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 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 use of interpolants from infeasible paths 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 poses its own challenges: (C1) exponential number of paths, and (C2) infinite-length paths in presence of unbounded loops.

We present TRACER, a SE-based verification tool 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 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

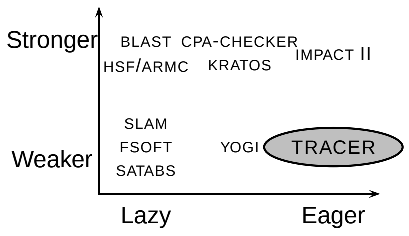


Fig. 1. State-of-the-art verifiers

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

*This paper extends the ideas published in [12, 13] by describing a method for computing weakest preconditions as interpolants as well as a detailed description of the architecture of the tool and a new experimental evaluation.

as a cheap refinement for CEGAR. One disadvantage is that it cannot recover from too-specific refinements (see program *diamond* in [18]). CPA-CHECKER and KRATOS encode loop-free blocks into Boolean formulas that are then subjected to an SMT solver in order to exploit its (learning) capabilities and avoid refinements due to coarser abstractions often used in CEGAR. 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 computing efficient approximated *weakest preconditions* as interpolants. To the best of our knowledge, TRACER is the first publicly available (`paella.d1.comp.nus.edu.sg/tracer`) verifier with these characteristics.

2 How TRACER Works

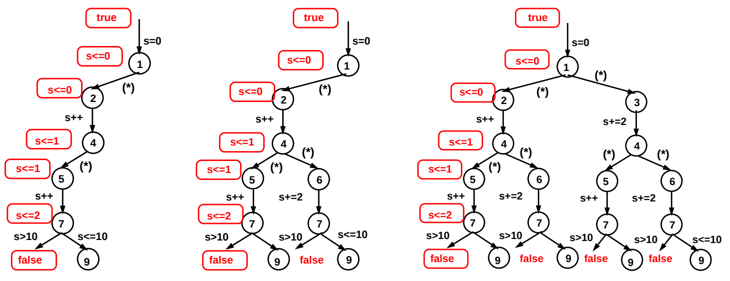
Essentially, TRACER implements classical symbolic execution [15] with some novel features that we will outline along this section. It takes symbolic inputs rather than actual data and executes the program considering those symbolic inputs. During the execution of a path all its constraints are accumulated 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 $PC_2 \equiv PC \wedge \neg\, C$. Then, it checks if either PC_1 or PC_2 is unsatisfiable. If yes, then the path is *infeasible* and the execution halts backtracking to the last choice point. Otherwise, it follows the path. The verification problem consists of building a *finite* symbolic execution tree that overapproximates all concrete reachable states and proving for every symbolic path the error location is unreachable.

The first key aspect of TRACER, originally proposed in [13] for symbolic execution, is the avoidance of full enumeration of symbolic paths by *learning* from infeasible paths computing *interpolants* [8]. Preliminary versions of TRACER [12, 13] computed interpolants based on *strongest postconditions (sp)*. Given two formulas A (symbolic path) and B (last guard where infeasibility 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 variables (i.e., variables occurring only in A). 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)*.

Example 1. The verification of the contrived program in Fig. 2(a) illustrates the need for wp as well as the essence of our approach to mitigate the "path-explosion" problem. Fig. 2(b) shows the first symbolic path explored by TRACER which is infeasible. `(*)` means that the evaluation of the guard can be *true* or *false*. After renaming we obtain the unsatisfiable constraints $s_0 = 0 \wedge s_1 = s_0 + 1 \wedge s_2 = s_1 + 1 \wedge s_2 > 10$. State-of-the-art interpolation techniques will annotate every location with its corresponding

