TracerX: Enhancing Dynamic Symbolic Execution with Weakest Precondition Interpolation

Arpita Dutta[1], Rasool Maghareh[2], Joxan Jaffar[3], Sangharatna Godboley[4], and Xiao Liang Yu[4]

School of Computing, National University of Singapore, Singapore[1,3,5], Lemurian Labs, Toronto[2], National Institute of Technology Warangal, India[4]

Test-Comp 2025,
[8 May 2025, Hamilton, Canada]

From KLEE To TracerX

- DFS Forward Symbolic Execution to find feasible paths (Similar to KLEE)
- Intermediate execution states preserved (Unlike KLEE)
- Path interpolants are generated for each path during backward tracking
- Tree interpolants are generated as conjunction of path interpolants
- Tree interpolants then used for subsumption at similar program points

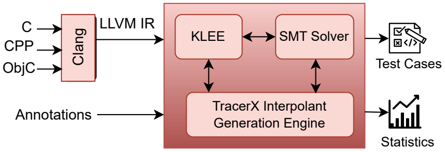


Figure: TracerX Framework

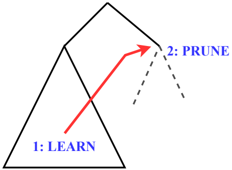
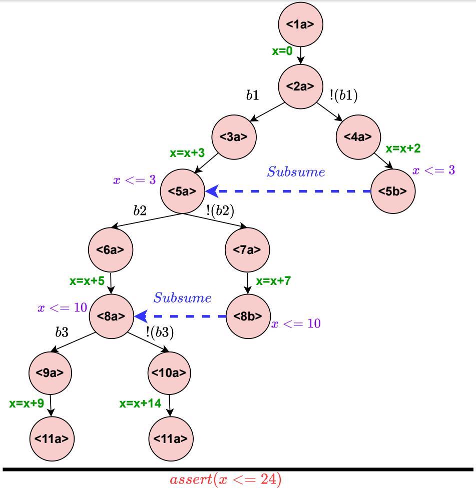


Figure: Pruning of subtree

Symbolic Execution Tree with Interpolation




```
x = 0;
if (b1) x += 3 else x += 2
if (b2) x += 5 else x += 7
if (b3) x += 9 else x += 14
assert(x <= 24)
```

- **Without interpolation:** The full tree is traversed.
- **With interpolation:**
  1. $\langle 8b \rangle$ context contains $x = 10$. It is subsumed with the tree interpolant from $\langle 8a \rangle$: $x \leq 10$.
  2. $\langle 5b \rangle$ context contains $x = 2$. Subsumed with the tree interpolant from $\langle 5a \rangle$: $x \leq 3$.
  3. Big subtree traversal is avoided.

NUS National University of Singapore

TracerX implementation of Path Based Weakest Precondition (PBWP)

- Ideal interpolant is the weakest precondition of the target. Unfortunately, PBWP is intractable to compute.
- For example, Assume $(b1 \wedge \neg b2 \wedge \neg b3)$ is **UNSAT**.
  WP before first "if-statement" is: $b1 \longrightarrow (\neg b2 \wedge b3 \wedge x \leq 7) \vee (b2 \wedge x \leq 4)$
  $\neg b1 \longrightarrow x < 3$
- Essentially, PBWP is exponentially disjunctive
- Challenge is to obtain a conjunctive approximation

A Path is a sequence of assignment and assume instructions:

1. Interpolant of Assignment instruction:
   - $\textsc{wp}(inst, \omega) = \cdots$ inverse transition of *inst* over ω
   - e.g. $\omega : x \leq 15$ and $inst : x = z + 2$, then $\textsc{wp}(inst, \omega) : z \leq 13$
2. Interpolant of Assume instruction (C is incoming Context): $\{C\}$ assume(B) $\{\omega\}$
   - PBWP Approximation: find $\bar{C}$ to replace C
   - ABDUCTION PROBLEM !!!

