TRACER: A Symbolic Execution Tool for Verification

JOXAN JAFFAR[1], VIJAYARAGHAVAN MURALI[1], JORGE A. NAVAS[2], AND ANDREW E. SANTOSA[3]

[1]National University of Singapore
[2]The University of Melbourne
[3]University of Sydney

Abstract. We present TRACER, a verifier for *finite-state* safety properties of sequential C programs. It is based on symbolic execution (SE) and its unique features are in how it makes SE finite in presence of unbounded loops and its uses of interpolants to tackle the *path-explosion* problem.

1 Introduction

Recently *symbolic execution (*SE*)* [15] has been successfully proven to be an alternative to CEGAR for program verification offering the following benefits among others [12, 18]: (1) it does not explore infeasible paths avoiding expensive refinements, (2) it avoids expensive *predicate image* computations (e.g., *Cartesian* and *Boolean* abstractions [2]), and (3) it can recover from *too-specific* abstractions as opposed to monotonic refinement schemes often used. Unfortunately, it posses its own challenges: (C1) exponential number of paths, and (C2) infinite-length paths in presence of unbounded loops.

We present TRACER, a verification tool based on SE for *finite-state* safety properties of sequential C programs. Informally, TRACER attempts at building a finite symbolic execution tree which overapproximates the set of all concrete reachable states. If the error location cannot be reached from any symbolic path then the program is reported as safe. Otherwise, either the program may contain a bug (TRACER reports a false alarm only if the theorem prover fails to prove a valid claim) or it may not terminate.The most innovative features of TRACER stem from how it tackles (C1) and (C2).

In this paper, we describe the main ideas behind TRACER and its implementation as well as our experience in running real benchmarks.

1.1 State-Of-The-Art Interpolation-Based Verification Tools

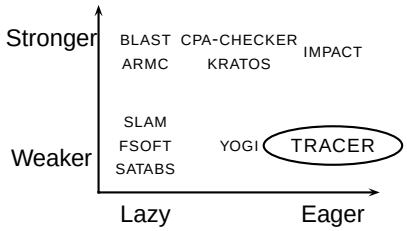

Fig. 1. Some state-of-the-art verifiers

Fig. 1 depicts one possible view of current verification tools based on two dimensions: *laziness* and *interpolation strength*. *Lazy* means that the tool starts from a coarsely abstracted model and then refines it while *eager* is its dual, starting with the concrete model and then removing irrelevant facts. CEGAR-based tools [1, 4, 7, 10, 21] are the best examples of lazy approaches while SE-based tools [12, 18] are for eager methods. Special mention is required for hybrid approaches such as YOGI [20], CPA-CHECKER [3], and KRATOS [5]. YOGI computes weakest preconditions from symbolic execution of paths as a cheap refinement for CEGAR. One disadvantage is that

it cannot recover from too-specific refinements (see program *diamond* in [18]). CPACHECKER and KRATOS encode loop-free fragments into a Boolean formula that can then be subjected to a SMT solver in order to avoid refinements if no loops are involved.

On the other hand, the performance of interpolation-based verifiers depends on the logical strength of the interpolants[1]. In lazy approaches, a weak interpolant may contain spurious errors and cause refinements too often. Stronger interpolants may delay convergence to a fixed point. In eager approaches, weaker interpolants may be better (e.g., for loop-free fragments) than stronger ones since they allow removing more irrelevant facts from the concrete model.

TRACER performs SE and computes efficient approximated *weakest preconditions* as interpolants. To the best of our knowledge, TRACER is the first publicly available (http://www.clip.dia.fi.upm.es/~jorge/tracer) verifier with these characteristics.

2 How TRACER Works

Essentially, TRACER implements classical symbolic execution [15] with some novel features that we will outline along this section. It takes symbolic inputs rather than actual data and executes the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated in a first-order logic (FOL) formula called *path condition (PC)*. Whenever code of the form if(C) then S1 else S2 is reached the execution forks the current symbolic state and updates path conditions along both the paths $PC_1 \equiv PC \wedge C$ and $PC_2 \equiv PC \wedge \neg\, C$, respectively. Then, it checks if either PC_1 or PC_2 is unsatisfiable. If unsatisfiable, then the path is *infeasible* and hence, the execution can halt and backtrack to the last choice point. Otherwise, the execution follows the path. Note that both PC_1 and PC_2 can be satisfiable simultaneously but both cannot be unsatisfiable. The verification problem consists of building a *finite* symbolic execution tree that contains at least all concrete reachable states and proving that for every symbolic path the error location is unreachable.

