Boosting k-Induction with Continuously-Refined Invariants

Dirk Beyer, Matthias Dangl, and Philipp Wendler

University of Passau, Germany

Abstract. *k*-induction is a promising technique to extend bounded model checking from falsification to verification. In software verification, *k*-induction works only if auxiliary invariants are used to strengthen the induction hypothesis. The problem that we address is to generate such invariants (1) automatically without user-interaction, (2) efficiently such that little verification time is spent on the invariant generation, and (3) that are sufficiently strong for a *k*-induction proof. We boost the *k*-induction approach to significantly increase effectiveness and efficiency in the following way: We start in parallel to *k*-induction a data-flow-based invariant generator that supports dynamic precision adjustment and refine the precision of the invariant generator continuously during the analysis, such that the invariants become increasingly stronger. The *k*-induction engine is extended such that the invariants from the invariant generator are injected in each iteration to strengthen the hypothesis. The new method solves the above-mentioned problem because it (1) automatically chooses an invariant by step-wise refinement, (2) starts always with a lightweight invariant generation that is computationally inexpensive, and (3) refines the invariant precision more and more to inject stronger and stronger invariants into the induction system. We present and evaluate an implementation of our approach, as well as all other existing approaches, in the open-source verification-framework CPACHECKER. Our experiments show that combining *k*-induction with continuously-refined invariants significantly increases effectiveness and efficiency, and outperforms all existing implementations of *k*-induction-based verification of C programs in terms of successful results.

1 Introduction

Advances in software verification in recent years have lead to increased efforts towards applying formal verification methods to industrial software, in particular operating-systems code [3, 4, 34]. One model-checking technique that is implemented by half of the verifiers that participated in the 2015 Competition on Software Verification [7] is bounded model checking (BMC) [16, 17, 22]. For unbounded systems, BMC can be used only for falsification, not for verification [15]. This limitation to falsification can be overcome by combining BMC with mathematical induction and thus extending it to verification [26]. Unfortunately, inductive approaches are not always powerful enough to prove the required verification conditions, because not all program invariants are inductive [2]. Using the more general *k*-induction [38] instead of standard induction is more powerful [37] and has already been implemented in the DMA-race analysis

A preliminary version of this article appeared as technical report [8].

tool SCRATCH [27] and in the software verifier ESBMC [35]. Nevertheless, additional supportive measures are often required to guide k-induction and take advantage of its full potential [25]. Our goal is to provide a powerful and competitive approach for reliable, general-purpose software verification based on BMC and k-induction, implemented in a state-of-the-art software-verification framework.

Our contribution is a new combination of k-induction-based model checking with automatically-generated continuously-refined invariants that are used to strengthen the induction hypothesis, which increases the effectiveness and efficiency of the approach. BMC and k-induction are combined in an algorithm that iteratively increments the induction parameter k (iterative deepening). The invariant generation runs in parallel to the k-induction proof construction, starting with relatively weak (but inexpensive to compute) invariants, and increasing the strength of the invariants over time as long as the analysis continues. The k-induction-based proof construction adopts the currently known set of invariants in every new proof attempt. This approach can verify easy problems quickly (with a small initial k and weak invariants), and is able to verify complex problems by increasing the effort (by incrementing k and searching for stronger invariants). Thus, it is both efficient and effective. In contrast to previous work [35], the new approach is sound. We implemented our approach as part of the open-source software-verification framework CPACHECKER [12], and we perform an extensive experimental comparison of our implementation against the two existing tools that use k-induction and against other common software-verification approaches.

Contributions. We make the following contributions:

- a novel approach for providing *continuously-refined invariants* from data-flow analysis with precision adjustment in order to repeatedly inject invariants to k-induction,
- an *effective and efficient tool* implementation of a framework for software verification with k-induction that allows to express all existing approaches to k-induction in a *uniform, module-based, configurable architecture*, and
- an extensive *experimental evaluation* of (a) all approaches and their implementations in the framework, (b) the two existing k-induction tools CBMC and ESBMC, and (c) the two different approaches predicate analysis and value analysis; the result being that the new technique outperforms all existing k-induction-based approaches to software verification.

Availability of Data and Tools. Our experiments are based on benchmark verification tasks from the 2015 Competition on Software Verification. All benchmarks, tools, and results of our evaluation are available on a supplementary web page [1].

Example. We illustrate the problem of k-induction that we address, and the strength of our approach, on two example programs. Both programs encode an automaton, which is typical, e.g., for software that implements a communication protocol. The automaton has a finite set of states, which is encoded by variable `s`, and two data variables `x1` and `x2`. There are some state-dependent calculations (lines 6 and 7 in both programs) that alternatingly increment `x1` and `x2`, and a calculation of the next state (lines 9

[1] `http://www.sosy-lab.org/~dbeyer/cpa-k-induction/`
(successfully evaluated by the CAV 2015 Artifact Evaluation Committee)

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;

    if ((s == 1) && (x1 != x2)) {
      // Valid safety property
      ERROR: return 1;
    }
  }
}
```

Fig. 1: Safe example program `example-safe`, which cannot be proven with existing k-induction-based approaches

