Proof-Guided Underapproximation Widening for Bounded Model Checking

Prantik Chatterjee[1]([✉]) , Jaydeepsinh Meda[2], Akash Lal[3],
and Subhajit Roy[1]

[1] Indian Institute of Technology Kanpur, Kanpur, India
prantik@cse.iitk.ac.in, subhajit@iitk.ac.in
[2] Oracle, Bengaluru, India
[3] Microsoft Research, Bengaluru, India
akashl@microsoft.com

Abstract. Bounded Model Checking (BMC) is a popularly used strategy for program verification and it has been explored extensively over the past decade. Despite such a long history, BMC still faces scalability challenges as programs continue to grow larger and more complex. One approach that has proven to be effective in verifying large programs is called Counterexample Guided Abstraction Refinement (CEGAR). In this work, we propose a complementary approach to CEGAR for bounded model checking of sequential programs: in contrast to CEGAR, our algorithm gradually widens underapproximations of a program, guided by the proofs of unsatisfiability. We implemented our ideas in a tool called LEGION. We compare the performance of LEGION against that of CORRAL, a state-of-the-art verifier from Microsoft, that utilizes the CEGAR strategy. We conduct our experiments on 727 Windows and Linux device driver benchmarks. We find that LEGION is able to solve 12% more instances than CORRAL and that LEGION exhibits a complementary behavior to that of CORRAL. Motivated by this, we also build a portfolio verifier, LEGION$^+$, that attempts to draw the best of LEGION and CORRAL. Our portfolio, LEGION$^+$, solves 15% more benchmarks than CORRAL with similar computational resource constraints (i.e. each verifier in the portfolio is run with a time budget that is half of the time budget of CORRAL). Moreover, it is found to be 2.9$\times$ faster than CORRAL on benchmarks that are solved by both CORRAL and LEGION$^+$.

Keywords: Verification · Bounded model checking ·
Underapproximation widening

1 Introduction

Bounded Model Checking (BMC) [11,20,26,33] is a popular option for program verification, primarily due to its ability of side-stepping the necessity of synthesizing complex invariants. BMC harnesses the power of modern SMT solvers to verify a bounded set of behaviors of a program. The user, if interested, may

re-attempt verification with larger bounds once the program is proven correct with small bounds.

BMC operates by constructing a logical formula that symbolically captures all states reachable by a program under a user-provided bound. A *query*, referred to as the *verification condition (VC)*, is constructed as a conjunction of the program semantics and the negation of the property, which is also expressed as a logical formula. If the verification condition is satisfiable, it implies that some program execution violated the property of interest, thus the program is faulty. If unsatisfiable, the program satisfies the property, i.e. the program is safe under the chosen bound.

However, for large programs, BMC faces scalability challenges as the verification condition for the program tends to grow large, posing difficulties for the SMT solver. Prior work has answered this challenge by using the popular *counterexample-guided abstraction refinement (CEGAR)* strategy: start off with the VC for an *abstraction* of the program, and incrementally refine the abstraction until the program is decided as safe or faulty. The *Stratified Inlining (SI)* [26] algorithm is an instance of this strategy. SI starts off with an abstraction of only the entry procedure of the program, and then incrementally inlines callees, guided by counterexamples. Not surprisingly, the dynamic inlining strategy of SI has been found to be significantly more scalable than algorithms that statically inline all procedures [25]. The SI algorithm is used in practice by the CORRAL [24] verifier that powers Microsoft's Static Driver Verifier (SDV) [4].

In this work, we propose a new algorithm that uses proofs of unsatisfiability to widen underapproximate models of the program en route to verification of sequential programs. Our algorithm starts off by constructing a partial verification condition for only the program entry procedure and blocks all paths that invoke calls to procedures that have not yet been inlined. This constructs an underapproximation of the original program (because paths are blocked). A satisfiable result on an underapproximation will indicate the presence of a bug. If the VC is unsatisfiable, we examine its *proof of unsatisfiability* in order to guide the inlining of called procedures. The program can be declared safe when the proof of unsatisfiability does not depend on any procedure call that has not been inlined yet. We implemented our ideas in a tool called LEGION.

Further, we found that our underapproximation widening algorithm and the abstraction refinement strategy (used by CORRAL) demonstrate complementary behaviors—many programs that CORRAL struggles on, yield to the underapproximation based technique, and vice-versa. This observation motivated us to build a portfolio verifier, LEGION$^+$, that runs both these techniques in parallel. We found that the portfolio is more effective than any of the tools alone (with similar computational resources, i.e. each verifier in the portfolio is run with a time budget that is half of the time budget of CORRAL). Both LEGION and LEGION$^+$ are available open-source at the *legion* branch of the *corral* repository[1].

Our experiments are conducted on 727 Windows and Linux device driver benchmarks on which CORRAL struggles, i.e., CORRAL is unable to solve any of

---

[1] https://github.com/boogie-org/corral.git (branch: `legion`).

these benchmarks in less than 200 s. We find that LEGION is able to solve 12% more instances than CORRAL with a time budget of 2 h per instance. Further, the portfolio verifier, LEGION$^+$, given half the time budget of CORRAL, solves 15% more benchmarks than CORRAL, and it is found to be 2.9× faster than CORRAL on benchmarks that are solved by both CORRAL and LEGION$^+$.

The primary contributions of this paper are as follows:

– We design a new algorithm, *Underapproximation Widening guided Stratified Inlining*, that uses proof-based artifacts to widen underapproximate models (in contrast to using counterexamples to refine overapproximate models).
– We implemented our ideas in a tool called LEGION for bounded program verification.
– We also design a portfolio verifier, LEGION$^+$, that includes both overapproximation refinement and underapproximation widening to verify a program in an attempt to reap the benefits of both worlds.
– We evaluate both LEGION and LEGION$^+$ on a set of 727 programs from Windows Device Drivers [31] from the SDV test-suite and Linux Device Drivers from SVCOMP [7] benchmarks.

2   Background

This section presents background material that we use in the rest of the paper.

A logical formula consists of literals. A literal is either a variable or the negation of a variable. A logical formula expressed in a *Conjunctive Normal Formal* (CNF) is a *conjunction* of clauses where each clause is a *disjunction* of literals. Given a logical formula, a *satisfiability* solver returns whether the formula is *satisfiable* (SAT) or *unsatisfiable* (UNSAT). If a formula is SAT, the solver provides a model in the form of a satisfying assignment of the variables. If a formula is UNSAT, the solver returns an *unsatisfiable core* (unsat core), which is a subset of clauses of the input formula whose conjunction is still UNSAT.