2.1 Cache-based Algorithm with Weakest Preconditions as Interpolants

The first key aspect of TRACER, originally proposed in [13] and used later in [12, 18], is the avoidance of full enumeration of symbolic paths by *learning* from infeasible paths by computing *interpolants* [8], in a similar spirit to the *nogood* learning in SAT. Preliminary versions of TRACER [12, 13] computed interpolants based on *strongest postconditions (sp)*. Given two formulas A (symbolic path) and B (last guard where infeasibilility is detected) such that $A \wedge B$ is unsat, an interpolant was obtained by $\exists \overline{x} \cdot A$ where $\overline{x}$ are A-local (i.e., variables occurring only in A) variables. However unlike CEGAR, TRACER starts from the concrete model of the program and then, deletes irrelevant facts. Therefore, the weaker the interpolant is the more likely it is for TRACER to avoid exploring other "similar" symbolic paths. This is the motivation behind an interpolation method based on *weakest preconditions (wp)*. The contrived example in Fig. 2 shows the need for wp as well as the essence of our approach to mitigate the "path-explosion" problem. The error location $\langle 8 \rangle$ is unreachable since the variable s can be at most 4. Note also the program has four symbolic paths. Assume the first path is $\langle 0 \rangle$-$\langle 1 \rangle$-$\langle 2 \rangle$-$\langle 4 \rangle$-$\langle 5 \rangle$-$\langle 7 \rangle$-$\langle 8 \rangle$

[1] Given formulas A and B such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* [8] I satisfies: (1) $A \models I$, (2) $I \wedge B$ is unsatisfiable, and (3) its variables are common to A and B. We say an interpolant I is stronger (weaker) than I' if $I \models I'$ ($I' \models I$).

⟨0⟩ s=0;
⟨1⟩ if(*) ⟨2⟩ s++; else ⟨3⟩ s+=2;
⟨4⟩ if(*) ⟨5⟩ s++; else ⟨6⟩ s+=2;
⟨7⟩if(s > 10) ⟨8⟩ error();
⟨9⟩

Fig. 2. Safe Code

which is infeasible since the formula (after renaming) $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 1 \wedge s_2 > 10$ is unsatisfiable. Then, TRACER using a reasonable interpolation method (e.g., FOCI [17], CLP-PROVER [22], and MATHSAT [6]) infers the interpolants $s_0 \leq 0$, $s_1 \leq 1$, and $s_2 \leq 2$ at program locations $\langle 1 \rangle$, $\langle 4 \rangle$, and $\langle 7 \rangle$, and memoizes them in a cache[2]. Unfortunately, those interpolants are not weak enough to avoid or *subsume*[3] the exploration of the other symbolic paths. That is, $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2$ (from $\langle 0 \rangle$-$\langle 1 \rangle$-$\langle 2 \rangle$-$\langle 4 \rangle$-$\langle 6 \rangle$-$\langle 7 \rangle$) $\not\models s_2 \leq 2$ and $s_0 = 0 \wedge s_1 = s_0 + 2$ (from $\langle 0 \rangle$-$\langle 1 \rangle$-$\langle 3 \rangle$-$\langle 4 \rangle$) $\not\models s_1 \leq 1$. . However, by weakest preconditions we can easily obtain from the first path the interpolants $s_0 \leq 8$, $s_1 \leq 9$, and $s_2 \leq 10$, which clearly subsume the other paths.

For efficiency, our algorithm under-approximates the weakest precondition by a mix of existential quantifier elimination, unsatisfiable cores, and some heuristics. Whenever an infeasible path is detected we compute $\neg\ (\exists \overline{y} \cdot G)$, the *postcondition* that we want to map into a *precondition*, where G is the guard where the infeasibility is detected and $\overline{y}$ are G-local variables. The two main rules for propagating wp's are:

(A) $wp(x := e, Q) = Q[e/x]$
(B) $wp(\text{if}(C)\ S1 \text{ else } S2, Q) = (C \Rightarrow wp(S1, Q)) \wedge (\neg\ C \Rightarrow wp(S2, Q))$

