SMT-based Software Model Checking: An Experimental Comparison of Four Algorithms

Dirk Beyer and Matthias Dangl

University of Passau, Germany

Abstract. After many years of successful development of new algorithms for software model checking, there is a need to consolidate the knowledge about the different algorithms and approaches. This paper gives a coarse overview in terms of effectiveness and efficiency of four algorithms. We compare the following different "schools of thought" of algorithms: bounded model checking, k-induction, predicate abstraction, and lazy abstraction with interpolants. Those algorithms are well-known and successful in software verification. They have in common that they are based on SMT solving as the back-end technology, using the theories of uninterpreted functions, bit vectors, and floats as underlying theory. All four algorithms are implemented in the verification framework CPAchecker. Thus, we can present an evaluation that really compares only the core algorithms, and keeps the design variables such as parser front end, SMT solver, used theory in SMT formulas, etc. constant. We evaluate the algorithms on a large set of verification tasks, and discuss the conclusions.

Keywords: Software Verification, Program Analysis, Bounded Model Checking, k-induction, Impact, Lazy Abstraction, SMT Solving

1 Introduction

In recent years, advances in automatic methods for software verification have lead to increased efforts towards applying software verification to industrial systems, in particular operating-systems code [3, 5, 13, 30]. Predicate abstraction [24] with counterexample-guided abstraction refinement (CEGAR) [18] and lazy abstraction [27], lazy abstraction with interpolants [33], and k-induction with auxiliary-invariants [8, 22] are some of the concepts introduced to scale verification technology from simple toy problems to real-world software. In the 5^{th} International Competition of Software Verification (SV-COMP'16) [7], ten out of the 13 candidates participating in category *Overall* used some of these techniques, and out of the remaining three, two are bounded model checkers [15]. Considering this apparent success, we revisit an earlier work that presented a unifying algorithm for lazy predicate abstraction (Blast-like) and lazy abstraction with interpolants (Impact-like), and showed that both techniques perform similarly [14]. We conduct a comparative evaluation of bounded model checking, k-induction, lazy predicate

abstraction, and lazy abstraction with interpolants, observe that the previously drawn conclusions about the two lazy-abstraction techniques still hold today, and show that k-induction has the potential to outperform the other two techniques. We restrict our presentation to safety properties; however, the techniques that we present can be used also for checking liveness [38].

Availability of Data and Tools. All presented approaches are implemented in the open-source verification framework CPAchecker [10], which is available under the Apache 2.0 license. All experiments are based on publicly available benchmark verification tasks from the last competition on software verification [7]. To ensure technical accuracy, we used the open-source benchmarking framework BenchExec[1] [12] to conduct our experiments. Tables with our detailed experimental results are available on the supplementary web page [2].

Related Work. Unfortunately, there is not much work available in rigorous comparison of algorithms. General overviews over methods for reasoning [6] and of approaches for software model checking [28] exist, but no systematic comparison of the algorithms in a common formal setting. This paper tries to give an abreast comparison of the effectiveness and efficiency of the algorithms.

Figure 1 tries to categorize the main approaches for software model checking that are based on SMT technology; we use this structure also to give pointers to other implementations of the approaches.

Bounded Model Checking. Many software bugs can be found by a bounded search through the state space of the program. Bounded model checking [15] for software encodes all program paths that result from a bounded unrolling of the program in an SMT formula that is satisfiable if the formula encodes a feasible program path from the program entry to a violation of the specification. Several implementations were demonstrated to be successful in improving software quality by revealing shallow program bugs, for example Cbmc [19], Esbmc [20], Llbmc [39], and Smack [35]. The characteristics to quickly verify a large portion of the state space without the need of computing expensive abstractions made the technique a basis component in many verification tools (cf. Table 4 in the report for SV-COMP 2016 [7]).

Unbounded — No Abstraction. The idea of bounded model checking (to encode large portions of a program as SMT formula) can be used also for unbounded verification by using an induction argument [40], i.e., a safe inductive invariant needs to be implied by all paths from the program entry to the loop head and by all paths starting from the assumed invariant (induction hypothesis) at the loop head through the loop body. The remaining problem, which is a main focus area of research on k-induction, is to compute a sufficient safe inductive invariant. The approach of k-induction is implemented in Cbmc [19], CPAchecker [8], Esbmc [36], Pkind [29], and 2ls [37]. The approach of k-induction with continuously-refining invariant generation [8] was independently reproduced later in 2ls [17].

[1] `https://github.com/sosy-lab/benchexec`
[2] `https://www.sosy-lab.org/~dbeyer/k-ind-compare/`

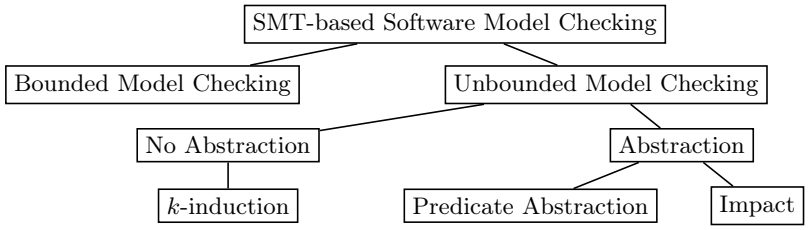


Fig. 1: Classification of Algorithms

Unbounded — With Abstraction. A completely different approach is to compute an over-approximation of the state-space, using insights from data-flow analysis [1, 31, 34]. The idea of state-space abstraction is often combined with the idea of counterexample-guided abstraction refinement (CEGAR) [18] and lazy abstraction refinement [27]. Several verifiers implement a predicate abstraction [24]: SLAM [4], BLAST [9], and CPACHECKER [10]. A safe invariant is computed by iteratively refining the abstract states by discovering new predicates during each CEGAR step. Interpolation [21, 32] is a successful method to obtain useful predicates from error paths. ULTIMATE AUTOMIZER [26] combines predicate abstraction with an automaton-based approach.

