Journal of Automated Reasoning (2025) 69:5
https://doi.org/10.1007/s10817-024-09702-9

Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification

Dirk Beyer[1] · **Nian-Ze Lee**[1] · **Philipp Wendler**[1]

Received: 11 August 2022 / Accepted: 6 July 2023 / Published online: 5 February 2025

Abstract
The article *Interpolation and SAT-Based Model Checking* (McMillan in: Proc. CAV 2003, LNCS, Springer [56]) describes a formal-verification algorithm, which was originally devised to verify safety properties of finite-state transition systems. It derives interpolants from unsatisfiable BMC queries and collects them to construct an overapproximation of the set of reachable states. Although 20 years old, the algorithm is still state-of-the-art in hardware model checking. Unlike other formal-verification algorithms, such as k-induction or PDR, which have been extended to handle infinite-state systems and investigated for program analysis, McMillan's interpolation-based model-checking algorithm from 2003 has not been used to verify programs so far. Our contribution is to close this significant, two decades old gap in knowledge by adopting the algorithm to software verification. We implemented it in the verification framework CPAchecker and evaluated the implementation against other state-of-the-art software-verification techniques on the largest publicly available benchmark suite of C safety-verification tasks. The evaluation demonstrates that McMillan's interpolation-based model-checking algorithm from 2003 is competitive among other algorithms in terms of both the number of solved verification tasks and the run-time efficiency. Our results are important for the area of software verification, because researchers and developers now have one more approach to choose from.

Keywords Software verification · Program analysis · Model checking · Interpolation · Interpolation-based model checking · CPAchecker · SMT · SAT

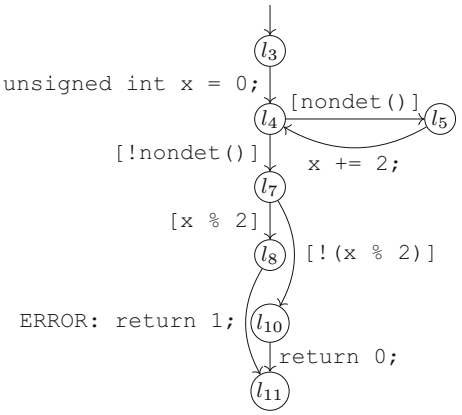
1 Introduction

Automatic software verification [48] is an active research field in which automated solutions of the following problem are studied: Given a program and a specification, decide whether the program satisfies the specification or not. In this paper, we focus on the verification of reachability-safety properties, asserting that some error location in the program should never be reached by the control flow. Other specifications, including termination, memory safety, concurrency safety, and overflows, are also investigated in the literature. Although the problem of software verification is in general undecidable, many important concepts, including various predicate-abstraction techniques [5, 40, 42, 46], counterexample-guided abstraction refinement (CEGAR) [34], large-block encoding [11, 20], interpolation [45, 57],

[1] LMU Munich, Munich, Germany

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

(a) C program



(b) Control-flow automaton

Fig. 1 An example C program (**a**) and its CFA (**b**) (adopted from loop-invariants/even.c in the benchmark set of the 2022 Competition on Software Verification (SV-COMP'22) [8])

together with the advances in SMT solving [7] and combinations with data-flow analysis [15], make it feasible to apply verification technology to industry-scale software, such as device drivers [4, 6, 23, 52], web services [31, 36], and operating systems [63].

To illustrate the reachability-safety verification of a program, consider the C program in Fig. 1a. The program first initializes the variable `x` to 0 and keeps incrementing `x` by 2 while the nondeterministic value returned from the function `nondet()` is nonzero. Once the nondeterministic value equals zero, the control flow exits the loop and tests whether `x` is odd. If `x` is odd, the control flow reaches the error location at line 8; otherwise the program terminates without errors. The goal of the reachability-safety verification is to either prove that the error location is unreachable by the control flow or find an execution path of the program reaching the error location.

As the verification of finite-state and infinite-state transition systems share much similarity, some classic model-checking algorithms for software (infinite-state systems), such as bounded model checking (BMC) [25, 35] or k-induction [13, 38, 51], were originally developed for hardware (finite-state systems). A well-known example of such technology transfer is property-directed reachability (PDR) [28]. After it obtained huge success in hardware model checking, many research efforts have been invested for its software-verification adoption [12, 26, 32, 50, 55].

1.1 Interpolation-Based Verification Approaches

McMillan's algorithm [56] from 2003 is another state-of-the-art approach for hardware model checking, prior to the invention of PDR. It utilizes Craig interpolation [37] to derive interpolants from unsatisfiable BMC queries and computes an overapproximation of the set of reachable states as the union of the interpolants. Its idea of abstracting objects with interpolants has been extended beyond state sets and underpinned various interpolation-based verification approaches and tools. Abstractions of transition relations [49], traces [43], predicates over program variables [5, 45], and function calls [61] have been studied in the literature. We classify in Fig. 2 different usages of Craig interpolation and highlight some

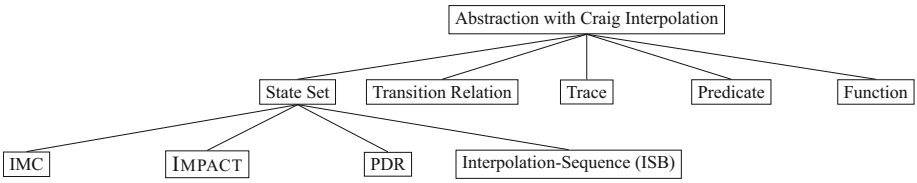


Fig. 2 Classification of different abstractions using Craig interpolation

important algorithms regarding state-set abstraction. An overview of several representative interpolation-based formal-verification approaches is provided in Sect. 2.

Despite its success in hardware model checking and profound theoretical impact on program analysis, McMillan's algorithm [56] from 2003 has not been investigated for software verification. We emphasize that McMillan's interpolation-based algorithm for model checking from 2003 should not be mistaken for other, later interpolation-based verification approaches. In the following, we refer to the algorithm proposed by McMillan from 2003 as *interpolation-based model checking* and abbreviate it as IMC.

One potential concern to apply IMC to software, raised by its inventor McMillan in his later paper [57] presenting the algorithm IMPACT, was the scalability of the underlying decision procedure to handle the entire unrolled program. Compared to IMC, IMPACT derives interpolants only for individual execution paths, reducing the workload of the solver. Fortunately, due to the advancements in SMT solving, delegating formulas encoding the entire unrolled program to the solvers has become feasible. Therefore, it is time to revisit IMC and evaluate its performance against the state of the art. Other SMT-based approaches have been thoroughly compared already in the literature [14].

1.2 Our Research Questions and Contributions

In this paper, we explore the applicability of the IMC algorithm to software verification. Specifically, we answer the following two research questions. First, we investigate *how to efficiently adopt IMC to software verification*. As mentioned earlier, IMC was originally invented to verify sequential Boolean-logic circuits (hardware), whose transition relations, required to perform IMC, are easy to derive: The downstream circuitry of the memory elements (i.e., registers) encodes the next-state function of the system, which can then be naturally expressed as a transition relation between system states. It is less straightforward, by contrast, to extract a transition relation from a program (software). Although representing a program as a transition relation with the program counter is in principle possible, such conversion mixes the reasoning of the control-flow structure and the program semantics, and hence, does not work for IMC in practice as we show in Sect. 4. To address this research question, we propose an efficient software adoption of IMC via large-block encoding, separating the analysis of the control flow and the program semantics by exploring the analogy between the execution paths of a sequential Boolean-logic circuit and a program. We also present the first implementation of the IMC algorithm for software verification and make it available in the open-source framework CPACHECKER [17, 19]. The details of the proposed adoption and implementation will be discussed in Sects. 5 and 6, respectively. Our second research question focuses on *evaluating the performance of the IMC adoption against the state of the art*. To address this research question, we compare the proposed implementation against other state-of-the-art software-verification algorithms, including PDR, BMC, k-induction, predicate abstraction,

and IMPACT, on the largest benchmark suite of C safety-verification tasks in Sect. 7. Our experimental results show that IMC is competitive in terms of both effectiveness (the number of solved tasks) and efficiency (the elapsed CPU time).

Novelty

(1) This paper closes the two decades old gap of knowledge by investigating the applicability of IMC to software verification. We analyze the characteristics of IMC in the context of software verification, and our empirical evaluation indicates its competitiveness against the state-of-the-art approaches. (2) Our replication of the IMC algorithm as open-source implementation broadens the spectrum of available software-verification techniques, which is important in practice because researchers, developers, and tool users now have more choices at their disposal. (3) While the application of large-block encoding to program analysis has a long history, to the best of our knowledge, using large-block encoding to represent an algorithm that originated from a different research community for software is a new idea, which may shed light on the efficient adoptions of other algorithms.

Significance

IMC is an important verification algorithm in hardware verification. It is a risk to leave the potential of it unexplored for the verification of software. Therefore, we believe that the knowledge about the algorithm's adoption to software is a significant improvement of the state of the art and has the potential to inspire other works in the area of software verification.

Correctness

We show the correctness of our algorithms in Theorem 1. Our implementation is based on components from the CPACHECKER framework [19], which is a well-maintained software project with lots of evidence that the components work well. Large-block encoding is a sound component from the literature [11, 20].

The effectiveness and efficiency of our implementation is empirically evaluated with experiments on a large benchmark set in Sect. 7. We discuss possible threats to validity that might affect the soundness of our conclusions from the experimental results in Sect. 7.5.

2 Related Work

IMC has popularized the idea of using interpolation for verification, and although IMC itself has not been applied to software so far, there are many approaches for software verification that make use of interpolation. Based on the classification in Fig. 2, we will discuss several representative interpolation-based approaches and tools, as summarized in Table 1. Interested readers are referred to the chapter [59] by McMillan in the Handbook of Model Checking for a broader survey. Of course, there exist many techniques for computing interpolants. We do not discuss them here as interpolant computation is typically orthogonal to the used verification algorithm. In our implementation, we use an off-the-shelf SMT solver for interpolation (MATHSAT5 [33]).