Rule (A) replaces all occurrences of x with e in the formula Q. Thus, the challenge is how to produce conjunctive formulas from rule (B) as weak as possible to increase the likelihood of subsumption. During the forward SE when an infeasible path is detected we discard *irrelevant* guards by using the concept of *unsatisfiable cores (UC)*[4] to avoid growing the wp formula unnecessarily. For instance, the formula $C \Rightarrow wp(S1, Q)$ can be replaced with $wp(S1, Q)$ if $C \notin \mathcal{C}$ where $\mathcal{C}$ is a (not necessarily minimal) UC. Otherwise, we underapproximate $C \Rightarrow wp(S1, Q)$ as follows. Let $d_1 \vee \ldots \vee d_n$ be $\neg\ wp(S1, Q)$ then $\forall\ d_i\ (1 \leq i \leq n)\ \cdot\ (\bigwedge_i (\neg\ (\exists\ \overline{x'} \cdot (C \wedge d_i))))$, where we use existential quantifier elimination to remove the post-state variables $\overline{x'}$. A very effective heuristic if the resulting formula is disjunctive is to delete those conjuncts that are not implied by $\mathcal{C}$ because they are more likely to be irrelevant to the infeasibility reason.

Remarks. TRACER separately discovers loop invariants during the forward symbolic execution. This ensures that any wp generated is necessarily *entailed* by the loop invariant. Thus, loops are not an issue in our wp computations. Finally, weakest preconditions may fail to generalize with some loops as Jhala et al. pointed out in [14] (Sec.1, page 2). Then, TRACER can compute other interpolants or be fed with inductive invariants from external tools (see Sec. 3).

2.2 Path Invariants via Widening with Counterexample-Guided Loop Unrolling

With unbounded loops the only hope to produce a proof is *abstraction*. In a nutshell, upon encountering a cycle TRACER computes the *strongest* possible loop invariants $\overline{\Psi}$

[2] Those interpolants can be also obtained by computing strongest postconditions and rewriting $x = y$ as $x \geq y \wedge y \geq x$.

[3] A symbolic state σ is *subsumed* or *covered* by another symbolic state σ' if they refer to same location and the set of states represented by σ is a subset of those represented by σ'. Alternatively, if σ and σ' are seen as formulas then σ is subsumed by σ' if $\sigma \models \sigma'$.

[4] Given a constraint set S whose conjunction is unsatisfiable, an *unsatisfiable core (UC)* S' is any unsatisfiable subset of S. An UC S' is *minimal* if any strict subset of S' is satisfiable.

by using widening techniques in order to make the SE finite. If a spurious abstract error is found then a *refinement phase* (similar to CEGAR) discovers an interpolant I that rules the spurious error out. After restart, TRACER strengthens $\overline{\Psi}$ by conjoining it with I and the symbolic execution checks *path by path* if the new strengthened formula is a loop invariant. If this test fails for a given path π, then TRACER will unroll only π at least one more iteration and continue with the process. Notice that the generation of invariants is *dynamic* in the sense that loop unrolls will expose new constraints producing new invariant candidates. For lack of space, we refer readers to [12].

```
⟨0⟩lock=0; new=old+1;
⟨1⟩while(new ≠ old) {
⟨2⟩   lock=1; old=new;
⟨3⟩   if(*){ ⟨4⟩ lock=0;new++;}
⟨5⟩}
⟨6⟩if(lock == 0) ⟨7⟩ error();
⟨8⟩
```

Fig. 3. Excerpt from a NT Windows driver

Remarks. Other SE-based tools (e.g., IMPACT [18]) may not terminate with our example in Fig. 3 (see [12] Sec.1, Ex.1) while TRACER converges in two iterations. On the other hand, our path invariant technique via widening is closely related to the widening "up to" S (∇^S) used in [9], where S contains the constraints inferred by the refinement phase. However, they use it to enhance CEGAR while SE poses different challenges (see [12] Sec.1, Ex.3). Finally, abstraction in TRACER also differs from CEGAR in a fundamental way. TRACER attempts at inferring the *strongest* loop invariants modulo the limitations of widening techniques while CEGAR, as well as hybrid tools like CPACHECKER and KRATOS, will often propagate coarser abstractions. Although stronger abstractions may be more expensive they may converge faster in presence of loops (see [12] Sec.1, Ex.4).