Instead of using predicate abstraction, it is possible to construct the abstract state space directly from interpolants using the IMPACT algorithm [33].

Combinations. Of course, the best features of all approaches should be combined into new, "hybrid" methods, such as implemented in CPACHECKER [41], SEAHORN [25], and UFO [2].

2 Algorithms

In the following, we will give a unifying overview over four widely-used algorithms for software model checking: bounded model checking (BMC), k-induction, predicate abstraction, and the IMPACT algorithm.

As shown in Fig. 1, all four algorithms are SMT-based model checking algorithms: They rely on encoding program paths as SMT formulas.

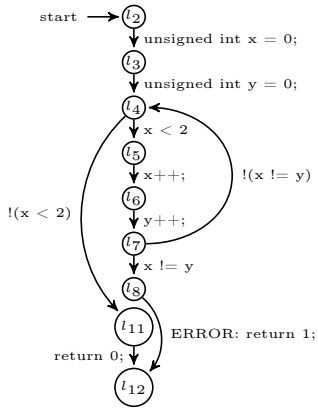
Preliminaries. We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[3] We use control-flow automata (CFA) to represent programs. A *control-flow automaton* consists of a set L of program locations (modeling the program counter), the initial program location $l_2 \in L$ (modeling the program entry), a target program location $l_E \in L$ (modeling the specification violation), and a set of control-flow edges (modeling the operation that is executed during the flow of control from one program location to another).

Example. Fig. 2 shows an example C program and the corresponding CFA. We will use this example to illustrate the algorithms. The displayed C program

[3] Our implementations are based on CPACHECKER [10], which supports C programs.

```
1  int main() {
2    unsigned int x = 0;
3    unsigned int y = 0;
4    while (x < 2) {
5      x++;
6      y++;
7      if (x != y) {
8        ERROR: return 1;
9      }
10   }
11   return 0;
12 }
```

(a) Safe program



(b) Control-flow automaton

Fig. 2: An example C program (a) and its CFA (b)

contains two variables `x` and `y`, which are both initialized to `0`. In the loop of lines 4–10, both variables are incremented as long as `x` is lower than `2`. The CFA nodes corresponding to this loop are l_4, l_5, l_6, and l_7, with l_4 being the loop head. At the end of the loop body in line 7, `x` and `y` are checked for equality. If the variables are not equal, control flows to the error location l_8 in line 8.

ABE: An SMT-formula-based program analysis. For the algorithms presented in this paper, it is frequently required to represent the reachability of a program state as a precise or over-approximated set of program paths that are encoded as SMT formulas. A configurable program analysis (CPA) for this purpose has been formally defined in previous work [11]. Adjustable-block encoding (ABE) is a forward-reachability analysis that unrolls the control-flow automaton (CFA) into an abstract reachability graph (ARG) while keeping track of arbitrarily-definable (adjustable) blocks. An abstract state in the ARG is defined as a triple consisting of a program location, an abstract-state formula, which represents an abstract over-approximation of the reachability of the block entry, and a concrete path formula, which for any state within a block represents the set of concrete paths from the block entry to the location of this state. This mechanism can be used to control if and when to apply abstraction by configuring the definition of block(s). Two of the algorithms we present, BMC and k-induction, do not use abstraction, while the other two, predicate abstraction and Impact, do. In the former case, the abstract-state formula is always *true* and has no effect. For consistency, however, we display the abstract-state formula in all our graphical representations.

Another feature required by the presented algorithms is the configuration of a limit for unrolling the control-flow automaton into an ARG, because a complete unrolling is not always desirable or even feasible. In addition to the configurability of the definition of blocks, we therefore introduce such a limit on unrolling the CFA as another parameter to configure the SMT-formula-based program analysis ABE. In the following, we will describe the presented algorithms informally and discuss their usage of configurable ABE program analysis as a convenient way to construct, manage, and apply SMT formulas.

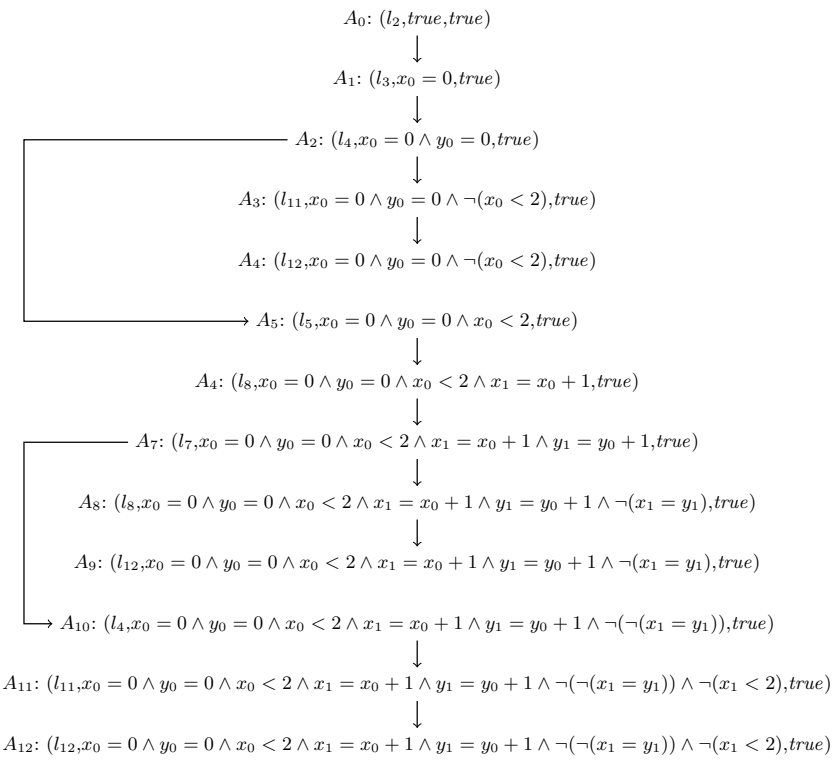


Fig. 3: ARG fragment for applying BMC to the example of Fig. 2

Bounded Model Checking. In BMC, the state space of the analyzed program is explored without using abstraction by unrolling loops up to a given bound k. In this setting, ABE is configured so that there is only one single block of unbounded size starting at the program entry. This way, there is never any abstraction computation. The limit for unrolling the CFA with ABE in the context of BMC is given by the loop-unrolling bound k.