```
int main() {
  unsigned int x1 = 0, x2 = 0;
  int s = 1;

  while (nondet()) {
    if (s == 1) x1++;
    else if (s == 2) x2++;

    s++;
    if (s == 5) s = 1;
  }

  if (s >= 4) {
    // Violation: s may be 4
    ERROR: return 1;
  }
}
```

Fig. 2: Unsafe example program `example-unsafe`, where some approaches may produce a wrong proof

and 10 in both programs). The state variable cycles through the range from 1 to 4. These calculations are done in a loop with a non-deterministic number of iterations. Both programs also contain a safety property (the label `ERROR` should not be reachable). The program `example-safe` in Fig. 1 checks that in every fourth state, the values of `x1` and `x2` are equal; it satisfies the property. The program `example-unsafe` in Fig. 2 checks that when the loop exits, the value of state variable `s` is not greater or equal to 4; it violates the property.

First, note that the program `example-safe` is difficult or impossible to prove with many classical software-verification approaches other than k-induction: (1) BMC cannot prove safety for this program because the loop may run arbitrarily long. (2) Explicit-state model checking fails because of the huge state space (`x1` and `x2` can get arbitrarily large). (3) Predicate analysis with counterexample-guided abstraction refinement (CEGAR) and interpolation is able to prove safety, but only if the predicate $x1 = x2$ gets discovered. If the interpolants contain instead only predicates such as $x1 = 1$, $x2 = 1$, $x1 = 2$, etc., the predicate analysis will not terminate. Which predicates get discovered is hard to control and usually depends on internal interpolation heuristics of the satisfiability-modulo-theory (SMT) solver. (4) Traditional 1-induction is also not able to prove the program safe because the assertion is checked only in every fourth loop iteration (when `s` equals 1). Thus, the induction hypothesis is too weak (the program state `s = 4`, `x1 = 0`, `x2 = 1` is a counterexample for the step case in the induction proof).

Intuitively, this program should be provable by k-induction with a k of at least 4. However, for every k, there is a counterexample to the inductive-step case that refutes the proof. For such a counterexample, set `s =` $-k$, `x1 = 0`, `x2 = 1` at the beginning of the loop. Starting in this state, the program would increment `s` k times (induction hypothesis) and then reach `s = 1` with property-violating values of `x1` and `x2` in iteration $k+1$ (inductive step). It is clear that `s` can never be negative, but this fact is not present in the induction hypothesis, and thus, the proof fails. This illustrates the general problem of k-induction-based verification: safety properties often do not hold in unreachable parts of

the state space of a program, and k-induction alone does not distinguish between reachable and unreachable parts of the state space. Therefore, approaches based on k-induction without auxiliary invariants will fail to prove safety for program `example-safe`.

This program could of course be verified more easily if it were rewritten to contain a stronger safety property such as $s \geq 1 \wedge s \leq 4 \wedge (s = 2 \Rightarrow x1 = x2 + 1) \wedge (s \neq 2 \Rightarrow x1 = x2)$ (which is a loop invariant and allows a proof by 1-induction without auxiliary invariants). However, our goal is to automatically verify real programs, and programmers usually neither write down trivial properties such as $s \geq 1$ nor more complex properties such as $s \neq 2 \Rightarrow x1 = x2$.

Our approach of combining k-induction with invariants proves the program safe with $k = 4$ and the invariant $s \geq 1$. This invariant is easy to find automatically using an inexpensive data-flow analysis, such as an interval analysis. For larger programs, a more complex invariant might be necessary, which might get generated at some point by our continuous strengthening of the invariant. Furthermore, stronger invariants can reduce the k that is necessary to prove a program. For example, the invariant $s \geq 1 \wedge s \leq 4 \wedge (s \neq 2 \Rightarrow x1 = x2)$ (which is still weaker than the full loop invariant above) allows to prove the program with $k = 2$. Thus, our strengthening of invariants can also shorten the inductive proof procedure and lead to better performance.

An existing approach tries to solve this problem of a too-weak induction hypothesis by initializing only the variables of the loop-termination condition to a non-deterministic value in the step case, and initializing all other variables to their initial value in the program [35]. However, this approach is not strong enough for the program `example-safe` and even produces a wrong proof (unsound result) for the program `example-unsafe`. This second example program contains a different safety property about `s`, which is violated. Because the variable `s` does not appear in the loop-termination condition, it is not set to an arbitrary value in the step case as it should be, and the inductive proof wrongly concludes that the program is safe because the induction hypothesis is too strong, leading to a missed bug and a wrong result. Our approach does not suffer from this unsoundness, because we add only invariants to the induction hypothesis that the invariant generation has proven to hold.

Related Work. The use of auxiliary invariants is a common technique in software verification [2, 9, 10, 18, 19, 20, 23, 30, 36], and techniques combining data-flow analysis and SMT solvers also exist [28, 31]. In most cases, the purpose is to speed up the analysis. For k-induction, however, the use of invariants is crucial in making the analysis terminate at all (cf. Fig. 1). There are several approaches to software verification using BMC in combination with k-induction.

Split-Case Induction. We use the *split-case k-induction* technique [26, 27], where the base case and the step case are checked in separate steps. Earlier versions of SCRATCH [27] that use this technique transform programs with multiple loops into programs with only one single monolithic loop using a standard approach [1]. The alternative of recursively applying the technique to nested loops is discarded by the authors of SCRATCH [27], because the experiments suggested it was less efficient than checking the single loop that is obtained by the transformation. We also experimented with single-loop transformation, but our experimental results suggest that checking all loops at once in each case instead of checking the monolithic transformation result (which also encodes all loops in one) has

