Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Handling Loops in Bounded Model Checking of C Programs via *k*-Induction

Mikhail Y. R. Gadelha, Hussama I. Ismail, and Lucas C. Cordeiro

Electronic and Information Research Center, Federal University of Amazonas, Brazil

Received: date / Revised version: date

Abstract The first attempts to apply the k-induction method to software verification are only recent. In this paper, we present a novel proof by induction algorithm, which is built on the top of a symbolic context-bounded model checker and uses an iterative deepening approach to verify, for each step k up to a given maximum, whether a given safety property ϕ holds in the program. The proposed k-induction algorithm consists of three different cases called *base case*, *forward condition*, and *inductive step*. Intuitively, 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 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. The algorithm was implemented in two different ways, a sequential and a parallel one, and the results were compared. Experimental results show that both forms of the algorithm can handle a wide variety of safety properties extracted from standard benchmarks, ranging from reachability to time constraints. And by comparison, the parallel algorithm solves more verification tasks in less time. This paper marks the first application of the k-induction algorithm to a broader range of C programs; in particular, we show that our k-induction method outperforms CPAChecker in terms of correct results, which is a state-of-the-art k-induction-based verification tool for C programs.

Keywords. software engineering, formal methods, verification, model checking, k-induction.

1 Introduction

Bounded Model Checking (BMC) techniques based on Boolean Satisfiability (SAT) [1] 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 [3,4,5]. The idea behind the BMC techniques is to check for the violation 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 the 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 to a given bound k. This limits the state space that needs to be explored during verification, leaving enough that real errors in applications [3,4,5,6] 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 Fig. 1a, in which the loop in line 2 runs an unknown number of times, depending on the initial non-deterministic value assigned to `x` in line 1. The assertion in line 3 holds independent of `x`'s initial value. Unfortunately, BMC tools like CBMC [3], LLBMC [4], or ESBMC [7] 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 Fig. 1b, line 5. This *unwinding assertion* causes the BMC tool to fail if k is too small.

One technique typically used to prove properties, for any given depth, is mathematical induction. In particular, the algorithm called k-induction was successfully ap-

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

Simple unbounded loop program.

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

Finite unwinding done by BMC.

Figure 2: Unbounded loop and finite unwinding.

plied to ensure that (restricted) C programs do not contain data races [8,9] and to respect time constraints specified during the design phase of a system [10]. Additionally, the k-induction is a well-established technique in hardware verification, where it is easier to be applied due to the monolithic transition relation present in hardware designs [10,11,12]. This paper contributes a new algorithm to prove correctness of (a large set of) C programs by mathematical induction in a completely automatic way (i.e., the user does not need to provide the loop invariant). Although the inductive step of our k-induction algorithm does not support heap-manipulating programs and unbounded recursion, we show in our (extensive) experiments that it is still useful for software verification.

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. The k-induction algorithm was implemented in two different ways, a sequential and a parallel one, where in the first, the three cases are performed in a sequential manner, starting with the base case, followed by the forward condition, and ending with the inductive step. In the parallel implementation, three processes are created and monitored by a main process to exploit the availablity of multi-core processors in model checking, as done by Holzmann et al. [13] and Kahsai et al. [14]. Each of the three processes performs a case of the k-induction algorithm and communicates the result to a main process.

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., [15,16]) to verify embedded systems written in C/C++ [7,17]. In Cordeiro et al. [7,18] 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 focus our contribution on the k-induction algorithm. 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 use a multi-process implementation of the k-induction algorithm, similar to Kahsai et al. [14], to speedup the verification time and to improve the quality of the results by solving more verification tasks in less time. Third, we show that our present implementation is also applicable to a broader range of verification tasks, where other existing approaches are unable to support [8, 9,11,19].

To validate the implementations of the algorithm, we used several benchmarks from the International Competition on Software Verification (SV-COMP) [20], where the sequential k-induction algorithm (in combination with the plain BMC) won the third place in the general ranking as well as a real application of a bicycle computer [21]. The experimental results show that, both the sequential and the parallel implementations, are able to verify that the user-specified properties (e.g., asserting time constraints) are met, and also indicates if the program has bugs related to the programming language (e.g., buffer or arithmetic overflow, division by zero, and pointers safety) [22]. The experiments also show that the parallel implementation presents better results, which is able to prove and falsify more properties in the verification tasks, while it requires less verification time. Last but not least, we show that we show that our k-induction method outperforms CPAChecker [19] in terms of correct results, which is a state-of-the-art k-induction-based verification tool for C programs.

The remainder of the paper is organized as follows. We first give a brief introduction to BMC and k-induction techniques that we will refer to the paper. In Section 4, the k-induction algorithm, and its parallel implementation, is described in terms of transformations on each step of the algorithm. In Section 5, we present the results of our experiments using several SV-COMP benchmarks and a real embedded system application. In Section 6, we discuss the related work. We conclude and describe future work in Section 7.

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

2 Background

This section provides the basic concepts for understanding BMC and k-induction techniques, which are described in terms of a state transition system.

2.1 State Transition System

A state transition system $M = (S, T, S_0)$ is an abstract machine that consists of a set of states S, where $S_0 \subseteq S$ represents the set of initial states, and $T \subseteq S \times S$ is the transition relation, i.e., pairs of states specifying how the system can move from state to state. A state $s \in S$ consists of the value of the program counter *pc* and the values of all program variables. An initial state s_0 assigns the initial program location. We identify each transition $T = (s_i, s_{i+1}) \in T$ between two states s_i and s_{i+1}, with a logical formula $T(s_i, s_{i+1})$ that captures the constraints on the corresponding values of the program counter *pc* and the program variables.

2.2 Bounded Model Checking (BMC)

In BMC, the program to be analyzed is modeled as a state transition system, which is extracted from the control-flow graph (CFG) [23]. This graph is built as part of a translation process from program code to single static assignment (SSA) form. A node in the CFG represents either a (non-) deterministic assignment or a conditional statement, while an edge in the CFG represents a possible change in the program's control location.

Given a transition system M, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a VC ψ such that ψ is satisfiable if and only if ϕ has a counterexample of length k or less [1]. The associated model checking problem is formulated by constructing the following logical formula:

$$\psi_k = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1}) \wedge \neg\phi(s_i), \qquad (1)$$

given that ϕ is a safety property, I is the set of initial states of M and $T(s_j, s_{j+1})$ is the transition relation of M between steps j and $j+1$. Hence, $I(s_0) \wedge \bigwedge_{j=0}^{i-1} T(s_j, s_{j+1})$ represents the executions of M of length i and (1) can be satisfied if and only if for some $i \leq k$ there exists a reachable state at step i in which ϕ is violated. If (1) is satisfiable, then the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counterexample. A counterexample for a property ϕ is a sequence of states $s_0, s_1, \cdots, s_k$ with $s_0 \in S_0$ and $T(s_i, s_{i+1})$ with $0 \leq i < k$.

If (1) is unsatisfiable, we can conclude that no error state is reachable in k steps or less. In this case, BMC techniques are not complete because there might still be a counterexample that is longer than k. Completeness can only be ensured if we know an upper bound on the depth of the state space, i.e., if we can ensure that we have already explored all the relevant behaviour of the system, and searching any deeper only exhibits states that have already been checked [24].

