A Transferability Study of Interpolation-Based Hardware Model Checking for Software Verification

DIRK BEYER, LMU Munich, Germany
PO-CHUN CHIEN, LMU Munich, Germany
MAREK JANKOLA, LMU Munich, Germany
NIAN-ZE LEE, LMU Munich, Germany

Assuring the correctness of computing systems is fundamental to our society and economy, and *formal verification* is a class of techniques approaching this issue with mathematical rigor. Researchers have invented numerous algorithms to automatically prove whether a computational model, e.g., a software program or a hardware digital circuit, satisfies its specification. In the past two decades, *Craig interpolation* has been widely used in both hardware and software verification. Despite the similarities in the theoretical foundation between hardware and software verification, previous works usually evaluate interpolation-based algorithms on only one type of verification tasks (e.g., either circuits or programs), so the conclusions of these studies do not necessarily transfer to different types of verification tasks. To investigate the transferability of research conclusions from hardware to software, we adopt two performant approaches of interpolation-based hardware model checking: (1) *Interpolation-Sequence-Based Model Checking* (Vizel and Grumberg, 2009) and (2) *Intertwined Forward-Backward Reachability Analysis Using Interpolants* (Vizel, Grumberg, and Shoham, 2013) for software verification. We implement the algorithms proposed by the two publications in the software verifier CPAchecker because it has a software-verification adoption of the first interpolation-based algorithm for hardware model checking from 2003, which the two publications use as a comparison baseline. To assess whether the claims in the two publications transfer to software verification, we conduct an extensive experiment on the largest publicly available suite of safety-verification tasks in the programming language C. Our experimental results show that the important characteristics of the two approaches for hardware model checking are transferable to software verification, and that the cross-disciplinary algorithm adoption is beneficial, as the approaches adopted from hardware model checking were able to tackle tasks unsolvable by existing methods. This work consolidates the knowledge in hardware/software verification and provides open-source implementations to improve the understanding of the compared interpolation-based algorithms.

CCS Concepts: • **Software and its engineering** → **Formal software verification**; *Formal methods*; • **Theory of computation** → *Verification by model checking*; *Program reasoning*; • **Hardware** → *Model checking*.

Additional Key Words and Phrases: Formal Verification, Craig Interpolation, Model Checking, Software Verification, Transferability, Replicability, Reproducibility

ACM Reference Format:
Dirk Beyer, Po-Chun Chien, Marek Jankola, and Nian-Ze Lee. 2024. A Transferability Study of Interpolation-Based Hardware Model Checking for Software Verification . *Proc. ACM Softw. Eng.* 1, FSE, Article 90 (July 2024), 23 pages. https://doi.org/10.1145/3660797

Authors' Contact Information: Dirk Beyer, LMU Munich, Munich, Germany, dirk.beyer@sosy-lab.org; Po-Chun Chien, LMU Munich, Munich, Germany, po-chun.chien@sosy.ifi.lmu.de; Marek Jankola, LMU Munich, Munich, Germany, marek.jankola@sosy.ifi.lmu.de; Nian-Ze Lee, LMU Munich, Munich, Germany, nian-ze.lee@sosy.ifi.lmu.de.

ACM 2994-970X/2024/7-ART90
https://doi.org/10.1145/3660797

1 INTRODUCTION

Formal verification aims to analyze computing systems with mathematical rigor. In the past few decades, it has received attention from both academy and industry. Since the modern society relies heavily on computing systems, formal verification is an indispensable pillar to assure their quality and correctness. Among various schools of formal verification, *model checking* [1] is a class of fully automatic methods for the following problem: Given a description of a computational model and a specification for the model as input, decide whether the model satisfies the specification. While the problem of model checking is undecidable in general, researchers have invented approaches applicable to practical systems.

Numerous model-checking techniques have been developed for different computational models. Early studies on model checking mainly target finite-state transition systems [2, 3]. Such a formalism is suitable for modeling sequential digital circuits. The behavior of a sequential circuit can be encoded as propositional formulas because of its finite state space, and *Boolean satisfiability* (SAT) solvers [4] can be used to reason about the circuit under verification. After the breakthrough in SAT solving in the early 2000s [5], SAT-based approaches have become mainstream for finite-state model checking and remained the state of the art up to now. Recent advancements in *satisfiability modulo theories* (SMT) [6] have further enabled SMT-based model-checking algorithms for infinite-state systems, for example, programs. Software verification is an active research field where SMT solving plays an essential role [7]. Many successful SAT-based techniques for hardware model checking, such as bounded model checking (BMC) [8], k-induction [9], and IC3/PDR [10], have been adapted to infinite-state systems and used to verify programs with the help of SMT solving. In addition to SAT/SMT solving, *Craig interpolation* [11] is a cornerstone for both hardware and software verification. It provides model-checking algorithms with the information relevant to an unsatisfiable formula,[1] which the algorithms can leverage to construct invariants of the model.

Despite the common theoretical foundation of hardware and software verification, it is unclear whether the results and observations reported for hardware model checking are transferable to software verification, and vice versa. In individual works, a newly proposed algorithm is usually implemented for one specific type of computational models (e.g., either a circuit or a program), and the conclusions about the characteristics of the algorithm are drawn solely from verification tasks of this type. Such a research practice creates a gap in the generalizability of existing publications and hinders the mutual learning between the two communities. While prior efforts [12–14] in extending algorithms for hardware model checking to software verification help to fill the gap partially, systematically investigating the transferability of results between hardware and software verification is essential for consolidating the knowledge of model checking.

As a step toward the exchange and unification of the knowledge in hardware and software verification, we perform a *transferability study* on interpolation-based hardware model checking to software verification. According to the *ACM SIGSOFT Empirical Standards for Software Engineering Research* [15, 16], transferability is defined as "the extent to which a study's results could plausibly apply to other sites, people, or circumstances." Transferability and generalizability of research results are gaining more and more attention in software engineering [17, 18]. They are especially important for model checking because (1) software verification is often inspired by hardware model checking, and the transferability from hardware to software is key to successful adaptation; (2) formal verification is challenging, and we should embrace every possibility in the literature by transferring advancements made in one community to others. In the following, we outline the design of our transferability study and highlight our contributions.

[1] Given an unsatisfiable formula $A_1 \wedge A_2$, a Craig interpolant τ of the formula satisfies three conditions: (1) $A_1 \Rightarrow \tau$ is valid, (2) $\tau \wedge A_2$ is unsatisfiable, and (3) τ only refers to common variables of A_1 and A_2.

1.1 Design and Outcomes of the Transferability Study

To study how well the results obtained from circuits can be transferred to programs, we select two publications on interpolation-based hardware model checking that report considerable improvement over the algorithm *interpolation-based model checking* (IMC) [19] from 2003, which was the first approach employing Craig interpolation to verify safety properties of sequential circuits, as the subjects of our transferability study. The two publications are (1) *Interpolation-Sequence-Based Model Checking* [20] from 2009 and (2) *Intertwined Forward-Backward Reachability Analysis Using Interpolants* [21] from 2013. The former proposes an algorithm called *interpolation-sequence-based model checking* (ISMC), and the latter proposes an algorithm named *dual approximated reachability* (DAR). Our goal is to validate the claims about IMC, ISMC, and DAR from the two publications in the context of software verification. In the rest of this paper, we refer to the algorithms and the corresponding publications interchangeably with the abbreviations.

Recently, IMC has been adopted to analyze programs and shown to compete well against other polished algorithms for software verification [22]. The software-verification adoption of IMC is implemented in an award-winning software verifier CPAchecker [23]. We choose to implement ISMC and DAR also in CPAchecker because (1) it provides reliable and well-maintained components needed for the two algorithms and (2) confounding variables (parser, SMT solver, library, etc.) can be kept to a minimum when ISMC and DAR are compared to IMC. In the following, we briefly describe the compared algorithms.

IMC unrolls (i.e., duplicating the combinational logic of the circuit) a sequential circuit like BMC, overapproximates reachable states within certain steps by interpolating unsatisfiable BMC queries, and attempts to construct a *fixed point* of the circuit. Similar to IMC, ISMC also unrolls the input circuit but differs from IMC in the way it performs Craig interpolation. Instead of a single interpolant, ISMC computes a sequence of interpolants [24, 25] from an unsatisfiable BMC query and keeps refining overapproximated sets of states by conjoining them with new interpolants. DAR maintains two sequences of overapproximated state sets and computes interpolants from both forward and backward directions. It aims to avoid unrolling the input circuit (referred to as *global strengthening* in the DAR paper) by posing small and local queries about the circuit. For the compared interpolation-based algorithms, the *convergence length* is defined to be the number of steps for overapproximation required to reach a fixed point.

The main claims in the ISMC [20] and DAR [21] papers are listed below. To assess whether they can be transferred to software verification, we compare IMC, ISMC, and DAR on the largest publicly available benchmark suite [26] of safety-verification tasks in the programming language C. The results of this experiment are reported in Sect. 6. On more than 8 000 verification tasks, we successfully transfer the important claims about the algorithmic characteristics of ISMC and DAR to software verification. However, some claims in the original papers do not generalize to our settings. We use a green check-mark (resp. red question mark) to denote that a claim transfers (resp. does not transfer) to software verification in our transferability study.

Claims in the ISMC Publication [20]. The authors compared ISMC and IMC on 136 verification tasks derived from industrial CPU designs. There are 67 tasks with property violation, and the other 69 tasks satisfy their specifications. Both ISMC and IMC were implemented in the same framework. A time limit of 10 000 s was imposed on each verification task, and the evaluation was conducted on a machine with 32 GB of memory. The authors draw the following conclusions about the characteristics of ISMC and IMC from their evaluation.

H1.A: ISMC is faster than IMC on tasks with property violation. (✓)
H1.B: ISMC is faster than IMC when IMC finds a proof only at high unrolling bounds. (?)
H1.C: Overall, ISMC is faster than IMC (by 30 % in this experiment). (?)