no negative performance impact, so for simplicity, we omit the transformation. SCRATCH also supports *combined-case k-induction* [25], for which all loops are cut by replacing them with k copies each for the base and the step case, and setting all loop-modified variables to non-deterministic values before the step case. That way, both cases can be checked at once in the transformed program and no special handling for multiple loops is required. When using combined-case k-induction, SCRATCH requires loops to be manually annotated with the required k values, whereas its implementation of split-case k-induction supports iterative deepening of k as in our implementation. Contrary to SCRATCH, we do not focus on one specific problem domain [26, 27], but want to provide a solution for solving a wide range of heterogeneous verification tasks.

Auxiliary Invariants. While both the split-case and the combined-case k-induction supposedly succeed with weaker auxiliary invariants than for example the inductive invariant approach [5], the approaches still do require auxiliary invariants in practice, and the tool SCRATCH requires these invariants to be annotated manually [25, 27]. There are techniques for automatically generating invariants that may be used to help inductive approaches to succeed (e.g. [2, 9, 20]. These techniques, however, do not justify their additional effort because they are not guaranteed to provide the required invariants on time, especially if strong auxiliary invariants are required. Based on previous ideas of supporting k-induction with invariants generated by lightweight data-flow analysis [24], we therefore strive to leverage the power of the k-induction approach to succeed with auxiliary invariants generated by a data-flow analysis based on intervals. However, to handle cases where it is necessary to invest more effort into invariant generation, we increase the precision of these invariants over time.

Invariant Injection. A verification tool using a strategy similar to ours is PKIND [28, 33], a model checker for Lustre programs based on k-induction. In PKIND, there is a parallel computation of auxiliary invariants, where candidate invariants derived by templates are iteratively checked via k-induction and, if successful, added to the set of known invariants [32]. While this allows for strengthening the induction hypothesis over time, the template-based approach lacks the flexibility that is available to an invariant generator using dynamic precision refinement [11], and the required additional induction proofs are potentially expensive. We implemented checking candidate invariants with k-induction as a possible strategy of our invariant generation component.

Unsound Strengthening of Induction Hypothesis. ESBMC does not require additional invariants for k-induction, because it assigns non-deterministic values only to the loop-termination condition variables before the inductive-step case [35] and thus retains more information than our as well as the SCRATCH implementation [25, 27], but k-induction in ESBMC is therefore potentially unsound. Our goal is to perform a real proof of safety by removing all pre-loop information in the step case, thus treating the unrolled iterations in the step case truly as "any k consecutive iterations", as is required for the mathematical induction. Our approach counters this lack of information by employing incrementally-refined invariant generation.

Parallel Induction. PKIND checks the base case and the step case in parallel, and ESBMC supports parallel execution of the base case, the forward condition, and the inductive-step case. In contrast, our base case and inductive-step case are checked sequentially, while our invariant generation runs in parallel to the base- and step-case checks.

2 k-Induction with Continuously-Refined Invariants

Our verification approach consists of two algorithms that run concurrently. One algorithm is responsible for generating program invariants, starting with an imprecise invariant, continuously refining (strengthening) the invariant. The other algorithm is responsible for finding error paths with BMC, and for constructing safety proofs with k-induction, for which it periodically picks up the new invariant that the former algorithm has constructed so far. The k-induction algorithm uses information from the invariant generation, but not vice versa. In our presentation, we assume that each program contains at most one loop; in our implementation, we handle programs with multiple loops by checking all loops together.

Iterative-Deepening k-Induction. Algorithm 1 shows our extension of the k-induction algorithm to a combination with continuously-refined invariants. Starting with an initial value for the bound k, e.g., 1, we iteratively increase the value of k after each unsuccessful attempt at finding a specification violation or proving correctness of the program using k-induction. The following description of our approach to k-induction is based on split-case k-induction [25], where for the propositional state variables s and s' within a state-transition system that represents the program, the predicate $I(s)$ denotes that s is an initial state, $T(s,s')$ states that a transition from s to s' exists, and $P(s)$ asserts the safety property for the state s.

Base Case. Lines 3 to 5 implement the *base case*, which consists of running BMC with the current bound k. This means that starting from an initial program state, all paths of the program up to a maximum path length $k-1$ are explored. If an error path is found, the algorithm terminates.

Forward Condition. Otherwise we check whether there exists a path with length $k' > k-1$ in the program, or whether we have already fully explored the state space of the program (lines 6 to 8). In the latter case the program is safe and the algorithm terminates. This check is called the *forward condition* [29].

Inductive Step. Checking the forward condition can, however, only prove safety for programs with finite (and short) loops. Therefore, the algorithm also attempts an inductive proof (lines 9 to 14). The *inductive-step case* checks if, after every sequence of k loop iterations without a property violation, there is also no property violation before loop iteration $k+1$. For model checking of software, however, this check would often fail inconclusively without auxiliary invariants [8]. In our approach, we make use of the fact that the invariants that were generated so far by the concurrently-running invariant-generation algorithm hold, and conjunct these facts to the induction hypothesis. Thus, the inductive-step case proves a program safe if the following condition is unsatisfiable:

