Augmenting Interpolation-Based Model Checking with Auxiliary Invariants

Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee

LMU Munich, Munich, Germany

Abstract. Software model checking is a challenging problem, and generating relevant invariants is a key factor in proving the safety properties of a program. Program invariants can be obtained by various approaches, including lightweight procedures based on data-flow analysis and intensive techniques using Craig interpolation. Although data-flow analysis runs efficiently, it often produces invariants that are too weak to prove the properties. By contrast, interpolation-based approaches build strong invariants from interpolants, but they might not scale well due to expensive interpolation procedures. Invariants can also be injected into model-checking algorithms to assist the analysis. Invariant injection has been studied for many well-known approaches, including k-induction, predicate abstraction, and symbolic execution. We propose an augmented interpolation-based verification algorithm that injects external invariants into *interpolation-based model checking* (McMillan, 2003), a hardware model-checking algorithm recently adopted for software verification. The auxiliary invariants help prune unreachable states in Craig interpolants and confine the analysis to the reachable parts of a program. We implemented the proposed technique in the verification framework CPAchecker and evaluated it against mature SMT-based methods in CPAchecker as well as other state-of-the-art software verifiers. We found that injecting invariants reduces the number of interpolation queries needed to prove safety properties and improves the run-time efficiency. Consequently, the proposed invariant-injection approach verified difficult tasks that none of its plain version (i.e., without invariants), the invariant generator, or any compared tools could solve.

Keywords: Software model checking · Program invariants · Invariant injection · Craig interpolation · Data-flow analysis · SMT · SAT

1 Introduction

Assuring that programs execute correctly with respect to their specifications is fundamental for deploying them in our daily lives. In the software industry, testing [2] is the most popular methodology to validate the quality of programs. However, software testing can only spot the presence of bugs in programs but not guarantee their absence. To prove the correctness of programs with mathematical rigorousness, automatic software model checking [3] has been extensively studied.

An extended version of this article is available on arXiv [1].

One of the greatest challenges to formally verify a program is to deduce suitable *program invariants* that can be used to prove the program's safety properties. A program invariant is a logical formula over some program variables that must hold at a certain program location for all feasible program paths.

There are numerous approaches to invariant generation, with different performance characteristics and strengths of the produced invariants. Data-flow analysis [4, 5, 6, 7, 8, 9, 10] is a category of methods that sacrifice path sensitivity for scalability. To reduce the size of the abstract reachability graph, data-flow analysis merges abstract program states arising from different execution paths when the control flow meets. Since merging abstract states usually loses information, the resulting invariants might not be strong enough to prove the safety properties. In contrast to data-flow analysis, techniques [11, 12, 13, 14, 15, 16] based on Craig interpolation [17] iteratively derive interpolants[1] to construct strong invariants. Although these approaches are capable of generating useful invariants, they might take too many refinement iterations and fail to converge.

To leverage the information carried by invariants, researchers also use invariant generators as an auxiliary component to aid various model-checking techniques. Algorithms that have been augmented by *invariant injection* include bounded model checking (BMC) [18, 19, 20], k-induction [21, 22, 23, 24, 25], predicate abstraction [26, 27], symbolic execution [28], and IC3/PDR [29]. The idea is to confine the scope of a verification algorithm to the reachable state space of a program, as safety properties often do not hold in the unreachable parts [22].

1.1 Our Research Goal

Despite the extensive study of injecting invariants to various software-verification methods, the possibilities of leveraging auxiliary invariants to assist *interpolation-based model checking* [11], a hardware model-checking algorithm published by McMillan in 2003 and adopted to software recently [30], have not been explored yet. Although this algorithm was invented two decades ago, it still remains state-of-the-art for safety verification. In this paper, our goal is to find out *whether invariant injection can enhance McMillan's interpolation-based model-checking algorithm from 2003 for software verification.* In particular, we aim to reduce the number of interpolation queries required to prove the safety property of a program, since interpolation is usually the most time-consuming step in interpolation-based algorithms. To avoid confusion between McMillan's algorithm from 2003 [11] and other interpolation-based verification approaches, we refer to McMillan's approach from 2003 as IMC from now on. IMC has been combined with expensive SAT-based invariant generators for hardware model checking [31, 32], but its characteristics when assisted by lightweight invariant generators [33] based on data-flow analysis remain unknown for software verification.

To motivate how auxiliary invariants could assist IMC, we briefly introduce the algorithm first. IMC extends BMC via constructing inductive invariants by Craig interpolants [17] arising from unsatisfiable BMC queries. Whenever a BMC

[1] An interpolant τ for an implication $A \Rightarrow B$ is a formula that uses only the common variables of formulas A and B such that $A \Rightarrow \tau$ and $\tau \Rightarrow B$ are valid.

query starting from initial states is unsatisfiable, IMC derives an interpolant from the query and replaces the initial states in the query with the interpolant. If the new query is unsatisfiable again, another interpolant can be obtained. The process is repeated until either (1) the newest interpolant is contained in the union of initial states and previous interpolants, or (2) the query starting from the newest interpolant becomes satisfiable. In case (1), an inductive invariant (i.e., a fixed point) is found and the property is proven. In case (2), the satisfied query might be a spurious counterexample, so IMC will continue to unfold the program.

Interpolants tend to abstract away irrelevant information and may intersect with the unreachable state space. Therefore, conjoining them with auxiliary invariants prunes away some unreachable states. In the IMC algorithm outlined above, if the newest interpolant is strengthened by an auxiliary invariant, it is more likely to be contained in the union of initial states and previous interpolants, and the query starting from a strengthened interpolant is less likely to be satisfiable. In other words, a fixed point might be found with fewer interpolation queries in case (1), and a spurious alarm plus extra program unrollings could be avoided in case (2). An example will be shown in Sect. 1.3 to illustrate these benefits.

1.2 Our Contributions

Augmenting IMC with Auxiliary Invariants. We devise two methods to augment IMC with auxiliary invariants. The first method confines the containment check between the newest interpolant and the union of initial states and previous interpolants with auxiliary invariants. The strengthened check is more likely to succeed, so a fixed point can be found with fewer unrollings and interpolation calls. The second method conjoins derived interpolants with auxiliary invariants. The BMC query starting from a strengthened interpolant is more likely to be unsatisfiable, and more interpolants can be derived to form a fixed point before IMC further unrolls the program. We rigorously prove the correctness of the proposed techniques. The proposed augmenting approaches are **novel** because they are the first invariant-injection techniques for IMC applied to program analysis. Moreover, the theoretical results in this paper are applicable to IMC for hardware verification even though we focus on software verification.

Open-Source Implementation and Extensive Evaluation. We implemented the proposed approaches in the open-source framework CPAchecker [34] and conducted an extensive evaluation on more than 1 600 difficult verification tasks of C programs from the 2022 Intl. Competition on Software Verification [35].[2] In our experiments, the *plain* IMC algorithm (i.e., without invariant injection), three other mature SMT-based algorithms in CPAchecker, and two other software verifiers were used as references to evaluate our implementation. Our experimental results show that invariant injection (1) effectively reduces the numbers of program unrollings and interpolation queries needed by plain IMC to prove safety properties, (2) reduces the wall-time usage, (3) proves 16 tasks that neither plain IMC nor the data-flow analyzer used to generate invariants could solve, and

[2] Due to space limitation, the pseudocode of the proposed approaches and some experimental results are not included in this article, but in the extended version [1].