2.1   Language Model

We consider a programming language that represents a *passified* form of BOO-GIE programs [8]. A program consists of multiple procedures (*Proc*). We assume an *entry-point* procedure called `main` where program execution starts. Each procedure can have any number of local variable declarations followed by a series of basic blocks (*BasicBlock*). We assume that local variables are initially unconstrained. A basic block is labeled by a unique identifier and consists of multiple statements (*Stmt*) followed by a single control statement (*ControlStmt*). A control statement is either a *goto*, which takes a sequence of basic block labels and non-deterministically picks one to jump to, or a *return* that returns control back to the caller. Returning from `main` terminates the program execution. A statement is either an *assume* command or a procedure *call*. The statement (*assume* φ) allows a feasible execution only if φ is satisfiable.

```
procedure main() {
  int x, y, z; bool c;
  L0:  assume x == 0;
       assume y == 0;
       goto L1, L2;
  L1:  assume c;
       call foo(x,y);
       goto L3;
  L2:  assume !c;
       call bar(x,y);
       goto L3;
  L3:  assume y != 0
       return; }

procedure foo(int x, int y) {
  bool d;
  L5:  goto L6, L7;
  L6:  assume d;
       call foo1(x, y);
       goto L8;
  L7:  assume !d;
       call foo2(x, y);
       goto L8;
  L8:  return; }
```

```
procedure foo1(int x, int y) {
  L9:  assume y == x + 1;
       return; }

procedure foo2(int x, int y) {
  L10:  assume y == x - 1;
        return; }

procedure bar(int x, int y) {
  bool e;
  L11:  goto L12, L13;
  L12:  assume e;
        call bar1(x, y);
        goto L14;
  L13:  assume !e;
        call bar2(x, y);
        goto L14;
  L14:  return; }

procedure bar1(int x, int y) {
  L15:  assume y == x + 10;
        return; }

procedure bar2(int x, int y) {
  L16:  assume y == x - 10;
        return; }
```

Fig. 1. A passified program

We leave the set of variable types (*Type*) and expressions (*Expr*) unspecified. In practice, we can use any expression language that can be directly encoded in SMT. Our implementation uses linear arithmetic, fixed-size bit-vectors, uninterpreted functions, and extensional arrays. This combination is sufficient to realistically translate C programs into our language representation [21, 24].

Note that the programs that we consider do not have global variables, return parameters of procedures, or assignments. These restrictions are without loss of generality [23]. Conversion of these additional feature into our language representation is readily available in tools like BOOGIE. A passified program makes it easy to describe the verification-condition generation process.

Given a program P, we consider the verification question of whether there exists a terminating execution of P. To be precise, we are interested in finding out whether there is any execution of main that reaches its *return* statement. If no such execution exists, then P is considered verified, or SAFE. Otherwise, we say that P is UNSAFE and return the execution trace with concrete variable values along the trace. Note that we consider a bounded version of the verification problem, i.e., we require that P does not contain any loops or recursive procedure calls. All such loops and recursive calls must be unrolled to a pre-determined
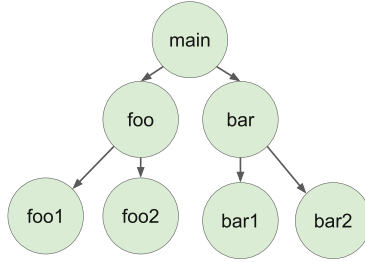
Fig. 2. Call graph of the program in Fig. 1.

$\text{pVC}(\texttt{main}) :$

$\qquad blk_{L0}$

$\wedge \quad (blk_{L0} \implies x == 0 \wedge y == 0$

$\qquad\qquad \wedge ((blk_{L1} \wedge flow(0) == 1) \vee (blk_{L2} \wedge flow(0) == 2)))$

$\wedge \quad (blk_{L1} \implies c \wedge blk_{L3} \wedge flow(1) == 3)$

$\wedge \quad (blk_{L2} \implies \neg c \wedge blk_{L3} \wedge flow(2) == 3)$

$\wedge \quad (blk_{L3} \implies y \neq 0)$

Fig. 3. Partial VC of `main()`

depth before proceeding with verification, and thus, the verification problem now becomes decidable (if the expression language of the program is decidable) [23].

2.2   VC Generation for a Procedure

Consider a procedure `baz` that does not contain any procedure calls. This section outlines one way of verifying `baz`, i.e., finding out if it has a terminating execution. We use a process called Verification Condition (VC) generation on `baz` to construct a logical formula Φ and feed it to an SMT solver. If Φ is UNSAT, then the *return* statement in `baz` is unreachable and `baz` is SAFE. Otherwise, we extract the satisfiable model from the SMT solver, construct the execution trace and return UNSAFE along with the trace. We now outline the VC-generation process.

Suppose that `baz` takes input arguments $\vec{x}$. For each basic block j in `baz`, we define a boolean variable blk_j that is termed as the *control-flow* variable. Let st_j denote the conjunction of all assume statements in basic block j. Let $successor(j)$ denote the targets of the *goto* statement in j, i.e., all the successor basic blocks in `baz`, to which control may jump non-deterministically from j. Let i_j be a unique integer constant representing basic block j. We also define an uninterpreted function $flow : \mathbb{Z} \rightarrow \mathbb{Z}$ that records the non-deterministic choice of the successor basic block of j. Given the above, we construct a logical formula ψ_j for each basic block j as follows:

$$blk_j \Rightarrow (st_j \wedge \bigvee_{s \in successor(j)} (blk_s \wedge (i_s == flow(i_j))))$$

If basic block j ends with a *return* statement instead of a *goto*, then ψ_j is:

$$blk_j \Rightarrow st_j$$

Assuming the first basic block of `baz`, where procedure execution begin, is labeled s, the VC of `baz` is constructed as follows:

$$blk_s \wedge \bigwedge_{l \in basicblocks(p)} \psi_l$$

In Fig. 3, we show the VC of `main` of the program in Fig. 1 as an example, where we ignore the procedure calls in `main` (i.e., treat them as (*assume true*)). We term such a VC (of a procedure where its calls are skipped) as the *partial VC* (pVC) of the procedure.

2.3   Static Versus Dynamic Inlining

Given a program `P` with a starting procedure `main`, one simple way to verify `P` would be to construct the VC of `main` by inlining all the procedure calls and check the satisfiability of VC(`main`) with an SMT solver. However, employing such a *static inlining* strategy can cause an exponential blowup in the size of the VC. Hence, we instead make use of *dynamic inlining* algorithm, called Stratified Inlining (SI) [26], that employs a Counterexample Guided Abstraction Refinement (CEGAR) technique [14] to *dynamically* inline procedure VCs. It has been shown that dynamic inlining scales better than static inlining [25]. Dynamic inlining produces more compact VCs during abstraction refinement which leads to significantly faster program verification.