Table 1 Important interpolation-based formal-verification approaches and tools

Approach	Year	Publication	Contribution
IMC	2003	[56]	First interpolation-based model-checking algorithm
Predicate abstraction	2004	[45]	Discovering relevant predicates from interpolants to refute false alarms
TR approximation	2005	[49]	Refining an abstract TR with interpolants to avoid exact image computation
IMPACT	2006	[57]	Performing lazy abstraction by computing sequences of interpolants on program paths
Slicing abstraction	2007	[29]	Splitting abstract states with interpolants
ISB	2009	[62]	Imitating BDD-based model checking by abstracting states with interpolants
Trace abstraction	2009	[43]	Refining an overapproximation of possible traces with interpolant automata
Lazy annotation	2010	[58]	Annotating a program with interpolants derived from Hoare triples
Function summaries	2011	[61]	Summarizing function calls with interpolants to reduce future analysis effort
Software PDR	2012	[32]	Combining IMPACT-like proof-based interpolants and PDR clause generation
CTIGAR	2014	[26]	Refining abstraction failures relative to single steps with interpolants
BLAST	2004	[16, 45]	First software model checker using interpolation
CSISAT	2008	[24]	First open-source interpolation engine
CPACHECKER	2009	[11, 19]	Large-block encoding and interpolation
WOLVERINE	2011	[54]	First public implementation of IMPACT
UFO	2012	[2]	Combining predicate and interpolation methods
DUALITY	2013	[60]	Solving constrained Horn clauses with interpolation
SPACER	2013	[53]	Combining proof-based approaches and CEGAR
SAFARI	2014	[3]	Backward IMPACT-like analysis with arrays

2.1 State Sets

The most closely related algorithm is IMPACT [57] from the same author, which is also based on the idea of computing a fixed point from interpolants. IMPACT applies interpolation to formulas of single program paths instead of the whole program and generates a *sequence of interpolants* for a spurious counterexample, one interpolant after each program statement on the execution path. It also computes fixed points of reachable states per program location instead of globally. One adaptation [32] of property-directed reachability (PDR) [28] to software computes sequences of sets of clauses for refuting spurious counterexamples, and these sequences also form valid sequences of interpolants. Under this view, the approach is similar to IMPACT, only differing in how the interpolants are computed. A hybrid approach with a combination of proof-based interpolation (as in IMPACT) and PDR-based clause generation has also been suggested [32]. CTIGAR [26] is another attempt to extend PDR to software. It combines Cartesian predicate abstraction with PDR and considers an abstract state as a conjunction of the predicates satisfied by the corresponding concrete state. Different from

other adaptations of PDR, CTIGAR avoids expensive pre-image computation by focusing on refinement relative to single steps of the transition relation.

A related approach for hardware model checking is *interpolation-sequence based model checking* (ISB) [62]. In contrast to IMC, which computes only one interpolant at a time that overapproximates states reachable within a certain number of steps, ISB derives a sequence of interpolants from an unsatisfiable BMC query, and each interpolant is an overapproximation of the states reachable within an increasing number of steps. This is similar to IMPACT, just with ISB computing sequences of interpolants for an unrolling of the whole transition relation instead of single program paths like IMPACT. In ISB, the fixed point is found if the interpolant derived at the last unrolled loop head implies the disjunction of all previous interpolants.

The approach of *lazy annotation* [58] combines symbolic execution and interpolation to generate Hoare-style annotations for a program in a similar way as a conflict-driven clause-learning SAT solver. An annotation on a program edge is a condition that will block any future execution from this edge to an error location. The method symbolically executes the input program along some chosen path to search for an error location. If the execution is blocked by an edge, it backtracks and produces an annotation by interpolation, which is a valid precondition of the edge's Hoare triple. This method is also applicable to program testing because it explores only feasible traces.

2.2 Predicates, Transition Relations, Traces, and Functions

Another popular use of interpolation for software verification is to derive predicates from interpolants for predicate abstraction [5, 45] in the refinement step of CEGAR, typically by breaking up the interpolants into atomic predicates. In contrast to IMC and IMPACT, which both create the final abstract model of the program (the overapproximation of the set of reachable states) directly from interpolants, predicate abstraction uses Boolean or Cartesian abstraction over the set of derived predicates and may generalize better. Interpolation has also been used to avoid the expensive exact image computation in predicate abstraction [49], refining an abstract transition relation to guarantee convergence given adequate predicates. Slicing abstraction [29] is another technique related to predicate abstraction. It splits abstract states using predicates obtained from Craig interpolants to refine the abstraction.

Trace abstraction [43, 44] extends the concept of abstracting information by Craig interpolation to representing program paths with interpolants. Given an unsatisfiable BMC query, it derives a sequence of interpolants and constructs an interpolant automaton out of them. This interpolant automaton excludes spurious traces that share the same reason of infeasibility with the current one. A novel counterexample-guided abstraction refinement scheme is proposed for trace abstraction to prove the correctness of a program.

Interpolants are also applied to summarize function calls in a program [61]. This approach replaces function calls with interpolants obtained in a previous analysis to reduce the subsequent verification effort. Given an unsatisfiable BMC query involving a function call, a summary of the function is computed as an interpolant between the function's corresponding formula and the rest of the BMC formula. Recently, Craig interpolation is also used to abstract sequences of transition relations to find deep counterexamples [27].

2.3 Tools Based on Craig Interpolation

Several software-verification tools are developed on top of Craig interpolation. The tool BLAST [16, 45] provides the first implementation of a software-verification tool that uses

interpolants for computing abstractions. The tool CSISAT [24] was the first freely available SMT solver with interpolation support. The verification framework CPACHECKER [19] applies Craig interpolation to large-block encodings of program code. The tool WOLVERINE [54] provides the first publicly available implementation of IMPACT, featuring a built-in interpolation procedure and some support for bit-vector operations. The framework UFO [2] is parameterized by definable components of abstract post, refinement, and expansion, allowing various verification techniques based on overapproximation and underapproximation. Craig interpolation has also been applied to solve constrained Horn clauses (CHC). The tool DUALITY [60] generalizes IMPACT to gradually unroll a program and solves the corresponding CHC formulas with interpolation until it yields valid inductive invariants. The tool SPACER [53] combines proof-based techniques with CEGAR, maintaining both an overapproximation and an underapproximation of the input program. The tool SAFARI [3] implements a backward reachability analysis with lazy abstraction based on the MCMT framework [41], which can be understood as a backward variant of IMPACT, to support reasoning of arrays with unknown length.

3 Background

In the following, Boolean connectives $\neg, \vee, \wedge, \rightarrow, \equiv$ are used in their conventional semantics. A first-order logical formula is also interpreted as a set of (program) states that satisfy the formula, and we use the two terms interchangeably when it is clear from the context.

3.1 Interpolation-Based Model Checking

Interpolation-based model checking (IMC) [56] is an algorithm for unbounded model checking to verify safety properties of state-transition systems. It can be considered as an extension of BMC, which is well-known for bug hunting. In order to describe IMC, we first define the notation to formalize a state-transition system. Second, we review Craig's interpolation theorem [37], which is the core concept to extend BMC to unbounded model checking.

3.1.1 State-Transition System

Let s and s' be two arbitrary states in the state space of a state-transition system. We formalize the state-transition system by three predicates over states. Predicate $I(s)$ evaluates to *true* if state s is an initial state of the system. Predicate $T(s, s')$ evaluates to *true* if the system can transit from state s to state s'. It is also called the *transition relation* of the system. Predicate $P(s)$ evaluates to *true* if state s satisfies the safety property to be verified.

In the above formulation of a state-transition system, we do not assume the state space to be finite or infinite. The working of IMC is similar in both cases, provided that the underlying constraint solver (SAT/SMT solver) supports the reasoning over the corresponding logical formulas.

3.1.2 Craig's Interpolation Theorem

Given two first-order logical formulas α and β, if $\alpha \Rightarrow \beta$, Craig's interpolation theorem [37] guarantees the existence of a logical formula γ such that $\alpha \Rightarrow \gamma$ and $\gamma \Rightarrow \beta$ hold, and γ only refers to the common variables of α and β. Formula γ is called an *interpolant* of α and β as it is *between* α and β. In the model-checking community, Craig's interpolation theorem is

usually stated in an equivalent form based on unsatisfiability: Given an unsatisfiable formula $A \wedge B$, C is an interpolant of this formula if (1) $A \Rightarrow C$, (2) $C \wedge B$ is unsatisfiable, and (3) C only refers to the common variables of A and B.

3.1.3 Algorithm Description

The overall procedure of IMC [56] can be decomposed into two phases. The first phase poses a BMC query by unrolling the transition relation k times and constructing a formula representing all possible execution paths from an initial state to a bad state (a state that violates the safety property) with k transitions.[1] We use variable s_i to denote the state after the i^{th} transition. Furthermore, to facilitate Craig interpolation in the second phase, the BMC query is partitioned into two formulas A and B (we omit $\wedge$ for brevity):

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

If this formula is satisfiable, a violation is found, and we conclude that the system does not fulfill the safety property. Otherwise, instead of simply increasing the unrolling upper bound, IMC tries to prove the safety property from the unsatisfiable BMC query in its second phase. According to Craig's interpolation theorem, there exists an interpolant $C(s_1)$ referring to the common variable s_1, such that the following two conditions hold:

$$I(s_0)T(s_0, s_1) \rightarrow C(s_1) \text{ and}$$
$$C(s_1)T(s_1, s_2) \ldots T(s_{k-1}, s_k)\neg P(s_k) \text{ is unsatisfiable.}$$

The above two conditions indicate that $C(s_1)$ is an overapproximation of the set of states reachable from the initial states with one transition, and that states in $C(s_1)$ will not violate the safety property after $(k-1)$ transitions.

An overapproximation of the set of reachable states can be built by iteratively computing these interpolants. Suppose the interpolant contains some noninitial states. Changing the variable used in the interpolant from s_1 to s_0, we pose another BMC query starting from the interpolant, that is, with $I(s_0)$ replaced by $C(s_0)$:

$$\underbrace{C(s_0)T(s_0, s_1)}_{A'(s_0,s_1)} \underbrace{T(s_1, s_2) \ldots T(s_{k-1}, s_k)\neg P(s_k)}_{B'(s_1,s_2,\ldots,s_k)}$$

If the formula is again unsatisfiable, another interpolant $C'(s_1)$ exists, which is an overapproximation of the set of states reachable from the initial states with two transitions. Such computation is repeated until the newly derived interpolant is contained in the union of the initial states and all previous interpolants. In other words, the procedure stops when the union of the initial states and all previous interpolants grows to a *fixed point*, i.e., a set of states that is inductive with respect to the transition relation and hence contains all reachable states. From the second condition of Craig's interpolation theorem, it is guaranteed that this fixed point implies the safety property, and hence the safety property is proved.

If any BMC query is satisfiable during the iteration in the second phase, we cannot conclude that the property is violated. The violation could be a wrong alarm, as some starting states in the interpolants might not be reachable. Therefore, we have to return back to the first phase,

[1] The original BMC query in McMillan's 2003 paper [56] encodes all possible execution paths violating the safety property with *at most* k transitions. In this work, we use an optimization discussed in Section 3.2 of the 2003 paper to perform IMC incrementally and consider the property violation only after the last transition.

increase the unrolling upper bound, and precisely check the existence of a violation starting from the initial states.

