LIV: Loop-Invariant Validation Using Straight-Line Programs

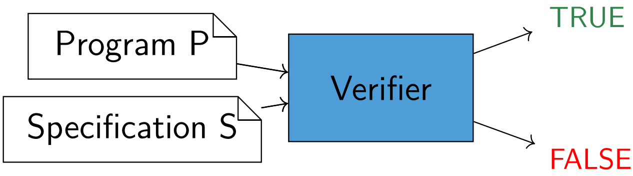
Dirk Beyer and **Martin Spiessl**

2023-09-14

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

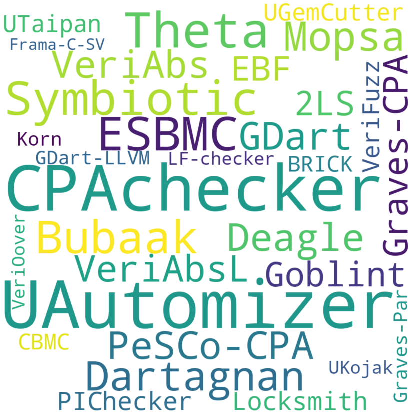
SoSy-Lab Software Systems

Automatic Software Verification



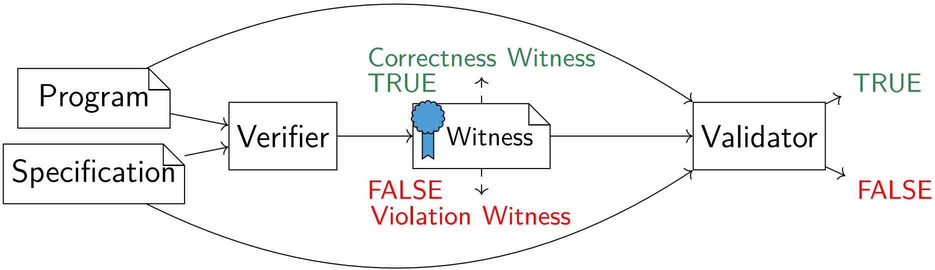
- Verification Task: answer whether $P \models S$
- In general undecidable!
- Many tools participating at the Competition on Software Verification (SV-COMP) [1]
- **Problem:** how can we trust their results?

⇒ Validate the results



C Verifiers in SV-COMP 2023

Validation of Verification Results



- Output a witness along with the verdict, i.e., information about the proof/counterexample
- Use a validator to check the proof/counterexamples [3, 2]
- **In this talk: focus on correctness-witness validation**

Correctness Witnesses Contain Loop Invariant Candidates

- not an Invariant:
  `sum==55`
- Invariant, safe but not inductive:
  `sum<=55`
- Invariant, inductive but not safe:
  `x<=10`
- Invariant, safe and inductive:
  `x>=0 && x<=10 && sum = x*(x+1)/2`

```
int x = 0;
int sum = 0 ;
//@ loop invariant I;
while (x<10) {
  x++;
  sum+=x;
}
assert(sum<=55);
```

Problem: current validators may validate witnesses successfully in all 4 cases!

Existing Correctness-Witness Validators

- Not enough validators for correctness witnesses (only 3, shown on the right)
- Witnesses **may** contain partial proofs
- Validators **may** ignore wrong or insufficient invariants
- Validators **may** default to solving the verification task
  $\Rightarrow$ **may** run for a long time or timeout
- **Our solution:** Design a new validator (LIV) that turns **"may"** above into **"must not"**

MetaVal

CPAchecker

UAutomizer

C Validators for Correctness Witnesses in SV-COMP 2023

Establish Full Proofs

```
int x = 0;
int sum = 0 ;
//@ loop invariant I;
while (x<10) {
  x++;
  sum+=x;
}
assert(sum<=55);
```

$$\dfrac{\{P\}s_0\{R\} \qquad \dfrac{R \Rightarrow I \qquad \{I \land C\}B\{I\} \qquad I \land \neg C \Rightarrow Q}{\{R\}\ \texttt{while}\ C\ \texttt{do}\ B\ \{Q\}}\ \text{while}}{\{P\}s_0\ \texttt{while}\ C\ \texttt{do}\ B\{Q\}}\ \text{comp}$$

Proof Obligations:

- $\{P\}s_0\{I\}$
- $\{I \land C\}B\{I\}$
- $I \land \neg C \Rightarrow Q$

From Proof Obligations to Straight-Line Programs

Proof Obligations:

- $\{P\}s_0\{I\}$ (Base Case)
- $\{I \land C\}B\{I\}$ (Inductiveness)
- $I \land \neg C \Rightarrow Q$ (Safety)