2.4   Verification with Stratified Inlining

The working of SI is shown in Algorithm 1. For the sake of simplicity, let us assume that each basic block in `P` may contain only a single procedure call. Every program point, from which a procedure is called, is termed as a callsite. For example, `main` in Fig. 1, has two callsites; `foo` and `bar` which are called from basic blocks `L1`, `L2` and `L3` respectively. A static instance of a callsite is denoted with a pair (l, c) where l denotes the basic block identifier from which a call to the procedure c is made. A dynamic callsite is defined as a stack of static callsites which represents the runtime stack during a program's execution with `main` being present at the bottom of the stack. For example, the dynamic callsite corresponding to the call `foo` from `L1` in `main` is given by $[\texttt{main}, (\texttt{L1}, \texttt{foo})]$. The call graph of the program in Fig. 1 is shown in Fig. 2.

Algorithm 1: Stratified Inlining (SI) algorithm.

Input: program P with starting procedure main
Input: An SMT solver $\mathcal{S}$
Output: SAFE, or UNSAFE(τ)
1 $C \leftarrow \{[\text{main}, s] \mid s \in callsites(\text{main})\}$
2 $\mathcal{S}.\text{Assert}(\text{pVC}(\text{main}, [\text{main}]))$
3 while *true* **do**
4 | $outcome \leftarrow \text{OVERREFSTEP}(\text{P}, C, \mathcal{S})$
5 | **if** *outcome == SAFE $\vee$ outcome == UNSAFE(τ)* **then**
6 | | **return** *outcome*
7 | **else**
8 | | **let** NODECISION$(\_, C') = outcome$
9 | | $C \leftarrow C'$

The SI algorithm takes as input a program P with a starting procedure main and an SMT solver $\mathcal{S}$. Initially, we add the dynamic callsites in main to a list C (Line 1) and then inline main, i.e., assert the pVC of main (Line 2). The callsites in C are termed as *open* callsites because they have not yet been inlined. The above steps construct an abstraction of P. The SI algorithm then iteratively calls the OVERREFSTEP routine on this abstraction (Line 4) to perform gradual refinement until we can reach a decision about whether P is SAFE or not. Each invocation of OVERREFSTEP can potentially inline more procedures by asserting their partial VC to the solver $\mathcal{S}$. Thus, the state of the solver, as well as the set of open callsites C change across invocations of OVERREFSTEP. We discuss the *Overapproximation Refinement Guided Stratified Inlining* (*OverRefSI*) strategy used by the OVERREFSTEP routine in Sect. 2.5.

2.5   Overapproximation Refinement Guided Stratified Inlining

The OVERREFSTEP routine given in Algorithm 2 demonstrates the inner workings of the *OverRefSI* strategy at each verification step. The *OverRefSI* strategy [26] for verifying a program works by iteratively firing overapproximation queries and gradually refining the abstraction of P. If the query returns UNSAT, then we can conclude that P is SAFE with respect to the given property. Otherwise, we extract all the open callsites that appear on the counterexample trace and refine the abstraction of P by inlining these callsites. If the counterexample trace contains no open callsites, then P is UNSAFE and we return the verdict along with the counterexample trace.

The OVERREFSTEP routine takes as input a program P, a set of open callsites C and an SMT solver $\mathcal{S}$. The OVERREFSTEP routine is called iteratively in order to verify the safety of P. We demonstrate the working of *OverRefSI* to verify the pVC of main of Fig. 1 in Table 1. At the beginning, the SI algorithm asserts the pVC of main to $\mathcal{S}$ and adds $[\text{main}, (\text{L1}, \text{foo})]$ and $[\text{main}, (\text{L2}, \text{bar})]$ to the list of open callsites C in step 0.

Algorithm 2: OVERREFSTEP(P, C, $\mathcal{S}$)

Input: procedure P, set of callsites C, SMT solver $\mathcal{S}$
Output: SAFE, UNSAFE(*trace*), NODECISION(τ, C)

1 *// Overapproximate check*
2 **if** $\mathcal{S}.Check()$ == UNSAT **then**
3 | **return** SAFE
4 **else**
5 | $\tau \leftarrow$ opencallsites($\mathcal{S}$.Model())
6 **if** τ == $\emptyset$ **then**
7 | **return** UNSAFE($\mathcal{S}$.Model())
8 **else**
9 | $C' \leftarrow \emptyset$
10 | **forall** $c \in \tau$ **do**
11 | | $C' \leftarrow$ INLINE(P, c)
12 | $C \leftarrow (C - \tau) \cup C'$
13 | **return** NODECISION(τ, C)

Next, the SI algorithm calls OVERREFSTEP with P, C and $\mathcal{S}$ as arguments. OVERREFSTEP fires an overapproximation query in Line 2. If the query is unsatisfiable, we return the SAFE verdict. If the query is satisfiable, we get the counterexample trace and extract all the open callsites on the trace in τ (Line 5). If τ is empty, i.e., the counterexample trace contains no open callsites, then the trace is not spurious and we can return an UNSAFE verdict with the trace (Line 7). Otherwise, we inline all the callsites in τ and add all the new callsites that opened up due to the inlinings in C' (Line 11). Inlining a callsite c consists of asserting the partial VC of the procedure that was invoked from c.

Subsequently, the inlined callsites are removed from the list of open callsites C and new callsites that opened up due to the inlinings are added to C (Line 12). For example, in step 1 of Table 1, OVERREFSTEP fires an overapproximation query that returns SAT with a counterexample trace that contains the callsite of foo, i.e., [main, (L1,foo)]. This callsite is then inlined by asserting the pVC of foo to the solver. This opens up the callsites of foo1 and foo2. Since we have not been able to arrive at a decision regarding the safety of P at this step, a verdict of NODECISION is returned along with the list of inlined callsites τ and the new list of open callsites C (Line 13).

Next, the SI algorithm calls OVERREFSTEP again and in step 2, it fires an overapproximation query again, which returns SAT with the counterexample trace containing the open callsite of foo1 that we inline by asserting the pVC of foo1. The verification process continues in this way by inlining the open callsites on the counterexample trace in every step, which gradually refines the pVC of main. Finally, in step 7, the overapproximation query returns UNSAT from which we can conclude that main is SAFE.