3 Usage and Implementation

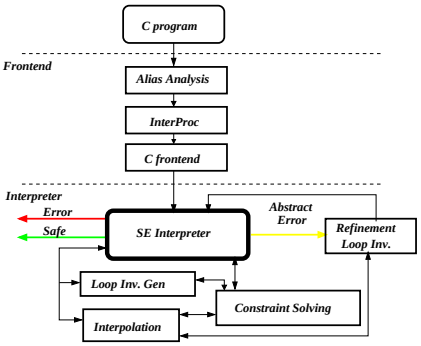

Fig. 4. Implementation of TRACER

Input. TRACER takes as input a C program with assertions of the form _TRACER_abort(*Cond*), where *Cond* is a quantifier-free FOL formula. Then, each path that encounters the assertion tests whether *Cond* holds or not. If yes, the symbolic execution has reached an error node and thus, it reports the error and aborts if the error is real, or refines if spurious. Otherwise, the symbolic execution continues normally.

Output. If the symbolic execution terminates and all _TRACER_abort assertions failed then the program is reported as safe and the corresponding symbolic execution tree is displayed as the proof object. If the program is unsafe then a counterexample is shown.

Implementation. Fig. 4 outlines the implementation of TRACER. It is divided into two components: a *frontend* and an *interpreter*. First, a C-frontend based on CIL [19] translates the input program into a constraint-based logic program. Both pointers and arrays are modeled using the theory of arrays. An alias analysis is used in order to yield sound and finer grained independent partitions (i.e., *separation*) as well as infer which scalars' addresses may have been taken. Optionally, it uses INTERPROC [16] (option `-loop-inv`) to provide loop invariants. The second component is an interpreter which

symbolically executes the constraint-based logic program and it aims at demonstrating that error locations are unreachable. This interpreter is implemented in a *Constraint Logic Programming* (*CLP*) system called CLP($\mathcal{R}$) [11], taking advantage of intrinsic features of CLP such as *backtracking*, efficient *existential quantifier elimination*, and *incremental* constraint solving. Its main sub-components are as follows:

- *Constraint Solving* relies on the CLP($\mathcal{R}$) solver to reason fast over linear arithmetic over reals ($\mathcal{R}$) without expensive calls to general-purpose theorem provers. A decision procedure for arrays (option `-mccarthy`) has been also implemented.
- *Interpolation* implements two methods with different logical strength. The first method uses *strongest postconditions* [12, 13] (`-intp sp`). The second computes *weakest preconditions* (`-intp wp`) as described in Sec. 2.1 but current implementation only allows reasoning in linear arithmetic over reals. TRACER also provides interfaces to other interpolation methods such as CLP-PROVER (`-intp clp`).
- *Loop Invariant Refinement*. Similar to CEGAR the effectiveness of the refinement phase usually relies on heuristics (`-h` option). But unlike CEGAR tools, SE only performs abstractions at loop headers. Thus, given a path that reaches an error location TRACER only needs to visit those abstraction points in the path and checks if one of the them caused the reachability of the error. If yes, it uses interpolation to choose which constraints can rule out the error. Otherwise, the error must be real.
- *Loop Invariant Generation*. Whenever a loop header is encountered TRACER records a set of *loop invariant* candidates. Then, if a cycle π is found TRACER performs a widening of the state at the loop header $c \nabla c'$ where c' is the candidate c after the execution of π. Current implementation of the ∇ operator follows $c \nabla c' \triangleq c$ if $c' \models c$ otherwise $\top$. Very importantly, if ∇ attempts at abstracting a constraint needed to exclude an error then it fails and the path is unrolled at least one more iteration. Although our experiments demonstrate that our implementation is quite fast and effective, it is clearly *incomplete* (in the sense that it may cause non-termination) for several reasons. First, the generation of candidates considers only constraints propagated by SE although TRACER allows enriching this set with inductive invariants provided by INTERPROC. Second, the implementation of ∇ is imprecise. Third, ∇ is applied to each candidate *individually*. By applying ∇ to *all candidate subsets* we could produce richer invariants, although this process would be exponential.