2.3 k-Induction

A feasible alternative to check properties in BMC is to prove that an invariant (assertion) is k-inductive [10,12]. The k-induction method has been successfully applied to verify hardware designs (represented as finite state machines) using a SAT solver, but the first attempts to apply this technique to software are only recent [8,9, 11,25,19]. In order to present the k-induction method, we use the notation of Eén and Sörensson [10], which describes the principle via temporal induction (i.e., the induction is carried out over the steps of the finite state machines). The simplest form of k-induction consists of two steps: the *base-case* and the *induction-step*. Let $I(s)$ and $T(s, s')$ encode the set of initial states and the transition relation of the finite transition system M, respectively. Let $\phi(s)$ denote states satisfying a safety property ϕ. The strengthened induction, as originally proposed in [10], is then defined by the following equation:

$$\begin{aligned} Base_k &= I(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \\ &\quad \wedge (\neg\phi(s_0) \vee \ldots \vee \neg\phi(s_k)) \\ Step_k &= \phi(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge \phi(s_k) \\ &\quad \wedge T(s_k, s_{k+1}) \wedge \neg\phi(s_{k+1}) \end{aligned} \qquad (2)$$

The intuitive interpretation of these two formulae are as follows: in the base-case, we aim to check that ϕ holds in all states reachable from an initial state within k steps (we assume that $k \geq 0$) and in the induction-step, we aim to check that whenever ϕ holds in k consecutive states $s_1, \ldots, s_k$, ϕ also holds in the next state s_{k+1} of the system. In both cases, we check whether formulae $Base_k$ and $Step_k$, as described above, are unsatisfiable. An algorithm can then be devised from these two formulae, which unwinds the system incrementally and checks whether $Base_k$ is satisfiable or $Step_k$ is unsatisfiable in order to determine termination. In particular, if $Base_k$ turns to be satisfiable in step k, then we have found a violation of the property. If $Step_k$ is unsatisfiable in step k, then the property holds.

Proving that an invariant is k-inductive with iterative deepening is a technique that was already presented by several authors [8,9,10,11,12,14,19,26]. The difference between the existing work and our present proposal is that the present proposal uses an additional forward condition to check whether loops have been fully unrolled and all states were reached within k iterations; and that the referenced implementation succeeds without manually providing program invariants.

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

Figure 3: Motivating example for the k-induction algorithm.

Similar to Donaldson et al. [8,9], we havoc only the variables that occur in the loop, i.e., all loops variables are assigned non-deterministic values. Then the loop is run $k-1$ times, where all post-loop states are assumed to be different; in the loop body, all assertions are replaced by assumptions, which ensures that the chosen values satisfy a consequence of the (unknown) loop invariant. Lastly, the loop is run one final time, before the invariant is checked for the final state. Differently from our approach, Große et al. [11] havoc all program variables, which makes it difficult to check for the reachability of an error since they do not provide enough information to constrain the havocked variables in the program.

3 Motivating Example

As a motivating example, we use a program from the SV-COMP [20] benchmarks; see Figure 3. Here, a is an integer constant that is set to 2, and variables i and sn are declared with a type larger than the type of the variable n to avoid arithmetic overflow. Mathematically, the code represents the implementation of the simple sum Equation (3):

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

We are required to show that the property represented by the assertion in line 9) holds for any value of n (i.e., for any unfolding of the program). BMC techniques find this difficult as the number of loop iterations, represented by n, is non-deterministically chosen; full unwinding would require $2^{32} - 1$ unfoldings. The intent of k-induction is to prove the property holds, without having to fully unwind the loop.

4 Induction-based Verification of C/C++ Programs

In this section, the transformations in each step of the k-induction algorithm are described using the Hoare notation [27]. From the implementation point of view, these transformations take place at the intermediate representation level, during the conversion of the C/C++ program into a GOTO-program, which simplifies the representation (e.g., replacement of *switch* and *while* by *if* and *goto* statements), and handles the unrolling of the loops and the elimination of recursive functions. For a detailed description of how these simplifications occur for C/C++ programs, we refer the reader to Cordeiro et al. [7] and Ramalho et al. [28].

4.1 The Proposed k-Induction Algorithm

Figure 4 shows an overview of the proposed k-induction algorithm. We do not add additional details about the transformations on each step of the algorithm; we keep it simple and describe the details in the next subsections so that one can have a big picture of the proposed method. The input of the algorithm is a C/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* (it does not succeed in computing an answer *true* or *false*).

In the base case, the algorithm tries to find a counterexample up to a maximum number of iterations k. In the forward condition, global correctness of the loop w.r.t. the property is shown for the case that the loop iterates at most k times; and in the inductive step, the algorithm checks that, if the property is valid in k iterations, then it must be valid for the next iterations. The algorithm runs up to a maximum number of iterations and only increases the value of k if it can not falsify the property during the base case. Note that k is incremented only at the start of the *else* branch in line 9 of Fig. 4. In our benchmarks, we noted that computational resources are wasted if we start with $k = 1$ in the forward condition and the inductive step since loops are usually unfolded at least two times.

4.1.1 Loop-free Programs

In the k-induction algorithm, the loop unwinding of the program is done incrementally from one to *max_iterations*, where the number of unwindings is measured by counting the number of *backjumps* [23]. On each step of the k-induction algorithm, an instance of the program that contains k copies of the loop body corresponds to checking a loop-free program, which uses only *if*-statements in order to prevent its execution in the case that the loop ends before k iterations.

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

```
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
```

Figure 4: An overview of the k-induction algorithm.

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

Therefore, the forward condition and the inductive step of our k-induction algorithm transform a program with loops into a loop-free program, such that correctness of the loop-free program implies correctness of the program with loops.

If the program consists of multiple and possibly nested loops, we simply set the number of loop unwindings globally, that is, for all loops in the program and apply these aforementioned translations recursively. Figure 5 shows how loop unwindings are applied to a program with nested loops. Note, however, that each case of the k-induction algorithm performs different transformations at the end of the loop, e.g., either to find bugs (base case) or to prove that enough loop unwindings have been performed (forward condition).

4.1.2 Program Translations

In terms of program translations, which are all done completely automatic by our proposed method, the base case simply inserts an unwinding assumption to the respective loop-free program P', consisting of the termination condition σ after the loop, to ensure that it finds a counterexample of depth k without reporting any false incorrect result (i.e., to avoid unfolding loops partially), as follows

$$I \wedge T \wedge \sigma \Rightarrow \phi$$

given that 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. [11], 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

$$A; while(c)\,\{S; E; U; R;\} \tag{4}$$

given that A is the code (or sequence of commands) responsible for assigning non-deterministic values to all loops variables, i.e., the state is havocked before the loop, c is the termination 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 original body of the *while* loop, U is the code to update all state variables with local values after executing E, and R is the code to remove redundant states.

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

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

The intuitive interpretation of S, U, and R is that if the current state (after executing E) is different from 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 from being included in the vector of statets. 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. In contrast, Große et al. [11] 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

$$A; B;\ while(c)\,\{S; E; D; U; R;\} \tag{5}$$

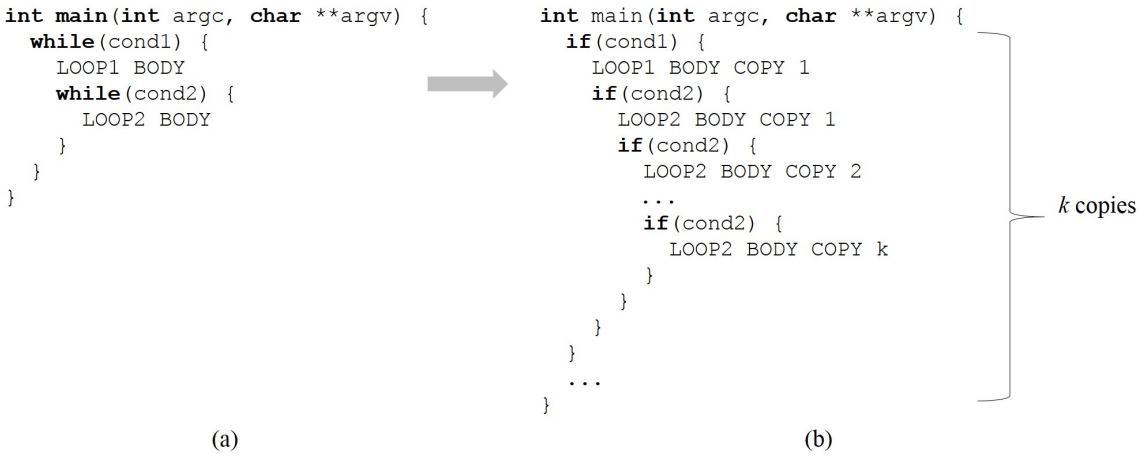

Figure 5: (a) A program with nested loops. (b) Iteration-based unwinding of the program in (a).

given that B is the initial condition of the loop, c is the termination 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 termination condition is checked. In particular, the loop

$$do\,\{E;\}\,while(c)$$

is converted into

$$A; E; while(c)\,\{S; E; U; R;\} \tag{6}$$

Finally, the expressions *if-then-else* are transformed as follows:

$$if(c)\ X\ else\ Y$$

is rewritten as

$$(\neg c \wedge Y) \vee (c \wedge X)$$

given that c is the condition of the *if* expression; the variables involved in c are initialized to non-deterministic values only whether that *if* expression is inside a given loop; X is the code to be executed if c is evaluated to *true*, and Y is the code to be executed if c is evaluated to *false*.