Table 1. Execution of *OverRefSI* on the program of Fig. 1

STEP	Action	Open callsites
0	Assert pVC(main)	[main, (L1,foo)]
		[main, (L2,bar)]
1	Overapprox check: SAT	
	Assert pVC(foo)	[main, (L2,bar)]
		[main, (L1, foo), (L6, foo1)]
		[main, (L1,foo), (L7, foo2)]
2	Overapprox check: SAT	
	Assert pVC(foo1)	[main, (L2,bar)]
		[main, (L1,foo), (L7, foo2)]
3	Overapprox check: SAT	
	Assert pVC(foo2)	[main, (L2,bar)]
4	Overapprox check: SAT	
	Assert pVC(bar)	[main, (L2,bar), (L12, bar1)]
		[main, (L2,bar), (L13, bar2)]
5	Overapprox check: SAT	
	Assert pVC(bar1)	[main, (L2,bar), (L13, bar2)]
6	Overapprox check: SAT	
	Assert pVC(bar2)	
7	Overapprox check: UNSAT	
	Return SAFE	

3   Overview

3.1   Underapproximation Widening

We propose a novel algorithm, *Underapproximation Widening Guided Stratified Inlining* (*UnderWidenSI*), that uses proofs of unsatisfiability to guide stratified inlining. *UnderWidenSI* maintains an underapproximated model of the target program and *widens* it until either the program is verified as *safe* or a bug is found.

We illustrate the *UnderWidenSI* strategy in Figs. 4a to 4d. Assume that we are trying to verify whether some required property holds on a program. The space contained by the *yellow* ovals show the reachable program states while the *red* ovals depict error states on which the required property does not hold. The objective of a verification algorithm is to construct a *model* of the program that is precise enough to show that the program can reach an error state or prove that the error states are unreachable. Figures 4a to 4c show a safe program while Fig. 4d depicts an unsafe program.

Consider Fig. 4a: the *UnderWidenSI* algorithm starts off with the partial verification condition of the entry procedure and "blocks" executions though all its open callsites.

(a) Underapproximate the program by blocking C_1 and C_2.

(b) Widen by inlining C_2 and construct a new underapproximation by blocking C_1, C_3 and C_4.

(c) Widen by inlining C_3, C_4 and verify as *safe*.

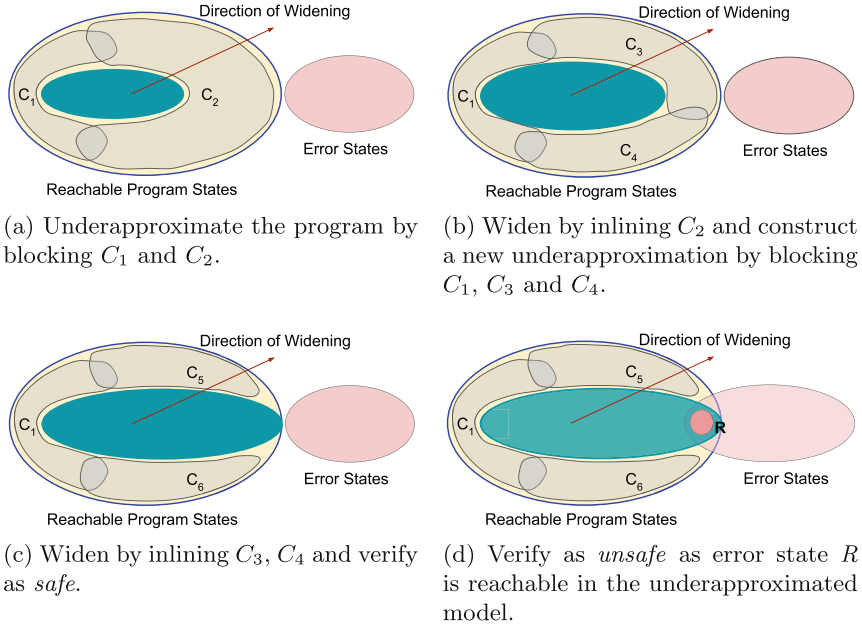(d) Verify as *unsafe* as error state R is reachable in the underapproximated model.

Fig. 4. How *UnderWidenSI* works

Definition (Blocked callsites). We use the term, *blocking* a callsite C, to imply that all paths that reach C are deemed infeasible. That is, blocking a callsite has the effect of replacing the callsite by (*assume false*).

Essentially, blocking callsites creates underapproximations of the set of feasible program paths. Such underapproximated VCs can be constructed by asserting additional *blocking* clauses corresponding to the control-flow variables of the open callsites. These blocks disallow reachability to certain program states. For example, in Fig. 4a, we construct an underapproximated model of the program by blocking the open callsites C_1 and C_2. The inner *green* oval depicts the program states that are reachable in the underapproximated model, whereas the outer *gray* regions demonstrate the states that are unreachable due to the blocks on C_1 and C_2.

If the verification query on this model (conjunction of the underapproximated model and the negation of the property) returns SAT, it implies that an error state in indeed reachable. On the other hand, if the query returns UNSAT (as shown in Fig. 4a), we need to *widen* the model to procure additional reachable executions. We guide this widening operation by extracting the reason for this unsatisfiability from a *minimal unsat core*[2] of the query, that returns the set of block clauses; the callsites corresponding to these blocking clauses constitutes

---

[2] Although there may exist multiple minimal unsat cores, we found via some preliminary experiments that the choice of the unsat core does not have a significant impact on the overall runtime of our algorithm (on an average).

(a) *OverRefSI* on correct programs.   (b) *OverRefSI* on buggy programs.
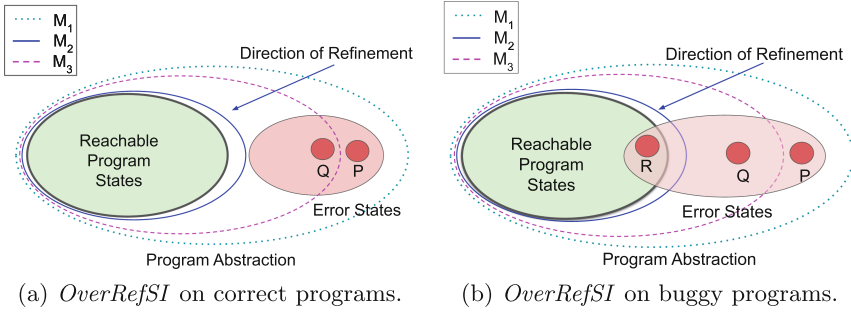
Fig. 5. How *OverRefSI* works

a *reason* of why the current underapproximate model is not able to reach any of the error states. Hence, we widen the model by unblocking exactly these callsites leading to a wider model (see Fig. 4b). The widening by inlining C_2 causes a stratified inlining step, and hence may open up new callsites, say C_3 and C_4.