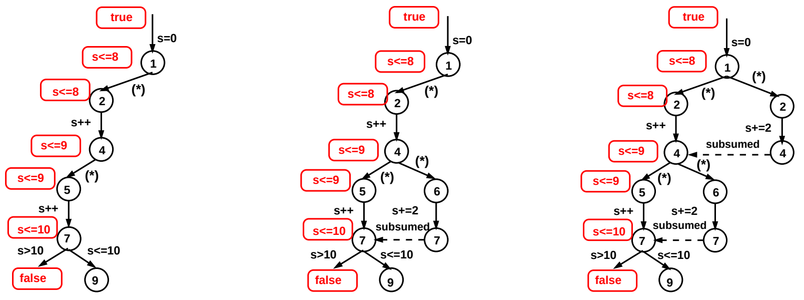
[1]Given formulas A and B such that $A \wedge B$ is unsatisfiable, a *Craig interpolant* [8] I satisfies: (1) $A \models I$, (2) $I \wedge B$ is unsatisfiable, and (3) its variables are common to A and B. We say an interpolant I is stronger (weaker) than I' if $I \models I'$ ($I' \models I$).

```
⟨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⟩
```



(a) (b) (c) (d)

Fig. 2. Symbolic Trees with Strongest Postconditions or CLP-PROVER (running TRACER on program in Fig. 2(a) with options `-intp sp` or `-intp clp`)



(a) (b) (c)

Fig. 3. Symbolic Trees with Weakest Preconditions (running TRACER with `-intp wp`)

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. 2(c) shows the tree after the second symbolic path has been explored. 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[2] by $\iota_7 : s_2 \leq 2$, the interpolant at 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. 2(d) if the new symbolic path can be subsumed by a previous explored path. Here, it tests if $s_0 = 0 \wedge s_1 = s_0 + 2$ implies $\iota_4 : s_1 \leq 1$ but again it fails. TRACER can prove the program is safe but the symbolic execution tree built is exponential on the number of program branches. □

For efficiency, TRACER 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:

[2] 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 $\sigma \models \sigma'$.

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

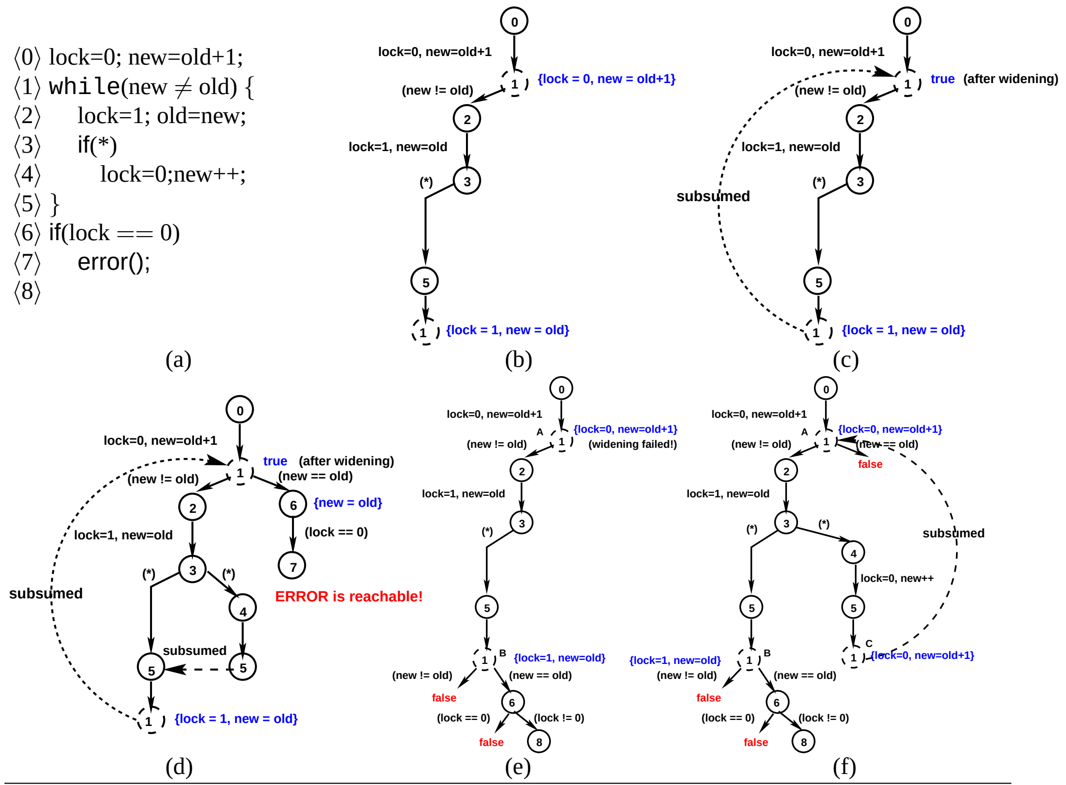
Rule (A) replaces all occurrences of x with e in the formula Q. The challenge is how to produce efficient (conjunctive) formulas from rule (B) as weak as possible to increase the likelihood of subsumption. During the forward SE when an infeasible path is detected we discard *irrelevant* guards by using the concept of *unsatisfiable cores (UC)*[3] 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 we compute $\bigwedge_{1 \leq i \leq n}(\neg\, (\exists\, \overline{x'} \cdot (C \wedge d_i)))$, where existential quantifier elimination removes 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.