(4) outperformed other well-established software verifiers. These observations are **significant** because they enhance the knowledge about the effect of auxiliary invariants on IMC. Furthermore, our open-source implementation of the proposed approaches helps other researchers to understand the details of the algorithms better and provides a solid baseline for future studies.

1.3 Motivating Example

We use the example C program in Fig. 1 to explain why invariant injection can reduce the numbers of program unrollings and interpolation queries for IMC. In the program, variables `x` and `i` are both initialized to 0. The loop will be executed nondeterministically many times depending on the values returned by function `nondet()`. The values of `x` and `i` are modified in each loop iteration: `x` will be incremented by 2; if `i` equals 3, `x` will be further incremented by 1. Variable `i` will be incremented by 1 and reset to 0 if its value equals 2 after being incremented. An error occurs if `x` is odd when the control flow exits the loop.

```
1  extern int nondet();
2  int main(void) {
3    unsigned int x = 0;
4    unsigned int i = 0;
5    while (nondet()) {
6      x += 2;
7      if (i == 3) x++;
8      i++;
9      if (i == 2) i = 0;
10   }
11   if (x % 2) {
12     ERROR: return 1;
13   }
14   return 0;
15 }
```

Fig. 1: An example C program to motivate how auxiliary invariants help IMC

IMC first checks if the error location line `12` is reachable by skipping the loop (i.e., assuming `nondet()` returns 0). As the conjunction of the initial states $x = 0 \wedge i = 0$ and the guard to the error location $x\%2 \neq 0$ is unsatisfiable, namely, line `12` is unreachable if the loop is not entered, IMC will try to build an inductive invariant at the loop head line `5` to prove that all paths entering the loop cannot reach the error location. IMC unrolls the program and builds a BMC query[3] starting from the initial location line `3`, going through the loop once (i.e., visiting line `5` twice), and ending at the error location line `12`. This BMC query is unsatisfiable, so an interpolant at the loop head line `5` can be derived to overapproximate the program states reachable after one loop iteration. Assume the interpolant is $x\%2 = 0$ (instantiated with variables of the initial states). IMC will replace the initial states $x = 0 \wedge i = 0$ in the BMC query by $x\%2 = 0$ and pose another query to derive the next interpolant. Unfortunately, the new query is satisfiable, indicating that there exists a feasible path from line `5` to the error location, which assumes `x` to be even at the beginning of the path and goes through the loop body once. Indeed, a solution to the new query is $(x, i) = (0, 3)$, which leads to $x = 3$ after exiting the loop. To decide whether this solution is a spurious counterexample or not, IMC has to further unroll the program and fails to converge to a fixed point under the current unrolling. The

[3] The query is $(x = 0 \wedge i = 0) \wedge (i = 3 \Rightarrow x' = x + 3) \wedge (i \neq 3 \Rightarrow x' = x + 2) \wedge (i + 1 = 2 \Rightarrow i' = 0) \wedge (i + 1 \neq 2 \Rightarrow i' = i + 1) \wedge (x'\%2 \neq 0)$, where the prime symbols indicate variables after a loop iteration, and the returned values of function `nondet()` are omitted for simplicity.

reason behind this situation is that the interpolant $x\%2 = 0$ contains unreachable program states allowing the infeasible assignment $i = 3$. In fact, the safety property of the program is fulfilled as variable i never grows beyond 2, and the problematic statement `x++;` at line `7` is unreachable.

By contrast, it is easy for data-flow analysis, e.g., one based on the abstract domain of intervals [33], to identify that $0 \leq i \leq 1$ is an invariant at line `5` of the program. If this invariant is injected to strengthen the interpolant $x\%2 = 0$, IMC reaches a fixed point $0 \leq i \leq 1 \wedge x\%2 = 0$ immediately, without further unrolling the program. This reasoning is confirmed by our implementation in CPAchecker, which injects auxiliary invariants produced by a continuously-refining interval analysis [33] into the plain IMC algorithm [30]. The table below summarizes the first interpolants obtained at line `5` and the numbers of unrollings and interpolation queries needed to prove the program in Fig. 1 for the plain and augmented IMC implementations in CPAchecker.

Algorithm	First interpolant at line `5`	Fixed point	#Unrolling	#Itp-queries
Plain IMC [30]	$x\%2 = 0$	No	3	7
Augmented IMC	$0 \leq i \leq 1 \wedge x\%2 = 0$	Yes	1	2

2 Related Work

Our paper is primarily related to invariant generation and verification approaches aided by auxiliary invariants.

2.1 Invariant Generation

Various approaches exist for invariant generation, ranging from lightweight ones based on data-flow analysis to computationally expensive ones based on SAT/SMT solving, predicate abstraction, or Craig interpolation.

Data-flow analysis has been extensively studied [5, 6, 7, 8, 9, 10]. Classic approaches usually consider program variables over an abstract domain structured as a semi-lattice. A standard fixed-point procedure is used to iteratively explore a program, and abstract states reached by different program traces are merged. When the procedure finishes, the merged abstract state at each program location contains the corresponding program invariant. Common choices of abstract domains include a value domain [36, 37] or an interval domain [38].

Interpolation-based approaches construct invariants by either predicates collected from interpolants [12] or interpolants themselves [11, 13, 14, 15]. Although they are able to build strong invariants, the expensive interpolation queries might hinder their scalability. Iterative SAT solving is used to prove the inductiveness of candidate invariants extracted from simulation data [31]. Invariant generation can also be guided by the safety property [39, 40] or the syntax of the program [41].

2.2 Invariant-Aided Verification

Injecting invariants to enhance model-checking algorithms is a popular technique. There are techniques to restrict the state space in BMC queries by high-level design information [19] or data mining [20]. It is well known that the induction hypothesis

of k-induction is often too weak on its own and has to be strengthened by auxiliary invariants from static analysis [21, 23] or property-directed reachability [25, 42]. Alternatively, k-induction can be used to prove candidate invariants instantiated from a template, and the confirmed invariants can be used in an induction proof [43]. Recently, an interval-based analysis that produces continuously-refined invariants is successfully applied to boost the performance of k-induction [22]. Transition relations can be strengthened by invariants from the octagon abstract domain to reduce the number of refinement loops in predicate abstraction [27]. Loop invariants can be used as annotations in symbolic execution [28]. Predicate abstraction is also employed to improve candidate invariants formed by Craig interpolants to reduce the number of refinements [44]. In hardware model checking, IMC is combined with inductive invariants to avoid spurious counterexamples [32]. Different from continuously-refined invariants commonly used in software verification, this approach computes a single invariant by SAT solving and uses it repeatedly throughout the entire verification process. Spacer [45], the backend engine of the verification framework SeaHorn [29], uses invariants produced by the abstract-interpretation tool Ikos [46] to speed up the analysis.

3 Background

In this section, we provide necessary background knowledge for the rest of the paper. All used logical formulas are quantifier-free and belong to the first-order theory of equality with uninterpreted functions, arrays, bit-vectors, and floats. We consider the satisfiability and validity of a logical formula with respect to this theory. A first-order predicate over state variables is interpreted interchangeably as a set of system states that satisfy the predicate.