We proceed in the same manner by blocking these open callsites and repeat the query. Finally, (in Fig. 4c) we construct an underapproximated model that still does not intersect with the error states. However, in this case, the unsat core does not contain any blocked clause, as none of the currently blocked callsites would have allowed widening in the direction of the error states.

The unsat core provides a *direction* for widening towards the error states. This also allows us to declare that the program is *safe* without requiring to widen the model to encompass the set of all reachable program states—if the verification query is UNSAT and the unsat core does not contain any blocked clause, then this forms a sufficient condition to declare the program *safe.*

Figure 4d shows how our algorithm proceeds for a faulty program: it incrementally widens the model in the direction of the error states till an error state R is reached. At this point, the *UnderWidenSI* algorithm declares the program as *unsafe.*

Let us now contrast the *UnderWidenSI* strategy with the *OverRefSI* strategy, popularly known as *counterexample-guided abstraction refinement (CEGAR)*, which currently drives the SI algorithm in CORRAL. *OverRefSI* starts off with an overapproximated model of the program: the pVC of the entry procedure with all callsites replaced by non-deterministic updates to its set of modified variables. For example, in Fig. 5a, *OverRefSI* constructs an abstract program/overapproximated model M_1 of the program by overapproximating the open callsites. If the resulting verification condition is SAT, it examines the generated counterexample to check if it spurious. If the counterexample is found to be a true bug, it declares the program *unsafe.* If the counterexample is spurious, the model is refined to eliminate this spurious counterexample. For example, in Fig. 5a, we find that there exists an error state/counterexample P within M_1, where the property can be violated. Hence, *OverRefSI* refines M_1 in Fig. 5a by

inlining the overapproximated callsites through which P is reachable. The refinement is done to rule out P as a counterexample, i.e., P becomes unreachable after refinement. We observe in Fig. 5a, that after the first round of refinement, P is no longer reachable in the overapproximated M_2, however, we can still find another counterexample Q. Hence, the abstraction M_2 is refined again. The program is declared *safe* when the model cannot reach any error state. Note that the algorithm can prove the safety of the program without requiring to precisely capture the exact set of reachable program state.

OverRefSI and *UnderWidenSI* are complementary: while *OverRefSI* maintains an overapproximated model and refines the model (shrinking the set of reachable states), *UnderWidenSI* maintains an underapproximated model and widens the model (expanding the set of reachable states) incrementally. In terms of the algorithmic details, the *OverRefSI* algorithm in CORRAL uses the models (the counterexamples) to drive refinements, whereas our *UnderWidenSI* algorithm uses the proof (the unsat core) to guide the widenings.

4   Algorithms

4.1   Underapproximation Widening Guided Stratified Inlining (*UnderWidenSI*)

The UNDERWIDENSTEP routine in Algorithm 3 demonstrates how the *UnderWidenSI* strategy works in each verification step. It takes as input a procedure P, a set of open callsites C and an SMT solver $\mathcal{S}$. The UNDERWIDENSTEP routine is called by the SI algorithm (instead of OVERREFSTEP in Line 4) iteratively in order to verify the safety of P.

In the beginning, we construct an underapproximated pVC of the input procedure P by blocking all calls through the open callsites in C (Line 4). Next, we fire an underapproximation query (Line 5). If the query returns SAT, then we return the verdict UNSAFE with the counterexample trace (Line 6). Otherwise, we get the minimal unsatisfiable core uc and extract all the blocked callsites which appear on uc in μ (Line 8).

If μ does not contain any blocked callsites, we deduce that P is SAFE. The proof of the safety of P is captured by uc. Hence, we return the verdict that P is SAFE. Otherwise, each of the callsites in μ are then inlined (Line 15) which constructs a refinement of P. The inlined callsites are then removed from the list of open callsites C and new callsites that opened up due to the inlinings are added to C (Line 16).

When the algorithm is unable to arrive at a decision regarding the safety of P, it returns a verdict of NODECISION along with the list of inlined callsites μ and the new list of open callsites C (Line 13).

Example. We demonstrate the working of *UnderWidenSI* to verify the pVC of main of Fig. 1 in Table 2. Initially, we assert the pVC of main and add [main, (L1, foo)] and [main, (L2, bar)] to the list of open callsites in step 0.

Algorithm 3: UNDERWIDENSTEP(P, C, $\mathcal{S}$)

    Input: procedure P, set of callsites C, SMT solver $\mathcal{S}$
    Output: SAFE, UNSAFE(*trace*), NODECISION(μ, C)

1   // *Underapproximate check*
2 $\mathcal{S}$.Push()
3   **forall** $c \in C$ **do**
4 | $\mathcal{S}$.Assert($\neg ControlVariable(c)$)

5   **if** $\mathcal{S}$.*Check()* == SAT **then**
6 | **return** UNSAFE($\mathcal{S}$.Model())
7   **else**
8 | $\mu \leftarrow$ BlockedCallsites($\mathcal{S}$.UnsatCore())

9 $\mathcal{S}$.Pop()
10 **if** $\mu == \emptyset$ **then**
11 | **return** SAFE

12 **else**
13 | $C' \leftarrow \emptyset$
14 | **forall** $c \in \mu$ **do**
15 | | $C' \leftarrow$ INLINE(P, c)
16 | $C \leftarrow (C - \mu) \cup C'$
17 | **return** NODECISION(μ, C)

Replacing each of the open callsites with (*assume false*) statement, i.e., blocking them, constructs an underapproximation of the program. If an SMT solver query on this underapproximation returns SAT, then the program is surely UNSAFE as the satisfiable model can only represent an execution trace that goes through inlined callsites. In that case, we can return the verdict UNSAFE along with an error trace constructed from the model. On the other hand, if the underapproximation check returns UNSAT, then we cannot return a verdict on the safety of the program immediately.

Following this, in step 1 (see Table 2), we push a new frame on the solver and assert $(\neg blk_{L1} \wedge \neg blk_{L2})$ to block executions through the callsites of `foo` and `bar` respectively to construct the underapproximated pVC of `main`. We query the solver with these constraints. Figure 1 shows that if we block executions through basic blocks `L1` and `L2`, the program cannot terminate, i.e., the *return* statement in `L3` is not reachable. Hence, the solver returns UNSAT. The reason for the unsatisfiability is blocking executions through both `L1` and `L2`.

To widen the underapproximated model of the program so that we may reach `L3`, we need to remove the block on at least one of them and inline the respective callsite. The unsat core, in this case, contains the callsite of var`bar` in basic block `L2`. Therefore, we pop the earlier solver frame containing blocked clauses and assert $(blk_{L2} \implies \text{pVC}(bar))$ in the solver. Inlining `bar`, opens up the callsites [`main`, (`L2`, `bar`), (`L12`, `bar1`)] and [`main`, (`L2`, `bar`), (`L13`, `bar2`)].