Due to the single ABE block that contains the whole program, the path formula of any state always represents a set of concrete program paths from the program entry to the program location of this state. After unrolling a loop up to bound k, the state-space exploration stops. Then, the disjunction of the path formulas of all states in the explored state space at error location l_E is checked for satisfiability using an SMT solver. If the formula is satisfiable, the program contains a real specification violation. If the formula is unsatisfiable, there is no specification violation in the program within the first k loop unrollings. Unless an upper bound lower than or equal to k for a loop is known, a specification violation beyond the first k loop iterations may or may not exist. Due to this limitation, BMC is usually not able to prove that a program satisfies its specification.

If we apply BMC with $k = 1$ to the example in Fig. 2, unrolling the CFA yields the ARG depicted in Fig. 3. The path formula of the ARG state A_8, which is the only ARG state at error location $l_E = l_8$, is unsatisfiable. Therefore, no

bug is reachable within one loop unrolling. The bound $k = 1$ is not large enough to completely unroll the loop; the second loop iteration, which is necessary to have the loop condition `x < 2` no longer satisfied, is missing from this ARG.

k-Induction. For ease of presentation, we assume that the analyzed program contains exactly one loop head l_{LH}. In practice, k-induction can be applied to programs with many loops [8]. k-induction, like BMC, is an approach that at its core does not rely on abstraction techniques. The k-induction algorithm is comprised of two phases. The first phase is equivalent to a bounded model check with bound k, and is called the *base case* of the induction proof. If a specification violation is detected in the base case, the algorithm stops and the violation is reported. Otherwise, the second phase is started. In the second phase, ABE is used to re-explore the state space of the analyzed program, with the analysis and the (single, unbounded) ABE block starting not at the program entry l_2, but at the loop head l_{LH}, so that the path formula of any state always represents a set of concrete program paths from the loop head to the program location of this state. The limit for unrolling the CFA is set to stop at $k + 1$ loop unrollings. Afterwards, an SMT solver is used to check if the negation of the disjunction of all path formulas for states at the error location l_E that were reached within k loop unrollings, implies the negation of the disjunction of all path formulas for states at the error location l_E that were reached within $k + 1$ loop unrollings. This step is called the *inductive-step case*. If the implication holds, the program is safe, i.e., the safety property is a k-inductive program invariant. Often, however, the safety property of a verification task is not directly k-inductive for any k, but only relative to some auxiliary invariant, so that plain k-induction cannot succeed in proving safety. In these cases, it is necessary to employ an auxiliary-invariant generator and inject these invariants into the k-induction procedure to strengthen the hypothesis of the inductive step case.

If we apply k-induction with $k = 1$ to the example in Fig. 2, the first phase, which is equivalent to BMC, yields the same ARG as in Fig. 3. Figure 4 shows the ARG of the second phase, which is constructed by unrolling the CFA starting at loop head $l_{LH} = l_4$ and using loop bound $k = 1$. The negation of the disjunction of the path formulas of the ARG states A_5 and A_{10} at the error location $l_E = l_8$, which were reached within at most one loop iteration, implies the negation of the disjunction of the path formulas of the ARG states A_5, A_{10}, and A_{18} at the error location $l_E = l_8$, which were reached within at most $k + 1 = 2$ loop iterations, which in combination with the base case (BMC) from the first phase proves that the program is safe. This inductive proof is strong enough to prove safety even if we replace the loop condition in line 4 of the sample program by a nondeterministic value.

Predicate Abstraction. Predicate abstraction with counterexample-guided abstraction refinement (CEGAR) directly applies ABE within the CEGAR loop. The abstraction-state formula of an abstract state over-approximates the reachable concrete states using a boolean combination of predicates over program variables from a given set of predicates (the *precision* π). This abstraction is computed by an SMT solver. Using CEGAR, it is possible to apply lazy abstraction, starting