3.1.4 Towards an Efficient Adoption

While IMC is described in terms of logical formulas in the above discussion, the adoption of this algorithm to a concrete state-transition system, such as a sequential Boolean-logic circuit (hardware) or a program (software), requires a conversion from the system under verification to the three predicates $I(s)$, $T(s, s')$, and $P(s)$. The conversion is simple for sequential Boolean-logic circuits, which IMC originally focused on, as the input wires to the registers of the circuit encode the function to compute the next state (i.e., the state after transition) via the downstream circuitry in terms of the output wires of the registers (i.e., the current state). This state-transition function can be naturally expressed as a transition relation. It is less straightforward, by contrast, to extract a transition relation from a program. Although a brute-force conversion is available, representing a program via a transition relation with symbolic program counters conceals the structural information of the program from the analysis. In Sect. 4, we examine why encoding a program as a transition relation with symbolic program counters is not suitable for adopting IMC to software verification. The main challenge towards an efficient adoption to software verification thus lies in obtaining all required predicates while taking the program's structure into consideration.

3.2 Program Representation

To facilitate the subsequent discussion of program analysis, here we provide some fundamental definitions for program representation from the literature [15, 16]. We consider an imperative programming language whose variables are all integers. The operations are either variable assignment or Boolean-expression evaluation. We represent such a program as a *control-flow automaton* (CFA) $A = (L, l_0, G)$. A CFA is a directed graph with a set L of nodes being program locations, an initial location $l_0 \in L$ indicating the entry point of the program, and a set $G \subseteq (L \times Ops \times L)$ being control-flow edges annotated with program operations.

A *reachability-safety verification task* consists of a CFA and an error location of the CFA. The task is to either prove that the error location is unreachable from the initial location or find a feasible error path to the error location otherwise. For instance, the CFA of the example C program in Fig. 1a is shown in Fig. 1b. The initial location of this CFA is l_3, and the error location is l_8.

3.3 Configurable Program Analysis

A configurable program analysis (CPA) [15, 17, 18] defines the abstract domain used for a program analysis. As we implemented the proposed adoption of IMC in the framework CPAchecker [19], which utilizes CPA as the core concept, we provide necessary background knowledge about CPA as follows. To simplify the presentation, we omit the dynamic precision adjustment of CPA because it is irrelevant for this paper. Please refer to the literature [14, 18] for further details.

3.3.1 Definition

A CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop})$ consists of an abstract domain D, a transfer relation $\rightsquigarrow$, and the operators merge and stop. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of a set C of concrete program states, a semilattice $\mathcal{E} = (E, \sqsubseteq)$ over a set E of abstract states and a partial order $\sqsubseteq$, and a concretization function $[\![\cdot]\!]$ to map an abstract state to the represented set of concrete program states. The transfer relation $\rightsquigarrow \subseteq E \times E$ computes abstract successor states. The merge operator $\mathsf{merge} : E \times E \to E$ specifies how to merge two abstract states when the control flow meets. The stop operator $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ determines whether an abstract state is covered by a given set of abstract states. The operators merge and stop can be chosen appropriately to influence the abstraction level of the analysis. Common choices include $\mathsf{merge}^{sep}(e, e') = e'$ (which does not merge abstract states) and $\mathsf{stop}^{sep}(e, R) = (\exists e' \in R : e \sqsubseteq e')$ (which determines coverage by checking whether the given abstract state is less than or equal to any other reachable abstract state according to the semilattice).

3.3.2 Fundamental CPAs and Composite CPA

Several fundamental CPAs are used in this paper: The *Location CPA* $\mathbb{L}$ [18] uses a flat lattice over all program locations to track the program counter explicitly; the *Loop-Bound CPA* $\mathbb{LB}$ [13, 14] tracks in its abstract states for every loop of the program how often the loop body has been traversed on the current program path. Another important CPA, namely the *Predicate CPA* $\mathbb{P}$ [14], serves as the core data structure underlying the proposed IMC adoption. The *Predicate CPA* $\mathbb{P}$ for *adjustable-block encoding* (ABE) [20] uses a triple $(\psi, l^{\psi}, \varphi)$ of an abstraction formula ψ, an abstraction location l^{ψ}, and a path formula φ as an abstract state. The abstraction formula ψ stores the abstraction of the program state computed at the program location l^{ψ}. The path formula φ syntactically encodes the program behavior from the abstraction location l^{ψ} to the current program location. Abstract states where the path formula φ is *true* are called *abstraction states*; other abstract states are *intermediate states*.

Several CPAs can be combined using a *Composite CPA* [17] to achieve synergy. The abstract states of the Composite CPA are tuples of one abstract state from each component CPA and the operators of the Composite CPA can delegate to the component CPAs' operators accordingly. We also use the *ARG CPA* $\mathbb{A}$ to store the predecessor-successor relationship between abstract states to track the *abstract reachability graph* (ARG).

3.3.3 CPA Algorithm

CPAs can be used by the CPA algorithm [14, 15, 17], which gets as input a CPA and an initial abstract state, for reachability analysis. The algorithm performs a classic fixed-point iteration by looping until all abstract states have been completely processed and returns the set of reachable abstract states. The proposed adoption of IMC relies on an extension of the CPA algorithm, named CPA++ [14]. Instead of an initial abstract state, the CPA++ algorithm takes a set of reached abstract states and a set of frontier abstract states awaiting processing. It additionally receives as input a function abort to determine whether it should abort early for some abstract state. Upon completion, the CPA++ algorithm returns the updated reached set and waiting list of abstract states.

4 A Straightforward Adoption with Symbolic Program Counters

Before presenting the proposed efficient adoption of IMC for software verification, we take a step back by first discussing a straightforward method to encode IMC with symbolic program counters, as mentioned in Sect. 3.1.4. We will also demonstrate, both conceptually and empirically, why such a brute-force encoding is not suitable for IMC on software.

4.1 Encoding Transition Relations with Symbolic Program Counters

The straightforward method derives the three predicates, namely, the initial condition, transition relation, and safety property, via introducing a variable pc to store the program counter. Given a CFA whose initial location is l_0, the initial condition can be encoded as $pc = l_0$. The transition relation of the CFA is the disjunction of the formula $pc = l_i \wedge op \wedge pc' = l_j$ for each edge (l_i, op, l_j) of the CFA, where op is the operation annotated to the edge, and the variable pc' stores the program counter after the operation is executed. Suppose a location l_E of the CFA is specified as the error location for a reachability-safety verification task, then the corresponding safety property of the task can be expressed as $pc \neq l_E$.

We use the example CFA in Fig. 1b to illustrate the encoding. Recall that a primed variable denotes the program variable after one transition. With the symbolic program counter pc, the initial condition is encoded as $I(pc) = (pc = l_3)$; the safety property is expressed as $P(pc) = (pc \neq l_8)$; the transition relation $T(pc, x, pc', x')$ of the CFA is captured as:

$$
\begin{aligned}
&(pc = l_3 \wedge x' = 0 \wedge pc' = l_4) \ \vee \\
&(pc = l_4 \wedge \texttt{nondet()} \neq 0 \wedge pc' = l_5 \wedge x' = x) \ \vee \\
&(pc = l_4 \wedge \texttt{nondet()} = 0 \wedge pc' = l_7 \wedge x' = x) \ \vee \\
&(pc = l_5 \wedge x' = x + 2 \wedge pc' = l_4) \ \vee \\
&(pc = l_7 \wedge x\%2 \neq 0 \wedge pc' = l_8 \wedge x' = x) \ \vee \\
&(pc = l_7 \wedge x\%2 = 0 \wedge pc' = l_{10} \wedge x' = x) \ \vee \\
&(pc = l_8 \wedge pc' = l_{11} \wedge x' = x) \ \vee \\
&(pc = l_{10} \wedge pc' = l_{11} \wedge x' = x)
\end{aligned}
$$

Note that for each program variable that is not assigned by an operation, the primed variable must be set equal to the respective unprimed variable. For example, x is not assigned by the edge $(l_4, [\texttt{nondet()}], l_5)$, so $x' = x$ needs to be added to the edge's disjunctive term.

Having derived the three predicates, we can perform IMC as described in Sect. 3.1. The following BMC query unrolls the transition relation k times and tests if the property can be violated after k transitions:

$$I(pc_0) \wedge T(pc_0, x_0, pc_1, x_1) \wedge \ldots \wedge T(pc_{k-1}, x_{k-1}, pc_k, x_k) \wedge \neg P(pc_k).$$

To ease the readability, we abbreviate $T(pc_i, x_i, pc_j, x_j)$ as $T_{i,j}$ in the following. Note that the correctness of the example program can be proved if the invariant $x\%2 = 0$ is established at the program location l_4.

4.2 Drawbacks of the Encoding

The obstacle hindering IMC to work with the encoding is the weak interpolants derived from the queries. In particular, when program-counter variables appear in unsatisfiable queries,

the resultant interpolants tend to concern themselves mainly with the program counter pc but seldom mention the program variables. The lack of strong interpolants arises from syntactically infeasible paths encoded in the BMC queries. We will discuss this problem using the example CFA in Fig. 1b. First, note that for this CFA, a potential error path from l_3 to l_8 must have an odd number of edges and at least three edges. In other words, a BMC query with an even number of transitions is trivially unsatisfiable, and the derived interpolant does not need to (and usually will not) refer to the program variable x.

For the BMC queries with a syntactically feasible path ($k = 3, 5, 7, 9, \ldots$), the derived interpolants may involve the program variable x. However, IMC could still fail to reach a fixed point in this situation (and does so in practice). As an example, we continue on the CFA in Fig. 1b, unroll the transition relation with $k = 5$, and pose the BMC query $I \wedge T_{0,1} \wedge T_{1,2} \wedge T_{2,3} \wedge T_{3,4} \wedge T_{4,5} \wedge \neg P$. This unsatisfiable BMC query allows syntactically feasible paths, and thus it is possible to yield useful interpolants. Suppose, with luck, a good interpolant $\tau_1 = (pc = l_4 \wedge x\%2 = 0)$ is derived, which is exactly the required invariant to prove the correctness of the program. In the following, we demonstrate how this valuable information might be discarded, unfortunately, if IMC is performed without knowledge of the program structure.