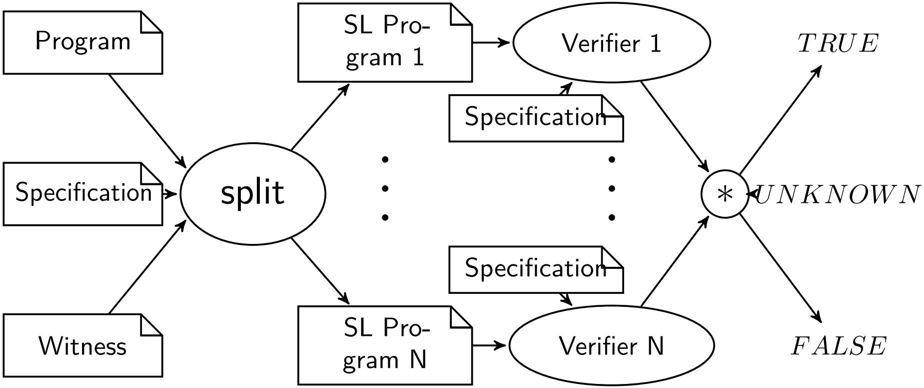
Straight-Line Programs:

```
int x = 0;
int sum = 0;
assert(I);
```

```
int x = nondet();
int sum = nondet();
assume(I && C);
x++;
sum += x;
assert(I);
```

```
int x = nondet();
int sum = nondet();
assume(I && !C);
assert(Q);
```

Workflow of LIV



- can use any off-the-shelf verifier from SV-COMP as backend
- small frontend using pycparser for AST-based splitting

Evaluation

Experiment 1: we run LIV on a set of 22 benchmarks with known, supposedly inductive and safe invariants

- **RQ 2:** Can LIV give additional feedback to the user?

Experiment 2: We run LIV on correctness witnesses of SV-COMP 2023 for the small subset of 22 C programs from experiment 1

- **RQ 1:** Is LIV an efficient validator?
- **RQ 3:** Are invariants from SV-COMP verifiers already inductive and safe?

RQ 2: Can LIV give additional feedback to the user?

- Experiment 1: we run LIV on a set of 22 benchmarks with known, supposedly inductive and safe invariants
- Result: we discovered three bugs in the benchmark set, where feedback from the tool helped to localize the cause; one is shown on the right ⇒

```
int k = nondet();
int j = nondet();
int n = nondet();
if (!(n>=1&&k>=n&&j==0))
  return 0;
//@ loop invariant j <= n
           && n <= k + j;
while (j<=n-1) {
  j++;k--;
}
assert(k>=0);
return 0;
```

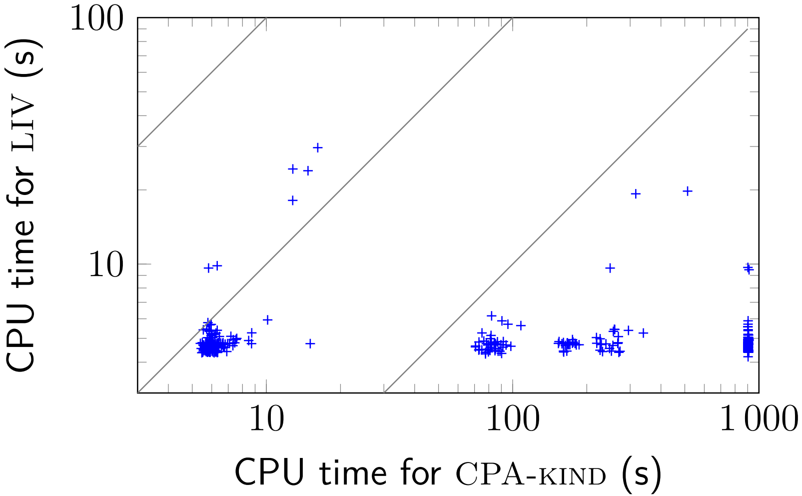
RQ 1: Is LIV an efficient validator?

Experiment 2:

- We run LIV on correctness witnesses of SV-COMP 2023 for a small subset of (22) C programs
- do the same with CPACHECKER and compare both validators

Comparison with CPAchecker's k-Induction-based Validation

LIV gives quick answers and does not run into timeouts



RQ 3: Are invariants from SV-COMP verifiers already inductive and safe?

Experiment 2:

- We run LIV on correctness witnesses of SV-COMP 2023 for a small subset of (22) C programs
- We will have a look at how many of those are contain already sufficient invariants for a proof

Witnesses Validated by LIV

some of the invariants are already sufficient

Verifier	# Tasks		LIV			
	total	non-trivial	confirmed	rejected	unknown	error
2LS	13	12	6	7	0	0
CBMC	7	7	1	5	1	0
CVT-ALGOSEL	16	11	2	13	1	0
CVT-PARPORT	19	5	4	15	0	0
CPACHECKER	21	6	5	14	0	2
GRAVES	22	9	5	14	2	1
PESCO	21	16	11	5	1	4
UAUTOMIZER	22	22	9	12	1	0
UKOJAK	21	21	10	10	1	0
UTAIPAN	22	22	6	16	0	0

Summary

- LIV: a correctness-witness validator
- more rigorous, confirms less witnesses, but terminates quickly
- splits validation into multiple straight-line programs
- delegates validation to verifiers
- allows insights into why a proof fails
- complements existing validators
- more information on our supplementary website ⇒

Supplementary Webpage:

sosy-lab.org/research/liv

Paper Preprint

Source Code Repo

Demonstration Video

DOI 10.5281/zenodo.8289101

Reproduction Artifact

References I

Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29

Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867