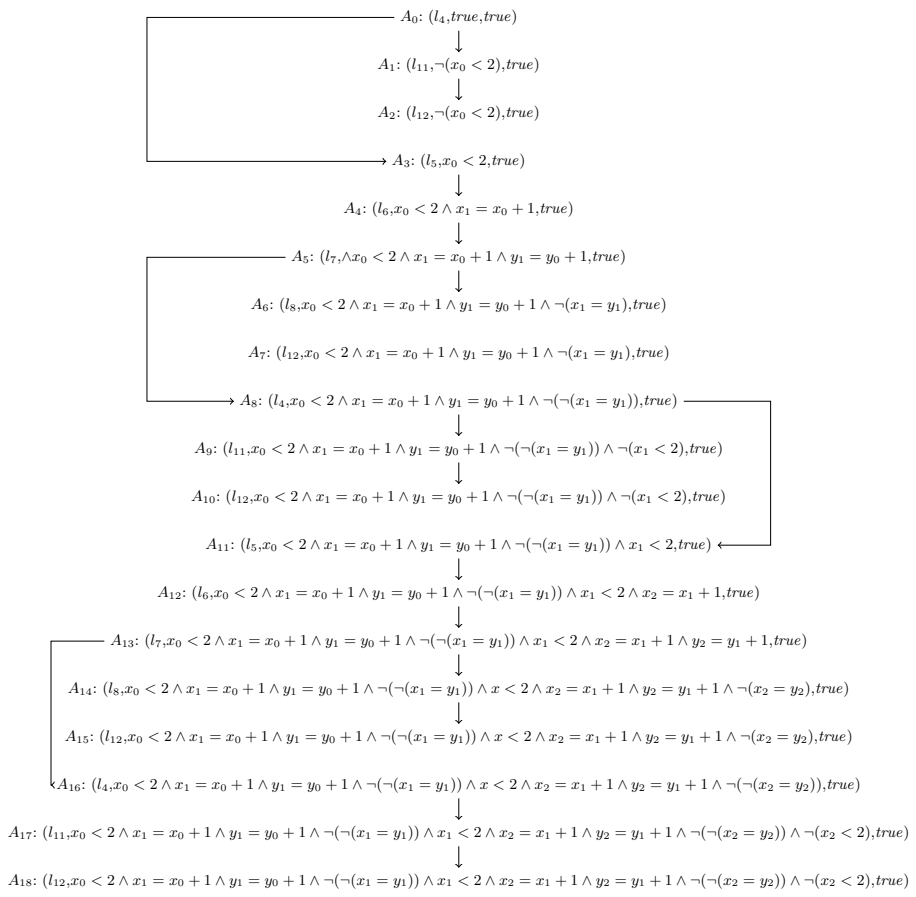


Fig. 4: ARG fragment for the inductive step case of k-induction applied to the example of Fig. 2

out with an empty initial precision. When the analysis encounters an abstract state at the error location l_E, the concrete program path leading to this state is reconstructed and checked for feasibility using an SMT solver. If the concrete error path is feasible, the algorithm reports the error and terminates. Otherwise, the precision is refined (usually by employing an SMT solver to compute Craig interpolants [21] for the locations on the error path) and the analysis is restarted. Due to the refined precision, it is guaranteed that the previously identified infeasible error paths are not encountered again.

For this technique, the blocks can be arbitrarily defined; in our experimental evaluation we define a block to end at a loop head. To enable CEGAR, the unrolling of the CFA must be configured to stop if the state-space exploration hits a state at the error location l_E.

If we apply predicate abstraction to the example in Fig.2 using a precision $\pi : \{x = y\}$ and defining all blocks to end at the loop head l_4, we obtain

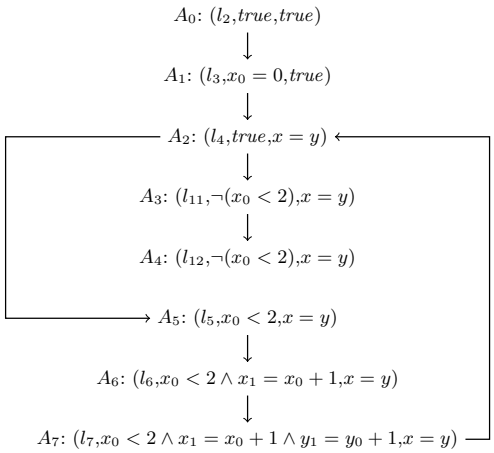


Fig. 5: ARG for predicate abstraction applied to the example of Fig. 2

the ARG depicted in Fig.5: The first block consists of the locations l_2 and l_3. If the ABE analysis hits location l_4, which is a loop head, the path formula $x_0 = 0 \wedge y_0 = 0$ is abstracted using the set of predicates π. Precision π contains only the predicate $x = y$, which is implied by the path formula and becomes the new abstraction formula, while the path formula for the new block beginning at l_4 is reset to *true*. From that point onwards, there are two possible paths: one directly to the end of the program the loop if `x` is greater than or equal to `2`, and another one into the loop if `x` is less than `2`. The path avoiding the loop is trivially safe, because from l_{11} or l_{12} there is no control-flow path back to the error location. The path through the loop increments both variables before encountering the assertion. Using the abstraction formula encoding the reachability of the block entry in combination with the path formula, it is easy to conclude that the assertion is true, so that the only feasible successor is at the loop head l_4, which causes the previous block to end. The abstraction computation yields again the abstraction formula $x = y$ at l_4, which is already covered by the ARG state A_2. Therefore, unrolling the CFA into the ARG completed without encountering the error location $l_E = l_8$. The algorithm thus concludes that the program is safe.

Impact. Lazy abstraction with interpolants, more commonly known as the Impact algorithm due to its first implementation in the tool Impact, also uses ABE to create an unwinding of the CFA similar to predicate abstraction. Impact, however, does not base its abstractions on an explicit precision. Initializing all new abstract-state formulas to *true*, the algorithm repeatedly applies the following three steps until no further changes can be made:

(1) *Expand(s)*: If the state s has no successors yet (s is currently a leaf node in the ARG) and is not marked as *covered*, the successor states of s are created with *true* as their initial abstract-state formula.

(2) *Refine(s)*: If s is an abstract state at the error location l_E with an abstract-state formula different from *false*, inductive Craig interpolants for the path from the root of the ARG to this state s are computed using an SMT solver. Each