4 Experience with Benchmarks

We ran TRACER on the ntdrivers-simplified and ssh-simplified benchmark suites from the SV-COMP[5]. Due to lack of space, we present in Fig. 5 only data that focuses on the two key features of TRACER: use of weakest preconditions as interpolants and how it handles unbounded loops computing strongest loop invariants. The experiments were run on Intel 2.33Ghz 3.2GB. Columns 2 and 3 compare the number of states of the symbolic execution tree (S) explored by TRACER using SP and WP, columns 4 and 5 compare the total verification time T(s) in seconds (excluding the C-frontend). Columns 6 and 7 compare the number of loop invariant refinements made (R), and the last two columns compare the time of computing loop invariants (InvG(s)), as outlined in Sec. 2.2. If

[5] *"1st Competition on Software Verification"* (sv-comp.sosy-lab.org). We also run statemate, from *WCET* (*Worst-Case Execution Time*) community, to prove upper bounds.

marked with ∞, TRACER did not finish within 900 seconds. WP often pays off with greater gains in programs where TRACER refines heavily, mainly because loop unrolls are expensive for SE and hence, more often subsumption is vital. For statemate, WP dramatically decreases the search space. However in diskperf and kbfiltr, WP is slower even though the size reduction due to the overhead of computing wp's. As expected, InvG increases if loop unrolls are more frequent and more loop headers are explored.

	S		T(s)		R		InvG(s)	
Program	SP	WP	SP	WP	SP	WP	SP	WP
cdaudio.i.cil.c	16547	13737	266	201	0	0	0.8	0.6
diskperf.i.cil.c	15715	9181	117	121	0	0	0.1	0.1
floppy.i.cil.c	16588	5421	212	91	6	2	1.5	0.5
kbfiltr.i.cil.c	1854	1484	3	6	0	0	0	0
s3_clnt_1.cil.c	19956	2656	156	11	27	8	57	2.7
s3_clnt_2.cil.c	∞	13018	∞	74	∞	58	∞	21
s3_clnt_3.cil.c	∞	10598	∞	58	∞	45	∞	15
s3_clnt_4.cil.c	35392	4130	275	17	63	17	92	4.2
s3_srvr_2.cil.c	39174	14066	241	83	49	42	56	18
s3_srvr_3.cil.c	9340	4081	48	15	6	6	12	3
s3_srvr_4.cil.c	35454	14065	346	102	33	33	147	44
s3_srvr_13.cil.c	∞	48323	∞	826	∞	127	∞	401
statemate	18995	1021	119	11	0	0	0.1	0

Fig. 5. TRACER: SP (`-intp sp`) vs WP (`-intp wp`)

Remarks. For a comparison with other SE methods and BLAST [4] we refer to [12] and our unpublished report available at arxiv.org/abs/1103.2027, respectively. For our experiments we turn off INTER-PROC because its inductive invariants did not make any impact on the convergence of TRACER. We were able to run CLP-PROVER(`-intp clp`) for statemate and the size of the symbolic tree was identical to using `-intp sp`. For the others, we could not run CLP-PROVER because its available implementation can only annotate one location per call, rather than all locations along the path per call, as commonly done, degrading the performance quickly.

References

1. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM. In *IFM'2004*.
2. T. Ball et al. Relative Completeness of Abstraction Refinement for Software Model Checking *TACAS'02*.
3. D. Beyer et al. Software Model Checking via Large-Block Encoding. In *FMCAD'09*.
4. D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. BLAST. *Int. J. STTT*, 2007.
5. A. Cimatti et al. Kratos - A Software Model Checker for SystemC. In *CAV'11*.
6. A. Cimatti et al. Efficient Interpolant Generation in SMT. In *TACAS'08*.
7. E. Clarke et al. Satabs: Sat-based Predicate Abstraction for ansi-C. In *TACAS'05*.
8. W. Craig. Three Uses of Herbrand-Gentzen Theorem in Relating Model and Proof Theory. *JSC'55*.
9. B. S. Gulavani et al. Refining Abstract Interpretations. *Inf. Process. Lett.*, 2010.
10. F. Ivancic et al. F-Soft: Software Verification Platform. In *CAV'05*.
11. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) System. *TOPLAS*, 1992.
12. J. Jaffar et al. Unbounded Symbolic Execution for Program Verification. In *RV'11*.
13. J. Jaffar, A. E. Santosa, and R. Voicu. An Interpolation Method for CLP Traversal. In CP'09.
14. R. Jhala et al. A Practical and Complete Approach to Predicate Refinement. In *TACAS'06*.
15. J . King. Symbolic Execution and Program Testing. *Com. ACM' 76*.
16. G. Lalire, M. Argoud, and B. Jeannet. The Interproc Analyzer http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc.
17. K. L. McMillan. An interpolating theorem prover. *TCS*, 2005.
18. K. L. McMillan. Lazy Annotation for Program Testing and Verification. In CAV'10.
19. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL. In *CC'02*.
20. A.V. Nori, S.K. Rajamani, S. Tetali, A.V. Thakur. The Yogi Project. In *TACAS'09*.
21. A. Podelski and A. Rybalchenko. ARMC. In *PADL'07*.
22. A. Rybalchenko and V. Sofronie. Constraint Solving for Interpolation. In *VMCAI'07*.