$$Inv(s_n) \wedge \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$$

where *Inv* is the currently available program invariant, and $s_n, \ldots, s_{n+k}$ is any sequence of states. If this condition is satisfiable, then the induction check is inconclusive, and the program is not yet proved safe or unsafe with the current value of k and the current invariant. If during the time of the satisfiability check of the step case, a new (stronger) invariant has become available (condition in line 14 is false), we immediately re-check

Algorithm 1 Iterative-Deepening k-Induction

Input:
the initial value $k_{init} \geq 1$ for the bound k,
an upper limit k_{max} for the bound k,
a function $\mathsf{inc} : \mathbb{N} \to \mathbb{N}$ with $\forall n \in \mathbb{N} : \mathsf{inc}(n) > n$ for increasing the bound k,
the initial states defined by the predicate I,
the transfer relation defined by the predicate T, and
a safety property P
Output: **true** if P holds, **false** otherwise

1: $k := k_{init}$
2: **while** $k \leq k_{max}$ **do**
3: $\quad base\_case := I(s_0) \wedge \bigvee_{n=0}^{k-1} \left(\bigwedge_{i=0}^{n-1} T(s_i, s_{i+1}) \wedge \neg P(s_n) \right)$
4: $\quad$ **if** $\mathsf{sat}(base\_case)$ **then**
5: $\quad\quad$ **return** **false**
6: $\quad forward\_condition := I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$
7: $\quad$ **if** $\neg \mathsf{sat}(forward\_condition)$ **then**
8: $\quad\quad$ **return** **true**
9: $\quad step\_case_n := \bigwedge_{i=n}^{n+k-1} (P(s_i) \wedge T(s_i, s_{i+1})) \wedge \neg P(s_{n+k})$
10: $\quad$ **repeat**
11: $\quad\quad Inv := \mathsf{get\_currently\_known\_invariant}()$
12: $\quad\quad$ **if** $\neg \mathsf{sat}(Inv(s_n) \wedge step\_case_n)$ **then**
13: $\quad\quad\quad$ **return** **true**
14: $\quad$ **until** $Inv = \mathsf{get\_currently\_known\_invariant}()$
15: $\quad k := \mathsf{inc}(k)$
16: **return** **unknown**

Algorithm 2 Continuous Invariant Generation using Configurable Program Analysis

Input:
a configurable program analysis with dynamic precision adjustment $\mathbb{D}$,
the initial states defined by predicate I,
a coarse initial precision π_0,
a safety property P
Output: **true** if P holds

1: $\pi := \pi_0$
2: $Inv := true$
3: **loop**
4: $\quad reached := \mathsf{CPAAlgorithm}(\mathbb{D}, I, \pi)$
5: $\quad$ **if** $\forall s \in reached : P(s)$ **then**
6: $\quad\quad$ **return** **true**
7: $\quad Inv := Inv \wedge \bigvee_{s \in reached} s$
8: $\quad \pi := \mathsf{RefinePrec}(\pi, reached)$

the step case with the new invariant. This can be done efficiently using an incremental SMT solver for the repeated satisfiability checks in line 12. Otherwise, we start over with an increased value of k.

Note that the inductive-step case is similar to a BMC check for the presence of error paths of length exactly $k+1$. However, as the step case needs to consider any consecutive $k+1$ loop iterations, and not only the first such iterations, it does not assume that the execution of the loop iterations begins in an initial state. Instead, it assumes that there is a sequence of k iterations without any property violation (induction hypothesis).

Continuous Invariant Generation. Our continuous invariant generation incrementally produces stronger and stronger program invariants. It is based on iterative refinement, each time using an increased precision. After each strengthening of the invariant, it can be used as injection invariant by the k-induction procedure. It may happen that this analysis proves safety of the program all by itself, but this is not its main purpose here.

Our k-induction module works with any kind of invariant-generation procedure, as long as its precision, i.e., its level of abstraction, is configurable. We implemented two different invariant-generation approaches: KI and DF, described below.

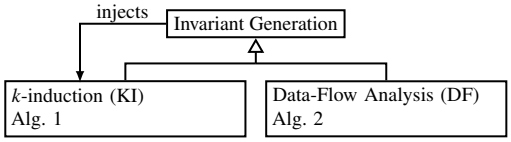

Fig. 3: Configurable design of a k-induction framework

We use the design of Fig. 3 to explain our flexible and modular framework for k-induction: k-induction is a verification technique, i.e., an invariant generation. In this paper, the main algorithm is thus the k-induction, as defined in Alg. 1. We denote the algorithm by KI. If invariants are generated and injected into KI, we denote this injection by KI$\leftarrow$. Thus, the use of generated invariants that are produced by a data-flow analysis (DF) are denoted by KI$\leftarrow$DF. If the invariant generator continuously refines the invariants and repeatedly *injects* those invariants into KI, this is denoted by KI$\leftarrow\!\!\circlearrowleft$, more specifically, if data-flow analysis with dynamic precision adjustment (our new contribution) is used, we have KI$\leftarrow\!\!\circlearrowleft$DF, and if the PKIND approach is used, i.e., KI is used to construct invariants, we have KI$\leftarrow\!\!\circlearrowleft$KI. Now, since the second KI, which constructs invariants for injection into the first KI, can again get invariants injected, we can further build an approach KI$\leftarrow\!\!\circlearrowleft$KI$\leftarrow\!\!\circlearrowleft$DF that combines all approaches such that the invariant-generating KI benefits from the invariants generated with DF, and the main KI algorithm that tries to prove program safety benefits from both invariant generators.

KI. PKIND [33] introduced the idea to construct invariants for injection in parallel, using a template-based method that extracts candidate invariants from the program and verifies their validity using k-induction [32]. If the candidate invariants are found to be valid, they are injected to the main k-induction procedure. We re-implemented the PKIND approach in our framework (KI$\leftarrow\!\!\circlearrowleft$KI), using a separate instance of k-induction to prove candidate invariants. Being based on k-induction, the power of this technique is continuously increased by increasing k. We derive the candidate invariants by taking the negations of assumptions on the control-flow paths to error locations. Similar to our Alg. 2, each time this k-induction algorithm succeeds in proving a candidate invariant, the previously-known invariant is strengthened with this newly generated invariant. In our tool, we used

an instance of Alg. 1 to implement this approach. We are thus able to further combine this technique with other auxiliary invariant-generation approaches.