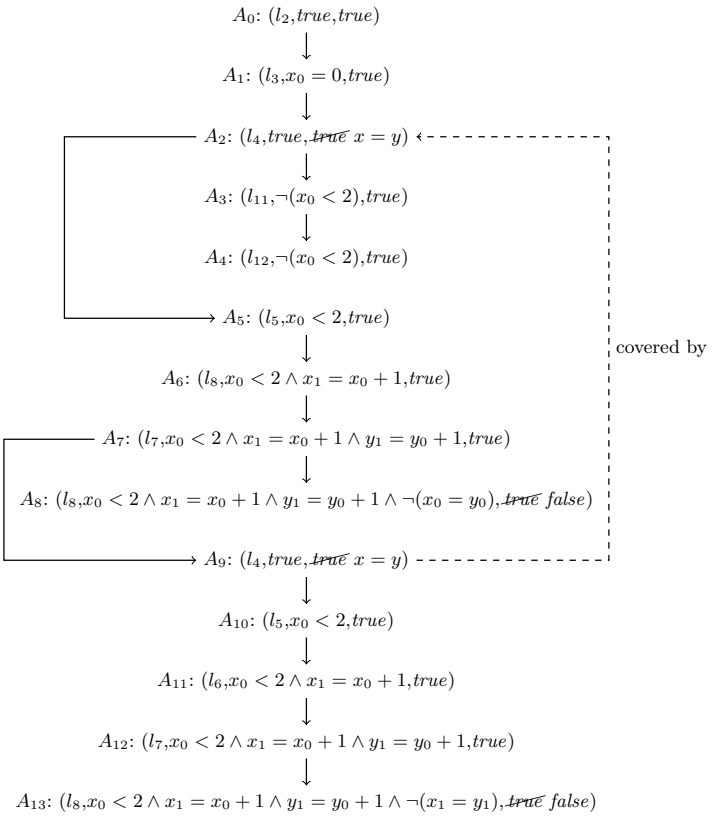


Fig. 6: Final ARG for applying the IMPACT algorithm to the example of Fig. 2

abstract state at an ABE block entry along this path is marked as *not covered*, and its abstract-state formula is strengthened by conjoining it with the corresponding interpolant, guaranteeing that if the state s is unreachable, the formula of s becomes *false*.

(3) *Cover(s_1,s_2)*: A state s_1 gets marked as *covered* by another state s_2 if neither s_2 nor any of its ancestors are covered, both states belong to the same program location, the abstract-state formula of s_2 is implied by the formula of s_1, s_1 is not an ancestor of s_2, and s_2 was created before s_1.

As in predicate abstraction, the ABE blocks can be arbitrarily defined; again, we define a block to end at a loop head in our experimental evaluation of the IMPACT algorithm. Since this algorithm is also based on CEGAR, the unrolling of the CFA must again be configured to stop when the state-space exploration hits a state at the error location l_E, so that interpolation can be used to compute the abstractions.

The original presentation of the IMPACT algorithm [33] also includes a description of an optimization called *forced covering*, which improves the performance significantly but is not relevant for understanding the fundamental idea of the algorithm and exceeds the scope of our summary.

If we apply the IMPACT algorithm to the example program from Fig. 2 defining blocks to end at the loop head l_4 and assuming that both interpolations that are required during the analysis yield the interpolant $x = y$, we obtain an ARG as depicted in Fig. 6: Starting with the initialization of the variables, we first obtain the ARG states A_0 and A_1; at A_2, however, we reset the path formula to *true*, because l_4 is a block entry. Note that at this point, the abstract-state formula for this block is still *true*. Unwinding the first loop iteration, we first obtain abstract states for incrementing the variables and then hit the error location $l_E = l_8$ with state A_8. An SMT check on the reconstructed concrete error path shows that the path is infeasible, therefore, we perform an interpolation. For the example we assume that interpolation provides the interpolant $x = y$, strengthen the abstract-state formula of A_2 with it, and set the abstract-state formula of A_8 to *false*. Then, we continue the expansion of A_7 towards l_4 with state A_9. Note that at this point, the abstract-state formula for A_9 is still *true*, so that it is not covered by A_2 with $x = y$. Also, A_2 cannot be covered by A_9, because A_2 is an ancestor of A_9. We unwind the loop for another iteration and again hit the error location l_8 with state A_{13}. Once again, the concrete path formula for this state is infeasible, so we interpolate. For the example we assume that interpolation provides again the interpolant $x = y$, use it to strengthen the abstract-state formula of A_9, and set the abstract-state formula of A_{13} to *false*. Now, a coverage check reveals that A_9 is covered by A_2, because neither A_9 nor any of its ancestors is covered yet, both belong to the same location l_4, $x = y$ implies $x = y$, A_9 is not an ancestor of A_2, and A_2 was created before A_9. Because A_9 is now covered, we need not continue expanding the other states in this block, and the algorithm terminates without finding any feasible error paths, thus proving safety.

Summary. We showed how to apply the four algorithms to the example presented in Fig. 2 and gave a rough outline of the concepts required to implement them. While BMC is very limited in its capacity of proving correctness, it is also the most straightforward of the four algorithms, because k-induction requires an auxiliary-invariant generator to be applicable in practice, and predicate abstraction and IMPACT require interpolation techniques. While invariant generator and interpolation engine are usually treated as a black box in the description of these algorithms, the efficiency and effectiveness of the techniques depends on the quality of these modules.

3 Evaluation

We evaluate bounded model checking, k-induction, predicate abstraction, and IMPACT, on a large set of verification tasks and compare the approaches.

Benchmark Set. As benchmark set we use the verification tasks from the 2016 Competition on Software Verification (SV-COMP'16) [7]. We took all 4 779 verification tasks from all categories except *ArraysMemSafety*, *HeapMemSafety*, *Overflows*, *Recursive*, *Termination*, and *Concurrency*, which are not supported by our implementations of the approaches. A total of 1 320 tasks in the benchmark set contain a known specification violation, while the rest of the tasks is assumed to be free of violations.