The IMC algorithm described in Sect. 3.1 aims at building a fixed point by iteratively computing more interpolants to overapproximate reachable states. After computing τ_1, the next interpolant is derived from a BMC query with the initial condition replaced by the previous interpolant with appropriately shifted indices: $\tau_1 \wedge T_{0,1} \wedge T_{1,2} \wedge T_{2,3} \wedge T_{3,4} \wedge T_{4,5} \wedge \neg P$. This query is trivially unsatisfiable because no path can go from l_4 to l_8 with five transitions, which are the start and end points enforced by τ_1 and $\neg P$, respectively. Therefore, the interpolant for this query usually concerns only the program counter and loses the information about program variables. For example, we might obtain $\tau_2 = (pc = l_5 \vee pc = l_7)$. Starting from τ_2, the next BMC query is satisfiable because there is a feasible path from l_5 to l_8 with five transitions. As a result, the IMC algorithm fails to reach a fixed point for $k = 5$ and has to go back to the BMC phase with an incremented unrolling bound. A better interpolant without the program counter, e.g., $(x\%2 = 0)$ instead of τ_2, could have prevented the loss of the information, but it is rare to get such high-quality interpolants even from state-of-the-art interpolation engines.

We conducted an experiment on 723 tasks from the category *ReachSafety-Loops* used in the 2022 Competition on Software Verification [9] to support our conceptual reasoning above. Among the 505 tasks without property violation, IMC with symbolic program counters only proved 11 tasks. Moreover, this encoding of IMC can prove the tasks because the numbers of loop iterations in these 11 tasks are bounded, not because it constructed a fixed point from interpolants. In our experiment, IMC with symbolic program counters never found a fixed point by interpolation for any task and usually got trapped in the suboptimal situation explained above.

4.3 Lessons Learned

To avoid weak interpolants that only concern the program counter and thus prevent reaching a fixed point, we must not pose BMC queries about syntactically infeasible paths. At the core of this issue is mixing the information about the control flow and program semantics in the transition relation when IMC is adopted with the brute-force conversion and symbolic program counters. Next, we will present another adoption of IMC based on large-block

encoding, which separates the analysis of the control flow from the fixed-point computation of IMC and hence improves the quality of derived interpolants.

5 An Efficient Adoption with Large-Block Encoding

In this section, we describe our proposed approach for adopting IMC to software verification. In essence, we utilize *large-block encoding* (LBE) [11] to draw an analogy between a program and the state-transition system discussed in Sect. 3. The idea is not only helpful for this paper but might also shed light on the efficient adoptions of other algorithms.

As explained in Sect. 4, explicitly encoding symbolic program counters into the transition relation of a CFA is not ideal for adopting IMC to software verification. Sequential Boolean-logic circuits, for which IMC was originally designed, usually have only one feedback loop. By contrast, a CFA could have arbitrarily many loops. To simplify the problem, we start by considering single-loop programs. As a program with multiple loops can be converted into a single-loop program by a standard transformation [1, 39], this simplification will not hurt the generality of the proposed approach. The effect of the single-loop transformation on the performance of IMC will be discussed in Sect. 5.1.

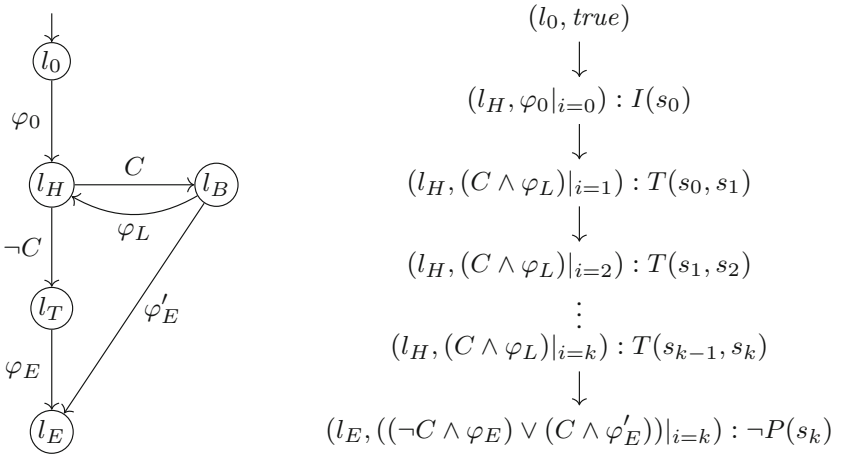
To obtain the transition relation of a single-loop program, we take advantage of LBE [11]. Given a CFA, LBE repeatedly rewrites the original CFA in order to summarize it. In the summarized CFA, each loop-free subgraph of the original CFA is represented by a single control-flow edge. The edge is annotated with a formula that encodes the program behavior of the represented subgraph of the original CFA.

For single-loop programs, applying LBE will always result in a summarized CFA with a structure as shown in Fig. 3a. It has an initial location l_0, a loop-head location l_H, a loop-body location l_B, a loop-tail location l_T, and an error location l_E. These locations correspond to program locations in the original single-loop CFA before summarization. The edges of the summarized single-loop CFA are labeled with the following formulas: Formula φ_0 summarizes the subgraph from l_0 to l_H, formula C is the loop condition, formula φ_L summarizes the subgraph from l_B back to l_H, and formulas φ_E, φ'_E summarize the subgraphs from l_T, l_B to l_E, respectively.

We notice that the summarized single-loop CFA has a natural analogy to those predicates used in Sect. 3.1: The initial-state predicate $I(s)$ is analogous to φ_0, the transition relation $T(s, s')$ is analogous to $C \wedge \varphi_L$, and the negated safety property $\neg P(s)$ is analogous to $(\neg C \wedge \varphi_E) \vee (C \wedge \varphi'_E)$. Using LBE, we successfully obtain the required predicates without explicitly encoding the program counter into the formulas.

Furthermore, in order to perform IMC, we have to unroll the summarized single-loop CFA and construct the BMC query Eq. (1). In Fig. 3b, we unroll the CFA by drawing all possible paths starting from l_0, iterating the loop k times ($k + 1$ visits to l_H), and finally reaching l_E. A node in Fig. 3b consists of a program location which the control flow is currently at and a formula σ to encode all possible paths starting from the program location of the preceding node. Note that σ is indexed with the unrolling counter i to distinguish between different iterations.

To discover the similarity between Eq. (1) and Fig. 3b, we additionally label a node in Fig. 3b with the subformula in Eq. (1) that σ corresponds to. From those labels, we observe that the formulas in the unrolled CFA nicely match the subformulas in Eq. (1). We name the formula matching $I(s_0)$ *prefix formula*, the formula matching $T(s_0, s_1)$ *loop formula*, and the formula matching $T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$ *suffix formula*.



(a) Summarized single-loop CFA **(b)** k -Unrolling

Fig. 3 A summarized single-loop CFA (**a**) and its k-unrolling (**b**)

We use the example CFA in Fig. 1b to illustrate how the LBE-based adoption of IMC separates the analysis of the control flow and the semantical reasoning about program states. The adoption avoids the usage of symbolic program counters in the formulas and hence often leads to more helpful interpolants. Recall that in Sect. 4, a copy of the transition relation means the execution of one program edge. Therefore, the BMC query $I(s_0) \wedge T(s_0, s_1) \wedge \neg P(s_1)$ is equivalent to $(pc_0 = l_3) \wedge (pc_0 = l_3 \wedge x_1 = 0 \wedge pc_1 = l_4) \wedge (pc_1 = l_8)$, which encodes only a single step from the initial program location l_3. Since the error location l_8 is syntactically unreachable from l_3 via one edge, the resulting interpolant does not need to concern the program variable x. In practice, an interpolant for this query could be $pc_1 \neq l_8$.

By contrast, if the required predicates $I(s)$, $T(s, s')$, $P(s)$ are obtained with LBE, the BMC query $I(s_0) \wedge T(s_0, s_1) \wedge \neg P(s_1)$ is equivalent to $x_0 = 0 \wedge \neg(r_0 = 0) \wedge x_1 = x_0 + 2 \wedge r_1 = 0 \wedge \neg(x_1 \% 2 = 0)$, where r_0 and r_1 denote the first and second returned values from the function `nondet()`, respectively. It represents the semantics of all syntactically feasible paths from l_3 to l_8 that visit the loop-head location l_4 twice. By Craig's interpolation theorem, the interpolant has to talk about the program variable x. In fact, as will be shown in Sect. 6.3, IMC is able to prove the correctness of the example CFA by deriving the loop invariant $x \% 2 = 0$ as an interpolant.

5.1 Effect of Single-Loop Transformation

In our approach, programs with multiple loops are transformed to single-loop programs before IMC is applied. The transformation introduces a fresh loop-head location, which is the unique entry to the new single loop, and a location variable to track which old loop should be entered next. Auxiliary logic is added to the CFA to redirect the control flow between the new loop head and the old ones based on the location variable.

Due to the existence of the location variable, there might be trivially infeasible program paths that enter a different loop from the one required by the location variable. As discussed in Sect. 4, trivially unsatisfiable queries often result in weak interpolants that prevent IMC from converging to a fixed point. One solution to this issue is to use a dedicated CPA to track

the location variable, which works similarly to the Location CPA. This CPA will not produce successors that enter a wrong loop, and when used in a composite CPA, it eliminates the aforementioned infeasible paths from the analysis.

In our experiments, we evaluated the IMC adoption with and without the CPA tracking the location variable. The latter treats the location variable as a normal variable and encodes it in the SMT formulas. No significant difference was observed from the empirical results. Unlike the issue of symbolic program counters discussed in Sect. 4, having the location variable (and the related infeasible program paths) in the BMC queries does not slow down the convergence of IMC. This is because the BMC queries obtained by LBE also include syntactically feasible paths. To refute these syntactically feasible paths, the interpolants cannot be trivial and have to concern the program variables. Since the performance of the two alternatives are similar, we stick to the one without the additional CPA for simplicity.

6 Implementation in CPAchecker

In this section, we will describe an implementation to adopt IMC with large-block encoding. We implemented the proposed adoption in the verification framework CPAchecker [19], leveraging its flexibility provided by configurable program analysis [17]. Before delving into implementation details, we emphasize that the idea to extract a transition relation with LBE is general and independent of the underlying framework. We chose to implement the proposed adoption in CPAchecker because it provides (1) the necessary components for the adoption, which are highly configurable, and (2) the implementations of various state-of-the-art software-verification algorithms, which is convenient for the evaluation.

6.1 Data Structures

The *Predicate CPA* for ABE [20] serves as the core data structure in our IMC adoption to store formulas that encode program semantics. We add to an abstract state $(\psi, l^{\psi}, \varphi)$ of the Predicate CPA a *block formula* σ, which encodes all possible paths from the previous abstraction location to the current abstraction location and is used to compute the abstraction formula. In the implementation of CPAchecker, a block formula is already stored in the data structure for abstraction formulas. We append it to an abstract state of the Predicate CPA in order to make the subsequent discussion more understandable.

