induction hypothesis of the inductive step consists of the conjunction between the postconditions ($Post$) and the termination condition (σ) of the loop.

C. Invariant Generation

To infer program invariants, we adopted the PIPS [13] 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 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 1 shows the proposed method, 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 an 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 3 of Algorithm 1 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 4). 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 prefix `#init`.

After identifying the `#init` structures in the invariants, the second part of Algorithm 1 performs line 10, which consists of reading 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 13). 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. During the execution of

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

Algorithm 1: Translation algorithm of invariants

this algorithm, a new instance of the code (`NewCodeInv`) is generated.

In the third (and final part) of Algorithm 1 (line 20), each line of the new code instance (`NewCodeInv`) is read to convert each PIPS invariant into expressions supported by the C language. This transformation consists in applying regular expressions (line 25) 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 code line to the new format, where this line is concatenated with the operator `&&` and added to the `__ESBMC_assume` function.

III. Experimental Evaluation

This section is split into two parts. The setup is described in Section III-A and Section III-B describes a comparison among DepthK [1], ESBMC [3], CBMC [1], and CPAchecker [5] using a set of C benchmarks from SV-COMP [14] and embedded applications [15], [16], [17].

A. 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. Fig. 1). To evaluate all tools, we initially adopted 142 ANSI-C programs of the SV-COMP

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

2015 benchmarks; in particular, the *Loops* subcategory; and 34 ANSI-C programs used in embedded systems: Powerstone [16] contains a set of C programs for embedded systems (*e.g.*, for automobile control and fax applications); while SNU real-time [17] 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 [15] contains C programs adopted for worst-case execution time analysis.

We also 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[2] to execute the tool; CBMC v5.0 with k-induction, running the script provided in [5]; CPAChecker[3] with k-induction and invariants at revision 15596 from its SVN repository. The options to execute the tool are defined in [5].

B. Experimental Results

After running all tools, we obtained the results shown in Table I for the SV-COMP 2015 benchmark and in Table II 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).

We evaluated all tools as follows: for each program we identified the verification result and time. We adopted the same scoring scheme that is used in SVCOMP 2015[4]. For every bug found, 1 score is assigned, for every correct safety proof, 2 scores are assigned. A score of 6 is subtracted for every wrong alarm (False Incorrect), and 12 scores are subtracted for every wrong safety proof (True Incorrect). According to [5], this scoring scheme gives much more value in proving properties than finding counterexamples, and significantly punishes wrong answers to give credibility for tool developers. It is noteworthy that for the embedded systems programs, we have used safe programs [3] since we intend to check whether we produce strong invariants to prove properties.

The experimental results related to *Loops* benchmarks had shown 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, CPAchecker *no-inv k-induction* achieved 101 scores, and CBMC achieved 53 scores. In the embedded systems benchmarks, 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, DepthK combined with k-induction and invariants, achieved 34 scores, and CBMC achieved 30 scores.

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

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

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

We observed that DepthK achieved a lower score in the embedded systems 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 low score in the embedded systems benchmarks is due to 35.30% of the results identified as `Unknown`, *i.e.*, when 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 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`. 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.

The result distribution shows that DepthK can prove more than 25.35% and 29.41% of the loops and embedded systems properties than ESBMC during the inductive step, respectively. These results lead to the conclusion that invariants help the k-induction algorithm to prove more properties. We also identified that DepthK did not find a solution in 33.09% of the programs in the SV-COMP benchmarks and 50% in the embedded systems benchmarks (producing `Unknown` and `Timeout`). This is explained due to the invariants generated from PIPS, which are not strong enough for 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 the SV-COMP benchmark, *i.e.*, 17.61% more than DepthK (adding `Unknown` and `Timeout`); and in the embedded benchmarks, ESBMC did not find a solution in 47.06%, then only 3.64% less than DepthK, thus providing evidences that the program invariants combined with k-induction can improve the verification results.