Table 1: Experimental results of the approaches for all 4 779 verification tasks, 1 320 of which contain bugs, while the other 3 459 are considered to be safe

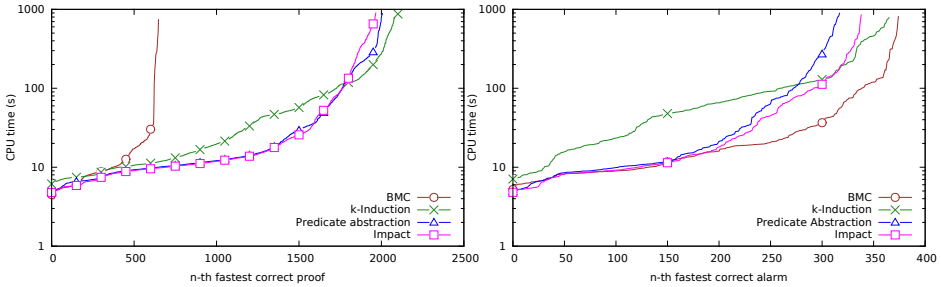
Algorithm	BMC	k-induction	Predicate Abstraction	Impact
Correct results	1 024	2 482	2 325	2 306
Correct proofs	649	2 116	2 007	1 967
Correct alarms	375	366	318	339
False alarms	1	1	0	0
Timeouts	2 786	2 047	1 646	1 607
Out of memory	180	98	75	104
Other inconclusive	788	151	733	762
Times for correct results				
Total CPU Time (h)	8.3	54	32	32
Avg. CPU Time (s)	29	79	49	50
Times for correct proofs				
Total CPU Time (h)	4.3	44	26	27
Avg. CPU Time (s)	24	75	47	50
Times for correct alarms				
Total CPU Time (h)	4.0	10	5.4	4.8
Avg. CPU Time (s)	38	100	61	51

Experimental Setup. Our experiments were conducted on machines with two 2.6 GHz 8-Core CPUs (Intel Xeon E5-2650 v2) with 135 GB of RAM. The operating system was Ubuntu 16.04 (64 bit), using Linux 4.4 and OpenJDK 1.8. Each verification task was limited to two CPU cores, a CPU run time of 15 min and a memory usage of 15 GB. We used version `cpachecker-1.6.8-vstte16` of CPAchecker, with MathSAT5 as SMT solver. We configured CPAchecker to use the SMT theories over uninterpreted functions, bit vectors, and floats. To evaluate the algorithms, we used ABE for Impact and predicate abstraction [14]. For Impact we also activated the forced-covering optimization [33], and for k-induction we use continuously-refined invariants from an invariant generator that employs an abstract domain based on intervals [8]. For bounded model checking we use a configuration with forward-condition checking [23].

Experimental Validity. We implemented all evaluated algorithms using the same software-verification framework, CPAchecker. This allows us to compare the actual algorithms instead of comparing different tools with different front ends and different utilities, thus eliminating influences on the results caused by such implementation differences unrelated to the actual algorithms.

Results. Table 1 shows the number of correctly solved verification tasks for each of the algorithms, as well as the time that was spent on producing these results. None of the algorithms reported incorrect proofs [4], there was one false alarm for bounded model checking, and one false alarm for k-induction. When an algorithm exceeds its time or memory limit, it is terminated inconclusively. Other

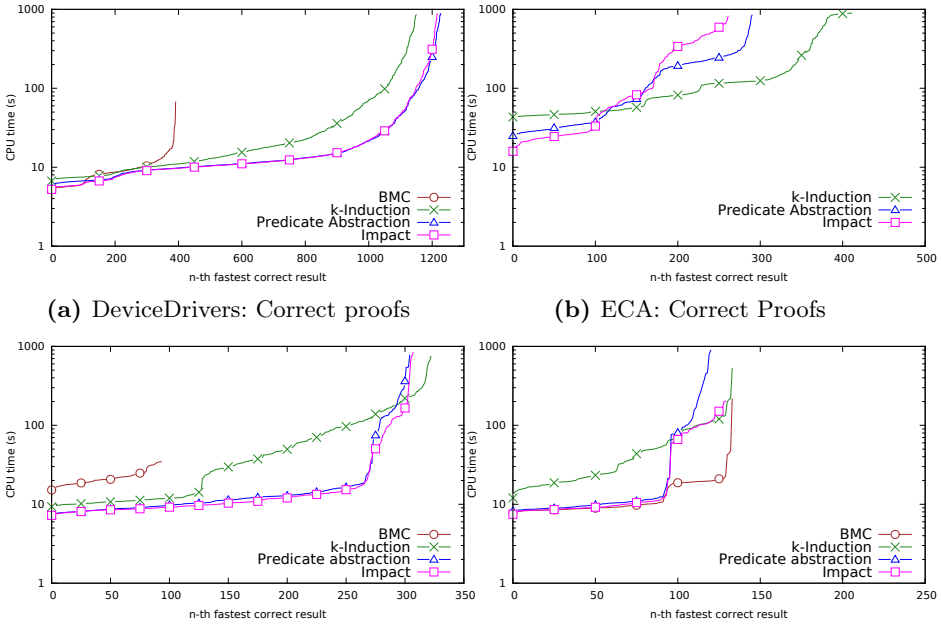
[4] For BMC, real proofs are accomplished by successful forward-condition checks, which prove that no further unrolling is required to exhaustively explore the state space.



