LIV: Loop-Invariant Validation using Straight-Line Programs

Dirk Beyer
LMU Munich, Germany

Martin Spiessl
LMU Munich, Germany

***Abstract*—Validation of program invariants (a.k.a. correctness witnesses) is an established procedure in software verification. There are steady advances in verification of more and more complex software systems, but coming up with good loop invariants remains the central task of many verifiers. While it often requires large amounts of computation to construct safe and inductive invariants, they are more easy to automatically validate. We propose LIV, a new tool for loop-invariant validation, which makes it more practical to check if the invariant produced by a verifier is sufficient to establish an inductive safety proof. The main idea is to apply divide-and-conquer on the program level: We split the program into smaller, loop-free programs (a.k.a. straight-line programs) that form simpler verification tasks. Because the verification conditions are not encoded in logic syntax (such as SMT), but as programs in the language of the original program, any off-the-shelf verifier can be used to verify the generated straight-line programs. In case the validation fails, useful information can be extracted about which part of the proof failed (which straight-line programs are wrong). We show that our approach works by evaluating it on a suitable benchmark. Supplementary website: https://www.sosy-lab.org/research/liv/**

I. Introduction

Constructing and validating program invariants is the main task of many verification approaches. Since invariants can be proposed by imprecise approaches, it is imperative to *validate* whether each candidate invariant is indeed an inductive and safe program invariant. Verification witnesses have been proposed as an exchange format for program invariants [1], but there is a shortage of validators of correctness witnesses in software verification. It is extremely important to not only *confirm* invariants that can be used for the proof of correctness, but also to reliably *reject* invariants that are not helpful for constructing the correctness proof (they might be not inductive or not implying the safety property). Recently, an approach was proposed to split the verification task (C program and specification) to a set of C programs (with assertions inlined) such that the original program is correct if all generated C programs are correct [2]. We adopt and implement this approach in LIV, and explore its potential for the use case of validation of correctness witnesses.

Contributions. We contribute the following results:

- the open-source tool LIV, which is a witness validator that implements the approach of generating straight-line programs in the language of the original program,
- using off-the-shelf verifiers to establish correctness, and
- an evaluation on programs from the SV-Benchmarks repository [3], demonstrating the effectiveness of LIV.

II. Approach

For ease of presentation, we consider a basic programming language where a program is a sequence of statements (which can consist of other statements). Let Σ denote the set of all possible statements, then the set of all programs is denoted by Σ^*. For the empty program we will use the symbol ϵ. Each statement can either be an atomic statement (denoted by a), a compound statement (denoted by S), or a special statement. Special statements are statements that affect the control flow, such as the break and continue statements which are frequently used in loops and the goto statement[1]. Iteration statements like `while` C `do` B and branching statements like `if` C `then` S `else` T are compound statements.

We support the iteration statements to be annotated by a loop invariant γ, and write `while`$^\gamma$ C `do` B. If no invariant is annotated, we will implicitly assume the weakest possible invariant, that is, $true$. Location invariants can be added to every statement in the same manner, i.e., S^γ is a statement with a location invariant that holds whenever that statement is reached and before it is executed. A *straight-line program* is a program that does not contain any loop statement (and therefore also no loop-invariant annotations).

A Hoare triple consists of a precondition $\{P\}$ from the set *Pred* of predicates, a program from Σ^*, and a post-condition $\{Q\}$ from *Pred*. A *straight-line Hoare triple* is a Hoare triple where the program does not contain any iteration statements.

Example. The program from Fig. 1a has the following statement structure:

$$s_0\ (\texttt{if}\ C_1\ (\texttt{while}^\gamma\ C_2\ \texttt{do}\ B)\ \texttt{else}\ s_1)\ s_2$$

From the structure of the program, we can construct the following four straight-line Hoare triples that correspond to the straight-line programs shown in Figs. 1b, 1c, 1d, and 1e:

- $\{P\}s_0\,[C_1]\,\{\gamma\}$
- $\{\gamma \wedge C_2\}\mathtt{B}\{\gamma\}$
- $\{\gamma \wedge \neg C_2\}s_2\{Q\}$
- $\{P\}\,[\,!C_1\,]\,s_1 s_2\{Q\}$

The brackets around an expression, for example in $[C_1]$, indicate an assume statement. The predicates P and Q are both *true* in the example.