This appendix contains more detailed information about how our tool TRACER *verifies the two program fragments shown in Fig. 2 and Fig. 3. It is included to help the reviewing process and would not be part of the final version.*

A Example of Fig. 2

In this section, we illustrate how TRACER verifies the snippet shown in Fig. 2 using two of the available interpolation methods in the tool. Fig. 6(a)-(c) show the details about how the symbolic execution tree is built by TRACER if state-of-the-art interpolation methods (e.g. FOCI, CLP-PROVER, or MATHSAT) are used. This can be achieved by running TRACER with options `-intp clp` or `-intp sp -convert-eq-to-ineq y`. On the other hand, Fig. 7(a)-(c) show the symbolic execution tree using our weakest preconditions method by running with option `-intp wp`. As a feature, TRACER displays the below symbolic execution trees in `dot` format.

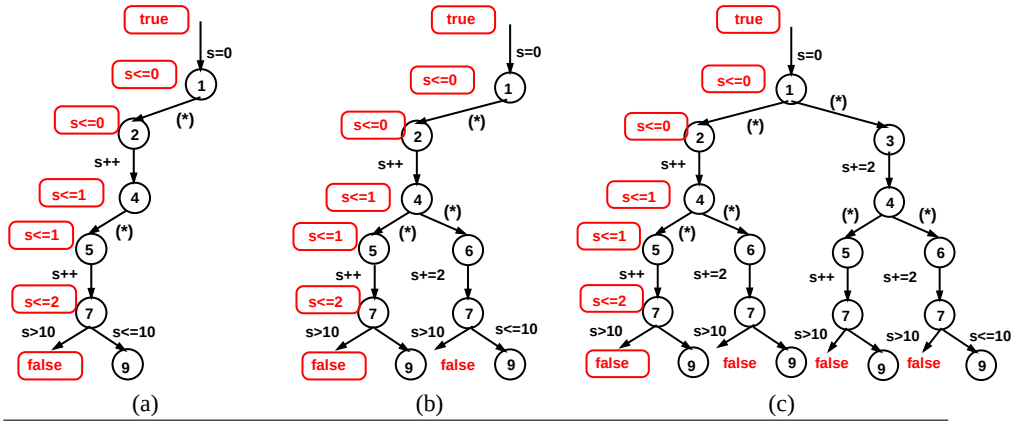

Fig. 6. Symbolic execution trees running TRACER with `-intp clp` or `-intp sp -convert-eq-to-ineq y` options.

Fig. 6(a) shows the first symbolic path explored by TRACER which is indeed infeasible. The symbol `(*)` means that the evaluation of the guard can be *true* or *false*. After renaming we obtain $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 1 \wedge s_2 > 10$ which is clearly unsatisfiable. State-of-the-art interpolation techniques will annotate every location with its corresponding interpolant: $\iota_1 : s_0 \leq 0$, $\iota_2 : s_0 \leq 0$, $\iota_4 : s_1 \leq 1$, $\iota_5 : s_1 \leq 1$, and $\iota_7 : s_2 \leq 2$ where ι_k refers to the interpolant at location k. In all figures, interpolants are enclosed in (red) boxes. Fig. 6(b) shows also the second symbolic path. Note that at location 7 of the second path TRACER tests if the current symbolic state $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2$ is subsumed by $\iota_7 : s_2 \leq 2$, the interpolant at location 7. However, this tests fails since $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2 \not\models s_2 \leq 2$. Similarly, TRACER attempts again at location 4 of the third path in Fig. 6(c) if the new symbolic path can be subsumed by a previous explored path. Here, it tests if $s_0 = 0 \wedge s_1 = s_0 + 2$