Approximation of Path Based Weakest Precondition

This algorithm is the heart of TracerX:

1. We compute finest partition so that $var(C_i) * var(C_j)$ $s.t.$ $i \neq j$: $\{C_1 * C_2 * C_3 * \ldots * C_n\}$ assume(B) $\{\omega_1 * \omega_2 * \omega_3 * \ldots * \omega_m\}$ ($*$ is as in separation logic).

2. Bunch C_i into three:

- **Target independent:** The C_i which are separate from B and ω.
  Action: Replace C_i with *true*, i.e. remove C_i.
- **Guard independent:** Consider $C_{gi} \equiv C_i$ s.t. $C_i * B$; and, $\omega_{gi} \equiv \omega_j$ s.t. $B * \omega_j$.
  Action: Replace C_{gi} by ω_{gi}.
- **Remainder of the C_i:** We do not capture exact WP for this group.
  e.g. $\{z == 5\}$ assume($x > z - 2$) $\{x > 0\}$ (Here, $z > 2$ is the WP)
  Action: No change to C_i, i.e. keep C_i.

Current Research Direction: Incremental Loop Analysis

Challenge

Verification of looping programs up to the maximum (including unbounded loops) bound.

Limitations of Existing Techniques

- Classic symbolic execution (e.g., KLEE[1]) does not support pruning.
- Static symbolic execution (e.g., CBMC[5]) requires the loop unrolling bound upfront.
- Iterative deepening is not incremental. Not recognizing the same program points on loop unroll.
- Interpolation techniques available for model checking are ineffective towards inductive reasoning (eg., CPAcheker[8]).

Our Ultimate Goal

Given a looping program:

1. Prove the program for a given loop bound.
2. Prove the program for as large as possible loop bound.
3. Prove the program for unbounded iterations.

Harness to Explain Incremental Behavior

```
#define loop for(;;)
#define MAXDEEP ∞
⟨1⟩ main() {
⟨2⟩         int B = 1, k, x = 0;  /* Initilization of variables */
⟨3⟩         loop {
⟨4⟩                 if ( * ) { /* Non deterministic choice */
⟨5⟩                      if (B == MAXDEEP) exit(0);  /* Fixed Bound Verification */
⟨6⟩                      B++;
⟨7⟩                 } else {
⟨8⟩                      k = B;
⟨9⟩                      while (k) {
⟨10⟩                          x = Ψ(x); /* Program under Test */
⟨11⟩                          k = k - 1;
⟨12⟩                     }
⟨13⟩                     assert(Φ(x)); /* Safety Condition Check */
⟨14⟩                     break;
⟨15⟩                }//EndIf
⟨16⟩        }//EndLoop
⟨17⟩ END:;
⟨18⟩ }
```

- In line <10>, $\Psi(x)$ is your program under test. It could contain non-determinism but must be free from unbounded loops.
- B and k are ghost variables.
- Line <5> check is used for fixed bound verification.

NUS National University of Singapore

Framework behind Incremental Analysis of Loops

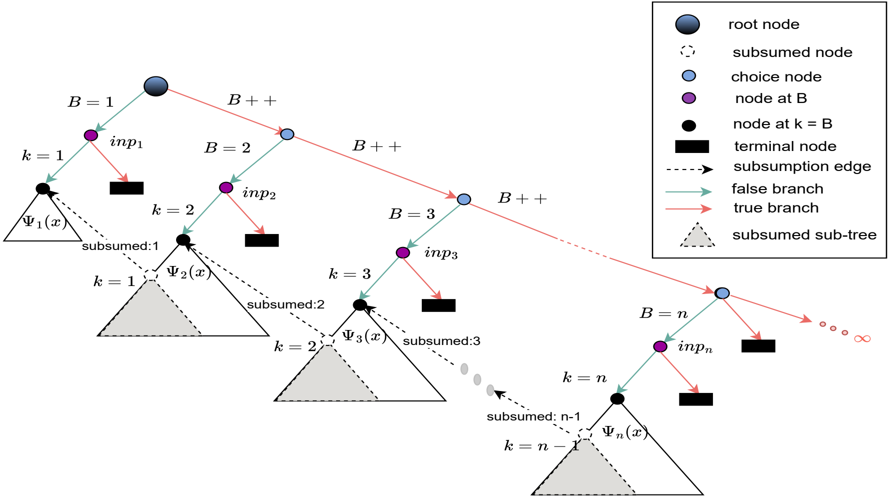


Figure: Incremental Deepening using Interpolation

Framework behind inductive reasoning and unbounded verification

For an unbounded program P:

We perform below two sequential checks to find the fixed point:

- First, confirm the tree explored for the most recent bound is subsumption closed.
  $\rightarrow$ i.e., No new path explored.
- Secondly, the safety condition at bound `k` is not weaker than at bound `k-1`.
  $\rightarrow$ We ensure this by equivalence check between the interpolants at bounds `k-1` and `k`. $[inp_{k-1} \equiv inp_k]$

Example 1: Incremental Reasoning

```
int B = 1, x = 0;
for (;;){
     if(*) B++;
     else{
          k = B;
          while (k){
               x = x + 2;
               k = k - 1;
          }
           assert(x != 99999);
       }
}
```

- TracerX verifies the program up to 10106 iterations in 3600 seconds.
- CBMC results in Timeout for bound $\sim\leq$ 300 with a limit of 3600 seconds.
- TracerX successfully utilizes the incremental behavior to reach a higher loop bound.
- TracerX unable to obtain a FixPoint but it can find FixPoint if assume $(x \geq 0)$.

Example 2: Inductive Reasoning when Assertion is Inductive

```
⟨1⟩ int x = 0;
⟨2⟩ int i = 0;
⟨3⟩ while (1) {
⟨4⟩         x += 3;
⟨5⟩         if (i == 3) x++;
⟨6⟩         i++;
⟨7⟩         if (i == 2) i = 0;
⟨8⟩         }
⟨9⟩ assert(x % 3 == 0)
```

Table: Comparison of Interpolation Algorithms for FixPoint Reasoning

Method	Fixed Point	#Unrolling	Invariant	Time
IMC [9]	No	41	-	900s
ISMC [8]	No	16	-	900s
DAR [8]	No	5	-	900s
TracerX	Yes	3	$i=0 \wedge x\%3=0$	0.07s

NUS National University of Singapore

Example 3: Inductive Reasoning when Assertion is Not Inductive

```
int B = 1, x = 1, y = 1;
for (;;){
        if(*) B++;
        else{
                k = B;
                while (k){
                        int temp = x;
                        x = x + y;
                        y = y + temp;
                        k = k - 1;
                }
                assert(y >= 1);
        }
}
```

- This example is taken from [10].
- Here, the invariant is not inductive.
- TracerX obtained the fixed point in 2 iterations.
- Obtained inductive interpolant is `(x + y >= 1)`.

Example 4: Inductive Parametric Reasoning

```
int B = 1, sum = 0;
for (;;){
      if(*) B++;
      else{
            k = B;
            while (k){
                  sum = 2 + sum;
                  k = k - 1;
            }
             assert(sum = 2*B);
      }
}
```

- TracerX obtained the fixed point in 2 iterations.
- Obtained interpolants:
  - At `k==1`: INP_1 is `(sum = (-2 + 2*B))`
  - At `k==2`: INP_2 is `(sum = (-4 + 2*B))`
- On replacing, `B` by `B+1` (since, step-size for `B` is 1) in INP_2, $\therefore$ $\text{INP}_1 \equiv \text{INP}_2$.
- This is true for every `k`th and `k+1`th interpolants.

Ex 5: Inductive Parametric Reasoning when Assertion is Not Inductive

```
int B = 1, x1 = 0, x2 = 1;
for (;;){
    if(*) B++;
    else{
        k = B;
        while (k){
            int temp = x1;
            x1 = x2;
            x2 = x1 + temp;
            k = k - 1;
        }
        assert(fib(B) == x2);
    }
}
```

- This is the Fibonacci sequence program.
- Function `fib(B)` returns the `B`th number from Fibonacci sequence.
- Here, the parametric loop invariant is itself not inductive.
- Work in progress.

Experimental Results

Data set: All C-programs (with at least 1 bug) from RERS Challenge [2012-2022] [6].