Example 2. Coming back to the program in Fig 2(a). Fig. 3(a) shows the same first symbolic path explored by TRACER but annotated with 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 $\iota_7 : s_2 \leq 10$. In this example, the wp computations are notably simplified since the guards are clearly irrelevant for the infeasibility of the path, and hence, only rule (A) 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. 3(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. 3(c) illustrates how the third symbolic path can be also subsumed at location 4 since $s_0 = 0 \wedge s_1 = s_0 + 2 \models s_1 \leq 9$. TRACER proves safety again but the size of the symbolic tree is now linear on the number of branches. □

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 $\overline{\Psi}$ 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 loop invariant. If this test fails for a path π, then TRACER unrolls π one more iteration and continues 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] for technical details. Here, we illustrate how TRACER handles unbounded loops through the classical example described in Fig 4(a).

Example 3. TRACER executes the program until a cycle is found and checks whether a certain set of loop candidates holds after the execution of the cycle. We obtain the symbolic path $\pi_1 \equiv lock_0 = 0 \wedge new_0 = old_0 + 1 \wedge (new_0 \neq old_0) \wedge lock_1 = 1 \wedge old_1 = new_0$ from executing the `else` branch, shown in Fig. 4(b). Assume a widening ∇ defined as $c\ \nabla\ c' \triangleq c$ if $c' \models c$ otherwise *true*, where c and c' are the