3.1 Model Checking

First, we recap the problem formulation of model checking. To simplify the presentation in Sect. 4, we describe model checking with the notation of state-transition systems and view a program as a state-transition system. The implementation of the proposed methods, discussed in Sect. 5, represents a program as a control-flow automaton and verifies C programs.

Describing State-Transition Systems. A state-transition system M is characterized by two predicates $I(s)$ and $T(s, s')$ over state variables: $I(s)$ evaluates to true if state s is an initial state of M; $T(s, s')$ evaluates to true if M can transit from state s to state s'. In the following, a state-transition system is represented by $M = (I(s), T(s, s'))$ and state variables after a transition are denoted with a prime. We write $R(s)$ to represent the set of all reachable states of M.

Verifying Safety Properties. Model checking concerns if a state-transition system fulfills a safety property. A safety property can be described as a predicate $P(s)$ that evaluates to true if state s satisfies the property. Given a state-transition system $M = (I(s), T(s, s'))$ and a safety property $P(s)$, M fulfills $P(s)$ if $R(s) \Rightarrow P(s)$ is valid, namely, the property is satisfied by all reachable states of M. Model-checking algorithms receive $M = (I(s), T(s, s'))$ and $P(s)$ as input and aim at proving or disproving $R(s) \Rightarrow P(s)$.

We discuss two criteria widely used by model-checking algorithms to establish $R(s) \Rightarrow P(s)$. First, some approaches construct an *inductive* set $F(s)$ that implies $P(s)$. That is, $F(s)$ must conform to the constraint below:

$$(I(s) \Rightarrow F(s)) \wedge (F(s) \wedge T(s, s') \Rightarrow F(s')) \wedge (F(s) \Rightarrow P(s)).$$

Since $R(s)$ is the smallest inductive set of M, we know that $R(s) \Rightarrow F(s)$ holds. The implication $R(s) \Rightarrow P(s)$ follows from $R(s) \Rightarrow F(s)$ and $F(s) \Rightarrow P(s)$. Interpolation-based approaches such as IMC [11] and interpolation-sequence-based model checking [14] fall into this category because they try to build an inductive state set $F(s)$ from Craig interpolants [17].

The second criterion takes advantage of *invariants*. An invariant $Inv(s)$ of M is a predicate that holds for all reachable states of M, namely, $R(s) \Rightarrow Inv(s)$ is valid. Given an invariant $Inv(s)$ of M, some methods produce a set $G(s)$ that is *relatively inductive* to $Inv(s)$ and implies $P(s)$. In other words, $G(s)$ must fulfill:

$$(I(s) \Rightarrow G(s)) \wedge (Inv(s) \wedge G(s) \wedge T(s, s') \Rightarrow G(s')) \wedge (G(s) \Rightarrow P(s)). \quad (1)$$

Because $Inv(s)$ holds for every reachable state and $G(s)$ is relatively inductive to $Inv(s)$, we have $R(s) \Rightarrow G(s)$. The implication $R(s) \Rightarrow P(s)$ is thus concluded from $R(s) \Rightarrow G(s)$ and $G(s) \Rightarrow P(s)$. IC3/PDR [40] is a prominent example based on this criterion. It chooses $G(s)$ to be the safety property $P(s)$ and generates clauses to form an invariant to which $P(s)$ is relatively inductive. As will be seen in Sect. 4, instead of producing invariants internally during model checking, we leverage external invariant generators and augment IMC to compute a relatively inductive set to the auxiliary invariants.

3.2 Interpolation-Based Model Checking

IMC is an algorithm proposed by McMillan in 2003 [11]. Originally designed for hardware model checking, it has been adopted to verify software programs recently [30]. IMC extends BMC by constructing inductive invariants (i.e., fixed points) from *Craig interpolants* [17].

Craig Interpolation. Given two formulas A and B, if $A \Rightarrow B$, Craig's interpolation theorem [17] assures the existence of a formula τ such that $A \Rightarrow \tau$ and $\tau \Rightarrow B$ are valid, and τ only involves variables appearing in both A and B. τ is called an *interpolant* of A and B as it is logically between A and B. In the model-checking community, Craig's interpolation theorem is usually stated in the equivalent form below.

Theorem 1. *Given an unsatisfiable formula $A \wedge B$, there exists an interpolant τ of this formula such that (1) $A \Rightarrow \tau$ is valid, (2) $\tau \wedge B$ is unsatisfiable, and (3) τ only refers to the common variables of A and B.*

Computational Stages in IMC. IMC [11] has two nested stages in its computation: The outer stage unrolls a state-transition system and poses BMC queries to satisfiability solvers; the inner stage derives Craig interpolants and constructs

fixed points. In the following, we name the outer stage *BMC stage* and the inner stage *interpolation stage*.

In the BMC stage, a state-transition system is unfolded into several copies, which is controlled by an unrolling counter. Suppose the value of the counter is k. A BMC query depicting all possible paths from an initial state to a property-violating state via at most k transitions is posed:

$$\underbrace{I(s_0)T(s_0,s_1)}_{A(s_0,\,s_1)}\underbrace{T(s_1,s_2)\ldots T(s_{k-1},s_k)(\neg P(s_1)\vee\ldots\vee\neg P(s_k))}_{B(s_1,\,s_2,\,\ldots,\,s_k)}. \tag{2}$$

In the above formula, variable s_i denotes the state variable after the i-th transition. If Eq. (2) is satisfiable, a feasible counterexample to the safety property is found. Otherwise, IMC enters the interpolation stage.

In the interpolation stage, a BMC query such as Eq. (2) is partitioned into two formulas A and B. According to Theorem 1, an interpolant $\tau_1(s_1)$ for Eq. (2) exists such that:

$$I(s_0)T(s_0,s_1)\Rightarrow\tau_1(s_1)\text{ is valid and}$$
$$\tau_1(s_1)\wedge\bigwedge_{i=1}^{k-1}T(s_i,s_{i+1})\wedge\bigvee_{i=1}^{k}\neg P(s_i)\text{ is unsatisfiable.}$$

Conceptually, $\tau_1(s_1)$ summarizes the reason why Eq. (2) is unsatisfiable: It over-approximates the set of states that are (1) reachable via one transition from an initial state and (2) do not violate the safety property within $(k-1)$ transitions.

A fixed point of the state-transition system may be constructed by computing such interpolants iteratively. Substituting variable s_0 into τ_1 and replacing the initial states $I(s_0)$ in Eq. (2) by $\tau_1(s_0)$, the interpolation stage of IMC poses another BMC query from the first interpolant τ_1. If the formula is still unsatisfiable, a second interpolant $\tau_2(s_1)$ can be derived, which overapproximates the set of states reachable from an initial state via two transitions. Replacing τ_1 with τ_2, one can derive the next interpolant τ_3 if the BMC query starting from τ_2 is again unsatisfiable. Suppose the above process is repeated n times, and a list of interpolants $\tau_1,\tau_2,\ldots,\tau_n$ is derived. Upon the derivation of the newest interpolant τ_n, IMC performs a *fixed-point check* to decide whether a fixed point has been reached: If τ_n is contained in the union of the initial states and previous interpolants, namely, $\tau_n\Rightarrow I\vee\bigvee_{j=1}^{n-1}\tau_j$ holds, the set $I\vee\bigvee_{j=1}^{n-1}\tau_j$ is an inductive invariant of the system [11]. The inductive invariant is also safe because every interpolant does not violate the safety property according to the second condition of Theorem 1. As a result, IMC concludes the system preserves the safety property.