DF. As a second invariant-generation approach (our contribution), we use the reachability algorithm CPAAlgorithm for configurable program analysis with dynamic precision adjustment [11]. Algorithm 2 shows our continuous invariant generation. The initial program invariant is represented by the formula *true*. We start with running the invariant-generating analysis once with a coarse initial precision (line 4). After each run of the program-invariant generation, we strengthen the previously-known program invariant with the newly-generated invariant (line 7, note that the program invariant *Inv* is not a safety invariant) and announce it globally (such that the *k*-induction algorithm can inject it). If the analysis was able to prove safety of the program, the algorithm terminates (lines 5 to 6). Otherwise, the analysis is restarted with a higher precision. The CPAAlgorithm takes as input a configurable program analysis (CPA), a set of initial abstract states, and a precision. It returns a set of reachable abstract states that form an over-approximation of the reachable program states. Depending on the used CPA and the precision, the analysis by CPAAlgorithm can be efficient and abstract like data-flow analysis or expensive and precise like model checking.

For invariant generation, we choose an abstract domain based on expressions over intervals [8]. Note that this is not a requirement of our approach, which works with any kind of domain. Our choice is based on the high flexibility of this domain, which can be fast and efficient as well as precise. For this CPA, the precision is a triple (Y, n, w), where $Y \subseteq X$ is a specific selection of important program variables, n is the maximal nesting depth of expressions in the abstract state, and w is a boolean specifying whether widening should be used. Those variables that are considered important will not be over-approximated by joining abstract states. With a higher nesting depth, more precise relations between variables can be represented. The use of widening ensures timely termination (at the expense of a lower precision), even for programs with loops with many iterations, like those in the examples of Figs. 1 and 2. An in-depth description of this abstract domain is presented in a technical report [8].

3 Experimental Evaluation

We implemented all existing approaches to *k*-induction, compare all configurations with each other, and the best configuration with other *k*-induction-based software verifiers, as well as to two standard approaches to software verification: predicate and value analysis.

Benchmark Verification Tasks. As benchmark set we use verification tasks from the 2015 Competition on Software Verification (SV-COMP'15) [7]. We took all 3 964 verification tasks from the categories *ControlFlow*, *DeviceDrivers64*, *HeapManipulation*, *Sequentialized*, and *Simple*. The remaining categories were excluded because they use features (such as bit-vectors, concurrency, and recursion) that not all configurations of our evaluation support. A total of 1 148 verification tasks in the benchmark set contain a known specification violation. Although we cannot expect an improvement for these verification tasks when using auxiliary invariants, we did not exclude them because this would unfairly give advantage to the new approach (which spends some effort generating invariants, which are not helpful when proving existence of an error path).

Experimental Setup. All experiments were conducted on computers with two 2.6 GHz 8-Core CPUs (Intel Xeon E5-2560 v2) with 135 GB of RAM. The operating system was Ubuntu 14.04 (64 bit), using Linux 3.13 and OpenJDK 1.7. Each verification task was limited to two CPU cores, a CPU run time of 15 min, and a memory usage of 15 GB. The benchmarking framework BENCHEXEC[2] ensures precise and reproducible results.

Presentation. All benchmarks, tools, and the full results of our evaluation are available on a supplementary web page.[3] All reported times are rounded to two significant digits. We use the scoring scheme of SV-COMP'15 to calculate a score for each configuration. For every real bug found, 1 point is assigned, for every correct safety proof, 2 points are assigned. A score of 6 points is subtracted for every wrong alarm (false positive) reported by the tool, and 12 points are subtracted for every wrong proof of safety (false negative). This scoring scheme values proving safety higher than finding error paths, and significantly punishes wrong answers, which is in line with the community consensus [7] on difficulty of verification vs. falsification and importance of correct results. We consider this a good fit for evaluating an approach such as k-induction, which targets at producing safety proofs.

In Fig. 4 and Fig. 5, we present experimental results using a plot of quantile functions for accumulated scores as introduced by the Competition on Software Verification [6], which shows the score and CPU time for successful results and the score for wrong answers. A data point (x,y) of a graph means that for the respective configuration the sum of the scores of all wrong answers and the scores for all correct answers with a run time of less than or equal to y seconds is x. For the left-most point (x,y) of each graph, the x-value shows the sum of all negative scores for the respective configuration and the y-value shows the time for the fastest successful result. For the right-most point (x,y) of each graph, the x-value shows the total score for this configuration, and the y-value shows the maximal run time. A configuration can be considered better, the further to the right (the closer to 0) its graph begins (fewer wrong answers), the further to the right it ends (more correct answers), and the lower its graph is (less run time).

Comparison of k-induction-based approaches. We implemented all approaches in the JAVA-based open-source software-verification framework CPACHECKER [12], which is available online [4] under the Apache 2.0 License. For the experiments, we used version `1.4.5-cav15` of CPACHECKER, with SMTINTERPOL [21] as SMT solver (using uninterpreted functions and linear arithmetic over integers and reals). The k-induction algorithm of CPACHECKER was configured to increment k by 1 after each try (in Alg. 1, $\mathsf{inc}(k) = k+1$). The precision refinement of the DF-based continuous invariant generation (Alg. 2) was configured to increment the number of important program variables in the first, third, fifth, and any further precision refinements. The second precision refinement increments the expression-nesting depth, and the fourth disables the widening.

We evaluated the following groups of k-induction approaches: (1) without any auxiliary invariants (KI), (2) with auxiliary invariants of different precisions generated by the DF approach (KI⟵DF), and (3) with continuously-refined invariants (KI⟲).