In Table I, the verification time of DepthK to the loops benchmarks is usually faster than the other tools, except for ESBMC, as shown in Fig. 2. This happens because DepthK has an additional time for the invariants generation. In Table II, we identified that the verification time of DepthK is only faster than CBMC (see Fig. 3). However, note that the DepthK verification time is proportional to ESBMC, since the time difference is 23.5min; we can argue that this difference is associated to the DepthK invariant generation.

We believe that DepthK verification time can be improved in two directions: fix errors in the tool implementation, because some results generated as `Unknown` are related to failures in the tool execution; and adjustments in the PIPS script parameters to generate stronger invariants, since PIPS has a broad set of commands for code transformation, parameter tuning might have a positive impact.

IV. Related Work

The k-induction application 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 [18], [19]. The authors have

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	190.38min	141.58min	742.58min	756.01min	1141.17min

TABLE I: Experimental results for the SVCOMP'15 loops subcategory.

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	77.68min	54.18min	1.8min	1.95min	286.06min

TABLE II: Experimental results for the Powerstone, SNU, and WCET benchmarks.

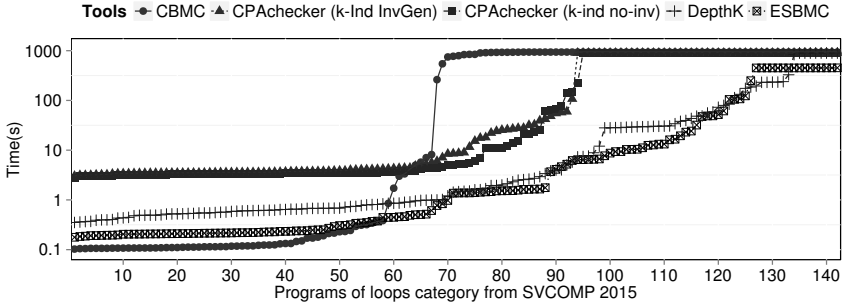

Fig. 2: Verification time to the loops subcategory.

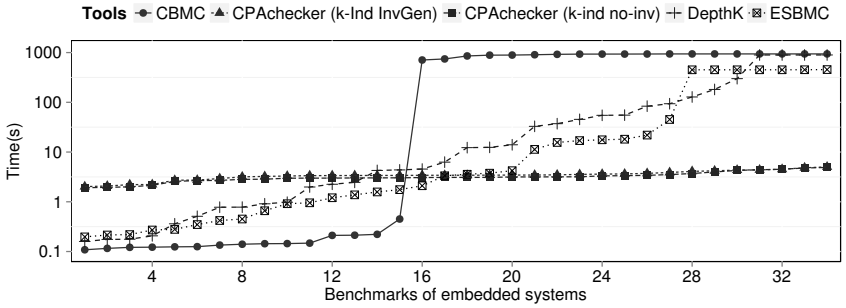

Fig. 3: Verification time to the embedded systems programs.

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 [18]; 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 [6], [9]. This complicates the automation of the verification process, unless other methods are used in combination to automatically compute the loop invariant [20], [21]. Similar to the approach proposed by [22], [7], 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 [1], [2]. This paper closes this gap, providing clear evidence that the *k*-induction algorithm in combination with invariants can be applied to a broader range of C programs without manual intervention.

Große et al. describe a method to prove properties of

Transaction Level Modeling designs in SystemC [9]. 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 [1]. 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 [6] to detect data races during Direct Memory Access in the CELL BE processor from IBM [6]. The approach used to verify C programs is *k*-induction, which is implemented in the Scratch tool using two steps: the base case and the inductive step. Scratch can 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 [23]. 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 happens because PKIND is used 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. [5].

V. Conclusions

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 (DepthK) 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, and *k*-induction with invariants (DepthK) was able to verify 66.19% of the benchmarks. 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. As future work, we will improve the robustness of DepthK and tune the PIPS parameters to produce stronger invariants.