The inductive step is thus represented by

$$\gamma \wedge \sigma \Rightarrow \phi \tag{7}$$

given that γ is the transition relation of $\hat{P}'$, which represents a loop-free program (cf. Definition 1) after applying transformations (4) and (5). 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.

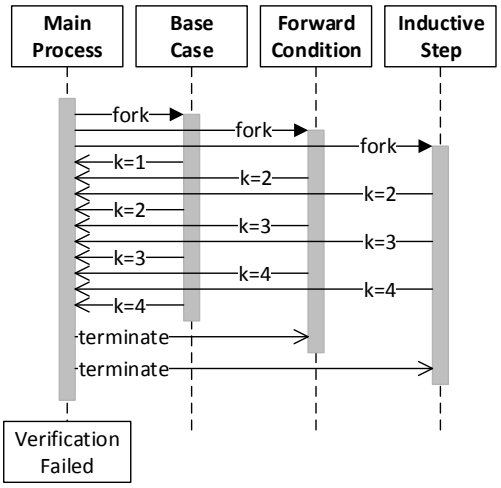

Figure 6: Base case falsified the property.

4.2 Parallel k-Induction Algorithm

After developing the k-induction algorithm (shown in Figure 4), we observed that the translations and the application of each step of the k-induction algorithm could be done completely independently. Therefore, in addition to a sequential execution of the k-induction algorithm, the execution of each step could also be carried out in parallel; specifically by splitting each step to a different processing core, which could potentially divide the verification wallclock time by a factor of three.

The approach chosen for the parallel implementation of the k-induction algorithm is the use of four different processes (running on different processing cores) instead of using threads; this choice is forced as the SMT solver currently used by ESBMC (Z3) is not thread safe. In

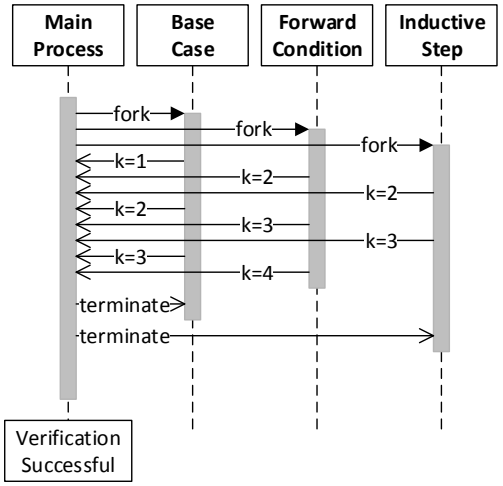

Figure 7: Forward condition proved the property.

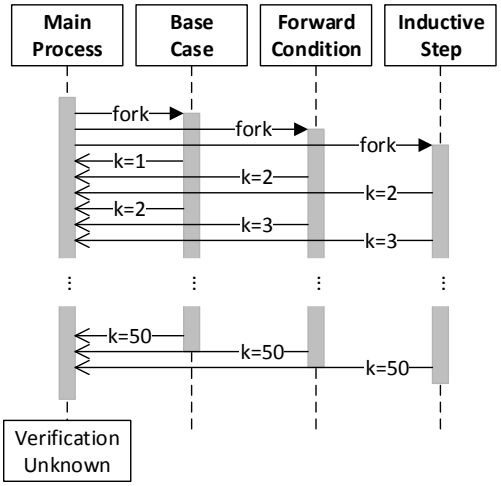

Figure 9: The algorithm could not prove or falsify the property.

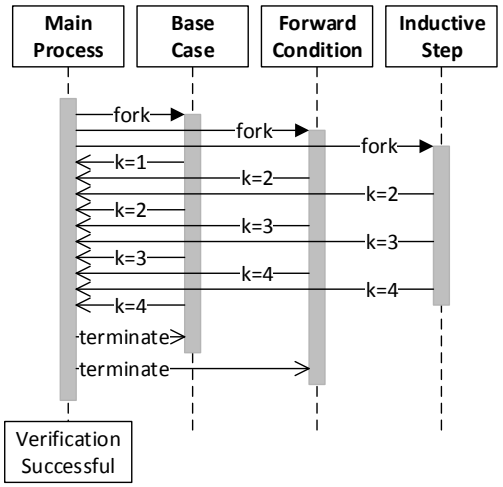

Figure 8: Inductive step proved the property.

terms of the parallel implementation of the k-induction algorithm, the parent process is responsible for initializing the three child processes, executing the logic of the k-induction algorithm and showing the final result of the verification process. Each child process is responsible for one step of the k-induction: base case, forward condition, and inductive step. Pairs of pipes are used for inter-process communication. For the inter-process communication, two pipes are used in each process [29].

Figures 6, 7, 8 and 9 show four possible scenarios during the parallel execution of the k-induction algorithm. In these, the arrows determine the direction of the message, from main process to child processes or vice versa; a filled arrow means a synchronous message while an open arrow means an asynchronous message.

Note that all child processes communicate the result of each iteration to the parent process and it, in turn, evaluates the decisions shown in Figure 4. This architecture was designed so that, if a process crashes for some reason, the other processes can still try to find bugs or prove correcness of the program, i.e., if the inductive step process crashes, the base case process can still find bugs and report the correct result.

In the first scenario, shown in Figure 6, the parent process starts three child processes and initiates the verification process. As an example, while checking the base case, a bug is found for step $k = 4$. At this point, the process of the base case communicates the result to the parent process, which sends terminate signals to the forward condition and the inductive step processes, to finalize them. At the end, the parent process shows that a bug was found in the program.

Scenarios two and three, shown in Figures 7 and 8, are similar to each other, with the difference that the former represents the successful result from the forward condition and the later, the successful result from inductive step. In such cases, the process in question informs the parent process that it has found a solution and then finishes its execution. At this point, the parent process sends terminate signals to the base case as well as to the the inductive step process, in the Figure 7, or to the forward condition process, in the Figure 8. At the end, the parent process shows which step was able to prove the validity of the safety property successfully. Note that in Figure 8, the base case process at least has to run to the same number of steps k as the successful inductive step process; otherwise, safety cannot be guaranteed.

The fourth and final scenario, shown in Figure 9, occurs when the three child processes have reached the maximum value of k and have not found a solution. In this case, each child process communicates to the parent process that it ended without finding a bug; and the parent process reports that the k-induction algorithm was unable to prove correctness.

4.3 Running Example

In this section, we explain how the k-induction algorithm (see Figure 4) can prove correctness of the C program shown in Figure 3.

4.3.1 The Base Case

In the base case, the k-induction algorithm attempts to find a counterexample up to a maximum number of iterations k. The pre- and postconditions of the loop shown in Figure 3, in SSA form, are as follows:

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

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

given that Pre and $Post$ are the pre- and postconditions to compute the sum given by Equation (3), respectively, and *nondet_uint* is a non-deterministic function, which can return any value of type *unsigned int*. In the preconditions, n_1 represents the first assignment to the variable n, which is a non-deterministic value greater than or equal to one. This ensures that the model checker explores all possible unwindings of the program. Additionally, sn_1 represents the first assignment to the variable sn and i_1 is the initial condition of the loop. In the postconditions, sn_k represents the assignment $n_1 + 1$ for the variable sn in Figure 3, which must be *true* if $i_k > n_1$. The resulting code of the base case transformations can be seen in Figure 10. 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 results.

4.3.2 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 Figure 3, 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+1. The resulting code of the forward condition transformations can be seen in Figure 11. The forward condition attempts to prove that the loop is unfolded deeply enough (by checking the loop invariant in line 11) and whether the property is valid in all states reachable within k iterations (by checking the assertion in line 11).

```
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   assume(i>n); // unwinding assumption
11   assert(sn==n*a);
12 }
```

Figure 10: Example code for the proof by mathematical induction, during base case.

```
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   assert(i>n); // check loop invariant
11   assert(sn==n*a);
12 }
```

Figure 11: Example code for the proof by mathematical induction, during forward condition.