Claims in the DAR Publication [21]. The authors compared DAR to IMC on 37 verification tasks derived from real-life industrial designs. There are at least four tasks with property violation.[2] All compared approaches were implemented in the same framework. A time limit of 1 800 s was imposed on each verification task, and the evaluation was conducted on a machine with 24 GB of memory. The authors draw the following conclusions about the characteristics of DAR and IMC from their evaluation.[3]

H2.A: For DAR, the ratio between iterations using global strengthening to the total number of iterations is less than 0.5 in most tasks. (✓)
H2.B: IMC finds a proof slower than DAR in many tasks even though it has a smaller convergence length. (?)
H2.C: DAR computes more interpolants than IMC. (✓)
H2.D: DAR's run-time is more sensitive to the sizes of interpolants than IMC. (?)
H2.E: Overall, DAR is faster than IMC (by 36 % in this experiment). (?)

1.2 Our Contributions

The transferability study presented in this paper makes the following contributions:

(1) It is the first systematic investigation of the transferability of the two interpolation-based algorithms (ISMC [20] and DAR [21]) for hardware model checking to software verification.
(2) Its evaluation confirms the important claims of the studied publications and discovers that the characteristics of the two algorithms are transferable to software verification.
(3) The study additionally compares IMC, ISMC, and DAR to predicate abstraction [24] and Impact [25], two interpolation-based approaches originated from the software-verification community. In the evaluation, the verification algorithms from the hardware domain solved about 20 % more tasks than predicate abstraction and Impact, showing that transferring hardware knowledge can improve software model checking.
(4) The open-source implementations of the two competitive algorithms for hardware model checking in CPAchecker enlarge the body of available software-verification techniques.

These contributions are **original** because of the new knowledge on the transferability, which was unknown before our study, and the novel analyses for software verification; They are also **important** as the transferability study consolidates the knowledge about the compared interpolation-based algorithms for hardware and software computational models. Formal verification is challenging, so it is imperative to leverage every possibility to advance the state of the art. The results in this paper can shed light on combining forces from the two research communities to invent more effective approaches for complex systems involving both hardware and software components.

2 RELATED WORK

This transferability study on interpolation-based hardware model checking to software verification is related to the following research areas.

2.1 Reliability and Transferability of Research Findings

Reliability and transferability of research results are fundamental to science and technology, but findings published in peer-reviewed venues are not always reliable or transferable [27]. This issue was first noticed in medicine [28] and has received broader attention in computer science and software engineering [29–33]. To mitigate the situation, empirical standards are proposed to assess the reliability and transferability of research results [15, 16]. Software-engineering conferences

[2]This can be seen from Table 1 of the paper [21]. The precise number is not explicitly stated.
[3]The authors also evaluated IC3/PDR [10]. We drop the claims about IC3/PDR because it is not based on Craig interpolation.

nowadays encourage or require authors to submit their research artifacts along with the manuscripts and organize a committee to evaluate the artifacts. There are also studies on the quality and community expectations of software artifacts [34–37] and methodologies to improve the reliability and transferability of results [17, 18]. Our transferability study contributes to the reliability of model-checking research on hardware and software. Especially, our open-source implementations of the compared algorithms offer a solid baseline for future studies. A recent work on Klee [38], a symbolic execution engine for software testing, emphasizes the importance of a well-maintained software infrastructure and reports that 27 % of the publications depending on Klee can be questioned [39].

2.2 Interpolation-Based Verification Techniques

Craig interpolation [11] is widely used in hardware and software model checking for abstracting objects appearing in the process of verification, such as sets of reachable states [19, 25], execution traces [40], transition relations [41], subroutines [42], and predicates over program variables [24]. Many state-of-the-art hardware model checkers and software verifiers also employ Craig interpolation [23, 43–46]. Therefore, it is important to understand how interpolation-based algorithms work and how well the results can be transferred from one type of verification tasks to another. Our transferability study answers this question for the two interpolation-based algorithms ISMC [20] and DAR [21] when they are adopted for software verification.

2.3 Applying Hardware Model Checking to Software Verification

Thanks to the similarities between finite-state and infinite-state model checking, algorithms for hardware model checking are often lifted to software verification. For example, BMC [8], *k*-induction [9], IMC [19], and IC3/PDR [10] are originally conceived for finite-state transition systems like sequential circuits. After becoming popular in the hardware community, they are also applied to program analysis [12–14, 22, 47, 48]. Such technology transfer is conducted under the assumption that *the observations made for hardware model checking are likely to hold for software verification*. The successful experiences reported in the above publications strengthen this assumption, but to what degree the assumption is correct remains unknown. This paper investigates this assumption by implementing two algorithms for interpolation-based hardware model checking [20, 21] in the software verifier CPAchecker [23] and applying them to verify a large set of software-verification tasks. We successfully transfer the important claims in the original publications about the characteristics of the two algorithms for hardware model checking to software verification.

3 BACKGROUND

In this section, we provide the preliminaries for the interpolation-based algorithms [19–21] compared in our transferability study and the software-verification framework CPAchecker [23] we used to implement them. The descriptions of the compared algorithms and their implementations in CPAchecker will be presented in Sect. 4 and Sect. 5, respectively. Logical connectives $\neg$, $\vee$, $\wedge$, and $\Rightarrow$ are used under their conventional semantics, and we use $\top$ and $\bot$ to represent logical *true* and *false*, respectively. A first-order predicate over state variables is interpreted interchangeably as the set of states satisfying the predicate.

3.1 Model Checking of Reachability Safety

First, we formulate the problem of model checking for a reachability-safety property. To simplify the presentation of the compared algorithms in Sect. 4, we base our formulation on state-transition systems. Section 3.3 will outline a generic approach to facilitate the adoption of hardware model checking to software verification [22].

3.1.1 State-Transition System. A state-transition system $\mathcal{M}$ can be described by two predicates $I(s)$ and $T(s, s')$, where s and s' are state variables. If state s is an initial state of $\mathcal{M}$, then $I(s)$ evaluates to $\top$, and if state s can transit to state s' via one step in $\mathcal{M}$, then $T(s, s')$ evaluates to $\top$.

3.1.2 Reachability-Safety Property. Model checking determines whether a state-transition system $\mathcal{M} = (I, T)$ fulfills a certain property. A reachability-safety property for $\mathcal{M}$ can be expressed as a predicate $P(s)$ over the state variable s, and is expected to hold at every *reachable* state of $\mathcal{M}$. A state s_i is reachable if there exists a sequence of states $\langle s_0, s_1, \ldots, s_i \rangle$ such that $I(s_0) \wedge T(s_0, s_1) \ldots \wedge T(s_{i-1}, s_i)$ evaluates to $\top$, i.e., there is a feasible path from an initial state s_0 to s_i via i transitions. $\mathcal{M}$ satisfies P if, for every reachable state s of $\mathcal{M}$, $P(s)$ holds. Otherwise, $\mathcal{M}$ violates P, and a sequence of states $\langle s'_0, s'_1, \ldots, s'_j \rangle$ exists such that $I(s'_0) \wedge T(s'_0, s'_1) \ldots \wedge T(s'_{j-1}, s'_j) \wedge \neg P(s'_j)$ evaluates to $\top$. The path from s'_0 to s'_j is called a *counterexample* to the reachability-safety property, and the safety-violating state s'_j is called a *bad* state.

3.2 Craig Interpolation

Craig interpolation is the foundation of the compared algorithms in this paper, facilitating the abstraction of infeasible counterexamples to invariants of the state-transition system. We briefly describe the properties of a Craig interpolant below.

3.2.1 Craig's Interpolation Theorem. Let A_1 and A_2 be two logical formulas. If $A_1 \wedge A_2$ is unsatisfiable, then Craig's interpolation theorem [11] ensures the existence of an *interpolant* τ, which is a logical formula satisfying the following properties:

- $A_1 \Rightarrow \tau$ is valid,
- $\tau \wedge A_2$ is unsatisfiable, and
- τ refers only to the common variables of A_1 and A_2.

3.2.2 Inductive Interpolation Sequence. Given a sequence of formulas $\langle A_1, \ldots, A_n \rangle$, with $\bigwedge_{j=1}^{n} A_j$ being unsatisfiable, a formula sequence $\langle \tau_0, \ldots, \tau_n \rangle$ is called an *inductive interpolation sequence* if

- $\tau_0 = \top$ and $\tau_n = \bot$,
- $\tau_{i-1} \wedge A_i \Rightarrow \tau_i$ is valid for $1 \leq i \leq n$, and
- τ_i refers only to the common variables of $\bigwedge_{j=1}^{i} A_j$ and $\bigwedge_{j=i+1}^{n} A_j$ for $1 \leq i < n$.

3.3 Adopting Hardware Model Checking to Software Verification

To adopt a model-checking algorithm designed for hardware, usually depicted as a state-transition system, extracting the three predicates I, T, and P from a software-verification task is necessary. We use the conversion proposed in the software-verification adoption of IMC [22], which applies *large-block encoding* (LBE) [49] to the program in order to take the its structure into account during the conversion. Since the conversion and the adoption of IMC [22] are available in CPAchecker [23], which also offers well-maintained components necessary for the compared algorithms, we chose CPAchecker as our implementation framework. In the following, we first recap the basic concepts of CPAchecker and then explain the conversion [22].

3.3.1 Program Representation. An imperative program can be represented graphically as a *control-flow automaton* (CFA) $C = (L, l_0, E)$ [50, 51], where L is the set of nodes corresponding to program locations, $l_0 \in L$ is the initial program location, and E is the set of directed edges between nodes annotated with program operations. A safety-reachability task of a CFA asks to decide the existence of a feasible program path from the initial location l_0 to an error location. Without loss of generality, we assume the CFA has exactly one error location $l_E \in L$.

3.3.2 Configurable Program Analysis. The software verifier CPAchecker is based on the concept of *configurable program analysis* (CPA) [52, 53]. A CPA defines an abstract domain for program analysis. For example, the *location CPA* $\mathbb{L}$ tracks the explicit program location; the *loop-bound CPA* $\mathbb{LB}$ counts the number of visits to a loop head on a program path; the *predicate CPA* $\mathbb{P}$ encodes program paths into logical formulas. A CPA has an initial abstract state representing the start of the program analysis. For example, the initial abstract state of the location CPA is the initial program location, and the initial abstract state of the predicate CPA uses $\top$ to encode the path formula because no program paths were traversed.

Multiple CPAs can be combined into a *composite CPA*, which can be utilized by the CPA++ algorithm [7] for reachability analysis. Given a set of already-reached abstract states (a *reached set*) and a list of abstract states to be processed (a *wait list*), the CPA++ algorithm explores the CFA of the input program, constructs an *abstract reachability graph* (ARG) in the abstract domains of the given CPAs, and returns the updated reached set and wait list. Our implementations of ISMC [20] and DAR [21] in CPAchecker use a composite CPA of the location, loop-bound, and predicate CPAs.