Acknowledgment

The authors acknowledge the support granted by FAPEAM through process number $1135/2011$ (PRONEX), $582/2014$ and $1722/2014$ (PRO-TI-PESQUISA), and by CNPq $475647/2013$-0 (UNIVERSAL).

References

[1] D. Kroening and M. Tautschnig, "CBMC - C Bounded Model Checker - (Competition Contribution)," in *TACAS*. Springer, 2014, pp. 389–391.

[2] F. Merz, S. Falke, and C. Sinz, "LLBMC: bounded model checking of C and C++ programs using a compiler IR," in *VSTTE*. Springer-Verlag, 2012, pp. 146–161.

[3] L. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-Based Bounded Model Checking for Embedded ANSI-C Software," in *TSE*. IEEE, 2012, pp. 957–974.

[4] F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai, "Model Checking C Programs Using F-SOFT," in *ICCD*. IEEE Computer Society.

[5] D. Beyer, M. Dangl, and P. Wendler, "Combining k-Induction with Continuously-Refined Invariants," *CoRR*, vol. abs/1502.00096, 2015. [Online]. Available: http://arxiv.org/abs/1502.00096

[6] A. F. Donaldson, D. Kroening, and P. Ruemmer, "Automatic Analysis of Scratch-pad Memory Code for Heterogeneous Multicore Processors," in *TACAS*. Springer, 2010, pp. 280–295.

[7] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software Verification Using k-Induction," in *SAS*. Springer, 2011, pp. 351–368.

[8] N. Eén and N. Sörensson, "Temporal Induction by Incremental SAT Solving," *ENTCS*, pp. 543–560, 2003.

[9] D. Große, H. M. Le, and R. Drechsler, "Induction-Based Formal Verification of SystemC TLM Designs," in *MTV*, 2009, pp. 101–106.

[10] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *FMCAD*. Springer, 2000, pp. 108–125.

[11] M. Ramalho, L. Cordeiro, A. Cavalcante, and V. L. Jr, "Verificação Baseada em Indução Matemática de Programas C/C++," in *SBESC*. SBC, 2013, pp. 1–6.

[12] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[13] M. ParisTech, "PIPS: Automatic Parallelizer and Code Transformation Framework," Available at http://pips4u.org, 2013.

[14] D. Beyer, "Software Verification and Verifiable Witnesses - (Report on SV-COMP 2015)," in *TACAS*. Springer, 2015, pp. 401–416.

[15] MRTC, "WCET Benchmarks," Mälardalen Real-Time Research Center. Available at http://www.mrtc.mdh.se/projects/wcet/benchmarks.html, 2012.

[16] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the Low-Power M*CORE Architecture," in *PDM*, 1998, pp. 145–150.

[17] SNU, "SNU Real-Time Benchmarks," Available at http://www.cprover.org/goto-cc/examples/snu.html, 2012.

[18] A. R. Bradley, "IC3 and beyond: Incremental, Inductive Verification," in *CAV*. Springer, 2012, p. 4.

[19] Z. Hassan, A. R. Bradley, and F. Somenzi, "Better generalization in IC3," in *FMCAD*. IEEE, 2013, pp. 157–164.

[20] R. Sharma, I. Dillig, T. Dillig, and A. Aiken, "Simplifying loop invariant generation using splitter predicates," in *CAV*. Springer-Verlag, 2011, pp. 703–719.

[21] C. Ancourt, F. Coelho, and F. Irigoin, "A Modular Static Analysis Approach to Affine Loop Invariants Detection," in *ENTCS*. Elsevier Science Publishers B. V., 2010, pp. 3–16.

[22] G. Hagen and C. Tinelli, "Scaling up the formal verification of Lustre programs with SMT-based techniques," in *FMCAD*. IEEE, 2008, pp. 109–117.

[23] T. Kahsai and C. Tinelli, "Pkind: A parallel k-induction based model checker," in *PDMC*, 2011, pp. 55–62.