(a) Proofs **(b)** Alarms

Fig. 7: Quantile plots for all correct proofs and alarms

inconclusive results are caused by crashes, for example if an algorithm encounters an unsupported feature, such as recursion or large arrays. For k-induction, there is sometimes a chance that while other techniques must give up due to such an unsupported feature, waiting for the invariant generator to generate a strong invariant will help avoid the necessity of handling the problem, which is why k-induction has fewer crashes but instead more timeouts than the other algorithms. The quantile plots in Fig. 7 shows the accumulated number of successfully solved tasks within a given amount of CPU time. A data point (x, y) of a graph means that for the respective configuration, x is the number of correctly solved tasks with a run time of less than or equal to y seconds. As expected, bounded model checking produces both the fewest correct proofs and the most correct alarms, confirming BMC's reputation as a technique that is well-suited for finding bugs. Having the fewest amount of solved tasks, BMC also accumulates the lowest total CPU time for correct results. Its average CPU time is on par with the abstraction techniques, because even though the approach is less powerful than the other algorithms, it still is expensive, because it has to completely unroll loops. On average, BMC spends 3.0 s on formula creation, 4.7 s on SMT-checking the forward condition, and 13 s on SMT-checking the feasibility of error paths. The slowest technique by far is k-induction with continuously-refined invariant generation, which is the only technique that effectively uses both available cores by running the auxiliary-invariant generation in parallel to the k-induction procedure, thus almost spending twice as much CPU time as the other techniques. Like BMC, k-induction also does not use abstraction and spends additional time on building the step-case formula and generating auxiliary invariants, but can often prove safety by induction without unrolling loops. Considering that over the whole benchmark set, k-induction generates the highest number of correct results, the additional effort appears to be mostly well spent. On average, k-induction spends 4.4 s on formula creation in the base case, 4.2 s on SMT-checking the forward condition, 4.8 s on SMT-checking the feasibility of error paths, 22 s on creating the step-case formula, 21 s on SMT-checking inductivity, and 11 s on generating auxiliary invariants, which shows that much more effort is required in the inductive step-case than in the base case. Predicate abstraction and the Impact algorithm both perform very similarly for finding proofs, which matches the observations from earlier work [14]. An interesting difference is



(a) DeviceDrivers: Correct proofs

(b) ECA: Correct Proofs

(c) ProductLines: Correct Proofs

(d) ProductLines: Correct Alarms

Fig. 8: Quantile plots for some of the categories

that the Impact algorithm finds more bugs. We attribute this observation to the fact that abstraction in the Impact algorithm is lazier than with predicate abstraction, which allows Impact larger parts of the state space in a shorter amount of time than predicate abstraction, causing Impact to find bugs sooner. For verification tasks without specification violations, however, the more eager predicate-abstraction technique pays off, because it requires fewer recomputations. Although in total, both abstraction techniques have to spend the same effort, this effort is distributed differently across the various steps: While, on average, predicate abstraction spends more time on computing abstractions (21 s) than the Impact algorithm (7.5 s), the latter requires the relatively expensive forced-covering step (13 s on average).

Although the plot in Fig. 7a suggests that k-induction with continuously-refined invariants outperforms the other techniques in general for finding proofs, a closer look at the results in individual categories, some of which are displayed in Fig. 8, reveals that how well an algorithm performs strongly depends on the type of verification task, but also reconfirms the observation of Fig. 7b that BMC consistently performs well for finding bugs. For example, on the safe tasks of the category on Linux device drivers in Fig. 8a, k-induction performs much worse than predicate abstraction and Impact. These device drivers are often C programs with many lines of code, containing pointer arithmetics and complex data structures. The interval-based auxiliary-invariant generator that we used for k-induction is not a good fit for such kinds of problems, and a lot of effort is wasted, while the abstraction techniques are often able to quickly determine that many operations on pointers and complex data structures are irrelevant to the

safety property. We did not include the plot for the correct alarms in the category on device drivers, because each of the algorithms only solves about 20 tasks, and although k-induction and BMC are slower than the abstraction techniques, which matches the previous observations on the correct proofs, there is not enough data among the correct alarms to draw any conclusions. The quantile plot for the correct proofs in the category of event condition action systems (ECA) is displayed in Fig. 8b. BMC in not included in this figure, because there is no single task in the category it could unroll exhaustively. These tasks usually only consist of a single loop, but each of these loops contains very complex branching structures over many different integer variables, which leads to an exponential explosion of paths, so unrolling them is very expensive in terms of time and memory. Also, because in many tasks, almost all of the variables are in some way relevant to the reachability of the error location within this complex branching structure, the abstraction techniques are unable to come up with useful abstractions, and perform badly. The interval-based auxiliary-invariant generator that we use for k-induction, however, appears to provide invariants useful for handling the complexity of the control structures, so that k-induction performs much better than all other techniques in this category. We did not include the plot for the correct alarms in this category, because the abstraction techniques are not able to detect a single bug, and only BMC and k-induction detect one single bug for the same task, namely `Problem10_label46_false-unreach-call.c`. Fig. 8c shows the quantile plot for correct proofs in the category on product lines. Similar to the proofs over all categories depicted in Fig. 7a, k-induction solves about as more tasks than the other techniques, but is becomes even more apparent how much slower than the other techniques it is. Fig. 8d shows the quantile plot for correct alarms in the same category. It is interesting to observe that the Impact algorithm distinctly outperforms predicate abstraction on the tasks requiring over 100 s of CPU time, whereas in the previous plots, the differences between the two abstraction techniques were hardly visible. While, ash shown in Fig. 8c, both techniques report almost the same amount of correct proofs (305 for predicate abstraction, 308 for Impact), Impact detects 130 bugs, whereas predicate abstraction detects only 121. This seems to indicate that the state-space spanned by the different product-line features can be explored more quickly by lazy abstraction of Impact than with the more eager predicate abstraction.

Individual Examples. The previous discussion showed that while overall, the algorithms perform rather similar (apart from BMC being inappropriate for finding proofs, which is expected), each of them has some strengths due to which it outperforms the other algorithms on certain programs. In the following, we will list some examples from our benchmark set that were each solved by one of the algorithms, but not by the others, and give a short explanation of the reasons.