If a BMC query starting from an interpolant is satisfiable, the safety property is not definitely violated. The corresponding error path might be spurious because the interpolant could involve unreachable states. In this situation, IMC will return to the BMC stage and increment the unrolling counter to confirm the existence of a feasible error path.

4 Augmenting IMC with Auxiliary Invariants

We propose two methods to augment IMC with auxiliary invariants. The first one strengthens the fixed-point checks, and the second one strengthens the interpolants derived in the interpolation stage. Given a state-transition system $M = (I(s), T(s, s'))$, we further assume that auxiliary invariants are inductive. In other words, an invariant $Inv(s)$ satisfies $I(s) \Rightarrow Inv(s)$ and $Inv(s) \wedge T(s, s') \Rightarrow Inv(s')$. This assumption is reasonable in practice because invariant generators, such as those based on data-flow analysis, usually perform a fixed-point iteration and produce inductive invariants.

4.1 Approach 1: Strengthening Fixed-Point Checks

The first approach strengthens fixed-point checks by restricting the newest interpolant with an auxiliary invariant. Let $\tau_1, \ldots, \tau_n$ be a list of interpolants derived in the interpolation stage. Instead of checking $\tau_n \Rightarrow I \vee \bigvee_{j=1}^{n-1} \tau_j$, we strengthen the check to $Inv \wedge \tau_n \Rightarrow I \vee \bigvee_{j=1}^{n-1} \tau_j$. The intuition is to confine the scope of fixed-point checks mainly within reachable states of the system such that the union of the initial states and previous interpolants is more likely to contain the newest interpolant. The correctness of this approach is stated in Theorem 2.

Theorem 2. *Given a state-transition system $M = (I(s), T(s, s'))$ and a safety property $P(s)$, let $\tau_1, \ldots, \tau_n$ be a list of interpolants derived in the interpolation stage of IMC. If the strengthened fixed-point check $Inv \wedge \tau_n \Rightarrow I \vee \bigvee_{j=1}^{n-1} \tau_j$ holds for some auxiliary invariant Inv, M fulfills the safety property $P(s)$.*

Proof. We rely on the criterion described by Eq. (1) to show that M fulfills P. Defining G to be $I \vee \bigvee_{j=1}^{n-1} \tau_j$, we will prove that G satisfies Eq. (1).

Proving $I \Rightarrow G$ is trivial. Moreover, Theorem 1 guarantees that $\tau_1, \ldots, \tau_n$ do not violate the safety property, which assures that $G \Rightarrow P$ holds.

To show $Inv(s) \wedge G(s) \wedge T(s, s') \Rightarrow G(s')$ is valid, recall $I(s) \wedge T(s, s') \Rightarrow \tau_1(s')$ and $\tau_j(s) \wedge T(s, s') \Rightarrow \tau_{j+1}(s')$ for $j = 1, \ldots, n-1$ both hold according to Theorem 1. Combining these conditions and the inductiveness of the auxiliary invariant Inv, we simplify the implication to

$$Inv(s) \wedge (I(s) \vee \bigvee_{j=1}^{n-1} \tau_j(s)) \wedge T(s, s') \Rightarrow Inv(s') \wedge (\bigvee_{j=1}^{n-1} \tau_j(s') \vee \tau_n(s')).$$

Since the strengthened fixed-point check $Inv \wedge \tau_n \Rightarrow I \vee \bigvee_{j=1}^{n-1} \tau_j$ holds, the right-hand side of the above implication further implies $(Inv(s') \wedge \bigvee_{j=1}^{n-1} \tau_j(s')) \vee (I(s') \vee \bigvee_{j=1}^{n-1} \tau_j(s'))$, which equals $G(s')$.

Therefore, we proved that G satisfies Eq. (1), and hence M fulfills P.

4.2 Approach 2: Strengthening Interpolants

The second approach strengthens interpolants by conjoining them with an auxiliary invariant. Unlike the first approach, which only restricts the newest interpolant

with an auxiliary invariant in a fixed-point check, the second approach replaces the original interpolants returned by the interpolation procedure with the strengthened ones. That is, given a BMC query starting from the initial states I or a previously strengthened interpolant $\tau_j \wedge Inv$, the interpolant τ_{j+1} is replaced by $\tau_{j+1} \wedge Inv$. Note that if interpolants are strengthened, fixed-point checks are effectively strengthened as well. Theorem 3 states that $\tau_{j+1} \wedge Inv$ is also an interpolant for the BMC query, and hence the correctness of the approach follows from the plain IMC algorithm.

Theorem 3. *Given a transition system $M = (I(s), T(s, s'))$ and a safety property $P(s)$, consider an unsatisfiable BMC query $\lambda(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=1}^{k} \neg P(s_i)$ posed in the interpolation stage of IMC, where $\lambda(s_0)$ is either $I(s_0)$ or a previously strengthened interpolant $\tau_j(s_0) \wedge Inv(s_0)$. Suppose $\tau_{j+1}(s_1)$ is an interpolant of the BMC query with $\lambda(s_0) \wedge T(s_0, s_1)$ labeled as formula A and $T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=1}^{k} \neg P(s_i)$ labeled as formula B. Then $\tau_{j+1}(s_1) \wedge Inv(s_1)$ is also an interpolant of the BMC query.*

Proof. To prove that $\tau_{j+1}(s_1) \wedge Inv(s_1)$ is also an interpolant, we have to show that it satisfies the three conditions in Theorem 1. Namely, (1) $\lambda(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_{j+1}(s_1) \wedge Inv(s_1)$ is valid; (2) $\tau_{j+1}(s_1) \wedge Inv(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=1}^{k} \neg P(s_i)$ is unsatisfiable; and (3) $\tau_{j+1}(s_1) \wedge Inv(s_1)$ only refers to the common variables of formulas A and B. Condition (2) holds because $\tau_{j+1}(s_1) \wedge T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \bigvee_{i=1}^{k} \neg P(s_i)$ is already unsatisfiable according to Theorem 1. Condition (3) holds because $\tau_{j+1}(s_1) \wedge Inv(s_1)$ only uses the common variable s_1. In the following, we prove condition (1) by splitting into two cases.

When $\lambda(s_0) = I(s_0)$, our proof goal is $I(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_{j+1}(s_1) \wedge Inv(s_1)$. Applying Theorem 1 to τ_{j+1}, we have $I(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_{j+1}(s_1)$. Since Inv is an invariant and hence contains all reachable states, we also have $I(s_0) \wedge T(s_0, s_1) \Rightarrow Inv(s_1)$. Therefore, our proof goal is achieved.