Next, in step 2, we again construct the underapproximated pVC of `main` by blocking executions through the callsites of `foo`, `bar1` and `bar2`. The solver

Table 2. Execution of *UnderWidenSI* on the program of Fig. 1

Step	Action	Open callsites
0	Assert pVC(`main`)	[`main`, (`L1`,`foo`)]
		[`main`, (`L2`,`bar`)]
1	Underapprox check: `UNSAT`	
	Assert pVC(`bar`)	[`main`, (`L1`,`foo`)],
		[`main`, (`L2`,`bar`), (`L12`, `bar1`)]
		[`main`, (`L2`,`bar`), (`L13`, `bar2`)]
2	Underapprox check: `UNSAT`	
	Assert pVC(`foo`)	[`main`, (`L1`,`foo`), (`L6`, `foo1`)]
	Assert pVC(`bar1`)	[`main`, (`L1`,`foo`), (`L7`, `foo2`)]
	Assert pVC(`bar2`)	
3	Underapprox check: `UNSAT`	
	Assert pVC(`foo1`)	
	Assert pVC(`foo2`)	
4	Underapprox check: `UNSAT`	
	Return Safe	

query returns UNSAT with *uc* containing the callsites of `foo`, `bar1` and `bar2` which are inlined.

In step 3, the callsites of `foo1` and `foo2` are now open. Blocking both of these callsites and making an underapproximation check returns UNSAT with *uc* containing the callsites of `foo1` and `foo2`. These callsites are now inlined.

In step 4, the underapproximation query returns UNSAT and *uc* contains no blocked callsites. This points to the fact that *uc* contains only inlined callsites, i.e., starting from step 0 if we only inline the callsites in *uc* and leave the remaining callsites overapproximated, we will still get an UNSAT. Therefore, *uc* is the proof of the safety of the program and we return the verdict that the pVC of `main` is safe.

Note that when the underapproximation query returns SAT, then the counterexample trace is constructed on the underapproximated program, i.e., the trace may contain only blocked and inlined callsites. The underapproximated program represents a subset of the paths in the original program, therefore, any counterexample trace present in the underapproximated program is sure to be present in the original program as well. Therefore, if the underapproximated program is unsafe, the original program is unsafe as well.

We have implemented the *UnderWidenSI* algorithm in Legion. We compare the performance of the *UnderWidenSI* algorithm in Legion against that of Corral which uses *OverRefSI*.

4.2 Portfolio Technique

The complementary behavior of the *OverRefSI* and the *UnderWidenSI* algorithms motivate us to design a portfolio approach for verifying a program. The

portfolio strategy incorporates both the *OverRefSI* algorithm used by CORRAL and the *UnderWidenSI* algorithm implemented in LEGION. We refer to the portfolio verifier as LEGION$^+$. For each program, LEGION$^+$ runs both CORRAL and LEGION in parallel. LEGION$^+$ terminates verification as soon as one of the algorithms finishes verification and reports the outcome. We discuss the performance of LEGION$^+$ against that of CORRAL and LEGION in Sect. 5.

5   Experimental Results

We have built a tool, LEGION, that implements our *UnderWidenSI* algorithm. To compare against *OverRefSI*, we use CORRAL [26], a state-of-the-art verifier used at Microsoft [24]. We also build a portfolio solver, LEGION$^+$, that runs both CORRAL and LEGION in parallel. Whenever one of the tools finish verification, LEGION$^+$ terminates the algorithms and reports the outcome.

We compare the performance of CORRAL against LEGION and LEGION$^+$ on a suite of Windows and Linux device driver benchmarks. The Windows device driver benchmarks are obtained by running Static Driver Verifier (SDV) [4] on real windows device drivers that exercise all features of the C language such as arrays, heaps, pointers, loops, recursion etc. SDV compiles these drivers into a suite of BOOGIE [8] programs, each of which is a device driver paired with property (compilation is detailed in [24]). Note that, although the suite of Windows device drivers compiled into BOOGIE programs are available as SDV benchmarks [31], the actual C programs are internal to Microsoft.

Along with this, we also use a set of Linux device drivers that are available as C programs as part of the SVCOMP benchmarks suite [7]. We used SMACK [36] to compile the Linux device drivers into BOOGIE programs. Overall, we elect to use a total of 727 hard programs, on which CORRAL took more than 200 s to verify or times out, from the SDV and SVCOMP benchmarks to run our experiments. We set the timeout for each verification task to 2 h for both CORRAL and LEGION. For all verification tasks, We use an unrolling length of 3 as advised in the benchmarks [31] and used in other works [11].

As LEGION$^+$ uses twice the computational resources compared to CORRAL and LEGION, we halve its time budget to 1 h to make a fair comparison. We also report the performance of LEGION$^+$ with a 2 h time budget (it can be seen as the *virtual best* of CORRAL and LEGION).

The experiments were performed on a machine with AMD EPYC 7452 processor (48 cores) and 384 GB of RAM. Both CORRAL and LEGION uses *Z3* [15] as the underlying SMT solver. We have used the default setting of a fixed random seed for *Z3* for all our experiments after verifying the fact that the choice of random seed does not have any significant impact on our results.

5.1   Corral Versus Legion

Figure 6 depicts the number of solved instances within the time budget by CORRAL and LEGION. In Fig. 6, a point (x, y) denotes the number of instances x,
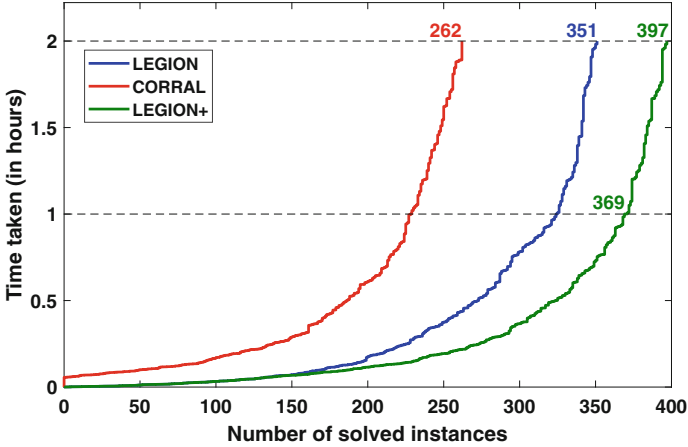
Fig. 6. Number of instances solved within time (in hours) for CORRAL vs LEGION vs LEGION⁺.