3.3.3 Software Programs as State-Transition Systems. The software-verification adoption of IMC [22] employs LBE [49] to obtain a state-transition system from a program. Without loss of generality, we assume the input program has at most one loop. A multi-loop program can be transformed into a single-loop program with an equivalent behavior by a standard preprocessing [54, 55] before its state-transition system is extracted [22]. Considering the loop-head location and the error location as the end of a large block, LBE constructs *path formulas* that capture the executions of the program between its initial location, loop-head location, and error location, from which an analogy to a state-transition system can be drawn. Specifically, the path formula between the initial program location and the loop-head location corresponds to the initial states I; the path formula between two consecutive visits to the loop-head location corresponds to the transition relation T; the path formula between the loop-head location and the error location corresponds to the negated safety property $\neg P$. The model-checking algorithms based on state-transition systems can be performed on a program using this analogy without a symbolic program counter.

4 DESCRIPTIONS OF THE COMPARED INTERPOLATION-BASED ALGORITHMS

In this transferability study, we adopt two interpolation-based hardware-model-checking algorithms ISMC [20] and DAR [21] to verify programs. Both algorithms are compared to IMC [19], the first model-checking algorithm based on Craig interpolation. Below we explain how these algorithms work and how they differ from each other.

4.1 Interpolation-Based Model Checking (IMC)

McMillan proposed IMC [19], the first interpolation-based algorithm for hardware model checking, in 2003. IMC extends BMC to unbounded verification by constructing a fixed point (i.e., inductive invariant) of the circuit's state from Craig interpolants. It has inspired numerous interpolation-based verification approaches, including ISMC [20] and DAR [21]. IMC consists of two nested computational stages: (1) The outer *BMC stage* unrolls the state-transition system and checks the reachability of bad states within some number of transitions; (2) The inner *interpolation stage* constructs fixed points via interpolating unsatisfiable BMC queries.

Given an unrolling bound k, a transition system is unwound into k copies in the BMC stage. A BMC query encoding all possible paths from an initial state (described by $I(s)$) to a bad state (described by $\neg P(s)$) via at most k transitions is then posed to a satisfiability solver:

$$\underbrace{I(s_0) \wedge T(s_0, s_1)}_{A_1(s_0,s_1)} \wedge \underbrace{T(s_1, s_2) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg P(s_1) \vee \ldots \vee \neg P(s_k))}_{A_2(s_1,s_2,\ldots,s_k)}, \tag{1}$$

where s_i denotes the state variable after the i-th transition. If Eq. (1) is satisfiable, a violation to the safety property is found. Otherwise, IMC proceeds to the interpolation stage.

During the interpolation stage, IMC tries to prove the safety property by constructing an overapproximation of reachable states from the unsatisfiable BMC query. According to Craig's interpolation theorem, an interpolant $\tau_1(s_1)$ for formulas A_1 and A_2 in Eq. (1) exists and satisfies: (1) $I(s_0) \wedge T(s_0, s_1) \Rightarrow \tau_1(s_1)$ is valid and (2) $\tau_1(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg P(s_i)$ is unsatisfiable. In other words, τ is an overapproximation of the set of states that (1) are reachable from an initial state via one transition and (2) do not violate the safety property within $k-1$ transitions.

Such overapproximation of states can be generated iteratively and accumulated into a fixed point. By replacing $I(s_0)$ by $\tau_1(s_0)$ in Eq. (1), another BMC query starting form the first interpolant τ_1 can be posed. If the query remains unsatisfiable, a second interpolant $\tau_2(s_1)$ can be derived, overapproximating the set of states via two transitions from initial states. The routine continues until, at some iteration n, $I \vee \bigvee_{i=1}^{n} \tau_i$ becomes inductive with respect to the transition relation T. That is, the union of the initial states and all computed interpolants grows into a fixed point. Since each interpolant satisfies the safety property thanks to the second criterion of Craig's interpolation theorem, the fixed point also satisfies the safety property. In this case, IMC proves the system safe at the unrolling bound of k and constructs a fixed point $I \vee \bigvee_{i=1}^{n} \tau_i$ at the convergence length of n.

In the other case, namely, some BMC query in the interpolation stage is satisfiable, we cannot be certain whether the safety property is violated. The violation could correspond to a spurious counterexample as the interpolant may contain unreachable states. To decide whether the counterexample is spurious, IMC will increment the unrolling bound k and return to the BMC stage, checking the feasibility of error paths starting from initial states.

4.2 Interpolation-Sequence-Based Model Checking (ISMC)

In 2009, Vizel and Grumberg introduced ISMC [20], which derives inductive interpolation sequences from unsatisfiable BMC queries to construct fixed points. Similar to IMC, ISMC also has a BMC stage and an interpolation stage, but the two stages in ISMC are executed sequentially. ISMC can be seen as a variation of McMillan's Impact algorithm [25] for software verification from 2006.

After checking there are no bad states in $I(s)$, ISMC starts the BMC stage with the unrolling bound set to one. Given an unrolling bound k, a BMC query that depicts all paths from an initial state to some bad state via *exactly* k transitions is posed:

$$\underbrace{I(s_0) \wedge T(s_0, s_1)}_{A_1(s_0,s_1)} \wedge \underbrace{T(s_1, s_2)}_{A_2(s_1,s_2)} \wedge \cdots \wedge \underbrace{T(s_{k-1}, s_k)}_{A_k(s_{k-1},s_k)} \wedge \underbrace{\neg P(s_k)}_{A_{k+1}(s_k)}. \tag{2}$$

If Eq. (2) is satisfiable, a bad state via k transitions from an initial state is discovered. Otherwise, ISMC proceeds to the interpolation stage.

During the interpolation stage, ISMC constructs a *reachability sequence* by interpolation sequences. A reachability sequence $\langle R_1, R_2, \ldots, R_k \rangle$ is a sequence of formulas, with each image R_i being an overapproximation of the set of states reachable via i transitions from an initial state. ISMC partitions the BMC formula in Eq. (2) into a sequence $\langle A_1, A_2, \ldots, A_{k+1} \rangle$, as indicated under Eq. (2). Since Eq. (2) is unsatisfiable, an inductive interpolation sequence $\langle \top, \tau_1^k, \tau_2^k, \ldots, \tau_k^k, \bot \rangle$ can be derived, where the superscript k indicates the current unrolling bound. According to the inductiveness condition, each τ_i^k is an overapproximation of reachable states after i transitions from the initial states. Note that the last interpolant τ_k^k contains no bad states because $\tau_k^k \wedge \neg P$ is unsatisfiable. To refine a reachability sequence by these interpolants, ISMC conjoins all interpolants derived in previous interpolation stages. That is, the image R_i for the i-step overapproximation at

the current unrolling bound k is computed as $\bigwedge_{j=i}^{k} \tau_i^j$. Note that each image R_i does not contain any bad state because $\tau_i^i \wedge \neg P$ is unsatisfiable.

After a reachability sequence $\langle R_1, R_2, \ldots, R_k \rangle$ is obtained, ISMC examines whether $\bigvee_{i=1}^{k} R_i$ has reached a fixed point. Since the images in the reachability sequence do not contain any bad state, ISMC can then conclude that the transition system satisfies the safety property in this case. If, nevertheless, the fixed-point check fails, ISMC will increment the unrolling bound by one and proceed to another BMC stage.

4.3 Dual Approximated Reachability (DAR)

In 2013, Vizel, Grumberg, and Shoham proposed DAR [21], which intertwines forward and backward derivation of Craig interpolants for unbounded formal verification of state-transition systems. The algorithm maintains two reachability sequences, one in the forward direction and the other backward, refining and extending the images by interpolation. A *forward reachability sequence* $\langle F_0, F_1, \ldots, F_n \rangle$ (resp. *backward reachability sequence* $\langle B_0, B_1, \ldots, B_n \rangle$) is a sequence of formulas such that (1) $F_0 = I$ (resp. $B_0 = \neg P$), (2) F_i contains no bad states (resp. B_i contains no initial states), and (3) F_i overapproximates the set of states reachable from an initial state via i transitions (resp. B_i overapproximates the set of states that can reach a bad state via i transitions). The existence of a forward or a backward reachability sequence demonstrates that there is no counterexample of length n.

The computation of DAR is partitioned into two stages: (1) a *local strengthening stage* that refines and extends forward and backward reachability sequences via interpolating *local* queries about two consecutive steps; (2) a *global strengthening stage* that unrolls the system when the local strengthening stage is not strong enough to refute potentially spurious counterexamples. Both stages refine the reachability sequences by iteratively computing interpolants from a pair of forward and backward overapproximations, which is called *iterative pairwise strengthening*.

After checking the initial states do not overlap with the bad states, DAR initializes the forward and backward reachability sequences as $\langle I \rangle$ and $\langle \neg P \rangle$, respectively, and enters the local strengthening stage. DAR attempts to find the smallest[4] index i such that $F_i(s) \wedge T(s, s') \wedge B_{n-i}(s')$ is unsatisfiable. If such an index i exists, it indicates that every state in F_i cannot reach a state in B_{n-i} via one transition, i.e., there is no counterexample of length $n+1$. In this case, DAR invokes iterative pairwise strengthening to refine and extend the reachability sequences based on this local unsatisfiability. For each $i \leq j < n$, an interpolant τ_{j+1} between $F_j(s) \wedge T(s, s')$ and $B_{n-j}(s')$, called a *forward interpolant*, is computed and used to refine F_{j+1}. Likewise, for each $n - i \leq j < n$, an interpolant τ'_{j+1} between $B_j(s') \wedge T(s, s')$ and $F_{n-j}(s)$, called a *backward interpolant*, is computed and used to refine B_{j+1}. At last, for $j = n$, the forward and backward interpolants, τ_{n+1} and τ'_{n+1}, are appended to the forward and backward reachability sequences, respectively.