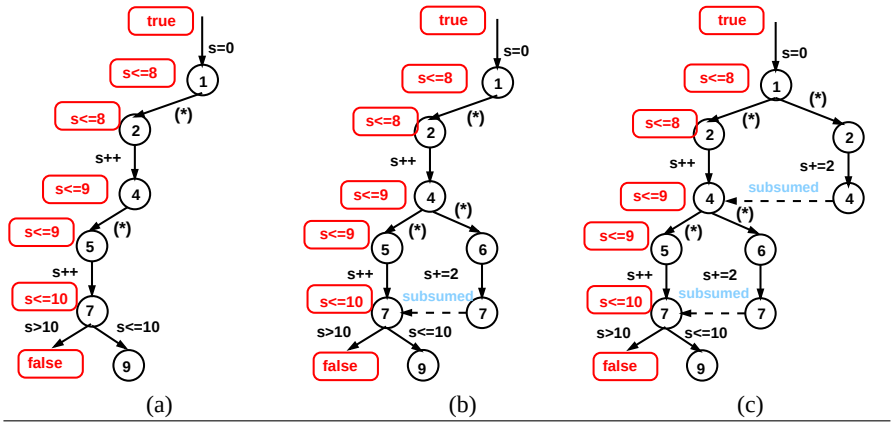

Fig. 7. Symbolic execution tree running TRACER with weakest preconditions (`-intp wp`)

implies $\iota_4 : s_1 \leq 1$ but again it fails. Finally, TRACER is able to prove the program is safe but notice that the symbolic execution tree built is exponential on the number of program branches.

Next, Fig. 7(a) shows the same first symbolic path explored by TRACER but annotated this time with the interpolants obtained by weakest preconditions: $\iota_1 : s_0 \leq 8$, $\iota_2 : s_0 \leq 8$, $\iota_4 : s_1 \leq 9$, $\iota_5 : s_1 \leq 9$, and $\iota_7 : s_2 \leq 10$.

For this example, the weakest precondition computations are notably simplified since the guards are not considered since they are clearly irrelevant for the infeasibility of the path. Therefore, only rule (A) from Sec.2.1 is triggered. For instance, $\iota_7 : s_2 \leq 10$ is obtained by $\neg\,(\exists \mathcal{V} \setminus \{s_2\} \cdot s_2 > 10) \equiv s_2 \leq 10$ where $\mathcal{V}$ is the set of all program variables (including renamed variables), and $\iota_6 : s_1 \leq 9$ is obtained by $wp(s_2 = s_1 + 1,\ s_2 \leq 10) = s_1 \leq 9$. Fig. 7(b) shows the second symbolic path but note that the path can be now subsumed at location 7 since the symbolic state $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 2 \models s_2 \leq 10$. Dashed edges represent subsumed paths and are labelled with "subsumed". Finally, Fig. 7(c) illustrates how the third symbolic path can be also subsumed at location 4 since $s_0 = 0 \wedge s_1 = s_0 + 2 \models s_1 \leq 9$.

B Example of Fig. 3

We provide here more details about how TRACER handles unbounded loops through the classical example in Fig 3. We still refer readers to [12] for technical discussion.

TRACER executes the program until a cycle is found and checks whether a certain set of loop candidates, created by propagation of SE, holds after the execution of the cycle. In our example, we obtain the symbolic path (after renaming) $\pi_1 \equiv lock_0 = 0 \wedge new_0 = old_0 + 1 \wedge (new_0 \neq old_0) \wedge lock_1 = 1 \wedge old_1 = new_0$ from executing the `else` branch, shown in Fig. 8(a). Assume a trivial widening operator ∇ defined as $c \nabla c' \triangleq c$ if $c' \models c$ otherwise $\top$ (i.e., *true*), where c and c' are the constraint versions before and after the execution of the cycle corresponding to one candidate. Then, widening our loop candidates (shown between curly brackets in the first occurrence of location