BMC. For example, only BMC can find bugs in the verification tasks `cs_lazy_false-unreach-call.i` and `rangesum40_false-unreach-call.i`. Surprisingly, by exhaustively unrolling a loop, BMC is the only of our four techniques that is able to prove safety for the tasks `sep20_true-unreach-call.i` and `cs_stateful_true-unreach-call.i`. All four of these tasks have in com-

mon that they contain bounded loops and arrays. The bounded loops are a good fit for BMC and enable it to prove correctness, while the arrays make it hard in practice for predicate abstraction and Impact to find good abstractions by interpolation. k-induction, which in theory is at least as powerful as BMC, spends too much time trying to generate auxiliary invariants and exceeds the CPU time limit before solving these tasks.

k-induction. k-induction is the only of our four techniques to prove the correctness of all of the safe tasks in the (non-simplified) `ssh` subset of our benchmark set, while none of the other three techniques can solve any of them. These tasks encode state machines, i.e., loops over switch statements with many cases, which in turn modify the variable that is considered by the switch statement. These loops are unbounded, so that BMC cannot exhaustively unroll them, and the loop invariants that are required to prove correctness of these tasks need to consider the different cases and their interaction across consecutive loop iterations, which is beyond the scope of the abstraction techniques but very easy for k-induction (cf. [8] for a detailed discussion of a similar example).

Predicate Abstraction. `toy_true-unreach-call_false-termination.cil.c` is a task that is only solved by predicate abstraction but by none of our other implementations. It consists of an unbounded loop that contains complex branching structure over integer variables, most of which only ever take the values `0`, `1` or `2`. Interpolation quickly discovers the abstraction predicates over these variables required to solve the task, but in this example, predicate abstraction profits from eagerly computing a sufficiently precise abstraction early after only 9 refinements while the lazy refinement technique used by Impact exceeds the time limit after 129 refinements, and the invariant generator used by k-induction fails to find the required auxiliary invariants before reaching the time limit.

Impact. The task `Problem05_label50_true-unreach-call.c` from the ECA subset of our benchmark set is only solved by Impact: BMC fails on this task due to the unbounded loop, and the invariant generator used by k-induction does not come up with any meaningful auxiliary invariants before exceeding the time limit. Predicate abstraction exceeds the time limit after only three refinements, and up to that point, over 80% of its time is spent on eagerly computing abstractions. The lazy abstraction performed by Impact, however, allows it to progress quickly, and the algorithm finishes after 7 refinements.

4 Conclusion

This paper presents an overview over four state-of-the-art algorithms for SMT-based software model checking. First, we give a short explanation of each algorithm and illustrate the effect on how the state-space exploration looks like. Second, we provide the results of a thorough experimental study on a large number of verification tasks, in order to show the effect and performance of the different approaches, including a detailed discussion of particular verification tasks that can be solved by one algorithms while all others fail. In conclusion, there is no

clear winner: there are disadvantages and advantages for each approach. We hope that our experimental overview is useful to understand the difference of the algorithms and the potential application areas.

Future Work. In our comparison, one well-known algorithm is missing: PDR (property-driven reachability) [16]. We plan to formalize this algorithm in our framework and implement it in CPAchecker as well.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV*, LNCS 7358, pages 672–678. Springer, 2012.
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. T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
6. B. Beckert and R. Hähnle. Reasoning and verification: State of the art and current trends. *IEEE Intelligent Systems*, 29(1):20–29, 2014.
7. D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses. In *Proc. TACAS*, pages 887–904. Springer, 2016.
8. D. Beyer, M. Dangl, and P. Wendler. Boosting k-induction with continuously-refined invariants. In *Proc. CAV*, LNCS 9206, pages 622–640. Springer, 2015.
9. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
10. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
11. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
12. D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS 9232, pages 160–178. Springer, 2015.
13. D. Beyer and A. K. Petrenko. Linux driver verification. In *Proc. ISoLA*, LNCS 7610, pages 1–6. Springer, 2012.
14. D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. impact. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012.
15. 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.
16. A. R. Bradley. SAT-based model checking without unrolling. In *Proc. VMCAI*, LNCS 6538, pages 70–87. Springer, 2011.
17. M. Brain, S. Joshi, D. Kröning, and P. Schrammel. Safety verification and refutation by k-invariants and k-induction. In *Proc. SAS*, pages 145–161. Springer, 2015.
18. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

19. E. M. Clarke, D. Kröning, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. TACAS*, LNCS 2988, pages 168–176. Springer, 2004.
20. L. Cordeiro, J. Morse, D. Nicole, and B. Fischer. Context-bounded model checking with ESBMC 1.17 (competition contribution). In *Proc. TACAS*, LNCS 7214, pages 534–537. Springer, 2012.
21. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
22. 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.
23. M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. Handling loops in bounded model checking of c programs via k-induction. *STTT*, pages 1–18, 2015.
24. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
25. A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A framework for verifying C programs (competition contribution). In *Proc. TACAS*, LNCS 9035, pages 447–450. Springer, 2015.
26. M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski. Ultimate Automizer with two-track proofs (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 950–953. Springer, 2016.
27. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
28. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4), 2009.
29. 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.
30. 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.
31. G. A. Kildall. A unified approach to global program optimization. In *Proc. POPL*, pages 194–206. ACM, 1973.
32. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
33. K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.
34. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
35. Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In *Proc. CAV*, LNCS 8559, pages 106–113. Springer, 2014.
36. H. Rocha, H. I. Ismail, L. C. Cordeiro, and R. S. Barreto. Model checking embedded C software using k-induction and invariants. In *Proc. SBESC*. IEEE, 2015.
37. P. Schrammel and D. Kröning. 2LS for program analysis (competition contribution). In *Proc. TACAS*, LNCS 9636, pages 905–907. Springer, 2016.
38. V. Schuppan and A. Biere. Liveness checking as safety checking for infinite state spaces. *Electr. Notes Theor. Comput. Sci.*, 149(1):79–96, 2006.
39. C. Sinz, F. Merz, and S. Falke. LLBMC: A bounded model checker for LLVM's intermediate representation (competition contribution). In *Proc. TACAS*, LNCS 7214, pages 542–544. Springer, 2012.

40. T. Wahl. The k-induction principle, 2013. Available at `http://www.ccs.neu.edu/home/wahl/Publications/k-induction.pdf`.
41. P. Wendler. CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In *Proc. TACAS*, LNCS 7795, pages 613–615. Springer, 2013.