When $\lambda(s_0) = \tau_j(s_0) \wedge Inv(s_0)$, our proof goal is $\tau_j(s_0) \wedge Inv(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_{j+1}(s_1) \wedge Inv(s_1)$. Applying Theorem 1 to τ_{j+1}, we have $\tau_j(s_0) \wedge Inv(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_{j+1}(s_1)$. Since Inv is inductive, we also have $Inv(s_0) \wedge T(s_0, s_1) \Rightarrow Inv(s_1)$. Our goal follows from the two implications.

As both cases are proved, we conclude that $\tau_{j+1}(s_1) \wedge Inv(s_1)$ is also an interpolant of the BMC query and can be used in the interpolation stage of IMC.

4.3 Comparison between the Two Approaches

The design spirit behind the two approaches is to make IMC capable of proving the safety of a system with fewer iterations in the BMC and interpolation stages. We attain this objective with the help of auxiliary invariants. In the first approach, instead of an inductive invariant, IMC generates a relatively inductive set of states, which might require fewer interpolation calls to produce. In the second approach, in addition to strengthening fixed-point checks, strengthened interpolants also make BMC queries in the interpolation stage more likely to be unsatisfiable. That is, IMC will remain in the interpolation stage more often, searching for a proof of the safety property. The second approach is more "aggressive" than the first one

because strengthened interpolants will change the interpolants obtained later, and IMC will follow a different computational footprint. By contrast, the interpolants encountered in the first approach are identical to the plain IMC algorithm if the interpolation procedure is the same. Moreover, the first approach can in principle be adopted by other verification algorithms with similar fixed-point checks, whereas the second approach is specifically tailored towards IMC. In our evaluation, we also experimented with a basic injection approach that conjoins the safety property with auxiliary invariants. However, this approach performed roughly the same as the plain IMC algorithm.

5 Implementation

We implemented the two approaches discussed in Sect. 4 on top of the plain IMC algorithm in CPAchecker [34], a software-verification framework for the programming language C. The pseudo code of the implemented procedure is outlined in the extended technical report [1]. The plain IMC implementation [30] in CPAchecker extracts transition relations from programs with *large-block encoding* [47]. Error labels in a program are used to specify the negation of safety properties. In addition to an IMC implementation, CPAchecker also has a continuously-refining invariant generator based on intervals [33], which is used to enhance k-induction [22]. Before continuing the discussion, we emphasize that the proposed approaches discussed in Sect. 4 are independent of the implementation framework and not limited to specific invariant generators.

CPAchecker is based on *configurable program analysis* (CPA) [48], which defines an abstract domain used in a program analysis. The plain IMC algorithm is implemented using several CPAs, with one of them tracking path formulas between program locations in its abstract states [49]. This CPA is configured to perform large-block encoding, which constructs formulas for complete loop unrollings as transition relations [30]. Thanks to its flexibility, it underpins the implementations of other SMT-based algorithms in CPAchecker [50], including predicate analysis, Impact, and k-induction. We will compare the proposed algorithm to these techniques in Sect. 6.

The continuously-refining invariant generator in CPAchecker uses a CPA with an abstract domain based on expressions over intervals [33]. Compared to methods concerning single intervals only, it can represent complex ranges like the disjunction and conjunction of intervals, e.g., $(x < 5 \vee x > 7) \wedge (1 \leq y \leq 8)$. Its precision includes a set of important variables and a maximum depth of expressions. The precision can be dynamically adjusted during the analysis [51] to produce more refined invariants.

Given a CPA and an initial abstract state, CPAchecker constructs a set of reachable abstract states by iteratively processing states and computing their abstract successors. To perform a specific algorithm, required information can be collected from the abstract states. For example, IMC collects the path formulas to assemble BMC queries and derive interpolants [30]. The interval-based invariant generator outputs the union of expressions as invariants [33]. In our implemen-

tation, we run IMC and the invariant generator in parallel and inject auxiliary invariants to augment IMC.

6 Evaluation

To understand the effects of the proposed invariant-injection approaches on the proof-finding ability of IMC, we pose the following research questions:

- Part 1: augmented IMC vs. plain IMC
  - **RQ1**: Can auxiliary invariants reduce the numbers of program unrollings and interpolation queries?
  - **RQ2**: Can augmented IMC prove the correctness of additional programs?
  - **RQ3**: Can invariant injection improve the run-time efficiency?
- Part 2: augmented IMC vs. other approaches and tools
  - **RQ4**: Can augmented IMC find more proofs than other state-of-the-art software-verification algorithms and tools?

6.1 Evaluated Approaches and Tools

To answer the above research questions, we evaluated the proposed invariant-injection methods for IMC against (1) its plain version plus three other SMT-based algorithms in the same verification framework CPAchecker [34] and (2) two other state-of-the-art software verifiers.

The plain IMC algorithm was recently adopted to verify programs [30]. The three other approaches in CPAchecker are predicate abstraction [12, 52], Impact [13], and k-induction boosted by continuously-refined invariants [22], whose characteristics were compared in a recent article on SMT-based software verification [50]. All proposed methods and compared algorithms above are implemented in the same framework, so the confounding variables are kept to a minimum (identical frontend, program encodings, SMT solvers, etc.) to facilitate the comparison of algorithmic differences. For invariant generation, we used the continuously-refining data-flow analysis (DF) [33] described in Sect. 5. The invariant generator can prove the safety properties of some programs on its own. In our experiments, we ignored the answers computed by DF because our goal is to study the effects of auxiliary invariants on the main analyses, namely, IMC and k-induction. In the following, we denote the injection of continuously-refined invariants generated by DF into k-induction as KI$\leftarrow\!\!\circlearrowleft$DF, and the augmented IMC algorithms with fixed-point checks and interpolants strengthened by auxiliary invariants as IMC$_{\mathbf{f}}\leftarrow\!\!\circlearrowleft$DF and IMC$_{\mathbf{i}}\leftarrow\!\!\circlearrowleft$DF, respectively.

To reflect the state of the art on software verification and invariant injection, we further compared our approaches to 2ls [23], which has a mature implementation of k-induction boosted by auxiliary invariants, and Symbiotic [53], which is based on symbolic execution and performs well in the Intl. Competitions on Software Verification [35] (the overall winner in 2022).

6.2 Benchmark Set

Since our goal is to investigate whether auxiliary invariants can improve IMC's capability of delivering correctness proofs, we selected verification tasks without known violation to their reachability-safety properties from the 2022 Intl. Competition on Software Verification (SV-COMP '22) [35]. Furthermore, to concentrate the evaluation on hard verification problems, we excluded the trivial tasks solvable by BMC of CPAchecker. To keep the program encodings consistent across all evaluated approaches in CPAchecker, we only considered single-loop programs in our experiments because the IMC implementation in CPAchecker needs to transform a multi-loop program into a single-loop program as preprocessing [30]. However, this is not a limitation of the proposed augmenting methods, since they can work on multi-loop programs as well if single-loop transformation [54, 55] is applied. In total, we collected 1 623 verification tasks, each of which contains a single-loop program and a challenging safety property.