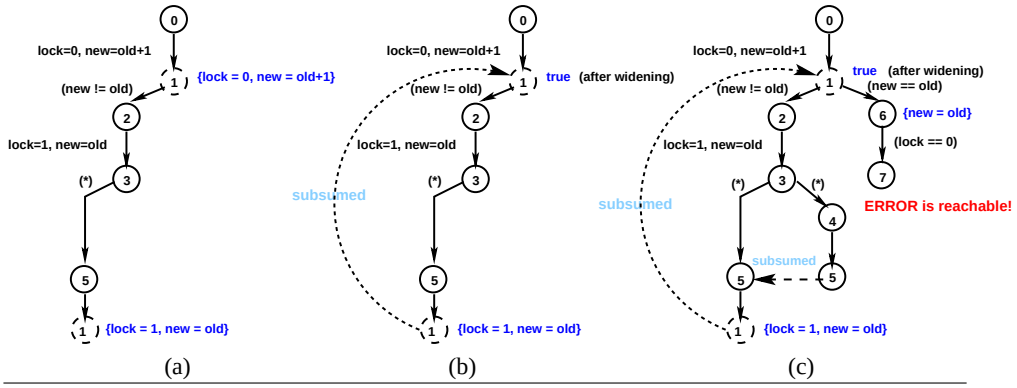

Fig. 8. First iteration of TRACER execution

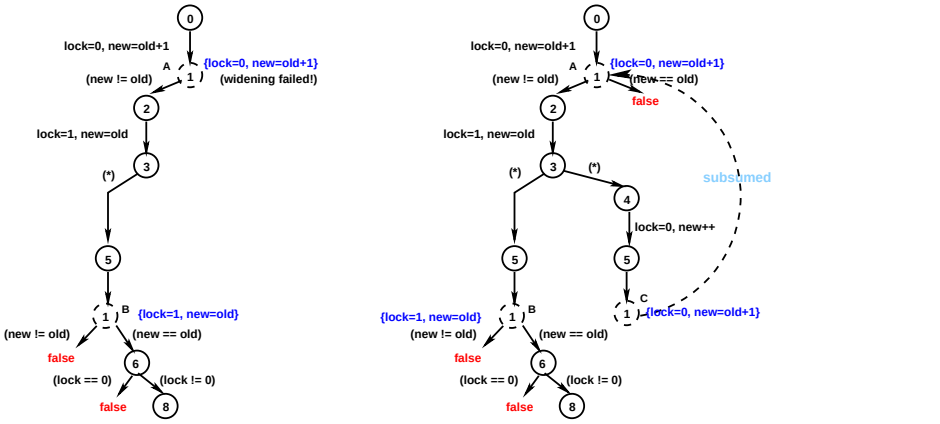

Fig. 9. Second iteration of TRACER execution

1) $\{lock_0 = 0, new_0 = old_0 + 1\}$ produces an abstracted symbolic state $\top$ ($lock_0 = 0 \nabla\ lock_1 = 1 \equiv \top$ and $new_0 = old_0 + 1 \nabla\ old_1 = new_0 \equiv \top$). The symbol $\top$ means *true*. The symbolic path π_1 after widening is shown in Fig. 8(b). Note that the symbolic state at the loop header is *true*, and as a result, we can halt the execution of the path and avoid unrolling the path π_1 forever since the child (second occurrence of location 1) is subsumed by its parent (first occurrence of 1).

We then continue and backtrack executing a second path π_2 from executing the `then` branch. For π_2, the candidates are indeed invariants but this is irrelevant since the execution of π_1 already determined that they were not invariant. As a result of the loss of precision of our abstraction, the exit condition of the loop $(new_0 = old_0)$ is now satisfied, and in fact, the error location is reachable by the symbolic path $\pi_3 \equiv (new_0 = old_0) \wedge (lock_0 = 0)$.

We then trigger a counterexample-guided refinement. First, we check that π_3 is indeed spurious due to the loop abstraction (i.e., $lock_0 = 0 \wedge new_0 = old_0 + 1 \wedge (new_0 = old_0) \wedge (lock_0 = 0)$ is unsatisfiable). Second, by weakest preconditions we infer an interpolant $I \equiv new_0 \neq old_0$ that suffices to rule out the counterexample. Third, we strengthen our loop abstraction $\top$ (*true*) with I, record that I cannot be abstracted further, and restart.

After restart, the execution of π_1 shown in Fig. 9(a) cannot be halted as before at location labelled with B since $(new_0 = old_0 + 1) \nabla (old_1 = new_0)$ is still $\top$ but this abstraction does not preserve $new_0 \neq old_0$, the interpolant from the refinement phase. As a result, we are not allowed to abstract the candidate $new_0 = old_0 + 1$ at location labelled with A and thus the path must be unrolled one more iteration. However, the unrolled path will not take the loop body anymore but follow the exit condition propagating the constraints $lock_1 = 1 \wedge new_1 = old_0$. Hence, the unrolled path is safe.

Finally, and in order to get a proof, we still need to explore π_2 from the `then` branch shown in Fig. 9(b). Fortunately, we can stop safely the exploration of π_2 as before since we do not need to perform any abstraction for this path and hence, $new_0 \neq old_0$ is preserved. As a result, the state of the child labelled with C is subsumed by its ancestor A.