4.3.3 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 true for $k + 1$. Several changes are performed in the original code during this step. First, a structure called *statet* is defined, containing all variables written within the loop. Then, a variable of type *statet* named *cs* (current state) is declared, which stores 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 *statet* vector of size equal to the current number of iterations of the loop is

also declared, called sv (state vector) that will store the values of all variables in each iteration of the loop.

Before starting the loop, all state variables are initialized to non-deterministic values and stored in the state vector on the first iteration of the loop so that the model checker can explore all possible states implicitly. Within the loop, after storing the current state and executing the current iteration of the loop, all state variables are updated with the current values of the current iteration and an *assume* instruction is inserted with the condition that the current state is different from the previous one, to prevent redundant states from being 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:

$$\begin{aligned} & sv_1[0] \neq cs_1 \\ & sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2 \\ & \ldots \\ & sv_1[0] \neq cs_1 \wedge sv_2[1] \neq cs_2 \wedge \ldots sv_k[i] \neq cs_k \end{aligned} \tag{8}$$

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 Equation (8) to be more efficient, leading to fewer timeouts when applied to the SV-COMP benchmarks.

Finally, after the whole execution of the unrolled loop an *assume* instruction is inserted, which is similar to that inserted in the base case. The pre- and postconditions of the loop shown in Figure 3, in SSA form, are defined as follows:

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

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

In the preconditions Pre, in addition to the initialization of the variables, the value of all variables contained in the current state cs must be assigned with non-deterministic values, where m is the number of loop variables. The postconditions do not change, as in the base case; they only contain the property that the algorithm is trying to prove. To implement the transformation in Equation (4) we implement the loop body as shown in instruction list Q, saving the *statet* vector cs before the loop body, and updating it afterwards, as follows:

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

In the instruction set Q, $sv[i-1]$ is the vector position to save the current state cs_i, E is the actual code inside the loop, and the assignments $cs_i.v_0 = v_{0i} \wedge \ldots \wedge cs_i.v_m = v_{mi}$ represent the value of the variables in iteration i being saved in the current state cs_i. The modified code for the inductive step, using the notation defined in Section 4.1, can be seen in Figure 12. Note that the *if*-statement (lines 17-24) in Figure 12 is copied k-times. As in the base case, the inductive step also inserts an *assume* instruction, which contains the exit condition. In contrast to the base case, the inductive step proves that the property, specified by the assertion, is valid for any value of n.

```
1  // variables inside the loop
2  typedef struct state {
3   uint64_t i, sn;
4   uint32_t 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 non-deterministc values
15  cs.i=nondet_uint();
16  cs.sn=nondet_uint();
17  cs.n=n;
18  if (cs.i<=cs.n) {          // c
19   sv[i-1]=cs;               // S
20   sn = sn + a;              // E
21   i = i + 1;                // E      } k copies
22   cs.i=i; cs.sn=sn; cs.n=n; //U
23   assume(sv[i-1]!=cs);      // R
24  }
25  assume(i>n); //unwinding assumption
26  assert(sn==n*a);
27 }
```

Figure 12: Example code for the proof by mathematical induction, during inductive step.

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

After the loop *while* is finished, the induction hypothesis $\{Post \wedge \neg(i \leq n)\}$ is satisfied on any number of iterations; in particular, the SMT solver can easily verify Lemma 1 and conclude that $sn == n * a$ is an inductive invariant relative to n.

5 Experimental Results

To evaluate the sequential and parallel implementation of the k-induction algorithm, we initially used the SV-

COMP 2013 benchmarks from the *loops*, *SystemC*, *Feature Checks*, and *BitVectors* directories [20]. The *loops* directory consists of 75 benchmarks. It was chosen because it has several programs that require analysis of bounded and unbounded loops. The *SystemC* directory consists of 62 programs derived from *SystemC* programs, which contain several unbounded loops [30]; they are converted into C programs by incorporating the scheduler into the C code. The *Feature Checks* directory consists of 67 programs that require the analysis of pointer aliases and function pointers for 32-bit machine model [31]. Finally, the *BitVectors* directory consists of 32 programs for which treatment of bit-operations is necessary [32]. Since the sequential implementation of the k-induction algorithm was already applied to all SV-COMP benchmarks [20,33], further evaluation of other categories only confirms the performance improvement of the parallel implementation over the sequential one.

After evaluating our k-induction algorithm (both sequential and parallel implementation), we compare it to CPAchecker (Configurable Software-Verification Platform) [19] using a set of C benchmarks from SV-COMP 2015 [34]. The benchmarks that are used in our comparison consist of 5205 C programs from SV-COMP. The benchmarks are split into 13 suites, as follows: *Arrays* contains 86 benchmarks with arrays, *BitVectors* contains 47 benchmarks with bit-operations, *Concurrency* contains 1003 multi-threaded benchmarks, *ControlFlowInteger* contains 48 benchmarks depend mostly on the control-flow structure and integer variables, *DeviceDrivers64* contains 1650 benchmarks that require the analysis of pointer aliases and function pointers (64-bit machine model), *ECA* contains 1140 benchmarks that represent event-condition-action systems, *Floats* contains 81 benchmarks to test float operations, *HeapManipulation* contains 80 benchmarks that require the analysis of data structures on the heap, pointer aliases, and function pointers, *Loops* contains 142 benchmarks with bounded and unbounded loops, *Product Lines* contains 597 benchmarks that represent 'products' and 'product simulators' that are derived using different configurations of product lines, *Recursive* contains 24 benchmarks with recursive functions, *Sequentialized* contains 261 benchmarks that were derived from SystemC programs, and *Simple* contains 46 benchmarks that, as the *ControlFlowInteger* category, depend mostly on control-flow structure and integer variables. Although SV-COMP 2015 consists of 15 categories, containing 5803 tests cases, two categories (*MemorySafety* and *Termination*) are not evaluated in this paper due to the lack of support on CPAchecker for these categories using k-induction, as reported by the authors.

Finally, we evaluate a bicycle computer program, which was chosen because it represents a real embedded system implementation and contains an unbounded loop that controls all operations modes of the system.

All experiments were conducted on a computer with Intel Core i7-2600, 3.40GHz with 24GB of RAM with Ubuntu 11.10 64-bit. For each test case, we set a time and a memory limit, of 900 seconds (15 minutes) and 16GB, respectively. We also set *max_iterations* to 100 (cf. Fig. 4). The tool used, with the implementation of the k-induction algorithm, is ESBMC v1.25.2 [33]. Although ESBMC supports incremental SMT solving, it did not work well in practice for the k-induction algorithm since the overhead of repeatedly checking the formulae satisfiability outweighs the amount of state space that is pruned by reusing previously learned clauses.

5.1 Results of the Sequential and Parallel k-Induction Algorithm

5.1.1 Results of the Category Loops

The set of programs is divided as follows: 49 benchmarks contain valid properties, i.e., ESBMC must be able to prove correctness; 50 benchmarks contain invalid properties, i.e., ESBMC must be able to falsify the property.

Figures 13 and 14 show the comparative result and the total verification time, respectively, of the sequential and parallel k-induction for the *Loops*, *Feature Checks*, and *SystemC* categories. Regarding the *Loops* category, it can be seen in Figure 13 that, using the sequential implementation, the verification has failed in 20 benchmarks, while using the parallel implementation, 7 benchmarks have failed. Most of the failures occurred due to time out since these benchmarks have many nested loops, which increase substantially the complexity of the generated VCs and hence the time required for checking their satisfiability. In particular, during the verification of the *ludcmp_bad* program using the parallel implementation, the base case is able to find the bug in iteration 7, while using the sequential implementation, the algorithm does not reach this iteration due to time out. The programs that caused ESBMC to abort, occurred in the inductive step, and in particular during the symbolic execution using the new intermediate representation recently introduced by Morse et al. [35]. These errors did not affect the verification of the parallel execution, because only the inductive step process aborts, and the base case and forward condition processes continue the verification process, which are later falsified by the base case or proved by the forward condition. From a total of 99 benchmarks, in the sequential implementation, the base case has found the solution in 44 benchmarks, while the forward condition has found the solution in 10 benchmarks, and the inductive step has found the solution in 25 benchmarks. In the parallel implementation, the base case has found the solution in 49 benchmarks, while the forward condition has found the solution in 13 benchmarks, and the inductive step has found the solution in 30 benchmarks.