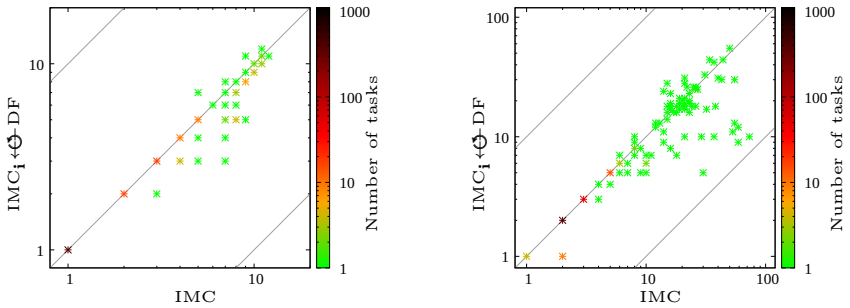
6.3 Experimental Setup

All experiments were conducted on machines having 33 GB of RAM and a 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units. The operating system was Ubuntu 22.04 (64 bit), running Linux 5.15. Each verification task was limited to 4 CPU cores, 15 min of CPU time, and 15 GB of RAM. In our evaluation, we utilized the benchmarking framework BenchExec [56] to ensure reproducibility of our results, and CPAchecker at revision 42901. All SMT and interpolation queries in CPAchecker were handled by MathSAT5 [57]. Since we want to observe IMC's behavior in the presence of auxiliary invariants, we limited the CPU time allocated to the invariant generator to 2.5 min such that IMC had enough time to perform its analysis. For the comparison to other software verifiers, we downloaded 2ls and Symbiotic from the tool archives of SV-COMP '22 [58].

6.4 Results

RQ1: Reduction of program unrollings and interpolation queries. To understand whether the invariant-injection techniques could reduce the numbers of program unrollings and interpolation queries required by plain IMC to find a proof, we conducted a case study on IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF. As discussed in Sect. 4.3, IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF derives different interpolants from plain IMC and is supposed to exhibit distinct computational behavior. We further identified 870 tasks for which DF was able to generate non-trivial and inductive invariants[4] because trivial invariants provide no additional information to help IMC. The scatter plots Fig. 2a and Fig. 2b show the comparison of IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF and IMC on the 870 tasks on the numbers of program unrollings and interpolation queries, respectively. A data point (x, y) in the plots means that there is a task solvable by both IMC and IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF, and IMC (resp. IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF) requires x (resp. y) times of the indicated operation. The color of a data point shows the number of tasks falling into this coordinate. Data points under the diagonal represent the tasks for which IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF needed fewer operations than plain IMC. Observe that IMC$_{\mathbf{i}\leftarrow\cancel{\mathcal{O}}}$DF often required fewer

[4] The trivial invariant $\top$ represents the entire state space.



(a) Program unrollings **(b)** Interpolation queries

Fig. 2: Comparison of the numbers of program unrollings and interpolation queries

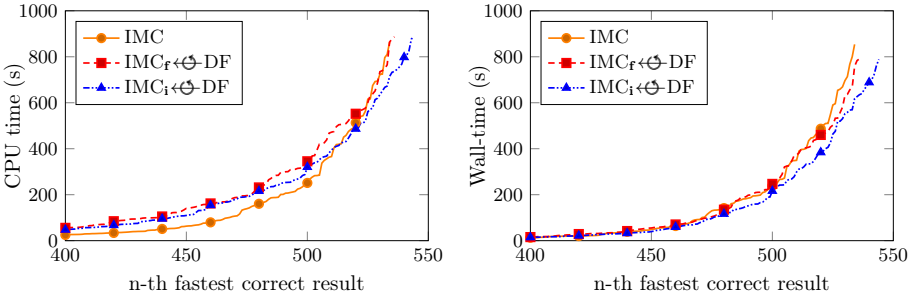
Table 1: Statistics of tasks where IMC$_i$ ↛ DF has significant improvement over IMC (time unit: s with two significant digits; ✓ for solved, and – for timeout)

Task	IMC result	IMC #unroll	IMC #itp	IMC wall-time	IMC itp-time	IMC$_i$ ↛ DF result	IMC$_i$ ↛ DF #unroll	IMC$_i$ ↛ DF #itp	IMC$_i$ ↛ DF wall-time	IMC$_i$ ↛ DF itp-time
Problem03_label03	–	8	50	870	740	✓	5	11	27	13
Problem03_label15	–	7	54	880	750	✓	5	11	35	19
Problem03_label51	–	10	57	880	750	✓	5	11	24	10
benchmark37_conj	–	317	316	890	850	✓	1	2	1.8	0.0080
pals_lcr-var-start	–	11	16	880	630	✓	10	24	760	480
pals_lcr.4.ufo.UNB	✓	11	23	200	160	✓	10	16	120	92
phases_2-2	–	1	1	900	900	✓	1	2	2.2	0.057
s3_srvr_1a.BV.c.cil	–	64	441	890	540	✓	5	13	6.3	1.0
s3_srvr_2a.BV.c.cil	–	39	290	890	790	✓	35	252	640	540
s3_srvr_2a_alt.BV	–	37	278	880	780	✓	19	125	72	52

program unrollings and interpolation calls. For programs correctly proved by both, our augmenting approach reduced the number of program unrollings in 35 tasks, and the number of interpolation calls in 56 tasks.

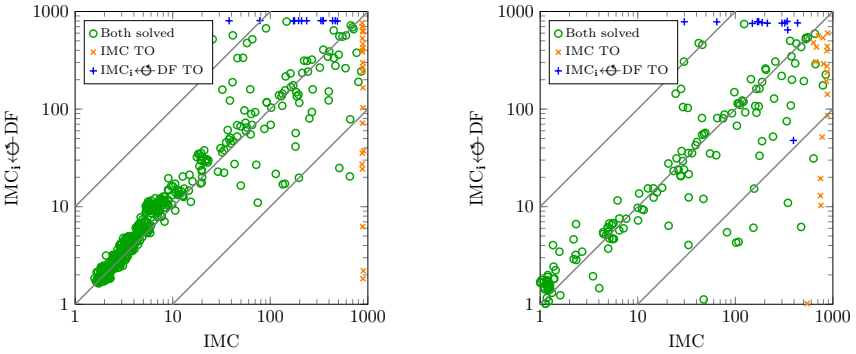
Table 1 shows the tasks for which IMC$_i$ ↛ DF exhibited significant performance improvement. (A similar table for IMC$_f$ ↛ DF can be found in the extended technical report [1].) Upon these tasks, plain IMC either ran into timeouts or required considerably more run-time, whereas the augmented version was able to deliver proofs efficiently. Under each algorithm, the table lists the verification result (solved or timeout), the numbers of program unrollings and interpolation queries, wall-time, and interpolation time (the time spent on all interpolation queries). Note that since plain IMC suffered from timeouts in most of these tasks, the reported numbers of unrollings and interpolation queries are lower bounds of the actual numbers required to compute proofs, which could be much higher.