[2] `https://github.com/dbeyer/benchexec`

[3] `http://www.sosy-lab.org/~dbeyer/cpa-k-induction/`

[4] `http://cpachecker.sosy-lab.org`

Table 1: Results of k-induction-based configurations in CPACHECKER for all 3 964 verification tasks with different approaches for generating auxiliary invariants

Approach	KI	KI←DF $(0,1,t)$	KI←DF $(8,2,t)$	KI←DF $(16,2,t)$	KI←DF $(16,2,f)$	KI↞KI	KI↞DF	KI↞KI↞DF
Score	2 246	3 944	4 117	4 062	3 992	3 535	4 249	**4 282**
Correct results	1 531	2 377	2 462	2 428	2 392	2 169	2 507	**2 519**
Wrong proofs	1	1	2	1	1	1	1	1
Wrong alarms	30	30	30	30	30	30	26	25
CPU time (h)	530	330	330	340	340	380	320	320
Wall time (h)	440	240	210	210	210	270	190	170
Times for correct results only:								
CPU time (h)	17	32	39	36	36	28	36	41
Wall time (h)	13	19	22	20	20	18	20	22
k-Values for correct safe results only:								
Max. final k	101	101	100	100	126	101	112	111
Avg. final k	1.7	1.4	1.7	1.8	1.8	1.8	1.8	1.9

The k-induction-based configuration using no auxiliary invariants (KI) is an instance of Alg. 1 where get_currently_known_invariant() always returns *true* as invariant and Alg. 2 does not run at all.

The configurations using generated invariants (KI←DF) are also instances of Alg. 1. Here, Alg. 2 runs in parallel, however, it terminates after one loop iteration. We denote these configurations with triples (s,n,w) that represent the precision (Y,n,w) of the invariant generation with s being the size of the set of important program variables ($s = |Y|$). For example, the first of these configuration, $(0,1,\mathit{true})$, has no variables in the set Y of important program variables (i.e., all variables get over-approximated by the merge operator), the maximum nesting depth of expressions in the abstract state is 1, and the widening operator is used. The remaining configurations we use are $(8,2,\mathit{true})$, $(16,2,\mathit{true})$, and $(16,2,\mathit{false})$. These configurations were selected because they represent some of the extremes of the precisions that are used during dynamic invariant generation. It is impossible to cover every possible valid configuration within the scope of this paper.

There are three configurations using continuously-refined invariants: (1) using the k-induction approach similar to PKIND to generate invariants, refining by increasing k, denoted as KI↞KI, (2) using the DF-based approach to generate invariants, refining by precision adjustment, denoted as KI↞DF, and (3) using both approaches in parallel combination, denoted as KI↞KI↞DF. All configurations using invariant generation run the generation in parallel to the main k-induction algorithm, an instance of Alg. 1.

Score and Reported Results. The configuration KI with no invariant generation receives the lowest score of 2 246, and (as expected) can verify only 1 531 programs successfully. This shows that it is indeed important in practice to enhance k-induction-based software verification with invariants. The configurations KI←DF using invariant generation produce similar numbers of correct results (around 2 400), improving upon the results of the plain k-induction without auxiliary invariants by a score of 1 700 to 1 800. Even though these configurations solve a similar number of programs, a closer inspection

reveals that each of the configurations is able to correctly solve significant amounts of programs where the other configurations run into timeouts. This observation explains the high score of 4 249 points achieved by our approach of injecting the continuously-refined invariants generated with data-flow analysis into the *k*-induction engine (configuration KI⇜DF). By combining the advantages of fast and coarse precisions with those of slow but fine precisions, it correctly solves 2 507 verification tasks, which is 45 more than the best of the chosen configurations without dynamic refinement. Using a *k*-induction-based invariant generation as done by PKIND (configuration KI⇜KI) is also a successful technique for improving the amount of solvable verification tasks, and thus, combining both invariant-generation approaches with continuously refining their precision and injecting the generated invariants into the *k*-induction engine (configuration KI⇜KI⇜DF) is the most effective of all evaluated *k*-induction-based approaches, with a score of 4 282, and 2 519 correct results. The few wrong proofs produced by the configurations are not due to conceptual problems, but only due to incompleteness in the analyzer's handling of certain constructs such as unbounded arrays and pointer aliasing.

Performance. Table 1 shows that by far the largest amount of time is spent by the configuration KI (no auxiliary invariants), because for those programs that cannot be proved without auxiliary invariants, the *k*-induction procedure loops incrementing *k* until the time limit is reached. The wall times and CPU times for the correct results correlate roughly with the amount of correct results, i.e., on average about the same amount of time is spent on correct verifications, whether or not invariant generation is used. This shows that the overhead of generating auxiliary invariants is well-compensated.

The configurations with invariant generation have a relatively higher CPU time compared to their wall time because these configurations spend some time generating invariants in parallel to the *k*-induction algorithm. The results show, however, that the time spent for the continuously-refined invariant generation clearly pays off as the configuration using both data-flow analysis and *k*-induction for invariant generation is not only the one with the most correct results, but at the same time one of the two fastest configurations with only 320 h in total. Even though they produced much more correct results, the configurations KI⇜KI⇜DF and KI⇜DF did not exceed the times of the chosen configurations using invariant generation without continuous refinement. The configuration KI⇜KI using only *k*-induction to continuously generate invariants is slower, but produces results for some programs where the configuration KI⇜DF fails. The results show that the combination of the techniques reaps the benefits of both.

These results show that the additional effort invested in generating auxiliary invariants is well-spent, as it even decreases the overall time due to the fewer timeouts. As expected, the continuously-refined invariants solve many tasks quicker than the configurations using invariant generation with high precisions and without refinement.