[1]We will not focus on goto statements in this paper, but plan to add support for this, which should be straight-forward.

```c
int x = nondet();
if (x>=0) {
  while (x>0) { // loop invariant: x>=0
    x--;
  }
} else x++;
y = x;
```

(a) Original program with a loop invariant

```c
assume(1);
int x = nondet();
assume(x>=0);
assert(x>=0);
```

(b) SLP 1: loop invariant holds after initialization

```c
int x = nondet();
assume(x>=0);
assume(x>0);
x--;
assert(x>=0);
```

(c) SLP 2: loop invariant is inductive

```c
int x = nondet();
int y = nondet();
assume(x>=0);
assume(x<=0);
y = x;
assert(1);
```

(d) SLP 3: continuing after the loop

```c
int x = nondet();
int y = nondet();
assume(1);
assume(x<0);
x++;
y = x;
assert(1);
```

(e) SLP 4: covering the else branch

Fig. 1: Example for splitting an original program into straight-line programs (SLPs) using a loop invariant; initializations in SLPs before the first assume are to produce valid C programs

Soundness. In general it is possible to prove the soundness of splitting procedures like the one we present here, which was done for VST-A [2], the main inspiration for LIV.

Our splitting procedure is implemented on top of the abstract syntax tree, and kept very simple on purpose. In addition, soundness bugs in our implementation can be expected to be discovered when applying LIV to the extensive benchmark set from the Competition of Software Verification, which covers many corner cases of the C language.

A. Tool Architecture

LIV is implemented in Python 3, using a modified version of *pycparser-ext*[2] as a frontend for parsing C programs. The splitting of the input program is done by traversing the abstract syntax tree of the input program. In addition, global variable and function definitions are collected and added to all generated straight-line programs. Verification of the generated straight-line programs is delegated to a backend verifier using CoVeriTeam [4]. The choice of the backend verifier can be configured (from a set of more than 45 verifiers for C programs [5]).

III. Evaluation

Our evaluation addresses the following research questions:

RQ 1 Can an (efficient) validator be constructed via splitting the original program into straight-line programs?

RQ 2 Can the validator give additional feedback to the user?

RQ 3 To which extent are the invariants provided by automatic software verifiers via correctness witnesses already enough to establish a complete, inductive proof?

[2]https://github.com/inducer/pycparserext

TABLE I: Results of running LIV on the benchmark set, using three off-the-shelf verifiers as backend

Verifier Backend	Correct (18 total)		Wrong (3 total)	
	Confirmed	Rejected	Confirmed	Rejected
CBMC	18	0	0	3
CPAchecker	18	0	0	3
CPA-LIA	17	1	1	2

A. Experiment Setup

We conduct two experiments to show the usefulness of LIV, the first targeting RQ 1 and RQ 2, the second targeting RQ 3. For both experiments, we will look at the subset of verification tasks from the SV-Benchmarks repository[3] called *loop-zilu*, consisting of 22 C programs. For the first experiment, we will take invariants for the benchmark tasks from the ACSL-Benchmarks repository[4] where inductive loop invariants that should be sufficient to prove the assertion of the programs are annotated to the programs in various formats. For the second experiment, we use correctness witnesses that were produced by verifiers participating in SV-COMP 2022 [6].

We run our experiments on compute nodes with an Intel Xeon E3-1230 CPU. Each experimental run uses all 8 available processing units and is limited to use 15 GB of memory and 900 s of CPU time.

We configure LIV to run with different off-the-shelf verifiers as backend. For the first experiment we limit ourselves to CBMC [7] and CPAchecker [8]. In addition, we use a special configuration of CPAchecker which we refer to as CPA-LIA. This configuration uses linear integer arithmetic (LIA) as internal encoding, which is imprecise by design and allows us to observe cases in which the internal encoding makes a difference for validation.

For the second experiment, we only use CBMC as backend. Since the generated programs do not contain loops, a mature, bounded model checker like CBMC will allow us to quickly check the generated verification tasks while supporting a large subset of the C language.