RQ2: Effectiveness of augmented IMC vs. plain IMC. We compared the number of tasks solved by augmented IMC versus that by plain IMC to evaluate the effectiveness of the proposed approaches. Figure 3b shows the results on the tasks with non-trivial invariants. A data point (x, y) in the plots indicates that x tasks were correctly solved by the respective algorithm within a time bound of y seconds. (The exact numbers can be found in the extended technical



(a) CPU time **(b)** Walltime

Fig. 3: Quantile plots comparing the run-time of plain and augmented IMC



(a) Walltime (s) **(b)** Interpolation time (s)

Fig. 4: Comparison of wall-time and interpolation time (TO stands for timeout)

report [1].) Observe that both augmented approaches solved more tasks than plain IMC. Specifically, IMC$_{\mathbf{i}}$↚DF (resp. IMC$_{\mathbf{f}}$↚DF) found proofs for 23 (resp. 6) tasks where plain IMC ran into timeouts. However, it was not able to solve 13 (resp. 4) tasks proven by plain IMC, partially due to the extra time spent on invariant generation (see Sect. 6.5 for more discussion). Overall, IMC$_{\mathbf{i}}$↚DF (resp. IMC$_{\mathbf{f}}$↚DF) proved 10 (resp. 2) additional tasks compared to plain IMC, and there were 16 tasks solvable by IMC$_{\mathbf{i}}$↚DF but not by plain IMC or DF. Although not extraordinary, the increase shows invariant injection can improve the effectiveness of plain IMC.

To account for the randomness in SMT solving and interpolation, we repeated the experiment with five different random seeds for the underlying SMT solver. In our evaluation, IMC$_{\mathbf{i}}$↚DF consistently delivered more proofs than plain IMC with each seed, demonstrating the robustness of the proposed invariant-injection approaches. The results are available in the extended technical report [1].

RQ3: Run-time efficiency of augmented IMC vs. plain IMC. Quantile plots comparing the CPU time and wall-time usage of plain and augmented IMC on the tasks with non-trivial invariants are shown in Fig. 3a and Fig. 3b, respectively.

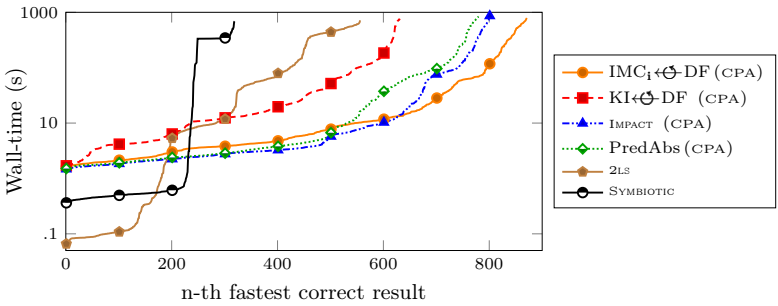


Fig. 5: Quantile plot comparing IMC$_i$↺DF with other approaches and tools

To focus on the performance differences for difficult tasks, we crop the first 400 tasks from the figures because they can be solved within 1 min. From Fig. 3a, observe that IMC dominated its two augmented variants when the CPU time was below 400 s because it did not have a parallel invariant generator that consumed extra time. However, as the elapsed CPU time increased, IMC$_i$↺DF took over and became the best among the three. By contrast, Fig. 3b shows a clear advantage of IMC$_i$↺DF regarding the wall-time consumption. It was able to solve the most tasks when the elapsed wall-time was beyond 200 s. In addition, although IMC$_f$↺DF did not not perform as well as IMC$_i$↺DF on these tasks, it still showed some wall-time improvement over plain IMC. The results indicate that our proposed augmenting approaches can not only prove the correctness of additional programs but also reduce the wall-time for solving difficult tasks.

The scatter plots in Fig. 4 compare the wall-time and interpolation time consumed by IMC and IMC$_i$↺DF on the tasks with non-trivial invariants. A data point (x, y) in the plots indicates a task that can be solved by IMC or IMC$_i$↺DF, for which the former took x seconds of wall-time (resp. interpolation time), and the latter took y seconds of wall-time (resp. interpolation time). Observe that invariant injection helps improve plain IMC's wall-time efficiency for time-consuming tasks because more data points are below the diagonal in the top-right quarter of Fig. 4a. For interpolation time, invariant injection generally lowered the summation of time spent on all queries, as can be seen from Fig. 4b. Moreover, there were 20 tasks showing reduction by an order of magnitude.

RQ4: Comparison with other algorithms and tools. To study whether the proposed invariant-injection approaches for IMC could deliver more proofs than other mature SMT-based methods in CPAchecker as well as other software verifiers, we evaluated IMC$_i$↺DF against KI↺DF [22], Impact [13], and predicate abstraction (PredAbs) [12, 52] in CPAchecker, along with 2ls [23] and Symbiotic [53].[5] Figure 5 plots the wall-time usage of the compared algorithms and tools, and shows that IMC$_i$↺DF solved the most tasks among them. (The exact numbers are summarized in the extended technical report [1].) For the comparison within CPAchecker, IMC$_i$↺DF outperformed other approaches in

[5] Plain IMC has been compared with other approaches in a technical report [30].

terms of both the number of solved tasks and the run-time efficiency. There were 324, 88, and 143 additional tasks solvable by IMC$_{\mathbf{i}}$↜↝DF, but not by KI↜↝DF, Impact, and predicate abstraction, respectively. In total, IMC$_{\mathbf{i}}$↜↝DF uniquely solved 10 tasks among the four compared algorithms. For the comparison with 2ls and Symbiotic, IMC$_{\mathbf{i}}$↜↝DF found significantly more proofs than both of them. In particular, 423 and 624 additional tasks were solved by IMC$_{\mathbf{i}}$↜↝DF, but not by 2ls and Symbiotic, respectively, and a total of 342 tasks were uniquely solved by IMC$_{\mathbf{i}}$↜↝DF. The results demonstrate the value of our invariant-injection methods for IMC and justify our contribution to the state of the art of software verification.

6.5 Discussion

In our experiments, strengthened interpolants decreased the number of interpolation queries (Fig. 2b) and the total interpolation time (Fig. 4b), but the effect came at a price: SMT and interpolation queries may become more challenging. There were several tasks solvable by plain IMC in a minute but stuck at one difficult SMT query with a strengthened interpolant for hundreds of seconds. Therefore, the effect of invariant injection on IMC is mixed: A similar trade-off between fewer refinement steps and extra computation effort to achieve the reduction was reported for the verifier Ufo [44]. In the evaluation, we observed a clear decrease in the numbers of unrollings and interpolation calls, but the number of solved tasks did not increase remarkably. Similar observations that the run-time does not necessarily benefit from auxiliary invariants and that the number of solved tasks might not increase significantly have also been reported in previous publications [23, 32] on invariant-aided verification.

7 Conclusion

We augmented IMC, an interpolation-based model-checking algorithm published by McMillan in 2003 [11], via injecting auxiliary invariants to reduce the numbers of program unrollings and interpolation queries needed to prove the correctness of programs. Invariants are used to strengthen (1) the checks for determining whether a fixed point is reached, or (2) the interpolants derived during the procedure. We rigorously proved the correctness of the proposed approaches and implemented both techniques in the verification tool CPAchecker. We empirically evaluated our implementations against four SMT-based verification approaches in CPAchecker and two state-of-the-art software verifiers over C programs whose safety properties are hard to prove. Our experiments show that the proposed techniques effectively reduced the numbers of program unrollings and interpolation calls and therefore reduced the wall-time usage compared to plain IMC. Furthermore, the proposed augmentation helped IMC deliver more proofs than the compared SMT-based algorithms in CPAchecker and solve 342 tasks unsolvable by the compared tools. For future work, as the strengthened interpolants might lead to extra time spent on SMT solving or interpolation, we plan to devise a strategy to selectively inject invariants into IMC only when they are likely to be helpful, in order to further improve the performance of the proposed methods.