Table 3. Total time taken by each verifier to solve instances

Verifier	Solved instances	Total time taken
CORRAL	262	109 h
LEGION	351	112 h
LEGION⁺	369	71 h

each of which was solved within time y. As we can observe, CORRAL is able to solve 262 out of 727 instances (36%) with a time budget of 2 h per instance, whereas LEGION solves 351 instances (48%) with the same time budget. Both of them fail to solve 330 instances (45%). Out of the 397 instances (55%) that are solved by either CORRAL or LEGION, 46 instances (12%) are solved exclusively by CORRAL, whereas 135 instances (34%) are solved exclusively by LEGION.

The scatter plot of verification times across LEGION and CORRAL is shown in Fig. 7. The spread in the scatter plots demonstrate that these two tools complement each other—the benchmarks on which CORRAL struggles are sometimes handled well by LEGION, and vice-versa. Picking the best of two verifiers solves a total of 397 out of 727 instances (55%). This motivated the design of LEGION⁺.

5.2 Performance of Legion⁺

As LEGION⁺ utilizes parallelism, in order to make a fair comparison we halve the time budget for LEGION⁺ on each verification instance to 1 h. This means that LEGION⁺ runs both the tools CORRAL and LEGION in parallel but with a time budget of 1 h each.

Figure 6 shows that the portfolio verifier LEGION⁺ solves 369 out of 727 instances (51%) with a 1 h time budget, whereas CORRAL solves only 262
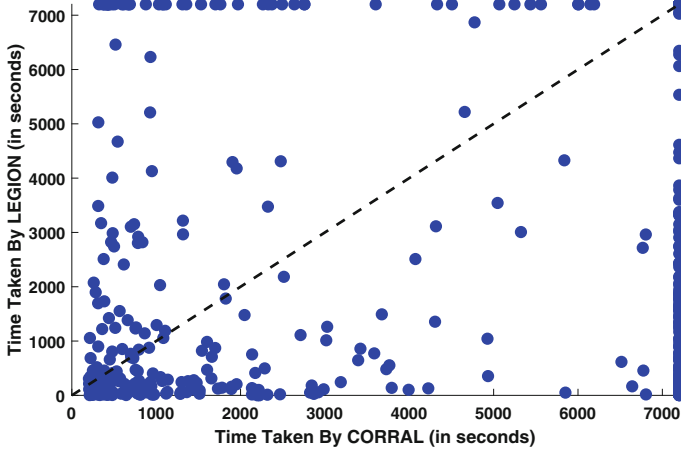
Fig. 7. Scatter plot of verification time of CORRAL vs LEGION.

instances (36%) with a total time budget of 2 h. There are only 14 instances that CORRAL solves but LEGION⁺ is unable to solve. Similarly, there are only 17 instances that LEGION solves but LEGION⁺ is unable to solve.

With a 2 h timeout, LEGION⁺ solves 397 instances in total (55%). This is essentially the *virtual best* of CORRAL and LEGION with a 2 h timeout.

Figure 8 shows the total time taken (in hours) by CORRAL, LEGION and LEGION⁺ to verify the instances that were solved by all three of them (total 213 instances). LEGION⁺ is 1.9× faster than LEGION and 2.9× faster than CORRAL.

Across the benchmarks that each of the tools solve individually, CORRAL takes 109 h to solve 262 benchmarks, LEGION takes 112 h to solve 351 benchmarks, whereas LEGION⁺ solves 369 benchmarks within only 71 h (see Table 3).

Note that the benchmarks used in our study are those on which CORRAL took greater than 200 s. On the rest of the benchmarks, clearly LEGION⁺ will perform at least as well as CORRAL. We chose to leave them out to ensure that the experiments run in a reasonable time: there were roughly 14000 of these easy cases. It allowed us to focus on benchmarks where speedup was important.

6   Related Work

The high-level idea of using proof-guided abstractions has been long known [3, 30]. Proofs of unsatisfiability have been used to derive abstractions for unbounded model checking in the context of microprocessor verification [30]. Amla et al. have also demonstrated that counterexample based abstraction is complementary to proof based abstraction and they can be combined in a judicious manner to reap the benefits of both the techniques for hardware verification tasks [3]. However, program verification has mostly been dominated by counterexample-guided abstraction refinement (CEGAR) based strategies. Of
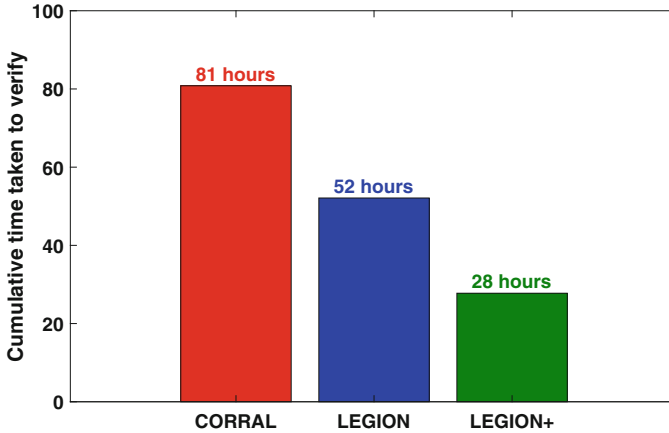
Fig. 8. Cumulative time taken (in hours) to verify 213 instances that were solved by all three verifiers.

the few proposals that use proof-guided underapproximation widening strategies, most of them focus on verification of multi-threaded programs [18,35]. These techniques perform underapproximation on the number of thread interleavings allowed, while eagerly inlining all procedures. One technique [18] constrains the number of interleavings to certain bounds, while the other [35] uses dynamically inferred invariants for constructing (potential) underapproximations on interleavings. Note that, these techniques are orthogonal to our approach. Eager inlining is not feasible for our benchmarks, which is precisely the problem that we address. Our proposal shows that proof-guided widening strategies can be effectively employed for verifying large sequential programs. Proof of unsatisfiability from underapproximated models have also been utilized to narrow down the search space for overapproximation refinement in order to decide finite precision bit vector arithmetic with arbitrary bit vector operations [9]. The underapproximation is done on the bit vector variables of a propositional logic formula where some of the bit vector variables are encoded with fewer boolean variables than their width.

Other than using proofs to guide widening heuristics, proof artifacts, like interpolants, have been used to construct annotations [1,2,27–29] that can be useful in constraining future search. Such techniques are orthogonal to underapproximation widening based techniques. However, they can be useful for LEGION and we plan to investigate them in the future.