With the help of ABE, we can achieve the effect of LBE via using the block-adjustment operator blk^l [20]. The operator blk^l will make the Predicate CPA convert an intermediate state to an abstraction state if the current program location is at the loop head or the error location. Under this configuration, the unrolled ARG, if projected to abstraction states, will have a similar structure to Fig. 3b. Therefore, we can easily obtain the required formulas by collecting and combining the block formulas from the corresponding abstraction states in the ARG.

It is worth noting that here we take advantage of the flexibility of the Predicate CPA: By choosing an appropriate implementation for the block-adjustment operator, we can configure the Predicate CPA to be suitable for IMC (together with the algorithms described in the following) without further changes to its definition. Other choices for its operators would allow it to implement different algorithms like Impact, predicate abstraction, and k-induction [14]. Using the Predicate CPA as common framework not only highlights conceptual differences and similarities between the approaches but also allows for comparing them experimentally with the set of confounding variables kept to a minimum.

6.2 Algorithmic Procedures

We present an algorithm for the adoption of IMC to software verification in Algorithm 1, which is based on the CPA++ algorithm [14]. The algorithm assumes single-loop programs as input. We apply single-loop transformation [1, 39] to input programs with multiple loops as a preprocessing. The algorithm takes as input an upper limit k_{max} for a counter k that tracks the number of loop iterations on a program path[2] and a composite CPA consisting of the Location CPA, the Predicate CPA, and the Loop-Bound CPA.

Algorithm 1 IMC: main procedure

Input: an upper limit k_{max} for the loop unrolling bound k,
a composite CPA $\mathbb{D}$ with components: the Location CPA $\mathbb{L}$,
the Predicate CPA $\mathbb{P}$, and the Loop-Bound CPA $\mathbb{LB}$
Output: **false** if an error path to l_E is found,
true if a fixed point is obtained,
unknown otherwise

1: $k := 1$
2: $e_0 := (l_0, (true, l_0, true, true), \{l_H \mapsto -1\})$ // Create initial abstract state at l_0
3: reached := waitlist := $\{e_0\}$
4: **while** $k \leq k_{max}$ **do**
5: (reached, waitlist) := CPA++($\mathbb{D}$, reached, waitlist, k)
6: $\sigma_p := \sigma \mid (l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto 0\}) \in$ reached
7: $\sigma_l := true$
8: **if** $k > 1$ **then**
9: $\sigma_l := \sigma \mid (l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto 1\}) \in$ reached
10: $\sigma_s := \bigwedge_{i=2}^{k-1} \sigma \mid (l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto i\}) \in$ reached $\wedge$
$\bigvee \{\sigma \mid (l_E, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto (k-1)\}) \in$ reached$\}$
11: **if** sat($\sigma_p \wedge \sigma_l \wedge \sigma_s$) **then**
12: **return false** // Found an error path via BMC query
13: **if** $k > 1$ **and** reach_fixed_point(σ_p,σ_l,σ_s) **then**
14: **return true** // Obtained a fixed point via interpolation
15: $k := k + 1$
16: **return unknown**

Algorithm 2 IMC: reach_fixed_point(σ_p,σ_l,σ_s)

Input: prefix formula σ_p, loop formula σ_l, and suffix formula σ_s
Output: **true** if a fixed point is reached, **false** otherwise

1: image := start := σ_p // Set current reachable and starting states to initial states
2: **while** ¬sat(start $\wedge \sigma_l \wedge \sigma_s$) **do**
3: τ := get_interpolant(start $\wedge \sigma_l, \sigma_s$) // formula A: start $\wedge \sigma_l$; formula B: σ_s
4: τ := shift_variable_index(τ, σ_p)
5: **if** ¬sat($\tau \wedge$ ¬image) **then**
6: **return true** // Interpolant implies image: fixed point
7: image := image $\vee \tau$ // Find new states: enlarge image
8: start := τ // Start next iteration from new states
9: **return false** // Reach error: might be wrong alarm

[2] While the algorithm CPA++ unrolls the program k times, the algorithm Algorithm 1 uses the last unrolling only for encoding the predicate $P(s)$ and thus only $k-1$ copies of $T(s, s')$ appear in its BMC query. This is done for consistency with other algorithms expressed on top of the same unifying framework [14].

An abstract state of the composite CPA is $(l, (\psi, l^{\psi}, \varphi, \sigma), \{l_H \mapsto i\})$, where the first element is an abstract state of the Location CPA representing the current program location l, the second element is an abstract state of the Predicate CPA as explained above, and the third element is an abstract state of the Loop-Bound CPA recording that the loop body starting from l_H has been completely traversed i times already. We also use the ARG CPA $\mathbb{A}$ to store the predecessor-successor relationship between abstract states. To increase readability, we simply use abstract states as elements in the ARG and explicitly give the unrolling upper bound k as a parameter to the CPA++ algorithm (instead of passing it via the precision of the initial abstract state as in the literature [14]). We also omit the aborting function and assume that the CPA++ algorithm never aborts early (i.e., we pass $\mathsf{abort}^{never} = \{\cdot \mapsto \mathit{false}\}$).

After unrolling the CFA with the CPA++ algorithm (line 5), we have to collect prefix, loop, and suffix formulas to pose a BMC query and perform the fixed-point computation via interpolation. The formula collection is described in lines 6 to 10, where we write $\sigma|(l, (\psi, l^{\psi}, \varphi, \sigma), \{l_H \mapsto i\})$ to denote the block formula σ of the abstract state $(l, (\psi, l^{\psi}, \varphi, \sigma), \{l_H \mapsto i\})$. The prefix formula σ_p is the block formula of the abstract state $(l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto 0\})$; if the loop body has been completely traversed at least once, i.e., $k > 1$, the loop formula σ_l is the block formula of the abstract state $(l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto 1\})$, otherwise, it is set to *true*; the suffix formula σ_s is the conjunction of the following two formulas: the conjunction of block formulas of the abstract state $(l_H, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto i\})$ for $i = 2, \ldots, (k-1)$ and the disjunction of the block formulas of the abstract states $(l_E, (\cdot, \cdot, \cdot, \sigma), \{l_H \mapsto (k-1)\})$.

Note that the above formula collection at abstract states whose locations equal l_H is unambiguous, meaning that there is a unique abstract state satisfying the conditions imposed by the Location CPA $\mathbb{L}$ and the Loop-Bound CPA $\mathbb{LB}$. This is because we assume single-loop programs and use LBE to summarize all paths between two adjacent abstraction states. After collecting these formulas, the BMC query is simply the conjunction of the prefix, loop, and suffix formulas. If the BMC query is unsatisfiable, we try to compute a fixed point using Algorithm 2, which implements the procedure described in Sect. 3.1 to iteratively derive interpolants from unsatisfiable BMC queries and grow a fixed point as their union.

Algorithm 2 first initializes both image, which stores an overapproximation of the reachable states, and start, which stores the starting states of BMC queries, to be the prefix formula. Using start $\wedge\, \sigma_l$ as formula A and σ_s as formula B, we derive an interpolant τ. As discussed in Sect. 3.1, the i^{th} interpolant is an overapproximation of the reachable states after i loop iterations. We change the variables used in the interpolant to those in the prefix formula and check whether the interpolant implies image. If so, a fixed point has been reached, and we conclude the property is true; otherwise, we enlarge image by adding the states contained in the interpolant to it and pose another BMC query starting from the interpolant. If any BMC query during the iteration is satisfiable, we return back to Algorithm 1 and increase the loop-unrolling counter k to check whether the violation is a wrong alarm.

6.3 Example

We demonstrate step-by-step how to apply Algorithm 1 and Algorithm 2 to verify the CFA in Fig. 1b. The ARG constructed by the CPA++ algorithm when $k = 2$ is shown in Fig. 4. In this figure, each abstract state is a tuple $(l, (\psi, l^{\psi}, \varphi, \sigma), \{l_4 \mapsto i\})$ of the abstract states of $\mathbb{L}$, $\mathbb{P}$, and $\mathbb{LB}$. Note that every abstract state in the ARG has an abstraction formula ψ (the first element in an abstract state of Predicate CPA) equal to *true* because IMC does not compute an abstraction formula. Instead, it relies on interpolants for the abstraction of program states.

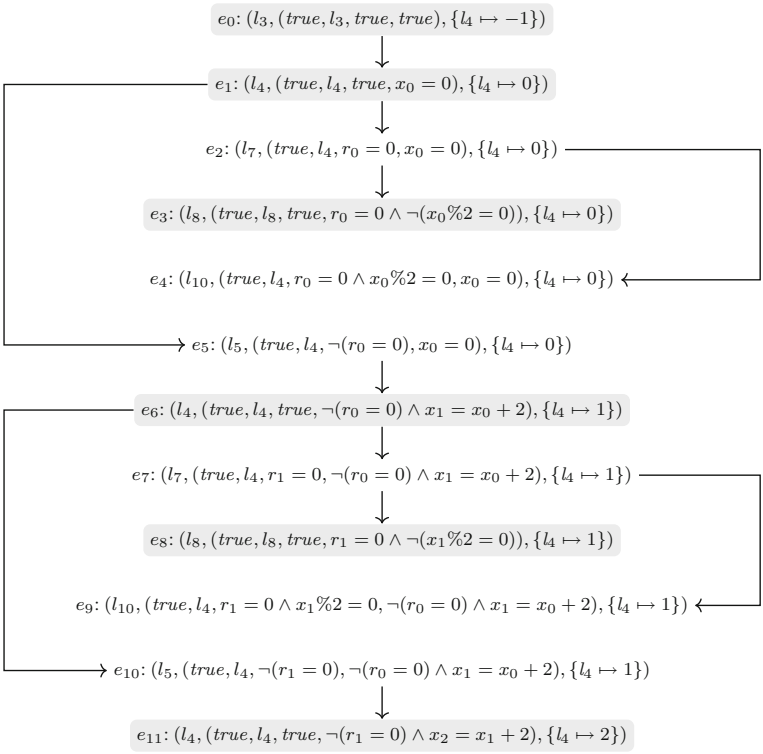


Fig. 4 ARG constructed by the CPA$\pitchfork$ algorithm [14] for the CFA in Fig. 1b ($k = 2$)

Abstract states whose predicate abstract state is an abstraction state (where the path formula is always reset to *true*) are highlighted in gray. We use r to denote the returned value of the function `nondet`.