[3]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.



```
⟨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. 4. TRACER execution for an excerpt from a NT Windows driver

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 1) $\{lock_0 = 0, new_0 = old_0 + 1\}$ produces an abstracted symbolic state *true* ($(lock_0 = 0) \nabla (lock_1 = 1) \equiv$ *true* and $(new_0 = old_0 + 1) \nabla (old_1 = new_0) \equiv$ *true*). The path π_1 after widening is shown in Fig. 4(c). Note that the symbolic state at the loop header is *true*, and as a result, we can stop executing 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 backtrack to 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 $(new_0 = old_0)$ (Fig. 4(d)) is now satisfied and the error location is reachable by the path $\pi_3 \equiv (new_0 = old_0) \wedge (lock_0 = 0)$. Then, a refinement is triggered. 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 *true* with I, record that I cannot be abstracted further, and restart.

After restart, the execution of π_1 shown in Fig. 4(e) cannot be halted at location labelled with B since $(new_0 = old_0 + 1) \nabla (old_1 = new_0)$ is still *true* 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 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, we explore π_2 from the `then` branch shown in Fig. 4(f). Fortunately, we can stop safely the execution of π_2 (as before) since no abstraction is needed for this path and hence, $new_0 \neq old_0$ is preserved. As a result, the state of the child C is subsumed by its ancestor A. $\square$

Remarks. It is known that wp may fail to generalize with some loops as Jhala et al. pointed out in [14]. TRACER can be fed with other interpolation methods and/or with inductive invariants from external tools (see Sec. 3). Also, 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, we would like to emphasize that abstraction in TRACER 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 CPA-CHECKER 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

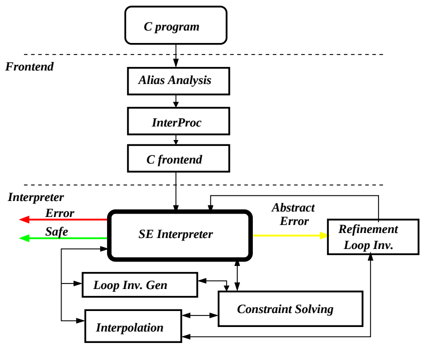


Fig. 5. Implementation of TRACER

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. 5 outlines the implementation of TRACER. It is divided into two components. First, a C-frontend based on CIL [19] translates the 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, INTERPROC [16] (option `-loop-inv`) can be used 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]. Its main sub-components are:

- *Constraint Solving* relies on the CLP($\mathcal{R}$) solver to reason fast over linear arithmetic over reals augmented with a decision procedure for arrays (option `-mccarthy`).
- *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`) but currently it only supports 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 check 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*. If a loop header is found TRACER records a set of *loop invariant* candidates by projecting onto the propagated symbolic state. When a cycle π is found it widens the state at the header by $c\nabla c'$ where c' is the candidate c after the execution of π. Current implementation of widening is $c\nabla c' \triangleq c$ if $c' \models c$ otherwise *true*. 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 show that our method for discovering loop invariants is fast and effective, it is *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 fairly naive. 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 benchmarks from SV-COMP (sv-comp.sosy-lab.org) and compare with two state-of-the-art tools: CPA-CHECKER [3] and HSF [21]. Fig. 6 shows the results of this comparison including the impact on TRACER using strongest postconditions (SP) and weakest preconditions (WP) as interpolants. Columns 2 and 3 compare the number of states of the symbolic execution tree (#S) explored by TRACER using SP and WP, and columns 4 and 5 compare the number of loop invariant refinements made (#R) using SP and WP. The rest of the columns show total time in seconds T (including compilation time) of TRACER (SP and WP), CPA-CHECKER (CPA), and HSF (HSF). For a fair comparison, TRACER did not use invariants from INTERPROC. ∞ indicates TRACER did not finish within 900 seconds.

Our results indicate that the use of WP pays off with greater gains in programs where TRACER refines heavily, mainly because loop unrolls are expensive for SE, and hence subsuming more often is vital. For ssh-simplified benchmarks (s3_clnt and s3_srvr) TRACER, with SP, was unable to finish for all but one program, where #S, #R and T

Program	#S		#R		T			
	SP	WP	SP	WP	SP	WP	CPA	HSF
cdaudio	4663	2138	0	0	12	10	3	529
diskperf	4565	2829	0	0	14	11	3	513
floppy	1758	1357	0	0	4	4	2	568
kbfiltr	319	230	0	0	2	2	2	5
s3_clnt_1	∞	6940	∞	33	∞	61	7	8
s3_clnt_2	∞	9871	∞	74	∞	115	12	5
s3_clnt_3	∞	17617	∞	114	∞	338	8	9
s3_clnt_4	∞	6990	∞	46	∞	80	5	8
s3_srvr_1	∞	5496	∞	12	∞	33	18	5
s3_srvr_2	∞	7295	∞	29	∞	120	98	11
s3_srvr_3	∞	5950	∞	14	∞	37	13	39
s3_srvr_4	47988	4349	143	12	372	27	25	10

Fig. 6. Comparison between TRACER and state-of-the-art verifiers on Intel 2.33Ghz 3.2GB.

were about 10-15 times more compared to WP. Compared with HSF, a "pure" CEGAR verifier, TRACER out-performed it in the ntdrivers-simplified benchmarks (first 4 rows) and was out-performed in the rest. This suggests that CEGAR may behave better when numerous loop unrolls are needed and SE may be more suitable when most of the infeasible paths affect safety (where CEGAR would perform many refinements). Comparing with CPA, a hybrid verifier and winner of SV-COMP'12, TRACER fares almost equally in the ntdrivers-simplified benchmarks and s3_srvr programs, but is out-performed in the s3_clnt benchmarks. Nevertheless, our evaluation demonstrates that TRACER is competitive with state-of-the-art verifiers.

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, J.A. Navas, and A. E. Santosa. 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. S.Grebenshchikov et.al. Synthesizing Software Verifiers from Proof Rules. In *PLDI'12*.
22. A. Rybalchenko and V. Sofronie. Constraint Solving for Interpolation. In *VMCAI'07*.