If such an index i cannot be found in the local strengthening stage, DAR enters the global strengthening stage to precisely analyze the existence of counterexamples of length $n + 1$. It tries to incrementally unroll the state-transition system and find the smallest unrolling bound m such that

$$\underbrace{I(s_0) \wedge T(s_0, s_1)}_{A_1(s_0, s_1)} \wedge \cdots \wedge \underbrace{T(s_{m-1}, s_m)}_{A_m(s_{m-1}, s_m)} \wedge \underbrace{B_{n-m+1}(s_m)}_{A_{m+1}(s_m)} \tag{3}$$

is unsatisfiable. If such a bound m exists, DAR concludes that a counterexample of length $n + 1$ does not exist because it is not possible to reach B_{n-m+1} in m steps. An interpolation sequence $\langle \top, \tau_1, \ldots, \tau_m, \bot \rangle$ is derived from the sequence $\langle A_1, A_2, \ldots, A_{m+1} \rangle$ of formulas, and τ_i is used to refine the forward reachability image F_i. Afterwards, iterative pairwise strengthening is invoked to refine

[4]The algorithm actually works with any such index. We follow the original publication [21] and use the smallest.

and extend the forward and backward reachability sequences. If such a bound m does not exist, i.e., Eq. (3) is satisfiable for $m = n + 1$, DAR discovers a counterexample of length $n + 1$ and terminates.

If both reachability sequences are refined and extended to $\langle F_0, F_1, \ldots F_{n+1}\rangle$ and $\langle B_0, B_1, \ldots B_{n+1}\rangle$ with the newly derived forward and backward interpolants, DAR examines whether the accumulated overapproximation of reachable states has grown into a fixed point in either direction. That is, DAR checks whether $\bigvee_{i=0}^{n+1} F_i$ or $\bigvee_{i=0}^{n+1} B_i$ is inductive. Since every forward reachability image F_i (except F_0) is initialized by some interpolant τ_i, with $\tau_i \wedge \neg P$ being unsatisfiable, F_i does not contain any bad state. Similarly, every backward reachability image B_i does not contain any initial state. Therefore, DAR successfully proves the system safe at a convergence length of $n + 1$, and $\bigvee_{i=0}^{n+1} F_i$ or $\neg(\bigvee_{i=0}^{n+1} B_i)$ is a safe invariant of the system. If a fixed point has not yet been reached, DAR will enter the local strengthening stage again.

4.4 Differences Between the Three Algorithms

Although IMC [19], ISMC [20], and DAR [21] all depend on Craig interpolation for abstracting reachable states, they differ in how the satisfiability queries are posed and how the overapproximations are constructed. IMC, ISMC, and DAR pose different BMC queries as shown in Eq. (1), Eq. (2), and Eq. (3), respectively. Unlike IMC and ISMC, which unroll the system when the currently computed overapproximated images fail to reach a fixed point, DAR tries to find a shorter unsatisfiable BMC query in order to avoid additional unrolling of the system. For constructing overapproximations, IMC forgets the previously computed abstractions and derives a new one from scratch after increasing the unrolling bound. By contrast, ISMC and DAR accumulate all interpolants derived throughout their executions. Furthermore, DAR poses local queries that involve only one copy of the transition relation, whereas IMC and ISMC rely solely on global queries of complete unrolling. The algorithmic differences of these algorithms result in their distinct strengths, which we will study in a large-scale empirical evaluation and report its results in Sect. 6.

5 ADOPTING ISMC AND DAR TO SOFTWARE VERIFICATION IN CPACHECKER

This section discusses the implementation details of ISMC [20] and DAR [21] in CPAchecker [23]. Both algorithms assume as inputs single-loop programs and apply a standard single-loop transformation [54, 55] to multi-loop programs as preprocessing. We utilize a composite CPA $\mathbb{D}$ of location, predicate, and loop-bound CPAs as well as other supportive CPAs[5] and the CPA^{++} algorithm [7] to unroll the program. The predicate CPA is configured to use LBE [49] for extracting the predicates I, T, and P from the input program.

5.1 Interpolation-Sequence-Based Model Checking (ISMC)

The main procedure of ISMC is summarized in Alg. 1. The algorithm CPA^{++} [7] unrolls the CFA of an input program into an ARG up to the given unrolling bound k. The subroutine extract_formulas() is used to collect the formulas for the initial states, transition relation, and (negated) safety property from the ARG [22], as discussed in Sect. 3.3.

After obtaining the formulas for the initial and bad states, ISMC inspects whether they intersect with each other at line 6. If not, ISMC initializes a reachability sequence images and tries to construct a fixed-point in the loop starting from line 9. Given an unrolling bound k, a BMC query at line 13 is posed to examine whether a counterexample of length k exists. If the query is unsatisfiable, ISMC enters the interpolation stage (lines 15 to 18) and computes an inductive interpolation sequence to refine and extend the reachability sequence. After the refinement, ISMC checks whether the

[5]These helper CPAs deal with features specific to software, such as the call stack and function pointers. We leave out the discussion on these CPAs to ease the presentation of the algorithm implementations.

Algorithm 1 ISMC: main procedure

Input: a composite CPA $\mathbb{D}$ of $\mathbb{L}$, $\mathbb{P}$, and $\mathbb{LB}$
Output: **true** if the program is proven to be safe; **false** if a feasible error path is found

```
1:  k ← 0;
2:  e_0 ← (L.get_initial(), P.get_initial(), LB.get_initial());   // Create an initial abstract state
3:  reached ← waitlist ← {e_0};
4:  reached, waitlist ← CPA++(D, reached, waitlist, k);
5:  ⟨I(s_0), ¬P(s_0)⟩ ← extract_formulas(reached);
6:  if sat(I(s_0) ∧ ¬P(s_0)) then
7:    return false;                                   // Initial state set contains bad states
8:  images ← ⟨⊥⟩;                  // Initialize a reachability sequence with ⊥ at the 0th step[6]
9:  while (true) do
10:   k ← k + 1;
11:   reached, waitlist ← CPA++(D, reached, waitlist, k);                    // Unroll program
12:   ⟨I(s_0), T(s_0, s_1), ..., T(s_{k−1}, s_k), ¬P(s_k)⟩ ← extract_formulas(reached);
13:   if sat(I(s_0) ∧ T(s_0, s_1) ∧ ... ∧ T(s_{k−1}, s_k) ∧ ¬P(s_k)) then
14:     return false;                                 // BMC finds a feasible error path
15:   ⟨⊤, τ_1, ..., τ_k, ⊥⟩ ← interpolate(I(s_0) ∧ T(s_0, s_1), T(s_1, s_2), ..., T(s_{k−1}, s_k), ¬P(s_k));
                                          // Compute inductive interpolation sequence
16:   for (i ← 1; i < k; i ← i + 1) do
17:     images⟨i⟩ ← images⟨i⟩ ∧ τ_i;                  // Refine reachability sequence
18:   images.append(τ_k);                             // Extend reachability sequence
19:   if check_fixed_point(images) then
20:     return true;                                  // Fixed point reached
```

Algorithm 2 check_fixed_point (used in both Alg. 1 and Alg. 3)

Input: a reachability sequence images
Output: **true** if the accumulated reachability images form a fixed point; **false** otherwise

```
1:  λ ← images⟨0⟩;                                    // Accumulation of reachability images
2:  for (i ← 1; i < images.len(); i ← i + 1) do
3:    if ¬sat(images⟨i⟩ ∧ ¬λ) then
4:      return true;                  // images⟨i⟩ is contained in the accumulation λ
5:    λ ← λ ∨ images⟨i⟩;                              // Accumulate images
6:  return false
```

current reachability sequence has converged to a fixed-point by the subroutine check_fixed_point() at line 19. If so, ISMC concludes the program is safe. Otherwise, it increments the unrolling bound by one and starts another iteration.

Algorithm 2 outlines the fixed-point checking procedure used by ISMC (and DAR in Alg. 3). It iterates the given reachability sequence and checks whether the image at the frontier is contained in the union of all previous images at line 3. A fixed point is found if the check succeeds. Otherwise, the frontier image is added to the union at line 5, and the check continues. In case no check succeeded after iterating the whole reachability sequence, the subroutine reports to the main procedure that a fixed point has not been reached.

Algorithm 3 DAR: main procedure

Input: a composite CPA $\mathbb{D}$ of $\mathbb{L}$, $\mathbb{P}$, and $\mathbb{LB}$
Output: **true** if the program is proven to be safe; **false** if a feasible error path is found

```
1:  k ← 0;
2:  e_0 ← (L.get_initial(), P.get_initial(), LB.get_initial());   // Create an initial abstract state
3:  reached ← waitlist ← {e_0};
4:  reached, waitlist ← CPA++(D, reached, waitlist, k);
5:  ⟨I(s_0), ¬P(s_0)⟩ ← extract_formulas(reached);
6:  if sat(I(s_0) ∧ ¬P(s_0)) then
7:    return false;                                   // Initial state set contains bad states
8:  for_seq ← ⟨I⟩; back_seq ← ⟨¬P⟩;                  // Initialize forward and backward sequences
9:  k ← 1;
10: reached, waitlist ← CPA++(D, reached, waitlist, k);   // Unroll once more to get T(s, s')
11: ⟨I(s_0), T(s_0, s_1), ¬P(s_1)⟩ ← extract_formulas(reached);
12: while ¬check_fixed_point(for_seq) ∧ ¬check_fixed_point(back_seq) do
13:   i ← find_smallest_unsat_index(for_seq, back_seq, T(s, s'));   // Find local unsatisfiability
14:   if i = −1 then          // −1 indicates index unfound: enter global strengthening stage
15:     n ← for_seq.len();
16:     for (i ← 1; i ≤ n; i ← i + 1) do
17:       if i > k then                                 // If additional program unrolling is required
18:         k ← i;
19:         reached, waitlist ← CPA++(D, reached, waitlist, k);   // Unroll program
20:       ⟨I(s_0), T(s_0, s_1), ..., T(s_{i−1}, s_i), ¬P(s_i)⟩ ← extract_formulas(reached);
21:       if ¬sat(I(s_0) ∧ T(s_0, s_1) ∧ ... ∧ T(s_{i−1}, s_i) ∧ back_seq⟨n − i⟩(s_i)) then
22:         break;                                      // No feasible error path of length n
23:       if i = n then          // The check in line 21 is precise: a feasible error path found
24:         return false;
25:     ⟨⊤, τ_1, ..., τ_i, ⊥⟩ ← interpolate(I(s_0) ∧ T(s_0, s_1), ..., T(s_{i−1}, s_i), back_seq⟨n − i⟩(s_i));
                                                  // Compute inductive interpolation sequence
26:     for (j ← 1; j ≤ min(i, n − 1); j ← j + 1) do
27:       for_seq⟨j⟩ ← for_seq⟨j⟩ ∧ τ_j;              // Refine forward reachability sequence
28:     i ← i − 1;          // Decrement i to match the precondition of iterative_strengthen()
29:   iterative_strengthen(for_seq, back_seq, T(s, s'), i);   // Iterative pariwise strengthening
30: return true;                                      // Fixed point reached
```

5.2 Dual Approximated Reachability (DAR)

We sketch the procedure of DAR in Alg. 3. Similar to ISMC, DAR first inspects if the initial states overlap with the bad states at line 6. If not, it initializes the forward reachability sequence for_seq with I and the backward reachability sequence back_seq with $\neg P$ at line 8. DAR unrolls the program one more time at line 10 to obtain the formula of the transition relation, required in the local strengthening stage at line 13. Before entering the local strengthening stage, DAR performs fixed-point checks (outlined in Alg. 2) on the forward and backward reachability sequences, respectively, at line 12. If either of the checks succeeds, DAR determines the program safe and terminates at line 30. Otherwise, it attempts to strengthen the reachability sequences locally.