Data-Availability Statement. To enhance the verifiability and transparency of the paper, all relevant materials, including used tools, benchmark tasks, and raw experimental data, are available in a supplemental reproduction package [59]. More information is available at https://www.sosy-lab.org/research/imc-df/.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

References

1. Beyer, D., Chien, P.C., Lee, N.Z.: Augmenting interpolation-based model checking with auxiliary invariants (Extended version). arXiv/CoRR **2403**(07821) (March 2024). https://doi.org/10.48550/arXiv.2403.07821
2. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley, 3rd edn. (2011). https://www.worldcat.org/isbn/978-1-119-20248-6
3. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
4. Kildall, G.A.: A unified approach to global program optimization. In: Proc. POPL. pp. 194–206. ACM (1973). https://doi.org/10.1145/512927.512945
5. Kam, J., Ullman, J.: Global data-flow analysis and iterative algorithms. J. ACM **23**, 158–171 (1976). https://doi.org/10.1145/321921.321938
6. Sharir, M., Pnueli, A.: Two approaches to interprocedural data-flow analysis. In: Program Flow Analysis: Theory and Applications. pp. 189–233. Prentice-Hall (1981). https://www.worldcat.org/isbn/978-0-137-29681-1
7. Kennedy, K.: A survey of data-flow analysis techniques. In: Program Flow Analysis: Theory and Applications, pp. 5–54. Prentice Hall (1981). https://www.worldcat.org/isbn/978-0-137-29681-1
8. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data-flow analysis and programs with recursive data structures. In: Proc. POPL. pp. 66–74. ACM (1982). https://doi.org/10.1145/582153.582161
9. Ryder, B.G.: Incremental data-flow analysis. In: Proc. POPL. pp. 167–176. ACM (1983). https://doi.org/10.1145/567067.567084
10. Reps, T.W., Horwitz, S., Sagiv, M.: Precise interprocedural data-flow analysis via graph reachability. In: Proc. POPL. pp. 49–61. ACM (1995). https://doi.org/10.1145/199448.199462
11. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
12. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
13. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV. pp. 123–136. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_14
14. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: Proc. FMCAD. pp. 1–8. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351148
15. McMillan, K.L.: Lazy annotation for program testing and verification. In: Proc. CAV. pp. 104–118. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_10
16. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proc. CAV. pp. 277–293. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_23

17. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. J. Symb. Log. **22**(3), 250–268 (1957). https://doi.org/10.2307/2963593
18. Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: Proc. DAC. pp. 1073–1076. ACM (2006). https://doi.org/10.1145/1146909.1147180
19. Ganai, M.K., Gupta, A.: Accelerating high-level bounded model checking. In: Proc. ICCAD. pp. 794–801. ACM (2006). https://doi.org/10.1145/1233501.1233664
20. Cheng, X., Hsiao, M.S.: Simulation-directed invariant mining for software verification. In: Proc. DATE. pp. 682–687. ACM (2008). https://doi.org/10.1109/DATE.2008.4484757
21. Donaldson, A.F., Haller, L., Kröning, D.: Strengthening induction-based race checking with lightweight static analysis. In: Proc. VMCAI. pp. 169–183. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_13
22. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
23. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
24. Rocha, H., Ismail, H.I., Cordeiro, L.C., Barreto, R.S.: Model checking embedded C software using k-induction and invariants. In: Proc. SBESC. pp. 90–95. IEEE (2015). https://doi.org/10.1109/SBESC.2015.24
25. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proc. FMCAD. pp. 85–92. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886665
26. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. FSE. pp. 227–236. ACM (2005). https://doi.org/10.1145/1081706.1081742
27. Jain, H., Ivancic, F., Gupta, A., Shlyakhter, I., Wang, C.: Using statically computed invariants inside the predicate abstraction and refinement loop. In: Proc. CAV. pp. 137–151. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_15
28. Pasareanu, C.S., Visser, W.: Verification of Java programs using symbolic execution and invariant generation. In: Proc. SPIN. pp. 164–181. LNCS 2989, Springer (2004). https://doi.org/10.1007/978-3-540-24732-6_13
29. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
30. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning (2024), accepted, preprint available via https://doi.org/10.48550/arXiv.2208.05046
31. Case, M.L., Mishchenko, A., Brayton, R.K.: Automated extraction of inductive invariants to aid model checking. In: Proc. FMCAD. pp. 165–172 (2007). https://doi.org/10.1109/FAMCAD.2007.12
32. Cabodi, G., Nocco, S., Quer, S.: Strengthening model checking techniques with inductive invariants. IEEE Trans. on CAD of Integrated Circuits and Systems **28**(1), 154–158 (2009). https://doi.org/10.1109/TCAD.2008.2009147
33. Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213
34. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

35. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
36. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proc. POPL. pp. 12–27. ACM (1988). https://doi.org/10.1145/73560.73562
37. Bodik, R., Anik, S.: Path-sensitive value-flow analysis. In: Proc. POPL. pp. 237–251. ACM (1998). https://doi.org/10.1145/268946.268966
38. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. Int. Symp. on Programming. pp. 106–130. Dunod (1976). https://www.di.ens.fr/~cousot/COUSOTpapers/publications.www/CousotCousot-ISOP-76-Dunod-p106--130-1976.pdf
39. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. **20**(4-5), 379–405 (2008). https://doi.org/10.1007/s00165-008-0080-9
40. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
41. Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Proc. TACAS. pp. 251–269. LNCS 10805, Springer (2018). https://doi.org/10.1007/978-3-319-89960-2_14
42. Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Proc. TACAS (1). pp. 3–21. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_1
43. Kahsai, T., Tinelli, C.: PKind: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification. pp. 55–62. EPTCS 72, EPTCS (2011). https://doi.org/10.4204/EPTCS.72.6
44. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: Proc. CAV, pp. 672–678. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_48
45. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Proc. CAV. pp. 846–862. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59
46. Brat, G., Navas, J.A., Shi, N., Venet, A.: Ikos: A framework for static analysis based on abstract interpretation. In: Proc. SEFM. pp. 271–277. LNCS 8702, Springer (2014). https://doi.org/10.1007/978-3-319-10431-7_20
47. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
48. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
49. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010). https://dl.acm.org/doi/10.5555/1998496.1998532
50. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6

51. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE. pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13
52. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV. pp. 72–83. LNCS 1254, Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
53. Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In: Proc. FMICS. pp. 207–221. LNCS 7437, Springer (2012). https://doi.org/10.1007/978-3-642-32469-7_14
54. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986). https://www.worldcat.org/isbn/978-0-201-10088-4
55. Donaldson, A.F., Kröning, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k-induction. FMSD **39**(1), 83–113 (2011). https://doi.org/10.1007/s10703-011-0124-2
56. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
57. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
58. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
59. Beyer, D., Chien, P.C., Lee, N.Z.: Reproduction package for SPIN 2024 submission 'Augmenting interpolation-based model checking with auxiliary invariants'. Zenodo (2024). https://doi.org/10.5281/zenodo.10548594