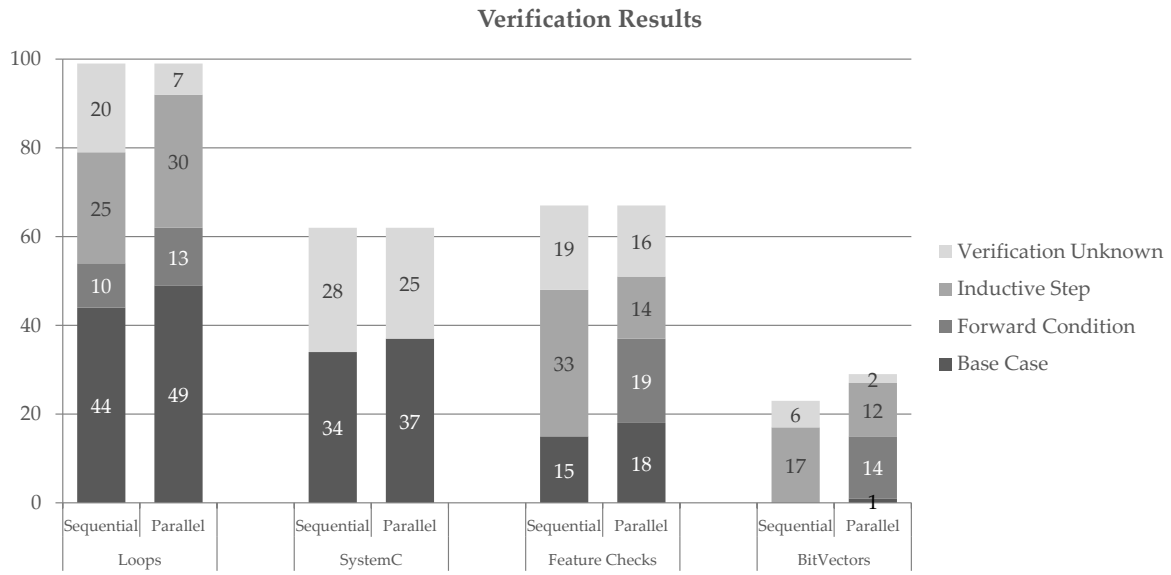

Figure 13: Comparative results of the verification for *Loops*, *SystemC*, *Feature Checks*, and *BitVectors* categories, using both sequential and parallel k-induction.

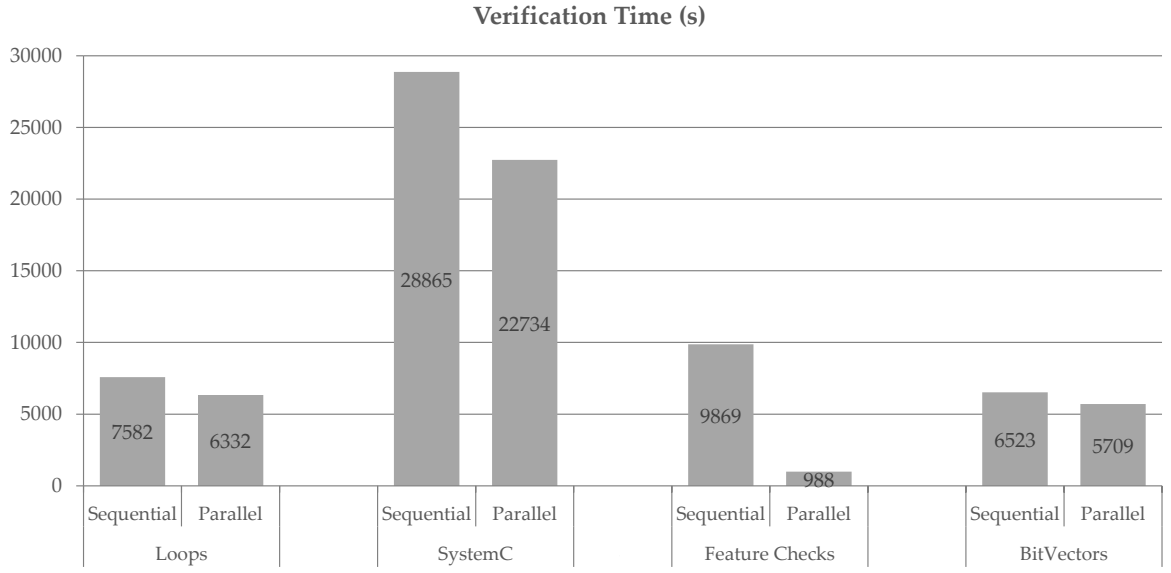

Figure 14: Comparative results of the verification time for *Loops*, *SystemC*, *Feature Checks*, and *BitVectors* categories, using both sequential and parallel k-induction.

In some benchmarks, e.g., *string, verisec_sendmail_ok* and *vowel*, each implementation of the algorithm has found the solution in a different stage of the k-induction algorithm. In these cases, using the sequential implementation, the proof occurs at a certain depth k during the inductive step. In the parallel implementation, the proof occurs at a depth $k + 1$ in the forward condition. As the process of checking the forward condition is usually faster than the inductive step, the parallel implementation finds the solution in a different step from that by the sequential implementation.

As shown in Figure 14, in the sequential implementation, all benchmarks were verified in 7582 seconds, which correctly verifies 79 out of 99 (79%)[2] while in the parallel implementation, all benchmarks were verified in 6332

[2] The benchmarks were verified using the command-line: `esbmc file.c --k-induction --k-step 100 --memlimit 15g --timeout 900s`

seconds, which correctly verifies 92 out of 99 (93%)[3]. Note that the parallel implementation is able to solve more verification tasks in less time than the sequential implementation.

5.1.2 Results of the *SystemC* Category

The *SystemC* category contains 25 benchmarks with valid properties and 37 benchmarks with invalid properties. As can be seen in Figure 13 that the sequential implementation is able to find bugs in 34 benchmarks, while the parallel implementation is able to find bugs in all (37) benchmarks that are declared as unsafe. The sequential and parallel implementation of the k-induction algorithm are not able to prove the correctness of any test case in that category, failing in 28 and 25 benchmarks, respectively; this happens mainly because of several unbounded loops present in the safe programs that do not produce redundant states; this makes it hard for the inductive step to prove correctness. As shown in Figure 14, the sequential implementation verified all benchmarks in 28865 seconds, which correctly verified 34 out of 62 benchmarks (54%), while the parallel implementation verified in 22734 seconds, which correctly verified 37 out of 62 benchmarks (59%).

5.1.3 Results of the *Feature Checks* Category

The *Feature Checks* category contains 51 benchmarks with valid properties and 16 benchmarks with invalid properties. As can be seen in Figure 13 that the sequential implementation failed in 19 benchmarks, while the parallel implementation failed in 16 benchmarks. Using the sequential and parallel implementations, the base case step has found bugs in 15 and 18 benchmarks, respectively; while the inductive step proved correctness in 33 and 14 benchmarks, respectively. The forward condition proved correctness in 19 benchmarks using the parallel implementation, but it has not proved correctness of any test case using the sequential implementation. Finally, as shown in Figure 14, the sequential implementation verified all benchmarks in 9869 seconds, which correctly verified 46 out of 67 benchmarks (68%), while the parallel implementation verified in 988 seconds, which correctly verified 49 out of 67 benchmarks (73%). In this category, the base case verified incorrectly two benchmarks that, in principle, fulfills the specification (i.e., it produced a false alarm). However, these false alarms do not happen because of the k-induction algorithm; they happen due to the memory model adopted by ESBMC [35], which is also present when verifying those two benchmarks using plain BMC.