In the local strengthening stage, given the forward and backward reachability sequences for_seq and back_seq of length n (which indicate no counterexample up to length $n-1$ exists), DAR searches

[6]The initial states I can also be used at the 0th step; We use $\bot$ for consistency with the ISMC paper [20].

Algorithm 4 iterative_strengthen

Input: sequences for_seq and back_seq of length n for forward and backward reachability, respectively, a transition relation $T(s, s')$, and an index i such that the formula for_seq$\langle i\rangle(s) \wedge T(s, s') \wedge$ back_seq$\langle n-1-i\rangle(s')$ is unsatisfiable

1: $n \leftarrow$ for_seq.len();
2: for_seq.append($\top$);
3: **for** ($j \leftarrow i$; $j < n$; $j \leftarrow j+1$) **do**
4: $\langle \top, \tau, \bot \rangle \leftarrow$ interpolate(for_seq$\langle j\rangle(s) \wedge T(s, s')$, back_seq$\langle n-1-j\rangle(s')$);
5: for_seq$\langle j+1\rangle \leftarrow$ for_seq$\langle j+1\rangle \wedge \tau$; // Strengthened by forward interpolant
6: back_seq.append($\top$);
7: **for** ($j \leftarrow n-1-i$; $j < n$; $j \leftarrow j+1$) **do**
8: $\langle \top, \tau, \bot \rangle \leftarrow$ interpolate(back_seq$\langle j\rangle(s') \wedge T(s, s')$, for_seq$\langle n-1-j\rangle(s)$);
9: back_seq$\langle j+1\rangle \leftarrow$ back_seq$\langle j+1\rangle \wedge \tau$; // Strengthened by backward interpolant

for the smallest index i such that the formula for_seq$\langle i\rangle(s) \wedge T(s, s') \wedge$ back_seq$\langle n-1-i\rangle(s')$ is unsatisfiable. Such an index i indicates that it is impossible to transit via one step from any state reachable via i steps from an initial state to a state that can reach a bad state via $n-i-1$ steps. In other words, this local unsatisfiable query shows that no counterexample of length n exists. Such local infeasibility of counterexamples is then leveraged by the iterative pairwise strengthening procedure at line 29 to refine and extend the reachability sequences.

The procedure of iterative pairwise strengthening is summarized in Alg. 4. Given the forward and backward reachability sequences for_seq and back_seq of length n, the transition relation, and an index i such that for_seq$\langle i\rangle(s) \wedge T(s, s') \wedge$ back_seq$\langle n-1-i\rangle(s')$ is unsatisfiable, the procedure computes forward and backward interpolants from pairs of forward and backward images in conjunction with the transition relation and uses the interpolants to refine and extend the reachability sequences in both directions. In lines 2 to 5 (resp. lines 6 to 9), the forward (resp. backward) sequence is extended and strengthened by forward (resp. backward) interpolants.

DAR repeats the process of identifying local infeasibility and iterative strengthening until either of the reachability sequences converges to a fixed point, or the sequences no longer demonstrate such local unsatisfiability. In the latter case, DAR enters the global strengthening stage from line 14, where counterexamples of length n are searched with multiple copies of the transition relation. Instead of directly posing the precise BMC query of length n, which is potentially expensive, DAR leverages the backward reachability sequence and gradually increases the number of transitions in the query at line 21 to show that the states that can reach a bad state in $n-i$ steps (overapproximated by back_seq$\langle n-i\rangle$) are unreachable via i steps from the initial states. This query overapproximates the precise BMC query, namely, no counterexample of length n exists if it is unsatisfiable. Additional program unrolling is performed at line 19 when needed.

If the query at line 21 is unsatisfiable for some number $i \leq n$, DAR refines the forward reachability sequence by the derived inductive interpolation sequence and invokes the iterative pairwise strengthening procedure at line 29 with i decremented by 1 at line 28 because it follows from the unsatisfiable query that for_seq$\langle i-1\rangle(s) \wedge T(s, s') \wedge$ back_seq$\langle n-1-(i-1)\rangle(s')$ is also unsatisfiable. The query becomes the precise BMC query when $i = n$, and DAR finds a counterexample of length n at line 24 if the query is satisfiable.

5.3 Design Choices in the Implementations

Two relevant design choices we made in the implementations are (1) the extraction of a transition relation from a program and (2) the encoding and solving of path formulas.

To extract a transition relation, we also tried a straightforward approach using single-block encoding with a symbolic program counter. However, the conversion based on LBE [22] outperformed the straightforward method in our evaluation because it takes the program structure into account. To encode the path formulas, we follow the default settings of CPAchecker [23] and use the SMT theory of equality with uninterpreted functions, arrays, bit-vectors, and floats. The SMT solver MathSAT5 [56] is used because it supports interpolation on the theory. This configuration of CPAchecker has been extensively evaluated and shown to perform well on our benchmark set [26].

We tried our best to faithfully implement the compared algorithms. Unfortunately, to our knowledge, there are no publicly available reference implementations of ISMC [20] and DAR [21], against which we can check our adoptions to software verification.

6 EVALUATION

To assess whether the claims listed in Sect. 1.1 about IMC, ISMC, and DAR are transferable to software verification or not, we evaluated the implementations of the three algorithms in the software verifier CPAchecker on a large set of safety-verification tasks in the programming language C. We used a much larger input data set than those in the original papers to make the experimental results more robust. In addition to examining the claims in the original papers [20, 21], we compared IMC, ISMC, and DAR to predicate abstraction (PredAbs) [24] and Impact [25], two state-of-the-art interpolation-based algorithms for software verification, to explore how the approaches originated from hardware model checking benefit conventional program analysis.

6.1 Benchmark Set

We used tasks from the 2023 Competition on Software Verification (SV-COMP '23) [26] in our evaluation. We considered the tasks whose safety property is the reachability of an error location and excluded the tasks from the categories *ReachSafety-Recursive* and *ConcurrencySafety-Main* because the implementations currently do not support them. In total, the benchmark set consists of 8 813 tasks, among which 2 793 contain a feasible execution path to the error location (referred to as *unsafe*), and the rest 6 020 are assumed to satisfy their specifications (referred to as *safe*). The benchmark set includes subcategories *Arrays*, *BitVectors*, *ControlFlow*, *ECA*, *Floats*, *Heap*, *Loops*, *ProductLines*, *Sequentialized*, *XCSP*, *Combinations*, and *Hardware* from the category *ReachSafety* and subcategories *AWS-C-Common-ReachSafety*, *BusyBox-ReachSafety*, *DeviceDriversLinux64-ReachSafety*, *DeviceDriversLinux64Large-ReachSafety*, and *uthash-ReachSafety* from the category *SoftwareSystems*.

6.2 Experimental Settings

All the five compared approaches (IMC, ISMC, DAR, PredAbs, and Impact) were implemented with a unifying framework in CPAchecker [7] to minimize confounding variables, such as the frontend parser and backend SMT solver, in the evaluation. CPAchecker at revision 45787 of branch `itp-mc-with-slt` was used in the evaluation, and the SMT solver MathSAT5 [56] was employed to handle all the SMT queries.

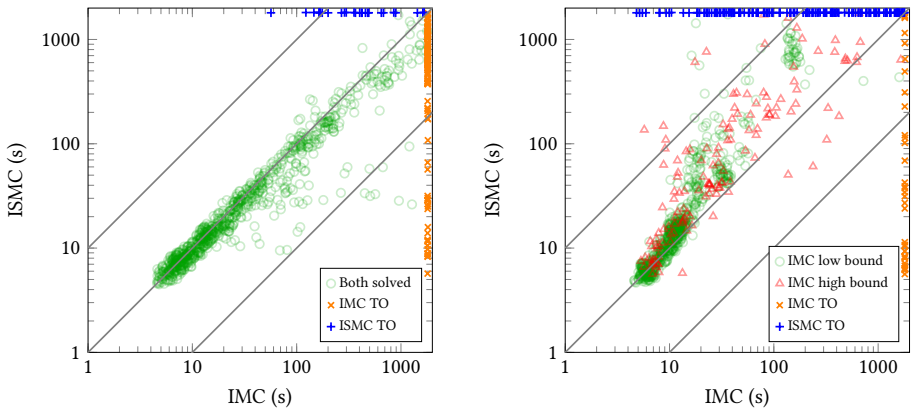
We ran the experiments on machines with a 3.40 GHz CPU (Intel Xeon E3-1230 v5) having 8 processing units and 32 GB of memory. The operating system was Ubuntu 22.04 (64 bit) running Linux 5.15 and OpenJDK 17.0. The resource limits imposed on each verification task were two CPU cores, 15 GB of memory, and 1 800 s of CPU time. The benchmarking framework BenchExec [57] was used to control the computational resources and process the measurement data.