The prefix formula σ_p is the block formula $x_0 = 0$ of the abstract state e_1, the loop formula σ_l is the block formula $\neg(r_0 = 0) \wedge x_1 = x_0 + 2$ of the abstract state e_6, and the suffix formula σ_s is the block formula $r_1 = 0 \wedge \neg(x_1 \% 2 = 0)$ of the abstract state e_8 (note that the block formula of the abstract state e_3, which also has location l_8, is not selected because $l_4 \mapsto 0$ does not match in line 10 of Algorithm 1). As the BMC query $x_0 = 0 \wedge \neg(r_0 = 0) \wedge x_1 = x_0 + 2 \wedge r_1 = 0 \wedge \neg(x_1 \% 2 = 0)$ is unsatisfiable, we try to compute a fixed point using Algorithm 2.

Variables image and start are initialized to $x_0 = 0$. Using $x_0 = 0 \wedge \neg(r_0 = 0) \wedge x_1 = x_0 + 2$ as formula A and $r_1 = 0 \wedge \neg(x_1 \% 2 = 0)$ as formula B, we can derive an interpolant τ from the unsatisfiable BMC query. Assume that τ is $x_1 \% 2 = 0$, referring to the common variable x_1 of formulas A and B. After shifting the variable to the one used in σ_p, we obtain $x_0 \% 2 = 0$. As the interpolant does not imply image, we enlarge the current image by disjoining it with the interpolant. The computation is then repeated again, with start equal to $x_0 \% 2 = 0$ this time. The BMC query in the second iteration becomes $x_0 \% 2 = 0 \wedge \neg(r_0 = 0) \wedge x_1 = x_0 + 2 \wedge r_1 = 0 \wedge \neg(x_1 \% 2 = 0)$, which is still unsatisfiable. Assume the interpolant is again $x_1 \% 2 = 0$. Obviously, we have reached a fixed point, as the newly derived interpolant implies image. Therefore, we conclude that the property holds.

6.4 Correctness

It is straightforward to see that Algorithm 1 is *precise*, i.e., does not produce wrong alarms, because if it returns **false**, then the BMC query for all paths from l_0 to l_E at line 11 is satisfiable, which implies that the CFA has a feasible path to l_E. More interesting is the soundness of Algorithm 1, i.e., whether it may produce wrong proofs, which we discuss in the following. Its soundness follows from that of large-block encoding [11] and the original IMC algorithm [56]. We state the soundness of Algorithm 1 when it is applied to a single-loop CFA in Theorem 1. For CFAs with multiple loops, the soundness will also depend on that of the single-loop transformation [1, 39].

Theorem 1 *Given a single-loop CFA A and its corresponding composite CPA $\mathbb{D}$ as input, if Algorithm 1 returns **true** upon $\mathbb{D}$, then A does not have a feasible path to l_E.*

Proof We prove the statement by contradiction. Suppose Algorithm 1 returns **true** when the value of the loop-unrolling counter k equals $\hat{k}$, but the single-loop CFA has a feasible path to l_E. We split into two cases based on the number $\hat{h}$ of the visits to l_H on the error path.

First, assume $\hat{h} \leq \hat{k}$. Thanks to the sound summarization of LBE, the formula of the error path must imply $\sigma_p \wedge \sigma_l \wedge \sigma_s$ when $k = \hat{h}$. Therefore, Algorithm 1 should have returned **false** at $k = \hat{h}$, because the BMC query at line 11 of Algorithm 1 is satisfiable. This result contradicts the assumption that Algorithm 1 returns **true**.

Second, assume $\hat{h} > \hat{k}$. Such an error path indicates the existence of a state $\hat{s}$ that is reachable from l_0 by traversing the loop $\hat{h} - \hat{k}$ times and will reach l_E after further traversing the loop $\hat{k} - 1$ times. We will show that Algorithm 2 will return **false** after discovering $\hat{s}$ via interpolation. Note that Algorithm 2 cannot return **true** before finding $\hat{s}$ because the state must be contained in the computed fixed point.

According to the property of the original IMC algorithm described in Sect. 3.1, the interpolant derived in the i^{th} while-loop iteration of Algorithm 2 is an overapproximation of the set of states reachable from l_0 by traversing the loop i times. Therefore, $\hat{s}$ must belong to the interpolant τ derived in the $(\hat{h} - \hat{k})^{th}$ while-loop iteration of Algorithm 2, which will be used as new starting states in the next iteration. Moreover, because of the soundness of LBE, the formula from the $(\hat{h} - \hat{k} + 1)^{th}$ l_H to l_E (involving $\hat{k}$ visits to l_H) on the error path must imply $\sigma_l \wedge \sigma_s$ when we enter Algorithm 2 with $k = \hat{k}$. Thus, in the beginning of the next iteration, the satisfiability query at line 2 of Algorithm 2 must be satisfiable, which makes Algorithm 2 return **false**. This in turn prevents Algorithm 1 from returning **true** when $k = \hat{k}$, contradicting our assumption.

Having analyzed the above two possibilities, we conclude that such a feasible error path does not exist, and hence Algorithm 1 is sound. □

6.5 Backward Derivation of Interpolants

Notice that in the example of Sect. 6.3, the "quality" of interpolants heavily affects the convergence of the fixed-point computation. For example, instead of $x_1 \% 2 = 0$, which is actually the loop invariant, suppose the interpolant derived by the solver is $x_1 = 2$. Starting from this interpolant, we might be trapped in a sequence of interpolants $x_1 = 4$, $x_1 = 6$, $x_1 = 8, \ldots$ and never reach a fixed point.

While in general it is difficult to control the interpolation process of the solver, there is a trick to mitigate this problem. First, we switch the labels of the two formulas, i.e., we label the

original formula B as the new formula A and the original formula A as the new formula B. Second, we ask the solver to derive an interpolant for the new formulas and then negate it. The negated interpolant is a valid interpolant for the original formulas A and B. In other words, instead of get_interpolant(A, B), we use ¬get_interpolant(B, A).

Using this trick in IMC, we are actually deriving interpolants backwards from the safety property. Therefore, we call it *backward derivation* of interpolants. With the backward derivation, we can in practice often avoid the bad interpolant $x_1 = 2$ and obtain the good one $x_1 \% 2 = 0$ for fast convergence of the example program in Sect. 6.3. Empirically, we found that the backward derivation performs slightly better than the forward derivation. This phenomenon might be attributed to the fact that deriving the interpolants backward from the safety property side is likely to yield interpolants summarizing information more relevant to proving the property. As a result, we use it as default in our implementation.

7 Evaluation

To evaluate the proposed adoption of IMC [56] and understand its characteristics, we carried out two parts of experiments to answer the research questions below:

- Part 1: IMC vs. other SMT-based algorithms
  - **RQ1**: Can IMC solve more safety-verification tasks?
  - **RQ2**: Can IMC solve safety-verification tasks faster?
  - **RQ3**: Can IMC solve tasks unsolvable by existing approaches?
- Part 2: IMC vs. IMPACT [57] (a closely related interpolation-based algorithm)
  - **RQ4**: Why can IMC deliver more proofs than IMPACT?

We evaluated the adoption of IMC based on large-block encoding against several state-of-the-art SMT-based algorithms on the largest publicly available benchmark suite of C safety-verification tasks [8]. We excluded the naive adoption of IMC with symbolic program counters from the evaluation because it was shown infeasible in the experiment described in Sect. 4.

7.1 Evaluated Approaches

We assessed IMC against five SMT-based verification algorithms, including BMC [25], k-induction [38], predicate abstraction [45], IMPACT [57], and PDR [28]. All of the compared approaches are implemented in CPACHECKER. The implementations of BMC, k-induction, predicate abstraction, and IMPACT are built on top of the CPA++ algorithm in a unified manner [14]. The implementation of PDR in CPACHECKER follows a software-verification adaptation named CTIGAR [26], which was compared against other PDR-related approaches recently in the literature [12]. We did not include other state-of-the-art verifiers in the evaluation to keep confounding variables at a minimum (same parser, same libraries, same SMT solver, etc.). We chose CPACHECKER because it is a flexible framework that performed well in the competitions. Empirical results of CPACHECKER against other software verifiers are available from the report [8] of the 2022 Competition on Software Verification (SV-COMP'22).

7.2 Benchmark Set

As the benchmark set, we used the verification tasks [9] from SV-COMP'22. We used only verification tasks where the safety property is the reachability of a program location. From those, we further excluded verification tasks that are not supported by at least one of the compared approaches, e.g., those from the categories *ReachSafety-Recursive* and *ConcurrencySafety-Main*. The resulting benchmark set consists of a total of 6024 verification tasks from the subcategories *AWS-C-Common-ReachSafety*, *DeviceDriversLinux64-ReachSafety*, *DeviceDriversLinux64Large-ReachSafety*, and *uthash-ReachSafety* of the category *SoftwareSystems* and from the following subcategories of the category *ReachSafety*: *Arrays*, *Bitvectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, *Sequentialized*, *XCSP*, and *Combinations*. A total of 1793 tasks in the benchmark set contain a known specification violation, while the other 4231 tasks are assumed to be correct.

7.3 Experimental Setup

Our experiments were performed on machines with one 3.4 GHz CPU (Intel Xeon E3-1230 v5) with 8 processing units and 33 GB of RAM each. The operating system was Ubuntu 22.04 (64 bit), using Linux 5.15 and OpenJDK 17.0.5. Each verification task was limited to two CPU cores, a CPU time of 15 min, and a memory usage of 15 GB. We used BenchExec[3] [22] to achieve reliable benchmarking and revision 43042 of branch `cfa-single-loop-transformation` of CPAchecker for evaluation. We configured CPAchecker to use the SMT theories of equality with uninterpreted functions, bit vectors, floats, and arrays. All SMT queries were handled by MathSAT5 [33].

7.4 Results

RQ1: Effectiveness of IMC

The experimental results of all compared approaches are summarized in Table 2. Observe that IMC produced the most correct results for both proofs and alarms among the interpolation-based approaches (IMC, PDR, predicate abstraction, and Impact) and was second only to k-induction in the evaluation. In comparison to the most-related approach Impact, IMC proved the safety of 328 more programs and found 25 more bugs (an increase of 21 % and 3 %, respectively). We will study the underlying mechanism that enables IMC to deliver more proofs than Impact in RQ4. Meanwhile, BMC generated the most correct alarms as expected, and k-induction correctly solved the most tasks, with the most correct proofs and the second-most correct alarms. Moreover, although IMC is a new addition to CPAchecker, it did not produce any wrong proof in the evaluation, identical to the other long-established approaches in the software-verification framework. We consider the 3 wrong alarms of IMC not caused by our implementation. They are related to the program encoding of CPAchecker, and other approaches, such as predicate abstraction, also failed to solve these tasks correctly.