[3] The benchmarks were verified using the command-line: `esbmc file.c --k-induction-parallel --k-step 100 --memlimit 15g --timeout 900s`

5.1.4 Results of the *BitVectors* Category

The *BitVectors* category contains 28 benchmarks with valid properties and 4 benchmarks with invalid properties. As can be seen in Figure 13, the sequential implementation was able to prove 17 correct results using the inductive step, while the parallel implementation produced 27 correct results, where it solves 1 benchmark using the base case, 14 using the forward condition, and 12 using the inductive step. The sequential implementation could not find a solution for 6 benchmarks (reporting verification unknown), while in the parallel implementation this number decreased for 2 benchmarks. In this category, both implementations incorrectly verified 2 benchmarks that, in principle, fulfills the specification (i.e., it produced a false alarm). As in the *Feature Checks* category, these false alarms do not happen because of the k-induction algorithm; they happen due to the memory model adopted by ESBMC [35].

As shown in Figure 14, to verify all benchmarks of this category, the sequential implementation spent 6523 seconds and the parallel implementation spent 5709 seconds; it decreased 12% of the total verification time. An important result is the number of timeouts: the sequential implementation had 7 timeouts, while the parallel implementation had only 1 timeout.

5.2 Comparison to CPAchecker

This subsection describes the evaluation of ESBMC (sequential and parallel k-induction) against CPAchecker. CBMC is another tool that provides a k-induction algorithm to prove correctness [9] but the comparison of CPAchecker and CBMC was already undertaken by Beyer et al. [19] and will not be covered by this paper. We invoked the tools using three scripts: one for ESBMC's sequential k-induction[4], one for ESBMC's parallel k-induction[5] and one for CPAchecker's k-induction, using Continuously-Refined Invariants[6].

Table 1 summarizes the results. Here, *Correct results* is the number of correct positive and negative results (i.e., the tool reports SAFE or UNSAFE correctly), *Wrong proofs* is the number of false positive results (i.e., the tool reports SAFE incorrectly), *Wrong alarms* is the number of false negative results (i.e., the tool reports UNSAFE incorrectly), *Timeout/Unknown* represents the number of time-outs (i.e., the tool was aborted

[4] `esbmc --k-induction --k-step 100 --z3 --no-unwinding-assertions --timeout 15m --memlimit 15g --64 -DLDV_ERROR=ERROR -Dassert=notassert -D_Bool=int --no-assertions --error-label ERROR`

[5] `esbmc --k-induction-parallel --k-step 100 --z3 --no-unwinding-assertions --timeout 15m --memlimit 15g --64 -DLDV_ERROR=ERROR -Dassert=notassert -D_Bool=int --no-assertions --error-label ERROR`

[6] `cpa.sh -bmc-induction -setprop cfa.useMultiEdges=true -setprop bmc.addInvariantsByInduction=false -spec PropertyERROR.prp`

after 900 seconds) or when the tool could not provide a proof or counterexample, *Memory out* represents the number of memory outs (i.e., the tool was aborted after the allocation of more than 15 GB), *Fail* is the number of internal errors during the verification of each benchmark and *Time* is the total verification time of all 5205 benchmarks.

Both ESBMC's sequential and parallel k-induction implementations are able correctly verify more benchmarks than CPAchecker, with the ESBMC's parallel k-induction providing 82 more correct results than ESBMC's sequential k-induction, and 804 more correct results than CPAchecker. However, ESBMC's sequential k-induction is able to provide fewer wrong alarms, 16 fewer than k-induction and 2 fewer than CPAchecker, and CPAchecker is able to provide fewer wrong proofs, 85 fewer than ESBMC's sequential k-induction and 239 fewer then ESBMC's parallel k-induction. Most of the incorrect results provided by ESBMC are related to bugs in the coding of the inductive step transformations, because we do not support *continue* and *return* (from inside a loop) statements. CPAchecker provides fewer timeouts/unknown results than the sequential k-induction implementation of ESBMC (252 fewer timeouts/unknown results) and ESBMC's parallel k-induction (737 fewer timeouts/unknown results).

ESBMC's parallel k-induction is able to verify all benchmarks without reporting any memory out or abort. This is due to the multiprocess approach; as the main process is responsible for reporting the result of the verification, if the other three processes are killed due to the allocation of more than the memory limit, or abort for some reason, the main process presents an unknown result. This behaviour also explains the increased number of correct results (as, if a process ends without providing a result, the other processes do not stop the verification), incorrect results (due to bugs mentioned earlier), and the number of timeout/unknown results. The parallel algorithm converts memory outs and aborts into successful, unsuccessful or unknown results.

Overall, ESBMC's sequential k-induction completed all categories in 1224634 seconds (approximately 14 days and 4 hours), the parallel k-induction completed all categories in 1384773 seconds (approximately 16 days) and CPAchecker's k-induction implementation completed all categories in 1041696 seconds (approximately 12 days). ESBMC's sequential k-induction correctly verifies 3071 benchmarks and presents 111 wrong proofs and 32 wrong alarms, while the parallel k-induction correctly verifies 3153 benchmarks and presents 265 wrong proofs and 48 wrong alarms, and CPAchecker correctly verifies 2349 benchmarks and presents 26 wrong proofs and 34 wrong alarms.

5.3 Verification of a Bicycle Computer

A proof of concept is also developed using the k-induction algorithm, in a bicycle computer, which uses the embedded platform Raspberry Pi [36] with the processor ARM1176JZF-S [37]. Note that this system is chosen since it contains an unbounded loop in the control software, which consists of a data acquisition stage, the application of an algorithm, and the output of a result. In particular, the system consists of a screen and two buttons, one for the circular modes of the computer operation and another button to restart the current mode of the computer. The operation modes of the computer include: travel (prints the current mileage of the trip), speed (shows the current speed), total (shows the total mileage traveled), and time (shows the total trip time). The bicycle computer control software contains approximately 125 lines of C code.

For each operation mode of this system, there are time constraints between the time of processing the information and on-screen display. In the travel mode, mileage should be shown every 200ms with a tolerance of 100ms. In the speed mode, the actual speed should be shown every 100ms. In total mode, the total mileage must be shown every 500ms and in time mode, the travel time should be displayed every second.

To verify the time constraints of the program, it is necessary to insert assertions into the program over the processing time of each travel mode. For example, if the computer is in speed mode, the scan time of the sensor and the speed calculation should be less than 100ms, which is the time to update the screen in this particular mode. To identify the processing time information of each mode, the code is converted into assembly instructions and, from the number of instructions, we calculate the CPU time. The CPU time is calculated as follows [38]:

$$T.\,de\,CPU = N_{ins} \times CPI \times T_c \tag{9}$$

given that N_{ins} is the generated number of instructions, CPI is the average number of processor cycles per instruction and the T_C is the cycle time of the processor (the inverse of the processor clock). In the experiments, values of CPI and T_C are obtained from the technical manual of the processor ARM1176JZF-S, presented in the platform Raspberry Pi used by the system. The values are as follows: $CPI = 1.1$ and $T_C = 1.42857 \times 10^{-9}$ sec (i.e., inverse of 700MHz).

The modified program with statements about the temporal properties is shown in Figure 15, where Tp is the processing time, Tb is the time in which the button is pressed (changing the system's mode), and Rt represents the temporal constraint of the current mode. The *current_time()* returns the timer register value (i.e., the number of processor clock ticks) multiplied by the cycle time of the processor; this measures the elapsed time.

Tool results	ESBMC Sequential	ESBMC Parallel	CPAchecker cont. refined
Correct results	3071	**3153**	2349
Wrong proofs	111	265	**26**
Wrong alarms	**32**	48	34
Timeout/Unknown	1254	1739	**1002**
Memory out	383	**0**	67
Fail	348	**0**	1727
Total time (days)	14	16	**12**

Table 1: Results of k-induction tools.

```
1 ...
2 Tb=current_time();
3 /* Processing and display information */
4 Tp=current_time();
5 assert(Tp - Tb < Rt);
6 ...
```

Figure 15: Modified program with assertions.

The *assert* statement then checks whether the elapsed time is less than the constraint for the current mode.