6.3 Assessment of the Claims about ISMC

Table 1 shows a summary of the three interpolation-based algorithms in our evaluation. Like the polished IMC implementation, the new implementations of ISMC and DAR did not report any

Table 1. Summary of the experimental results for 8 813 safety-verification tasks

Algorithm	(#tasks)	IMC	ISMC	DAR
Correct results	8 813	**2 791**	2 723	**2 791**
proofs	6 020	**1 886**	1 713	1 815
alarms	2 793	905	**1 010**	976
Incorrect results		2	2	2
proofs		0	0	0
alarms		2	2	2
Timeouts		2 367	2 257	2 281
Out of memory		437	662	524
Other inconclusive		3 216	3 169	3 215



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

Fig. 1. Comparing the CPU time of ISMC and IMC on (a) unsafe and (b) safe tasks (TO: timeout)

incorrect proofs. All three analyses reported incorrect alarms on two verification tasks. These wrong alarms are related to the program encoding of CPAchecker and not caused by our implementations as other mature approaches in CPAchecker, like PredAbs [24], cannot solve them correctly, either.

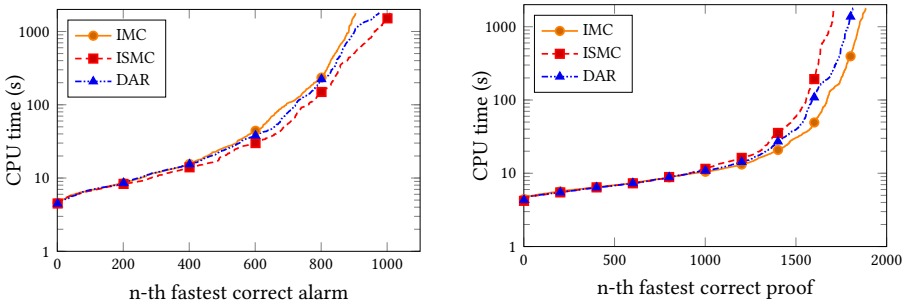
Based on the data collected in our evaluation, we investigate whether the claims in the ISMC [20] and DAR [21] papers can be transferred to software verification below.

H1.A: ISMC is faster than IMC on tasks with property violation (✓). Figure 1a compares the CPU time ISMC (on the y-axis) and IMC (on the x-axis) took to report alarms in the verification tasks. Observe that more data points are below the diagonal, indicating that ISMC is faster than IMC at bug hunting. Moreover, there are 127 tasks (orange marks) for which IMC reached the time limit but ISMC can find a bug. There are only 27 tasks (blue crosses) the other way around. Therefore, we conclude that this claim holds in our evaluation.

H1.B: ISMC is faster than IMC when IMC finds a proof only at high unrolling bounds (?). Figure 1b compares the CPU time ISMC (on the y-axis) and IMC (on the x-axis) took to find proofs in the verification tasks. The proofs found by completely unrolling the loops in the programs instead of deriving fixed points were excluded. Because the original claim did not specify what a *high* unrolling bound for IMC is, we interpret it as the first quantile (i.e., higher than 75 %) of the unrolling bounds required by IMC to find fixed points on the whole benchmark set. These tasks are labelled with red triangles in Fig. 1b. Since most of the red triangles are above the diagonal, IMC is still faster than ISMC even it finds a proof at high unrolling bounds. Therefore, we conclude

Table 2. Comparing the CPU time of IMC vs. ISMC/DAR on tasks they both correctly solved (time unit: s)

	CPU time				CPU time			
	#tasks	IMC	ISMC	ratio	#tasks	IMC	DAR	ratio
Total	2 549	**143 000**	167 000	1.17	2 631	**163 000**	180 000	1.10
Proofs	1 676	**49 400**	99 700	2.02	1 762	**72 200**	95 800	1.34
Alarms	873	93 300	**67 300**	0.72	869	90 400	**84 400**	0.93



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

Fig. 2. Quantile plots for all correct (a) alarms and (b) proofs of IMC, ISMC, and DAR

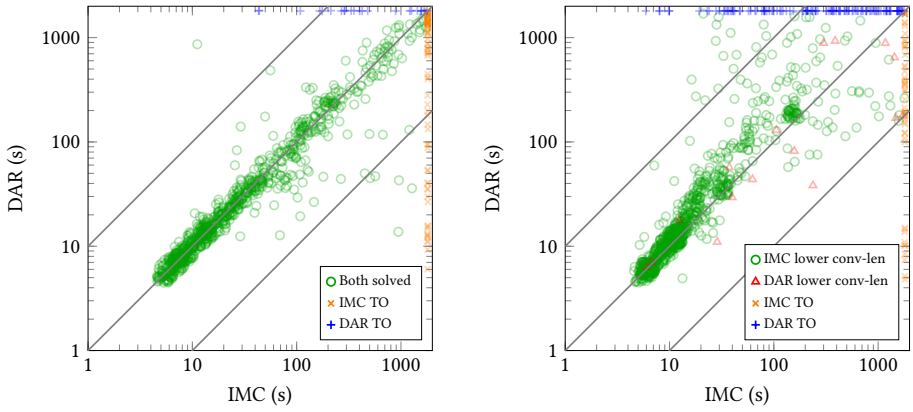
that this claim does not hold according to our interpretation of high unrolling bounds. It is also worth noting that IMC never had a higher unrolling bound than ISMC in our evaluation.

H1.C: Overall, ISMC is faster than IMC (?). The authors of the ISMC paper [20] report that ISMC was overall faster than IMC by 30 % in their experiment. We decompose the assessment into finding alarms and proofs. In the left half of Table 2, we report the numbers of alarms and proofs found by both IMC and ISMC as well as the summation of CPU time they took to solve these tasks. ISMC is faster than IMC by 27 % at bug hunting but twice slower at delivering proofs. The quantile plots in Fig. 2 also show that ISMC is faster at bug hunting but slower at proof finding. Overall, ISMC spent 17 % more CPU time than IMC to solve these tasks, so we conclude that this claim does not hold in our evaluation.

6.4 Assessment of the Claims about DAR

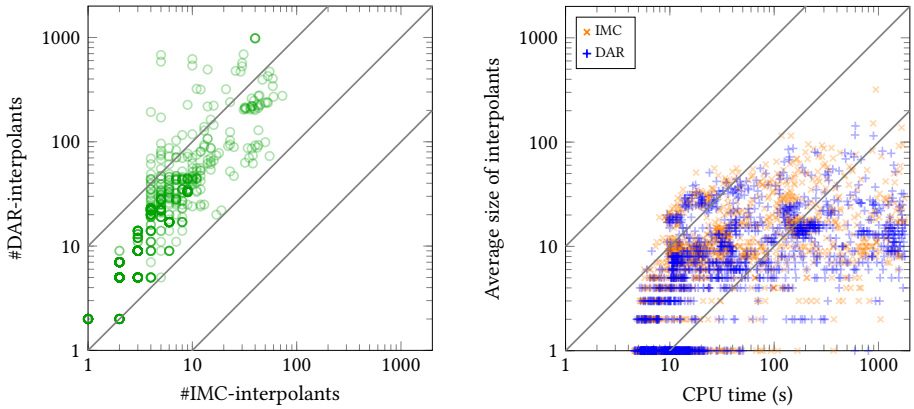
H2.A: For DAR, the ratio between iterations using global strengthening to the total number of iterations is less than 0.5 in most tasks (✓). DAR is designed to avoid large and expensive BMC queries as much as possible. It achieves this goal by first trying to show that the BMC query is unsatisfiable with shorter and possibly cheaper satisfiability checks in the local strengthening stage. Out of the 1 815 safe tasks DAR correctly solved, 979 were proven by constructing fixed points, while the rest were proven by completely unrolling the loops in the programs. In the tasks where fixed points were derived, the average ratio of the number of iterations in which DAR entered the global strengthening stage to the number of total iterations is 0.097. Specifically, there are 785 tasks in which DAR never performed global strengthening in any iteration. Therefore, we conclude that the claim holds in our evaluation and that DAR's key insight of using local checks to avoid large and expensive BMC queries is transferable to software verification.

H2.B: IMC finds a proof slower than DAR in many tasks even though it has a smaller convergence length (?). Figure 3b shows a scatter plot of the CPU time elapsed for DAR (on the y-axis) and IMC (on the x-axis) to find proofs by reaching fixed points. The plot confirms that IMC



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

Fig. 3. Comparing the CPU time of DAR and IMC on (a) unsafe and (b) safe tasks (TO: timeout)



(a) Numbers of interpolants of DAR vs. IMC **(b)** Size of interpolants vs. CPU time

Fig. 4. (a) The numbers of interpolants derived by DAR and IMC; (b) The size of interpolants (measured by average numbers of atoms) versus the CPU time elapsed to solve a task

usually has a smaller convergence length than DAR (the data points shaped as green circles). In fact, DAR found a proof with a smaller convergence length than IMC only in 16 tasks. However, the scatter plot contradicts the claim that DAR is faster than IMC. In our evaluation, IMC usually found a proof faster than DAR, as shown by the large number of data points above the diagonal. Therefore, we conclude that this claim does not hold in our evaluation.

H2.C: DAR computes more interpolants than IMC (✓). Figure 4a is a scatter plot of the numbers of interpolants derived during the computation of DAR and IMC, showing that IMC never computed more interpolants than DAR on all tasks. Therefore, we conclude that the claim holds in our evaluation.

H2.D: DAR's run-time is more sensitive to the sizes of interpolants than IMC (?). To assess this claim, we define the *size* of an interpolant as the number of atoms it contains. An *atom* is a predicate applied to terms without any boolean connectives. To evaluate if the CPU time of DAR

Table 3. Summary of the experimental results for 4 790 tasks consisting of programs with at most one loop

	Algorithm	(#tasks)	IMC	ISMC	DAR	Impact	PredAbs
	Correct results	4 790	2 439	2 372	**2 446**	2 107	2 026
	proofs	3 188	**1 755**	1 592	1 691	1 433	1 438
	alarms	1 602	684	**780**	755	674	588
	Incorrect results		2	2	2	1	1
	proofs		0	0	0	0	0
	alarms		2	2	2	1	1
	Timeouts		1 878	1 778	1 828	1 823	2 231
	Out of memory		317	535	401	368	82
	Other inconclusive		154	103	113	491	450
Correct results by category	*ReachSafety-ECA*	1 263	562	546	**582**	571	492
	proofs	783	**431**	319	384	398	362
	alarms	480	131	**227**	198	173	130
	ReachSafety-Sequentialized	461	**281**	251	245	255	204
	proofs	131	**45**	23	19	27	13
	alarms	330	**236**	228	226	228	191
	ReachSafety-Loops	445	157	**174**	172	110	117
	proofs	314	93	**105**	101	79	85
	alarms	131	64	69	**71**	31	32
	Other categories	2 621	1 439	1 401	**1 447**	1 171	1 213

is more sensitive to the sizes of its interpolants, we plot the CPU time and the average sizes of interpolants needed by DAR and IMC to solve a task in Fig. 4b.