Underapproximation widening has also been used in program synthesis [37,39,40]. Instead of unleashing the search for the program on the whole search space, such techniques search for the desired program in an underapproximated search space. While prior approaches [37] used a pre-defined widening sequence, later approaches [39,40] use proofs of unsatisfiability to guide the widening sequence. Similar techniques have also been used in the synthe-

sis of boolean functions [16,17]. Manthan [16,17] constructs an initial guess of the boolean function by sampling the specification and constructing a decision-tree classifier from the resulting data. It, then, uses a proof-guided technique to "repair" the learnt model into a desired function.

There have also been applications of the maximal satisfiable set (MAXSAT) on an unsatisfiable formula for program debugging. BugAssist [19] attempts to infer the set of suspicious locations using a MAXSAT formulation over an failing program trace and the specifications. Bavishi et al. [6] extend the formulation to provide a ranking over the suspicious locations such that the locations higher up in the rankings are less likely to cause regressions.

Another line of work is to use fuzzers to sample concrete instances and gradually build approximations of program behavior for the purpose of deductive verification [22] and symbolic execution [34]. However, such approaches use test instances and do not apply a proof-guided strategy.

LEGION is inspired by many of the above algorithms and, there is potential of incorporating more of these ideas in LEGION in the future.

7 Conclusion

Bounded model checking approaches for program verification predominantly focuses on CEGAR based strategies. In this work, we propose a proof-guided underapproximation widening strategy which behaves in a complementary manner to the CEGAR technique. The complementary nature allows us to build a portfolio strategy that takes advantage of both proof-guided underapproximation widening and CEGAR to deliver a significant speed up in verification time over both.

Our current approach only looks at the predicates corresponding to the call-sites to figure out which are most relevant to the proof of unsatisfiability of the underapproximated model. In the future, we aim to extract additional information from the unsat core which would allow us to explore more involved widening strategies. Furthermore, combining the underapproximation techniques that work on the domain of thread interleavings to deal with a large space of sequential behaviors (via lots of procedures) and concurrent behaviors (via lots of interleavings) would be another interesting direction to explore. We also believe that underapproximation widening may yield improvement performance on our distributed bounded model checker, HYDRA [11,12]. Another interesting direction that we want to pursue is on combining bounded model checking algorithms (both overapproximation refinement and underapproximation widening) with dynamic analysis [5,13,38] and statistical testing [10,32] based approaches.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: a framework for abstraction-and interpolation-based software verification. In: CAV (2012)
2. Alt, L., et al.: Hifrog: Smt-based function summarization for software verification. In: TACAS (2017)
3. Amla, N., McMillan, K.L.: A hybrid of counterexample-based and proof-based abstraction. In: FMCAD (2004)
4. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In: International Conference on Integrated Formal Methods (2004)
5. Ball, T., Larus, J.R.: Efficient path profiling. In: Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (1996)
6. Bavishi, R., Pandey, A., Roy, S.: To be precise: regression aware debugging. ACM SIGPLAN Notices (2016)
7. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, 6–11 April, 2019, Proceedings, Part III (2019)
8. Boogie: An intermediate verification language. https://boogie-docs.readthedocs.io/en/latest/. Accessed June 2022
9. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: TACAS (2007)
10. Chatterjee, P., Chatterjee, A., Campos, J., Abreu, R., Roy, S.: Diagnosing software faults using multiverse analysis. In: IJCAI (2020). https://doi.org/10.24963/ijcai.2020/226
11. Chatterjee, P., Roy, S., Diep, B.P., Lal, A.: Distributed bounded model checking. In: FMCAD (2020)
12. Chatterjee, P., Roy, S., Diep, B.P., Lal, A.: Distributed bounded model checking. Formal Methods in System Design (2022). https://doi.org/10.1007/s10703-021-00385-1
13. Chouhan, R., Roy, S., Baswana, S.: Pertinent path profiling: tracking interactions among relevant statements. In: CGO (2013). https://doi.org/10.1109/CGO.2013.6494983
14. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV (2000)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: TACAS (2008)
16. Golia, P., Roy, S., Meel, K.S.: Manthan: a data-driven approach for boolean function synthesis. In: CAV (2020)
17. Golia, P., Roy, S., Slivovsky, F., Meel, K.S.: Engineering an efficient boolean functional synthesis engine. In: ICCAD (2021)
18. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided under-approximation-widening for multi-process systems. In: POPL (2005)
19. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. ACM SIGPLAN Notices (2011)
20. Kroening, D., Tautschnig, M.: Cbmc-c bounded model checker. In: TACAS (2014)
21. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL (2008)
22. Lahiri, S., Roy, S.: Almost correct invariants: synthesizing inductive invariants by fuzzing proofs. In: ISSTA (2022)

23. Lal, A., Qadeer, S.: Reachability modulo theories. In: Reachability Problems - 7th International Workshop, RP (2013)
24. Lal, A., Qadeer, S.: Powering the static driver verifier using Corral. In: FSE (2014)
25. Lal, A., Qadeer, S.: DAG inlining: a decision procedure for reachability-modulo-theories in hierarchical programs. ACM SIGPLAN Notices (2015)
26. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: CAV (2012)
27. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV (2003)
28. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV (2006)
29. McMillan, K.L.: Lazy annotation revisited. In: CAV (2014)
30. McMillan, K.L., Amla, N.: Automatic abstraction without counterexamples. In: TACAS (2003)
31. Microsoft: Static Driver Verifier Benchmarks. https://github.com/boogie-org/sdvbench
32. Modi, V., Roy, S., Aggarwal, S.K.: Exploring program phases for statistical bug localization. In: PASTE (2013). https://doi.org/10.1145/2462029.2462034
33. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: Esbmc 1.22. In: TACAS (2014)
34. Pandey, A., Kotcharlakota, P.R.G., Roy, S.: Deferred concretization in symbolic execution via fuzzing. In: ISSTA (2019). https://doi.org/10.1145/3293882.3330554
35. Prabhu, S., Schrammel, P., Srivas, M., Tautschnig, M., Yeolekar, A.: Concurrent program verification with invariant-guided underapproximation. In: ATVA (2017)
36. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: CAV (2014)
37. Roy, S.: From concrete examples to heap manipulating programs. In: SAS (2013). https://doi.org/10.1007/978-3-642-38856-9_9
38. Roy, S., Srikant, Y.N.: Profiling k-iteration paths: A generalization of the ball-larus profiling algorithm. In: CGO (2009). https://doi.org/10.1109/CGO.2009.11
39. Verma, A., Kalita, P.K., Pandey, A., Roy, S.: Interactive debugging of concurrent programs under relaxed memory models. In: CGO (2020)
40. Verma, S., Roy, S.: Synergistic debug-repair of heap manipulations. In: ESEC/FSE (2017)