The k-induction algorithm is able to successfully verify all properties in less than one second and no bug has been found[7]. It was also noted that the processing time and the time to display the information on-screen is extremely small in relation to the time constraints. For example, in the travel mode, only 55 instructions are necessary to calculate the speed and to show the results on the display. The calculated CPU time for this particular case is:

$$\begin{aligned} T.\,de\,CPU &= 55 \times 1,1 \times 1,42857 \times 10^{-9} \\ &= 86ns \end{aligned} \tag{10}$$

This means that the CPU speed of the system, which determines how many instructions it can perform in one second of time, is faster than what is needed for the implementation of the bicycle computer.

6 Related Work

The application of the k-induction method is gaining popularity in the software verification community. Recently, Bradley et al. introduced "property-based reachability" (or IC3) procedure for the safety verification of systems [39,40]. The authors have shown that IC3 can scale on certain benchmarks where k-induction fails to succeed. We do not compare k-induction with IC3 as this was performed by Bradley [39]; we focus our comparison on related k-induction procedures.

Other work has explored some proofs by mathematical induction of hardware and software systems with some limitations, e.g., requiring changes in the code to introduce loop invariants [8,9,11]. This complicates the automation of the verification process, unless other methods are used in combination to automatically derrive the loop invariant [41,42,43,44,45]. Similar to the approach proposed by Hagen and Tinelli [26], our method is completely automatic and does not require the user to provide loops invariants as the final assertions after each loop. On the other hand, state-of-the-art BMC tools have been widely used, but as bug-finding tools since they typically analyze bounded program runs [3, 4]; completeness can only be ensured if the BMC tools know an upper bound on the depth of the state space, which is not generally the case. This paper closes the gap between these literatures, 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 for proving properties of TLM designs (Transaction Level Modelling) in SystemC [25,11]. The approach consists of three steps; starting with the transformation of a SystemC program into a C program, followed by the generation and addition of logics to monitor TLM properties (using assertions and finite state machines), and the verification of the C program using the CBMC tool [3]. In CBMC, the proof of the properties is done by mathematical induction, using the k-induction algorithm implemented by the authors. A producer-consumer program is tested, however, with various restrictions. The tool is not able to finalize the verification in the given time with the increase of the number of producers and consumers; additionally, the technique can only prove correctness for a limited number of iterations. The k-induction algorithm, implemented by the authors, is similar to the one described in this paper, which uses three steps: base case, forward condition, and inductive step. The difference lies on the transformations carried out in the forward condition. In the k-induction algorithm, proposed by Große et al., during the forward condition, transformations sim-

[7] The benchmarks are verified using the command-line: esbmc file.c −−k-induction −−k-step 10

ilar 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. Although the tool is targeted at SystemC programs, it needs to convert the SystemC program into a C program instead of directly verifying in the target language; this introduces additional delays before starting the verification process; in our k-induction algorithm, this an additional step does not occur.

Sheeran et al. describe a tool called Lucifer to verify hardware designs based on mathematical induction [12]. The proposed algorithm consists of two steps, the base case and the inductive step. Several forms of the algorithm are presented, in a incremental way, always presenting improvements over the previous version. The problem is modeled as a graph; the base case checks whether a property holds in the first i states, and the inductive step then checks the disjunction of two expressions, the first is the negation of the satisfiability of the shortest path between the initial states and a $i+1$ state, and the second is the negation of the satisfiability of the shortest path between the initial state and the negation of the property. There are several differences between the algorithm proposed by Sheeran et al. to our k-induction algorithm. The first is that our algorithm has an extra step before the inductive step (i.e., the forward condition) to check whether the loops are sufficiently unrolled. The second difference is that, in their algorithm, no transformation is made during the inductive step; and the inductive step only checks whether the next state is reachable, whereas our algorithm checks that, if the property holds for the current state, it should hold for the next unwindings of the program. Moreover, since at that time, sophisticated SMT solvers built over efficient SAT solvers were just about to be born, the authors did not exploit the use of SMT solvers to check the satisfiability of the generated VCs, as done by the ESBMC tool [7,17]. Since Lucifer is a hardware verification tool, the authors also do not handle directly programs written in the C/C++ programing languages.

Similar to Sheeran et al., Hagen and Tinelli describe a k-induction algorithm for Lustre programs that does not require the user to provide loops invariants [26,41]. Kahsai and Tinelli also propose a parallel k-induction based model checker for Lustre programs [14]. Differently from Sheeran et al., Hagen and Tinelli approach automatically translates Lustre programs into a first-order logic (FOL) specification language. In particular, Hagen and Tinelli exploit decidable fragments of FOL supported by typical SMT solvers: uninterpreted functions, linear arithmetic, arrays, tuples, and records. However, the authors do not exploit bit-vector arithmetic in their k-induction algorithm, which limits its analysis. If the program variables are modeled as bit-vectors of a fixed size, then the result of the analysis can be precise (w.r.t. the program semantics) depending on the size considered for the bit-vectors [7]. In contrast, if the program variables are modeled using the abstract numerical domains, as proposed by Hagen and Tinelli, then the result of the analysis is independent from the actual binary representation, but it may not be precise when arithmetic expressions are involved.

Eén et al. propose a verification method based on SAT solvers to check properties, using temporal induction, in hardware designs modeled as finite state machines [10]. This verification method is implemented in the TIP tool and some benchmarks, written in the SMV (symbolic model verification) language [46], are verified. Simplifications are implemented to the algorithm to prune the state space explored during the proof by induction, which considerably reduced the verification time. Differently from our work, the induction algorithm uses only two steps for the proof, i.e., the base case and the inductive step; the latter includes the forward condition, but not as an (independent) step. Additionally, the inductive step does not perform any transformation in the original program and the proof condition of the algorithm is simply a conjunction of the negation of the inductive step result and the forward condition result. The main conclusion of this paper is that the iterative and deepening verification, starting from a low iterations limit, is usually more effective than using a random iterations limit. However, Eén et al. do not apply this approach to the verification of software systems and they focus only on hardware designs written in the SMV language.

Donaldson et al. describe a verification tool called Scratch [8] to detect data races during Direct Memory Access (DMA) in the CELL BE processor from IBM [8]. The approach used to verify C programs is the k-induction technique. The tool inserts assertions into the program to model the behavior of the memory control-flow, and it attempts to prove the correctness of the program, using the k-induction algorithm. The k-induction algorithm implemented in the Scratch tool 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. Differently from the algorithm proposed in this paper, Donaldson et al. proposed the k-induction with two steps, the base case and the inductive step. 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. The tool supports C programs, which is used to program the CELL processor. In this paper, in addition to check C/C++ programs, a more general class of problems is evaluated, using the (traditional) SV-COMP benchmarks.

In another related work, Donaldson et al. describe two tools for proving correctness of programs: K-Boogie and K-Inductor [9]. The K-Boogie is an extension to the model checker of the Boogie language and allows the

proof of correctness, using the k-induction method, of several programming languages, including Boogie, Spec, Dafny, Chalice, VCC, and Havoc. K-Inductor is a model checker for C programs, which is built on top of the CBMC tool [3]. The authors use an algorithm called combined-case k-induction, which uses the base case and the inductive step, but optimizations are inserted into the code to remove nested loops, turning them into programs with only one loop. K-Inductor is used to verify the same benchmarks shown in a previous work of the authors, in which the tool Scratch is presented [8]. The K-inductor has shown similar results in terms of correct verification of programs, but with gains in verification time. The difference between the work of Donaldson et al. and the work proposed in this paper, appears when the algorithm of mathematical induction is analyzed. Their k-induction algorithm uses two steps, the base case and the inductive step. The absence of the forward condition in the algorithm proposed by the authors does not interfere in the verification of the cases described in the paper. However, the benchmarks are manually changed, in order to model data races, which thus complicates the automation of the verification process.