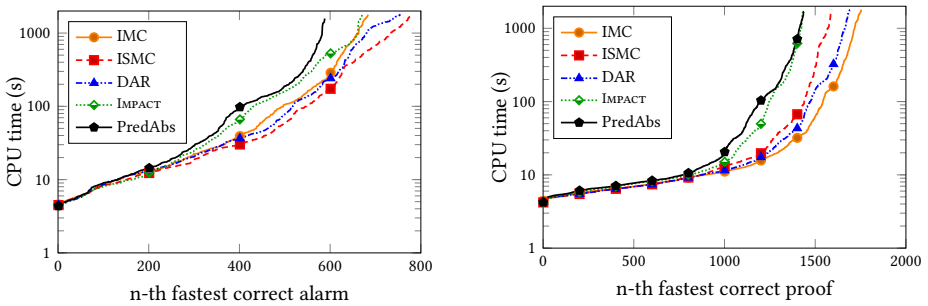
From the plot, we observe for both DAR and IMC that the CPU time elapsed to solve a task and average sizes of interpolants are positively co-related. However, the plot does not indicate that one algorithm is more sensitive to the size of interpolants than the other. We also evaluated this claim with different metrics for measuring the size of an interpolant, including the numbers of Boolean operations and variables in it, but did not find clear evidence to support the claim. Therefore, we conclude that the claim does not hold in our evaluation.

H2.E: Overall, DAR is faster than IMC (?). The authors of the DAR paper [20] report that DAR was overall faster than IMC by 36 % in their experiment. In the right half of Table 2, we report the numbers of alarms and proofs found by both IMC and DAR as well as the summation of CPU time they took to solve these tasks. DAR is faster than IMC by 7 % at bug hunting but 34 % slower at delivering proofs. The quantile plots in Fig. 2 and the scatter plots in Fig. 3 also show that DAR performs similarly to IMC for finding bugs and that IMC is more efficient in finding proofs. Overall, DAR spent 10 % more CPU time than IMC to solve these tasks, so we conclude that this claim does not hold in our evaluation.

6.5 Comparison with Other Interpolation-Based Software-Verification Approaches

Besides validating the claims in previous publications [20, 21], we compare IMC, ISMC, and DAR to PredAbs [24] and Impact [25]. Note that PredAbs and Impact are inherently able to handle multi-loop programs, whereas IMC, ISMC, and DAR require single-loop transformation as preprocessing. Therefore, to eliminate the difference in program encoding caused by the transformation, we focus the comparison on benchmark tasks consisting of programs with at most one loop.

Table 3 summarizes the results of the five compared algorithms on 4 790 safety-verification tasks, among which 3 188 are safe and 1 602 are unsafe. In the evaluation, DAR was able to solve the most tasks in total, IMC found the most proofs, and ISMC was the best bug-hunting algorithm. Notably,



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

Fig. 5. Quantile plots for all correct (a) alarms and (b) proofs on the tasks with at most one loop

all three hardware-verification algorithms produced more correct results than both software-verification algorithms, and the overall increase in the correct results is about 20 %. IMC, ISMC and DAR were able to solve 512, 572, and 528 tasks that were unsolvable by either IMPACT or PredAbs, respectively. On nearly 300 tasks, PredAbs and IMPACT posed SMT queries that MATHSAT5 encountered errors, which results in their high numbers of "Other inconclusive" in Table 3. For the two tasks where IMC, ISMC, and DAR reported incorrect alarms, PredAbs and IMPACT delivered an incorrect alarm on one and failed during SMT solving on the other. The quantile plots in Fig. 5 further demonstrate that the three hardware-verification algorithms are not only more *effective* but also more *efficient* than the two software-verification algorithms.

To gain more insights on the strengths of hardware-verification algorithms, we conducted a detailed analysis on selected subcategories of the benchmark set. The results divided by subcategories are shown in the second half of Table 3. The subcategory *ReachSafety-ECA* consists of 1263 programs modeling event-condition-action systems [58]. IMC and ISMC showcased their unique capabilities in this subcategory by finding the most proofs and alarms, respectively. In comparison, PredAbs solved the fewest tasks in the subcategory. Profiling its run-time, we found that the abstraction computation of PredAbs, which involves an expensive model-enumeration step [7, 59], took up a significant amount of its CPU time and led to timeouts in many tasks. IMC and ISMC avoid the expensive abstraction computation by taking the union of interpolants as overapproximations.

The subcategory *ReachSafety-Sequentialized* is another example where IMC demonstrated its outstanding proof-finding ability. The 461 programs in this subcategory were obtained by sequentializing the execution of concurrent multi-threaded programs [60, 61]. IMC was able to delivered 11 proofs that none of the other four algorithms could find. In the subcategory *ReachSafety-Loops*, all three hardware-verification algorithms outperformed the two software-verification algorithms at both proof finding and bug hunting. Particularly, on several programs involving nonlinear arithmetics, IMC, ISMC, and DAR were able to detect bugs, while PredAbs and IMPACT got stuck at some difficult interpolation queries.

Our detailed analysis shows that the algorithms originated from hardware model checking can improve the state of the art of software verification by more than 40 % in some benchmark families, demonstrating the importance of systematically transfer knowledge between the two communities.

6.6 Threats to Validity

External Validity. The conclusions of our study are based on the used benchmark set, which is the largest and most diverse open-source collection of verification tasks in the programming language C. Although our experiments show that IMC is faster than ISMC and DAR, the benchmark set contains

many more safe than unsafe tasks. Therefore, the improvement of ISMC and DAR over IMC is biased since IMC is more effective in finding proofs.

We adopted algorithms that were originally designed for hardware circuits to software, and it was unclear which of the claims in the original papers will hold for software. Our study shows that the characteristics of the algorithms are transferable to software on the used benchmark set and that the investigated algorithms are robust regarding the representation of the verification tasks. However, it is still unclear which claims will hold for software with different features that are not covered in the used benchmark set. The comparison in Sect. 6.5 was performed on a subset of benchmark tasks containing at most one loop. Therefore, it is possible that the performance characteristics of the evaluated algorithms are different on multi-loop programs.

Internal Validity. To minimize confounding variables in the evaluation, the implementations of the two algorithms and the compared approaches are all realized in the mature and well-maintained software verifier CPAchecker. The evaluation is performed on more than 8 000 software-verification tasks to make the results more robust. Note that the number of verification tasks in our evaluation is much larger than the two previous works [20, 21].

For executing the experiments, we used the popular benchmarking framework BenchExec [57], which employs modern features of the Linux kernel, such as cgroups for resource measurement and control, name spaces for process isolation to prevent interference, and overlay file systems to prevent experiment runs from changing the state of the system. To mitigate interference from shared hardware resources, we make sure to never run two executions on the same physical cores (no hyper-threading across executions). However, the effectivity is more important than the CPU time in our experiments, therefore, the impact is limited. We set the CPU time limit to 1 800 s as in the DAR paper. In principle, we could use 10 000 s as in the ISMC paper, but the experiments will require considerably more time given the size of our benchmark set.

7 CONCLUSION

Hardware and software verification techniques deal with the same problem conceptually and rely on common theoretical cornerstones such as satisfiability and Craig interpolation. Even though the areas are closely related, there is a knowledge gap in how well the results obtained in one community are transferable to the other. This transferability study contributed to filling this gap. ISMC [20] and DAR [21] are two successful interpolation-based algorithms for hardware model checking. We implemented them in the software verifier CPAchecker to analyze C programs. To observe which claims about their characteristics in the original papers are transferable to software verification, we evaluated them against the software-verification adoption [22] of IMC [19], the baseline approach used in the original studies. The experiments were executed on an extensive benchmark set.

From the results, we confirmed the claims about the characteristics of the two algorithms for software verification: ISMC was faster than IMC to find bugs in programs, and local strengthening was often enough to refute spurious error paths and avoid expensive BMC queries in DAR. However, the claims in the original papers about the speedup of ISMC and DAR over IMC did not transfer to software verification. Overall, IMC was the fastest among the three algorithms on the software-verification benchmark set used in our evaluation. When compared to the state-of-the-art software-verification algorithms, IMC, ISMC, and DAR demonstrated superior effectiveness and efficiency. Most importantly, all three algorithms were able to solve tasks that the others could not, proving their indispensable values for software verification. Our work improves the knowledge about the transferability of ISMC and DAR to software verification and reveals the potential for improving software verification by adopting methods from hardware model checking. Our results will shed light on exchanging and unifying research findings between the two areas in both directions.

DATA-AVAILABILITY STATEMENT

A reproduction package [62] for the transferability study, including the implementations of the algorithms in CPAchecker, the evaluated benchmark set, and the resulting experimental data, is available on Zenodo. Additional information is available at https://www.sosy-lab.org/research/dar-ismc-transferability/.