Final value of k. The bottom of Table 1 shows some statistics about the final values of *k* for the correct safety proofs. There are only small differences between the maximum *k* values of most of the configurations. Interestingly, the configuration using non-dynamic invariant generation with high precision has a higher maximum final value of *k* than the others, because for the verification task `afnp2014_true-unreach-call.c.i`, a strong invariant generated only with this configuration allowed the proof to succeed. This effect is also observable in the continuously-refined configurations using invariants

Table 2: Results of k-induction-based tools for all 3 964 verification tasks

Tool	CBMC	ESBMC		CPACHECKER
Configuration		sequential	parallel	KI⟲-KI⟲-DF
Score	−4 372	1 674	1 716	**4 282**
Correct results	1 949	2 050	2 059	**2 519**
Wrong proofs	666	156	152	**1**
Wrong alarms	**5**	9	13	25
CPU time (h)	360	290	370	320
Wall time (h)	360	290	200	170
Times for correct results only:				
CPU time (h)	3.9	16	26	41
Wall time (h)	3.9	16	13	22
k-Values for correct safe results only:				
Max. final k	50	2 048	1 952	111
Avg. final k	1.1	5.3	7.1	1.9

generated by data-flow analysis: They are also able to solve this verification task, and, by dynamically increasing the precision, find the required auxiliary invariant even earlier with loop bounds 112 and 111, respectively. There is also a verification task in the benchmark set, `gj2007_true-unreach-call.c.i`, where most configurations need to unroll a loop with bound 100 to prove safety, while the strong invariant generation technique allows the proof to succeed earlier, at a loop bound of 16. The continuously-refined configurations benefit from the same effect: KI⟲-DF and KI⟲-KI⟲-DF solve this task at loop bounds 22 and 19, respectively.

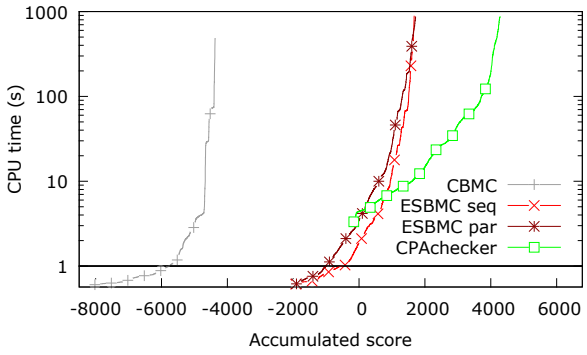

Fig. 4: Quantile functions of k-induction-based tools (CPACHECKER in configuration KI⟲-KI⟲-DF) for accumulated scores showing the CPU time for the successful results; linear scale between 0 s and 1 s, logarithmic scale beyond

Comparison with other tools. For comparison with other k-induction-based tools, we evaluated ESBMC and CBMC, two software model checkers with support for k-induction. For CBMC, we used version 5.1 in combination with a wrapper script for split-case k-induction provided by M. Tautschnig. For ESBMC we used version 1.25.2 in combination with a wrapper script that enables k-induction (based on the SV-COMP'13 submission [35]). We also provide results for the experimental parallel k-induction of ESBMC, but note that our benchmark setup is not focused on parallelization (using only two CPU cores and a CPU-time limit instead of wall time).

The CPAchecker configuration in this comparison is the one with continuously-refined invariants and both invariant generators (KI⇜↻-KI⇜↻-DF). Table 2 gives the results; Fig. 4 shows the quantile functions of the accumulated scores for each configuration. The results for Cbmc are not competitive, which may be attributed to the experimental nature of its *k*-induction support.

Score. CPAchecker in configuration KI⇜↻-KI⇜↻-DF successfully verifies almost 500 tasks (20 %) more than Esbmc. Furthermore, it has only 1 missed bug, which is related to unsoundness in the handling of some C features, whereas Esbmc has more than 150 wrong safety proofs. This large number of wrong results must be attributed to the unsound heuristic of Esbmc for strengthening the induction hypothesis, where it retains potentially incorrect information about loop-modified variables [35]. We have previously also implemented this approach in CPAchecker and obtained similar results [8]. The large number of wrong proofs reduces the confidence in the soundness of the correct proofs. Consequently, the score achieved by CPAchecker in configuration KI⇜↻-KI⇜↻-DF is much higher than the score of Esbmc (4 282 compared to 1 674 points). This clear advantage is also visible in Fig. 4. The parallel version of Esbmc performs somewhat better than its sequential version, and misses fewer bugs. This is due to the fact that the base case and the step case are performed in parallel, and the loop bound *k* is incremented independently for each of them. The base case is usually easier to solve for the SMT solver, and thus the base-case checks proceed faster than the step-case checks (reaching a higher value of *k* sooner). Therefore, the parallel version manages to find some bugs by reaching the relevant *k* in the base-case checks earlier than in the step-case checks, which would produce a wrong safety proof at reaching *k*. However, the number of wrong proofs is still much higher than with our approach, which is conceptually sound. Thus, the score of the new, sound approach is more than 2 500 points higher.

Performance. Table 2 shows that our approach needs only 10 % more CPU time than the sequential version of Esbmc for solving a much higher number of tasks, and even needs less CPU and wall time than the parallel version of Esbmc. This indicates that due to our invariants, we succeed more often with fewer loop unrollings, and thus in less time. It also shows that the effort invested for generating the invariants is well spent.

Final value of k. The bottom of Table 2 contains some statistics on the final value of *k* that was needed to verify a program. The table shows that for safe programs, CPAchecker needs a loop bound that is (on average) only about one third of the loop bound that Esbmc needs. This advantage is due to the use of generated invariants, which make the induction proofs easier and likely to succeed with a smaller number of *k*. The verification task `array_true-unreach-call2.i` is solved by Esbmc after completely unwinding the loop, therefore reaching the large *k*-value 2 048. In the parallel version, the (quicker) detached base case hits this bound while the inductive step case is still at $k = 1\,952$.

Comparison with other approaches. We also compare our combination of *k*-induction with continuously-refined invariants with other common approaches for software verification. We use for comparison two analyses based on CEGAR, a predicate analysis [13] and a value analysis [14]. Both are implemented in CPAchecker, which allows us to compare the approaches inside the same tool, using the same run-time environment, SMT solver, etc., and focus only on the conceptual differences between the analyses.

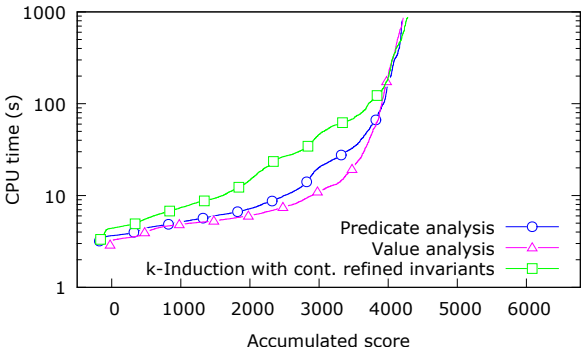

Fig. 5: Quantile functions of different approaches implemented in CPACHECKER (*k*-induction in configuration KI⟲-KI⟲-DF) for accumulated scores showing the CPU time for the successful results