Kahsai et al. describe PKIND, a parallel version of the tool KIND [41], used to verify invariant properties of programs written in Lustre [14]. PKIND makes use of a multi-process approach, similar to ESBMC, with the difference that the communication between process is message-based for PKIND (using MPI API [47]), and pipe-based for ESBMC [29]. In order to verify a Lustre program, PKIND starts three processes, one for base case, one for inductive step and one for invariant generation [41], note that unlike ESBMC, the k-induction algorithm used by PKIND does not have a forward condition step. The base case starts the verification with $k = 0$, and increments its value until it finds a counterexample or it receives a message from the inductive step process that a solution was found. Similarly, the inductive step verifies the program with increasing values of k until it receives a message from the base case process or a solution was found. The invariant generation process generates a set of candidate invariant from predefined templates and constantly feeds the inductive step process with them, as done recently by Beyer et al. [19], which combines k-induction with continuously-refined invariants. It is important to note that when the inductive step finds a solution for an arbitrary value of k, it sends a message to the base case process that checks if the properties holds up to that value, before showing the result.

7 Conclusions

This paper presented a new algorithm, called k-induction, to prove correctness by mathematical induction of C programs. This algorithm has been implemented in the ESBMC tool, using a sequential and a parallel implementation. The main contributions of this work are the design, implementation, and evaluation of the k-induction algorithm in a verification tool as well as the use of the technique for the automated verification of reachability and time constraints properties in heteregenous programs. To the best of our knowledge, this paper marks the first application of the k-induction algorithm to a broader range of C programs if compared to existing implementations of k-induction-based verification.

To validate the k-induction algorithm, experiments were performed involving several benchmarks, of which the sequential k-induction was able to successfully verify 70% of the benchmarks in 52839 seconds, and the parallel k-induction was able to correctly verify 80%, in 35763 seconds, which gives a speedup of roughly 32% faster than its sequential version. Given a fixed timeout, this speedup can also improve the quality of the results, because more programs can be verified if their verification would otherwise be interrupted by the time limit.

Additionally, we compare the sequential and parallel implementation of the k-induction algorithm against the CPAChecker tool using the SV-COMP 2015 benchmarks. In summary, ESBMC's sequential k-induction was able to successfully verify 59% in 1224634 seconds, ESBMC's parallel k-induction was able to correctly verify 60%, in 1384773 seconds, and CPAchecker was able to successfully verify 45% of the benchmarks in 1041696 seconds which, although faster, gives less coverage than ESBMC's sequential or parallel k-induction by 14% and 15%, respectively.

Last but not least, our k-induction algorithm was able to prove and falsify safety properties and temporal constraints in a bicycle computer. In the two proposed approaches, the parallel implementation of the k-induction produced better results, with the largest number of benchmarks being successfully verified in less time.

For future work, we intend to investigate whether redundant computations (or constraints) in the k-induction algorithm can be avoided, possibly by using the results of already completed steps using the Green solver interface [48].

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications., IOS Press (2009)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: 26. In: Satisfiability Modulo Theories. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (February 2009) 825–885
3. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of

Lecture Notes in Computer Science., Springer Berlin Heidelberg (2004) 168–176
4. Merz, F., Falke, S., Sinz, C.: 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'12, Springer-Verlag (2012) 146–161
5. Cordeiro, L.: SMT-Based Bounded Model Checking of Multi-threaded Software in Embedded Systems. University of Southampton, Southampton, UK (2011)
6. Ivanicic, F., Shlyakhter, I., Gupta, A., Ganai, M.K.: Model checking c programs using F-Soft. In: Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on. (Oct 2005) 297–308
7. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. **38**(4) (2012) 957–974
8. Donaldson, A.F., Kroening, D., Rümmer, P.: SCRATCH: A tool for automatic analysis of dma races. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming. PPoPP '11, ACM (2011) 311–312
9. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: Proceedings of the 18th International Conference on Static Analysis. SAS'11, Springer-Verlag (2011) 351–368
10. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4) (2003) 543 – 560
11. Große, D., Le, H.M., Drechsler, R.: Induction-based formal verification of SystemC TLM designs. In: Microprocessor Test and Verification (MTV), 2009 10th International Workshop on. (Dec 2009) 101–106
12. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design. FMCAD '00, Springer-Verlag (2000) 108–125
13. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Transaction of Software Engineering **37**(6) (November 2011) 845–857
14. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Proceedings 10th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011, Snowbird, Utah, USA, July 14, 2011. (2011) 55–62
15. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. TACAS'08/ETAPS'08, Springer-Verlag (2008) 337–340
16. Brummayer, R., Biere, A.: 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 2009,. TACAS '09, Springer-Verlag (2009) 174–177
17. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proceedings of the 33rd International Conference on Software Engineering. ICSE '11, ACM (2011) 331–340
18. Cordeiro, L., Fischer, B., Marques-Silva, J.: Continuous Verification of Large Embedded Software Using SMT-Based Bounded Model Checking. In: Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. ECBS '10, Washington, DC, USA, IEEE Computer Society (2010) 160–169
19. Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. CoRR **abs/1502.00096** (2015)
20. Beyer, D.: Second competition on software verification. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7795 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 594–609
21. Morse, J., Cordeiro, L.C., Nicole, D., Fischer, B.: Model checking LTL properties over ANSI-C programs with bounded traces. Software and System Modeling **14**(1) (2015) 65–81
22. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in c. In: Proceedings of the 25th International Conference on Software Engineering. ICSE '03, Washington, DC, USA, IEEE Computer Society (2003) 385–395
23. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1997)
24. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: CAV. Volume 6806 of Lecture Notes in Computer Science. (2011) 557–572
25. Große, D., Le, H.M., Drechsler, R.: A semantics-based translation method for automated verification of SystemC TLM designs. Volume 29., Norwell, MA, USA, Kluwer Academic Publishers (October 2013) 685–695
26. Hagen, G., Tinelli, C.: Scaling up the formal verification of lustre programs with smt-based techniques. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design. FMCAD '08, Piscataway, NJ, USA, IEEE Press (2008) 15:1–15:9
27. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (October 1969) 576–580
28. Ramalho, M., Lopes, M., Sousa, F., Marques, H., Cordeiro, L., Fischer, B.: SMT-Based Bounded Model Checking of C++ Programs. In: Proceedings of ECBS 13. (2013) 147–156
29. Mitchell, M., Samuel, A.: Advanced Linux Programming. New Riders Publishing, Thousand Oaks, CA, USA (2001)
30. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. Volume 7610 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 1–6
31. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying systemc: A software model checking approach. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. FMCAD '10, Austin, TX, FMCAD Inc (2010) 51–60

32. Franz, A.: Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT. PhD thesis, University of Trento (2010)
33. Morse, J., Cordeiro, L.C., Nicole, D., Fischer, B.: Handling unbounded loops with ESBMC 1.20. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 7795 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 619–622
34. Beyer, D.: Software verification and verifiable witnesses. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 9035 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2015) 401–416
35. Morse, J., Ramalho, M., Cordeiro, L.C., Nicole, D., Fischer, B.: ESBMC 1.22. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 8413 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2014) 405–407
36. Kiepert, J.: Creating a raspberry pi-based beowuf cluster. Boise State University (2013) 1–17
37. ARM: Arm1176jzf-s technical reference manual. (2009)
38. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Revised 4th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
39. Bradley, A.: IC3 and beyond: Incremental, inductive verification. In: Computer Aided Verification. Volume 7358 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2012) 4–4
40. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013. (2013) 157–164
41. Kahsai, T., Ge, Y., Tinelli, C.: Instantiation-based invariant discovery. In: Proceedings of the Third International Conference on NASA Formal Methods. NFM'11, Springer-Verlag (2011) 192–206
42. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV'11, Berlin, Heidelberg, Springer-Verlag (2011) 703–719
43. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electron. Notes Theor. Comput. Sci. **267**(1) (October 2010) 3–16
44. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Nonlinear loop invariant generation using gröbner bases. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '04, New York, NY, USA, ACM (2004) 318–329
45. Hoder, K., Kovács, L., Voronkov, A.: Interpolation and symbol elimination in vampire. In: Proceedings of the 5th International Conference on Automated Reasoning. Volume 6173 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer-Verlag (2010) 188–195
46. Yang, J., Mok, A.K., Wang, F.: Symbolic model checking for event-driven real-time systems. ACM Trans. Program. Lang. Syst. **19**(2) (1997) 386–412
47. Pacheco, P.S.: Parallel Programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
48. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: Reducing, reusing and recycling constraints in program analysis. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. FSE '12, New York, NY, USA, ACM (2012) 58:1–58:11