[3] https://github.com/sosy-lab/benchexec

Table 2 Summary of the results for 6024 reachability-safety verification tasks

Algorithm	IMC	PDR	BMC	k-Induction	Predicate Abstraction	IMPACT
Correct results	2766	1599	2388	3157	2346	2413
Proofs	1871	1136	1211	2158	1531	1543
Alarms	895	463	1177	999	815	870
Wrong proofs	0	0	0	0	1	0
Wrong alarms	3	1	1	1	2	2
Timeouts	2018	3373	2227	1841	1955	1725
Out of memory	160	23	363	221	13	107
Other inconclusive	1077	1028	1045	804	1707	1777

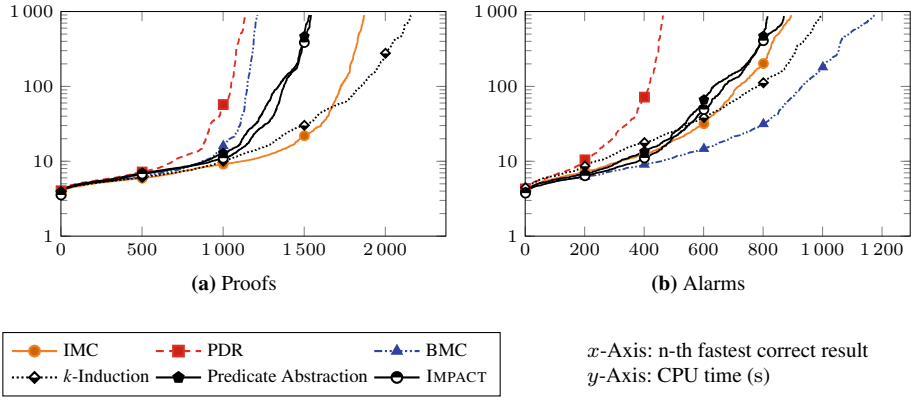


Fig. 5 Quantile plots for all correct proofs and alarms

RQ2: Efficiency of IMC

To study the run-time efficiency of IMC, we present the quantile plots for the compared approaches in Fig. 5 and the scatter plots for CPU time spent on correctly solved tasks in Fig. 6. The quantile plots for the correct proofs and alarms of the compared approaches are shown in Fig. 5a and b, respectively. A data point (x, y) in the plots indicates that there are x tasks correctly solved by the respective algorithm within a CPU time of y seconds each. Note that IMC is not only effective in producing proofs and alarms but is also efficient. From Fig. 5, we see that it is the most efficient and effective interpolation-based approach in the evaluation.

The scatter plots for the correctly solved tasks (including both proofs and alarms) of the compared approaches are shown in Fig. 6. We omitted the scatter plot for BMC as it is mainly inclined to bug hunting, while other approaches have more balanced behavior. A data point (x, y) in the plots indicates that there is a task correctly solved by both IMC and a compared approach, while IMC took a CPU time of y seconds and the other approach took a CPU time of x seconds. Observe that IMC is often more efficient than a compared approach. For example, while it solved fewer tasks compared to k-induction, its time efficiency is often better than k-induction on the tasks which can be solved by both algorithms. This phenomenon could be explained by the fact that, unlike k-induction, which relies on an external procedure to generate auxiliary invariants, IMC generates interpolants from BMC queries and uses them to

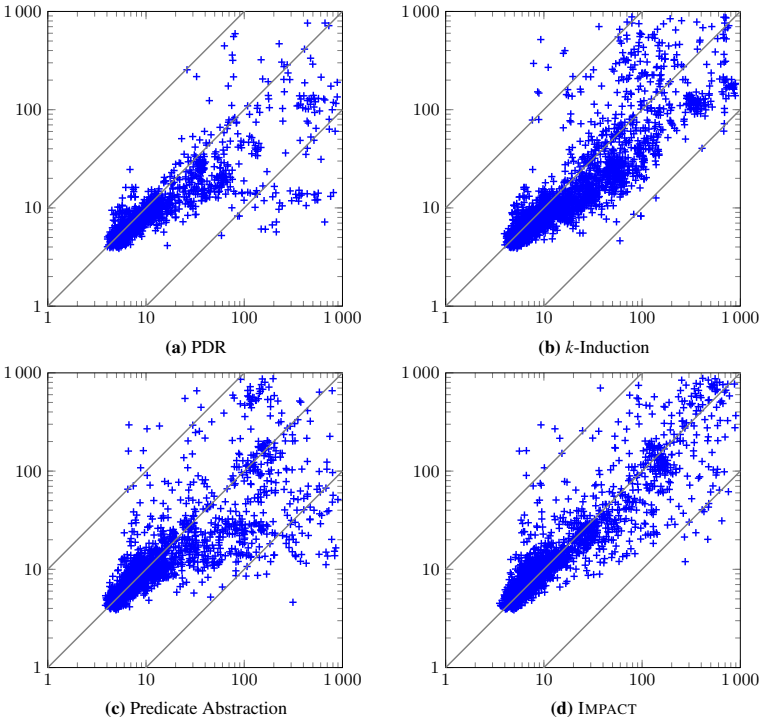


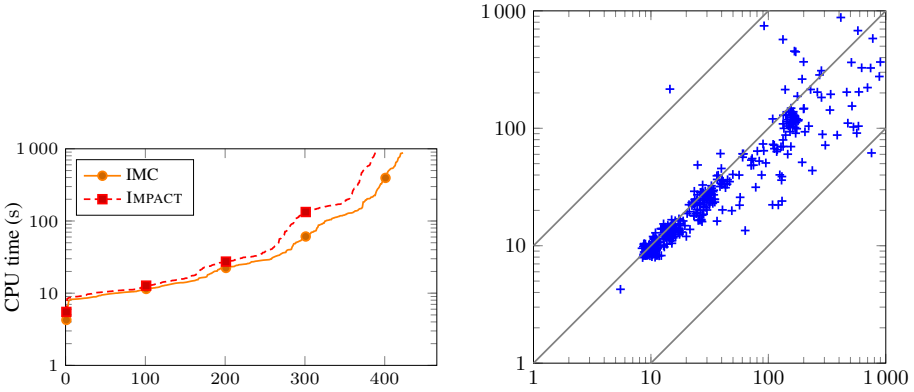
Fig. 6 Scatter plots of CPU time in seconds for all correct results with IMC in y-axis and compared approaches in x-axis

construct fixed points purely internally. Moreover, when representing *helpful* loop invariants, i.e., those that help to prove the safety property of a program, requires complex formulas, the interval-based data-flow analysis [10] used by the default configuration [13] of k-induction in CPACHECKER is disadvantageous because the expressiveness of candidate invariants is limited. By contrast, IMC is favorable in such cases since it constructs a candidate fixed point (also a loop invariant) as a union of previously derived interpolants, which in principle can encode any combination of reachable states.

RQ3: Enhancing Software Verification with IMC

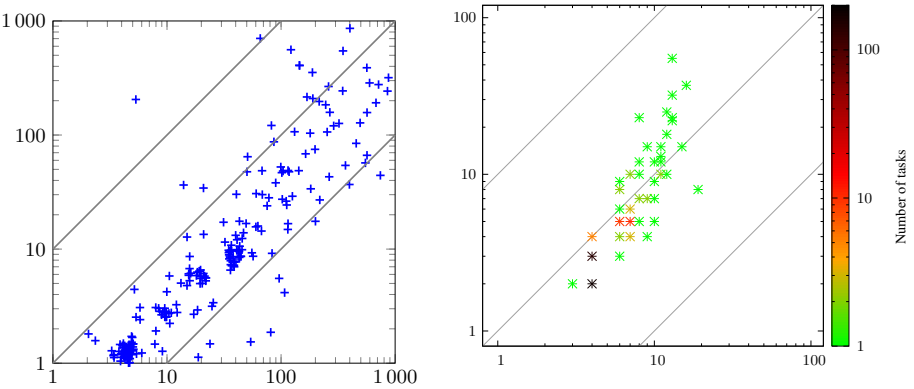
To highlight IMC's contribution to software verification, we report the numbers of tasks solvable by IMC but not by a compared approach because it ran out of resources. In our evaluation, IMC solved 1199, 929, 100, 323, and 185 tasks for which PDR, BMC, k-induction, predicate abstraction, and IMPACT, respectively, failed to solve within the time or memory limits. Overall, it uniquely solved 7 tasks for which all other approaches ran out of resources.

IMC performed best for the category *ReachSafety-ECA*. These event-condition-action (ECA) [47] programs have a loop to receive external inputs, generate outputs, and update internal variables based on the ECA rules, implemented by a complex control flow inside the loop. Conceptually, the working of these programs is similar to that of sequential Boolean-logic circuits. IMC naturally performs well on them because it originated from hardware verification. Out of a total 1265 ECA programs, IMC solved a second most 565 tasks, while



(a) n-th fastest correct proof **(b)** CPU time (s)

Fig. 7 Comparing IMC and IMPACT on safe ECA tasks: (**a**) quantile plot for proofs and (**b**) scatter plot for elapsed CPU time of proofs with IMC in y-axis and IMPACT in x-axis



(a) Total interpolation time (s) **(b)** Numbers of interpolation queries

Fig. 8 Scatter plots of (**a**) total interpolation time and (**b**) numbers of interpolation queries used to prove safe ECA tasks with IMC in y-axis and IMPACT in x-axis

predicate abstraction, IMPACT, and k-induction solved 476, 555, and 607 respectively. We will use the ECA tasks without property violation (namely, safe ECA tasks) to study the performance characteristics of IMC and answer why it can deliver more correctness proofs than IMPACT, a closely related interpolation-based approach.

RQ4: Performance Characteristics of IMC

IMC is the best interpolation-based verification algorithm in our evaluation. To profile its performance characteristics and understand why it can deliver more proofs, we compared IMC and IMPACT on the *ReachSafety-ECA* tasks without property violation. Among the 785 safe ECA tasks, IMC and IMPACT proved the correctness of 423 and 390 of them, respectively. The quantile plot in Fig. 7a shows that IMC not only delivered more proofs than IMPACT but also spent less CPU time finding the proofs. The scatter plot in Fig. 7b further demonstrates that IMC usually obtained a proof faster than IMPACT when both methods succeeded.

To understand the advantage of IMC over IMPACT for finding proofs, we investigate the most essential and time-consuming step in their computation, namely, interpolation. The total interpolation time and numbers of interpolation queries used by IMC and IMPACT to produce proofs are compared in the scatter plots Fig. 8a and 8b, respectively. In Fig. 8b, the color of a data point indicates the number of tasks falling into this coordinate. From the scatter plots, observe that IMC usually required fewer interpolation queries and less interpolation time to prove a task. Among the 382 safe ECA tasks proved by both IMC and IMPACT, IMC invoked fewer interpolation calls for 354 of them. This phenomenon indicates that the quality of interpolants derived by IMC were high, which enabled it to generalize better than IMPACT on these tasks.