FUNDING STATEMENT

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

REFERENCES

[1] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem. 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. 1990. Symbolic model checking: 10^{20} states and beyond. In *Proc. LICS*. IEEE, 428–439. https://doi.org/10.1109/LICS.1990.113767

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. 1990. Sequential circuit verification using symbolic model checking. In *Proc. DAC*. ACM, 46–51. https://doi.org/10.1145/123186.123223

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press. ISBN: 978-1-58603-929-5

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proc. DAC*. ACM, 530–535. https://doi.org/10.1145/378239.379017

[6] C. Barrett and C. Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[7] D. Beyer, M. Dangl, and P. Wendler. 2018. A unifying view on SMT-based software verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. https://doi.org/10.1007/s10817-017-9432-6

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic model checking without BDDs. In *Proc. TACAS (LNCS 1579)*. Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[9] M. Sheeran, S. Singh, and G. Stålmarck. 2000. Checking safety properties using induction and a SAT-solver. In *Proc. FMCAD*. Springer, 127–144. https://doi.org/10.1007/3-540-40922-X_8

[10] A. R. Bradley. 2011. SAT-based model checking without unrolling. In *Proc. VMCAI (LNCS 6538)*. Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[11] W. Craig. 1957. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593

[12] E. M. Clarke, D. Kröning, and F. Lerda. 2004. A tool for checking ANSI-C programs. In *Proc. TACAS (LNCS 2988)*. Springer, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[13] A. F. Donaldson, L. Haller, D. Kröning, and P. Rümmer. 2011. Software verification using k-induction. In *Proc. SAS (LNCS 6887)*. Springer, 351–368. https://doi.org/10.1007/978-3-642-23702-7_26

[14] A. Cimatti and A. Griggio. 2012. Software model checking via IC3. In *Proc. CAV (LNCS 7358)*. Springer, 277–293. https://doi.org/10.1007/978-3-642-31424-7_23

[15] P. Ralph. 2021. ACM SIGSOFT empirical standards released. *ACM SIGSOFT Softw. Eng. Notes* 46, 1 (2021), 19. https://doi.org/10.1145/3437479.3437483

[16] P. Ralph, S. Baltes, D. Bianculli, Y. Dittrich, M. Felderer, R. Feldt, A. Filieri, C. A. Furia, D. Graziotin, P. He, R. Hoda, N. Juristo, B. A. Kitchenham, R. Robbes, D. Méndez, J. S. Molléri, D. Spinellis, M. Staron, K. Stol, D. A. Tamburri, M. Torchiano, C. Treude, B. Turhan, and S. Vegas. 2021. Empirical standards for software engineering research. *arXiv/CoRR* 2010, 03525 (March 2021). https://doi.org/10.48550/arXiv.2010.03525

[17] K. Petersen and Ç. Gencel. 2013. Worldviews, research methods, and their relationship to validity in empirical software engineering research. In *Proc. IWSM-Mensura*. IEEE Computer Society, 81–89. https://doi.org/10.1109/IWSM-MENSURA.2013.22

[18] K.-J. Stol and B. Fitzgerald. 2018. The ABC of software engineering research. *ACM Trans. Softw. Eng. Methodol.* 27, 3 (2018), 11:1–11:51. https://doi.org/10.1145/3241743

[19] K. L. McMillan. 2003. Interpolation and SAT-based model checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. https://doi.org/10.1007/978-3-540-45069-6_1

[20] Y. Vizel and O. Grumberg. 2009. Interpolation-sequence based model checking. In *Proc. FMCAD*. IEEE, 1–8. https://doi.org/10.1109/FMCAD.2009.5351148

[21] Y. Vizel, O. Grumberg, and S. Shoham. 2013. Intertwined forward-backward reachability analysis using interpolants. In *Proc. TACAS (LNCS 7795)*. Springer, 308–323. https://doi.org/10.1007/978-3-642-36742-7_22

[22] D. Beyer, N.-Z. Lee, and P. Wendler. 2022. Interpolation and SAT-based model checking revisited: Adoption to software verification. *arXiv/CoRR* 2208, 05046 (July 2022). https://doi.org/10.48550/arXiv.2208.05046

[23] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from proofs. In *Proc. POPL*. ACM, 232–244. https://doi.org/10.1145/964001.964021

[25] K. L. McMillan. 2006. Lazy abstraction with interpolants. In *Proc. CAV (LNCS 4144)*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[26] D. Beyer. 2023. SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2023 and Test-Comp 2023). Zenodo. https://doi.org/10.5281/zenodo.7627783

[27] M. Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533 (2016), 452–454. https://doi.org/10.1038/533452a

[28] J. P. A. Ioannidis. 2005. Why most published research findings are false. *PLoS medicine* 2, 8 (2005), e124:0696–e124:0701. https://doi.org/10.1371/journal.pmed.0020124

[29] S. Krishnamurthi and J. Vitek. 2015. The real software crisis: Repeatability as a core value. *Commun. ACM* 58, 3 (2015), 34–36. https://doi.org/10.1145/2658987

[30] N. Juristo and O. S. Gómez. 2012. Replication of software engineering experiments. In *Empirical Software Engineering and Verification*. Springer, 60–88. https://doi.org/10.1007/978-3-642-25231-0_2

[31] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. 2008. Replication's role in software engineering. In *Guide to Advanced Empirical Software Engineering*. Springer, 365–379. https://doi.org/10.1007/978-1-84800-044-5_14

[32] C. S. Collberg and T. A. Proebsting. 2016. Repeatability in computer-systems research. *Commun. ACM* 59, 3 (2016), 62–69. https://doi.org/10.1145/2812803

[33] J. Vitek and T. Kalibera. 2011. Repeatability, reproducibility, and rigor in systems research. In *Proc. EMSOFT*. ACM, 33–38. https://doi.org/10.1145/2038642.2038650

[34] S. Winter, C. S. Timperley, B. Hermann, J. Cito, J. Bell, M. Hilton, and D. Beyer. 2022. A retrospective study of one decade of artifact evaluations. In *Proc. ESEC/FSE*. ACM, 145–156. https://doi.org/10.1145/3540250.3549172

[35] R. Heumüller, S. Nielebock, J. Krüger, and F. Ortmeier. 2020. Publish or perish, but do not forget your software artifacts. *Empirical Software Engineering* 25 (2020), 4585–4616. https://doi.org/10.1007/s10664-020-09851-6

[36] B. Hermann, S. Winter, and J. Siegmund. 2020. Community expectations for research artifacts and evaluation processes. In *Proc. ESEC/FSE*. ACM, 469–480. https://doi.org/10.1145/3368089.3409767

[37] C. S. Timperley, L. Herckis, C. L. Goues, and M. Hilton. 2021. Understanding and improving artifact sharing in software engineering research. *Empirical Software Engineering* 26, 4 (2021). https://doi.org/10.1007/s10664-021-09973-5

[38] C. Cadar, D. Dunbar, and D. R. Engler. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*. USENIX Association, 209–224. https://dl.acm.org/doi/10.5555/1855741.1855756

[39] E. F. Rizzi, S. Elbaum, and M. B. Dwyer. 2016. On the techniques we create, the tools we build, and their misalignments: A study of Klee. In *Proc. ICSE*. ACM, 132–143. https://doi.org/10.1145/2884781.2884835

[40] M. Heizmann, J. Hoenicke, and A. Podelski. 2009. Refinement of trace abstraction. In *Proc. SAS (LNCS 5673)*. Springer, 69–85. https://doi.org/10.1007/978-3-642-03237-0_7

[41] R. Jhala and K. L. McMillan. 2005. Interpolant-based transition relation approximation. In *Proc. CAV (LNCS 3576)*. Springer, 39–51. https://doi.org/10.1007/11513988_6

[42] O. Sery, G. Fedyukovich, and N. Sharygina. 2012. Interpolation-based function summaries in bounded model checking. In *Proc. HVC (LNCS 7261)*. Springer, 160–175. https://doi.org/10.1007/978-3-642-34188-5_15

[43] R. Brayton and A. Mishchenko. 2010. ABC: An academic industrial-strength verification tool. In *Proc. CAV (LNCS 6174)*. Springer, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[44] A. Goel and K. Sakallah. 2020. AVR: Abstractly verifying reachability. In *Proc. TACAS (LNCS 12078)*. Springer, 413–422. https://doi.org/10.1007/978-3-030-45190-5_23

[45] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software model checking for people who love automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[46] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. 2015. The SeaHorn verification framework. In *Proc. CAV (LNCS 9206)*. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[47] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-induction with continuously-refined invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. https://doi.org/10.1007/978-3-319-21690-4_42

[48] T. Lange, M. R. Neuhäußer, T. Noll, and J. Katoen. 2020. IC3 software model checking. *Int. J. Softw. Tools Technol. Transf.* 22, 2 (2020), 135–161. https://doi.org/10.1007/S10009-019-00547-X

[49] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. 2009. Software model checking via large-block encoding. In *Proc. FMCAD*. IEEE, 25–32. https://doi.org/10.1109/FMCAD.2009.5351147
[50] D. Beyer, S. Gulwani, and D. Schmidt. 2018. Combining model checking and data-flow analysis. In *Handbook of Model Checking*. Springer, 493–540. https://doi.org/10.1007/978-3-319-10575-8_16
[51] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. https://doi.org/10.1007/s10009-007-0044-z
[52] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV (LNCS 4590)*. Springer, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51
[53] D. Beyer, T. A. Henzinger, and G. Théoduloz. 2008. Program analysis with dynamic precision adjustment. In *Proc. ASE*. IEEE, 29–38. https://doi.org/10.1109/ASE.2008.13
[54] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. ISBN: 978-0-201-10088-4
[55] A. F. Donaldson, D. Kröning, and P. Rümmer. 2011. Automatic analysis of DMA races using model checking and *k*-induction. *FMSD* 39, 1 (2011), 83–113. https://doi.org/10.1007/s10703-011-0124-2
[56] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. 2013. The MathSAT5 SMT solver. In *Proc. TACAS (LNCS 7795)*. Springer, 93–107. https://doi.org/10.1007/978-3-642-36742-7_7
[57] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable benchmarking: Requirements and solutions. *Int. J. Softw. Tools Technol. Transfer* 21, 1 (2019), 1–29. https://doi.org/10.1007/s10009-017-0469-y
[58] F. Howar, M. Isberner, M. Merten, B. Steffen, and D. Beyer. 2012. The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In *Proc. ISoLA (LNCS 7609)*. Springer, 608–614. https://doi.org/10.1007/978-3-642-34026-0_45
[59] T. Ball, A. Podelski, and S. K. Rajamani. 2001. Boolean and Cartesian abstraction for model checking C programs. In *Proc. TACAS (LNCS 2031)*. Springer, 268–283. https://doi.org/10.1007/3-540-45319-9_19
[60] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. 2010. Verifying SystemC: A software model checking approach. In *Proc. FMCAD*. FMCAD Inc., 51–59. https://ieeexplore.ieee.org/document/5770933
[61] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. 2014. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *Proc. CAV (LNCS 8559)*. Springer, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39
[62] D. Beyer, P.-C. Chien, M. Jankola, and N.-Z. Lee. 2024. Reproduction package for FSE 2024 article 'A transferability study of interpolation-based hardware model checking for software verification'. Zenodo. https://doi.org/10.5281/zenodo.11070973

Received 2023-09-28; accepted 2024-04-16