- Removed Programs with more than 300K LOC.
- Total Programs: 137 - 19 (Large) - 6 (From 2022) = 112
- All programs are event-condition-action (ECA) type with unbounded loops.

Table: Program Characteristics

Year	# Programs	LOC		# Predicates		Size of Predicates		# Target
		Min	Max	Min	Max	Min	Max	
2012	19	595	184.9K	127	17917	2	30	61
2013	18	2.4K	153.3K	279	11152	4	12	60
2014	24	1K	285K	134	22671	2	19	100
2016	14	1.8K	155.7K	199	2862	2	20	100
2017	9	1.9K	140.8K	216	2883	2	20	100
2018	9	1.2K	114.2K	211	2742	2	21	100
2019	11	1.1K	88.6K	200	2702	2	20	100
2020	8	1.1K	127K	199	2825	2	20	100
2022	6	3.2K	270.2K	-	-	-	-	-

- Removed shallow bugs using KLEE[1] for 3600 seconds.

Experimental Results

- Both the systems **CBMC** [5] and **TracerX** are run for 3600 seconds
- We first run TracerX to determine the loop bound for CBMC.
- TracerX(Win): Programs not finished by CBMC.
- CBMC(Win): CBMC finished faster than TracerX.

Table: Comparison of TracerX with CBMC

RERS Year	Total Programs	TracerX (Win)	CBMC (Win)	DeepBug Detected*	Bound Covered	
					TracerX [Min - Max]	CBMC [Min - Max]
2012	19	14	5	4	[4 - 523]	[8 - 377]
2013	18	14	4	1	[4 - 15]	[5 - 15]
2014	24	15	9	0	[4 - 20]	[4 - 20]
2016	14	9	5	3	[5 - 16]	[9 - 16]
2017	9	6	3	1	[3 - 11]	[9 - 11]
2018	9	5	3	1	[4 - 15]	[4 - 11]
2019	11	8	3	2	[3 - 11]	[10 - 12]
2020	8	4	3	0	[4 - 12]	[10-12]

Resources on TracerX

1. Website: `https://tracer-x.github.io/`
2. Github: `https://github.com/tracer-x/`
3. **TracerX: Dynamic Symbolic Execution with Interpolation**
   J. Jaffar, R. Maghareh, S. Godboley, X.L. Ha, 2020
   `https://arxiv.org/abs/2012.00556`
4. **TracerX: Dynamic Symbolic Execution with Interpolation (competition contribution)**
   J. Jaffar, R. Maghareh, S. Godboley, X.L. Ha, *FASE 2020*
5. **Toward Optimal MC/DC Test Case Generation**
   S. Godboley, J. Jaffar, R. Maghareh, A. Dutta, *ISSTA 2021*
6. **TracerX: Pruning Dynamic Symbolic Execution with Deletion and Weakest Precondition Interpolation (competition contribution)**
   A. Dutta, R. Maghareh, J. Jaffar, S. Godboley, X. L. Yu, *FASE 2024*

References

[1] C. Cadar et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, 2008.

[2] J. Jaffar et al. TRACER: A symbolic execution tool for verification. In: CAV, 2012.

[3] J. Jaffar et al. TracerX: Dynamic symbolic execution with interpolation (competition contribution) . In: FASE, 2020.

[4] A. Dutta et al. TracerX: Pruning Dynamic Symbolic Execution with Deletion and Weakest Precondition Interpolation (competition contribution). In: FASE, 2024.

[5] D. Kroening et al. CBMC-C Bounded Model Checker. In: TACAS 2014.

[6] http://rers-challenge.org/

[7] D. Beyer et al. Augmenting interpolation-based model checking with auxiliary invariants. In International Symposium on Model Checking Software, 2024.

[8] D. Beyer et al. A transferability study of interpolation-based hardware model checking for software verification. In: FSE, 2024.

[9] D. Beyer et al. Interpolation and SAT-based model checking revisited: Adoption to software verification. In: J. Autom. Reasoning, 2024.

[10] A. R. Bradley. Understanding IC3. In: International Conference on Theory and Applications of Satisfiability Testing, 2012.