Fig. 5 shows a quantile plot to compare the configuration KI⟲-KI⟲-DF with CPACHECKER's predicate analysis and value analysis. The predicate analysis solves 2463 verification tasks in a total of 280 CPU hours, and achieves a score of 4201. The value analysis solves 2367 verification tasks in a total of 303 CPU hours, and achieves a score of 4216 because it has a few wrong results less. The higher number of solved tasks (2519) and the higher score (4282) of the *k*-induction-based configuration show that *k*-induction is clearly competitive with the state-of-the-art in software verification, if it is boosted by injecting continuously-refined invariants.

Acknowledgments. We thank M. Tautschnig and L. Cordeiro for explaining the optimal available parameters for *k*-induction, for the verifiers CBMC and ESBMC, respectively.

4 Conclusion

We have presented the novel idea of injecting invariants into *k*-induction that are generated using data-flow analysis with dynamic precision adjustment, and contribute a publicly available implementation of our idea within the software-verification framework CPACHECKER. Our extensive experiments show that the new approach outperforms all existing implementations of *k*-induction for software verification, and that it is competitive compared to other, more mature techniques for software verification. We showed that a sound, effective, and efficient *k*-induction approach to general-purpose software verification is possible, and that the additional resources required to achieve these combined benefits are negligible if invested judiciously. At the same time, there is still room for improvement of our technique. An interesting improvement would be to add an information flow between the two cooperating algorithms in the reverse direction. If the *k*-induction procedure could tell the invariant generation which facts it misses to prove safety, this could lead to a more efficient and effective approach to generate invariants that are specifically tailored to the needs of the *k*-induction proof. Already now, CPACHECKER is parsimonious in terms of unrollings, compared to other tools. The low *k*-values required to prove many programs show that even our current invariant generation is powerful enough to produce invariants that are strong enough to help cut down the necessary number of loop unrollings. *k*-induction-guided precision refinement might direct the invariant generation towards providing weaker but still useful invariants for *k*-induction more efficiently.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. M. Awedh and F. Somenzi. Automatic invariant strengthening to prove properties in bounded model checking. In *Proc. DAC*, pages 1073–1076. ACM/IEEE, 2006.
3. T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proc. IFM*, LNCS 2999, pages 1–20. Springer, 2004.
4. T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.
5. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proc. PASTE*, pages 82–87. ACM, 2005.
6. D. Beyer. Second competition on software verification. In *Proc. TACAS*, LNCS 7795, pages 594–609. Springer, 2013.
7. D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.
8. D. Beyer, M. Dangl, and P. Wendler. Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau, January 2015. arXiv:1502.00096.
9. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proc. VMCAI*, LNCS 4349, pages 378–394. Springer, 2007.
10. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM, 2007.
11. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
12. D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
13. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
14. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
15. A. Biere. *Handbook of Satisfiability*. IOS Press, 2009.
16. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
17. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.
18. N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997.
19. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.
20. A. R. Bradley and Z. Manna. Property-directed incremental invariant generation. *FAC*, 20(4-5):379–405, 2008.
21. J. Christ, J. Hoenicke, and A. Nutz. SMTINTERPOL: An interpolating SMT solver. In *Proc. SPIN*, LNCS 7385, pages 248–254. Springer, 2012.
22. L. Cordeiro, B. Fischer, and J. P. M. Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proc. ASE*, pages 137–148. IEEE, 2009.
23. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*, pages 84–96, 1978.

24. A. F. Donaldson, L. Haller, and D. Kröning. Strengthening induction-based race checking with lightweight static analysis. In *Proc. VMCAI*, LNCS 6538, pages 169–183. Springer, 2011.
25. A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. Software verification using k-induction. In *Proc. SAS*, LNCS 6887, pages 351–368. Springer, 2011.
26. A. F. Donaldson, D. Kröning, and P. Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *Proc. TACAS*, LNCS 6015, pages 280–295. Springer, 2010.
27. A. F. Donaldson, D. Kröning, and P. Rümmer. Automatic analysis of DMA races using model checking and k-induction. *FMSD*, 39(1):83–113, 2011.
28. P. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *Proc. NASA Formal Methods*, LNCS 7871, pages 139–154. Springer, 2013.
29. D. Große, H. M. Le, and R. Drechsler. Proving transaction and system-level properties of untimed SystemC TLM designs. In *Proc. MEMOCODE*, pages 113–122. IEEE, 2010.
30. A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Proc. CAV*, LNCS 5643, pages 634–640. Springer, 2009.
31. A. Gurfinkel, A. Albarghouthi, S. Chaki, Y. Li, and M. Chechik. UFO: Verification with interpolants and abstract interpretation (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 637–640. Springer, 2013.
32. T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-based invariant discovery. In *Proc. NASA Formal Methods*, LNCS 6617, pages 192–206. Springer, 2011.
33. T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. In *Proc. Int. Workshop on Parallel and Distributed Methods in Verification*, EPTCS 72, pages 55–62, 2011.
34. A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.
35. J. Morse, L. Cordeiro, D. Nicole, and B. Fischer. Handling unbounded loops with ESBMC 1.20 (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 619–622. Springer, 2013.
36. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proc. VMCAI*, LNCS 3385, pages 25–41. Springer, 2005.
37. M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proc. FMCAD*, LNCS 1954, pages 127–144. Springer, 2000.
38. T. Wahl. The k-induction principle, 2013. Available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`.