We attribute the quality of interpolants to two factors. First, IMC is known to generalize better than approaches based on interpolation sequences in hardware verification [30]. Unlike algorithms based on interpolation sequences [57, 62], IMC derives interpolants from not only the initial states but also the previous interpolants. Such eager abstraction decreases the numbers of interpolation queries required to reach a fixed point. We observed the same effect for software verification, as exhibited in Fig. 8b. Second, the proposed adoption of IMC analyzes the control-flow structures separately and only encodes syntactically feasible program paths without using symbolic program counters in the formulas. Therefore, the underlying SMT solver can focus on the semantics of the program and derive useful interpolants about the actual program variables. The proposed adoption is crucial for unlocking the potential of IMC for software verification.

Answers to the Research Questions

For the first part of our evaluation where IMC was compared against five SMT-based verification approaches, the proposed approach with large-block encoding is effective and efficient. Adopted with the proposed method, IMC, the first interpolation-based formal-verification approach ever invented, is competitive against other state-of-the-art algorithms, which have been investigated much more by the research community. The conclusion is well supported by the experimental results: Our IMC implementation not only solved the second most verification tasks (Table 2) in the evaluation but was also efficient compared to other SMT-based approaches (Fig. 6). In our experiments, it was the most efficient and effective interpolation-based approach (Fig. 5). It uniquely solved 7 tasks for which all other approaches ran out of resources, indicating its unique value to complement existing approaches.

In the second part of the evaluation, IMC was compared to IMPACT on a subset of the SV-COMP'22 benchmark set to study its strength to find proofs. We observed that IMC spent less effort on interpolation than IMPACT (Fig. 8), indicating that it derives high-quality interpolants and generalizes better. The reason behind this phenomenon is that IMC eagerly computes interpolants from not only the initial states but also the previous interpolants. The same effect is also reported for hardware verification [30].

7.5 Threats to Validity

Here we discuss some threats that may affect the validity of our conclusions and how we limited them. To ensure internal validity, all the compared algorithms are implemented in the verification framework CPACHECKER [19]. This practice minimizes the confounding variables (front ends and utilities) and rules out differences unrelated to the algorithms. We also use BENCHEXEC [22] to ensure best possible measurement accuracy. To reduce the external threat

resulting from the selection bias of verification tasks, we conduct the experiments using the largest publicly available benchmark set [9] of C safety-verification tasks. Other external threats arise from the selection of the compared approaches and underlying framework. It is clear from the literature [14] that the compared approaches in this paper indeed represent the state of the art of software verification; the only missing main related state-of-the-art approach is trace abstraction [44], for which the implementation in the framework is not yet mature enough. Moreover, the chosen platform CPACHECKER is a well-maintained software project that performs well in the competitions, and the relative performance between CPACHECKER and other verifiers is available from SV-COMP'22 [8].

8 Conclusion

Software verification is a hard problem, and it is imperative to leverage as much knowledge of the verification community as possible. Interpolation-based model checking (McMillan, 2003 [56]) is a successful hardware-verification algorithm, but in contrast to many other interpolation-based verification approaches, this algorithm was not yet adopted to software verification, and the characteristics of the algorithm when applied to software systems were unknown. This paper presents the first theoretical adoption and practical implementation of the algorithm for software verification, providing a base-line for other researchers to build on. Surprisingly, it has taken two decades to close this significant gap of knowledge by investigating the applicability to software verification. We present the novel idea of utilizing the well established technique of large-block encoding to extract transition relations from programs, without encoding the control-flow structure of the program into the formulas via symbolic program counters. The proposed adoption was implemented in the open-source software-verification framework CPACHECKER and evaluated against other state-of-the-art software-verification algorithms on a large benchmark set of C verification tasks for reachability properties.

Among the competing approaches, our implementation achieved a comparable performance, evaluated in terms of both effectiveness (the number of correctly solved tasks) and efficiency (the CPU time to solve tasks). Our IMC implementation was the most effective and efficient interpolation-based approach in the evaluation. Furthermore, the new approach was able to solve 7 programs for which all other approaches ran out of resources (15 min CPU time or 15 GB memory usage), which shows that the new approach *improves* the state of the art and *complements* the other approaches. We hope that our promising results stimulate other researchers to further improve the approach for software verification and that our open-source implementation in the flexible framework CPACHECKER helps other researchers to understand the details of the algorithm.

Funding Open Access funding enabled and organized by Projekt DEAL. This project was funded in part by the LMU Postdoc Support Fund.

Data Availability To ensure verifiability and transparency of the results reported in this paper, all used software, input programs, and raw experimental results are available in a supplemental reproduction package [21]. For convenient browsing through the results, interactive tables are available at https://www.sosy-lab.org/research/cpa-imc. Current versions of CPACHECKER are also available at https://cpachecker.sosy-lab.org.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Boston (1986). https://www.worldcat.org/isbn/978-0-201-10088-4
2. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: Proc. CAV, LNCS 7358, pp. 672–678. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_48
3. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: An extension of lazy abstraction with interpolation for programs with arrays. Form. Methods Syst. Des. **45**(1), 63–109 (2014). https://doi.org/10.1007/s10703-014-0209-9
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Proc. IFM, LNCS 2999, pp. 1–20. Springer (2004). https://doi.org/10.1007/978-3-540-24756-2_1
5. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI, pp. 203–213. ACM (2001). https://doi.org/10.1145/378795.378846
6. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL, pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274
7. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_11
8. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2), LNCS 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
9. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5831003
10. 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
11. 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
12. Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Proc. TACAS (1), LNCS 12078, pp. 3–21. Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_1
13. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV, LNCS 9206, pp. 622–640. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
14. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
15. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
16. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. **9**(5–6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
18. 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
19. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV, LNCS 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
20. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010). https://ieeexplore.ieee.org/document/5770949
21. Beyer, D., Lee, N.Z., Wendler, P.: Reproduction package for article 'Interpolation and SAT-based model checking revisited'. Zenodo (2023). https://doi.org/10.5281/zenodo.8245824
22. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
23. Beyer, D., Petrenko, A.K.: Linux driver verification. In: Proc. ISoLA, LNCS 7610, pp. 1–6. Springer (2012). https://doi.org/10.1007/978-3-642-34032-1_1

24. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Proc. CAV, LNCS 5123, pp. 304–308. Springer (2008). https://doi.org/10.1007/978-3-540-70545-1_29
25. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS, LNCS 1579, pp. 193–207. Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
26. Birgmeier, J., Bradley, A.R., Weissenbacher, G.: Counterexample to induction-guided abstraction-refinement (CTIGAR). In: Proc. CAV, LNCS 8559, pp. 831–848. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_55
27. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: Proc. TACAS, LNCS 13243, pp. 524–542. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_29
28. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI, LNCS 6538, pp. 70–87. Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
29. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. In: Proc. FSEN, LNCS 4767, pp. 17–32. Springer (2007). https://doi.org/10.1007/978-3-540-75698-9_2
30. Cabodi, G., Nocco, S., Quer, S.: Interpolation sequences revisited. In: Proc. DATE, pp. 1–6. IEEE (2011). https://doi.org/10.1109/DATE.2011.5763056
31. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM, LNCS 9058, pp. 3–11. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
32. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Proc. CAV, LNCS 7358, pp. 277–293. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_23
33. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS, LNCS 7795, pp. 93–107. Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
34. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
35. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS, LNCS 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
36. Cook, B.: Formal reasoning about the security of Amazon web services. In: Proc. CAV (2), LNCS 10981, pp. 38–47. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_3
37. 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
38. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS, LNCS 6887, pp. 351–368. Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
39. 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
40. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proc. POPL, pp. 191–202. ACM (2002). https://doi.org/10.1145/503272.503291
41. Ghilardi, S., Ranise, S.: Goal-directed invariant synthesis for model checking modulo theories. In: Proc. TABLEAUX, LNCS 5607, pp. 173–188. Springer (2009). https://doi.org/10.1007/978-3-642-02716-1_14
42. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Proc. CAV, LNCS 1254, pp. 72–83. Springer (1997). https://doi.org/10.1007/3-540-63166-6_10
43. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS, LNCS 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
44. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV, LNCS 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
45. 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
46. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
47. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Proc. ISoLA, LNCS 7609, pp. 608–614. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_45
48. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. (2009). https://doi.org/10.1145/1592434.1592438
49. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Proc. CAV, LNCS 3576, pp. 39–51. Springer (2005). https://doi.org/10.1007/11513988_6
50. Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: Proc. FMCAD, pp. 85–92. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886665

51. Kahsai, T., Tinelli, C.: PKIND: A parallel k-induction based model checker. In: Proc. Int. Workshop on Parallel and Distributed Methods in Verification, EPTCS 72, pp. 55–62. EPTCS (2011). https://doi.org/10.4204/EPTCS.72.6
52. Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference, LNCS 5947, pp. 165–176. Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14
53. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in SMT-based unbounded software model checking. In: Proc. CAV, LNCS 8044, pp. 846–862. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_59
54. Kröning, D., Weissenbacher, G.: Interpolation-based software verification with WOLVERINE. In: Proc. CAV, LNCS 6806, pp. 573–578. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_45
55. Lange, T., Neuhäußer, M.R., Noll, T.: IC3 software model checking on control flow automata. In: Proc. FMCAD, pp. 97–104 (2015). https://doi.org/10.1109/FMCAD.2015.7542258
56. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV, LNCS 2725, pp. 1–13. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
57. McMillan, K.L.: Lazy abstraction with interpolants. In: Proc. CAV, LNCS 4144, pp. 123–136. Springer (2006). https://doi.org/10.1007/11817963_14
58. McMillan, K.L.: Lazy annotation for program testing and verification. In: Proc. CAV, LNCS 6174, pp. 104–118. Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_10
59. McMillan, K.L.: Interpolation and model checking. In: Handbook of Model Checking, pp. 421–446. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_14
60. McMillan, K.L., Rybalchenko, A.: Computing relational fixed points using interpolation. Tech. Rep. https://www.microsoft.com/en-us/research/publication/computing-relational-fixed-points-using-interpolation/, Microsoft Research (2013)
61. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Proc. HVC, LNCS 7261, pp. 160–175. Springer (2011). https://doi.org/10.1007/978-3-642-34188-5_15
62. 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
63. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. Program. Comp. Softw. **41**(1), 49–64 (2015). https://doi.org/10.1134/S0361768815010065