In order to compare this to state-of-the art correctness validators, the second experiment also contains a comparison with CPAchecker's correctness-witness validation, which is based on incremental k-induction, and which we refer to as CPA-kind. Since the approach of LIV is more similar to 1-induction, we also compare with a modified version of CPAchecker's k-induction, where the induction bound k is fixed to 1, hence we refer to it as CPA-1ind.

B. Evaluation Results

RQ 1. Table I shows the results for the first experiment. For our benchmark set, all correct witnesses are confirmed by the off-the-shelf verifiers CBMC and CPAchecker. These also correctly reject three witnesses for which the invariant is actually not sufficient. One such example of an inductive invariant that is not

[3]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[4]https://gitlab.com/sosy-lab/research/data/acsl-benchmarks

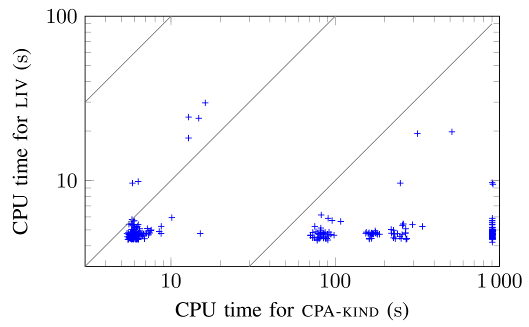
```c
int main() {
    int k = __VERIFIER_nondet_int();
    int j = __VERIFIER_nondet_int();
    int n = __VERIFIER_nondet_int();

    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;
    __VERIFIER_assert(k>=0);
    return 0;
}
```

(a) **benchmark04_conjunctive.c**: Inductive but unsafe invariant (invariant holds after initialization, is inductive, but does not imply the assertion)

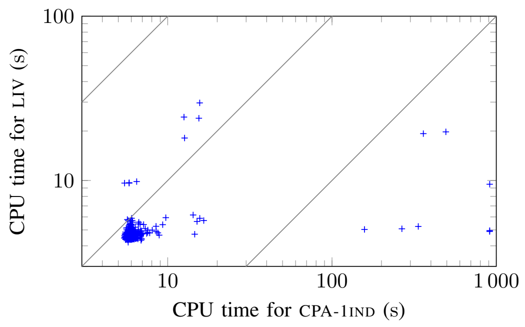
```c
int main() {
    int i = __VERIFIER_nondet_int();
    int c = __VERIFIER_nondet_int();
    if (!(c==0 && i==0)) return 0;
    //@ loop invariant 2 * c == (i-1) * i
      ↪ && 0 <= i && i <= 100;
    while (i<100) {
      c = c+i;
      i = i+1;
      if (i<=0) break;
    }
    //@ assert c >= 0;
    __VERIFIER_assert(c>=0);
    return 0;
}
```

(b) **benchmark10_conjunctive.c**: Invariant not inductive (integer variable `c` can have negative values due to overflow)

Fig. 2: Example for benchmarks with insufficient invariants



(a) Comparison with CPA-KIND



(b) Comparison with CPA-1IND

Fig. 3: Scatter plots comparing LIV with CPACHECKER-based correctness-witness validation; all results are shown, independent of the verification result

sufficient to prove the assertion is shown in Fig. 2a. CPA-LIA misses rejecting one incorrect witness and instead rejects one witness that is actually correct. The incorrect invariant that is accepted by CPA-LIA is shown in Fig. 2b.

Regarding efficiency, Fig. 3a shows that the execution of LIV is also efficient, i.e., it finishes after a few seconds in the majority of cases and never runs into a timeout for the given benchmark set. This is also mostly true for CPA-1IND (Fig. 3b); there are only a few outliers and timeouts caused by the employed SMT solver. The fact that CPA-1IND confirms more witnesses than LIV is due to the fact that CPA-1IND unrolls the loop body one additional time (to encode assertions inside the loop body) before checking the assertion after the loop. CPA-KIND completely unrolls the loops in several programs that have a finite loop bound, which is why it confirms significantly more witnesses.

In sum, RQ 1 can be answered positively.

RQ 2. Initially we expected all the witnesses from the benchmark set to be confirmed. Surprisingly, upon inspection of the witnesses that LIV rejected, we indeed confirmed that some invariants are not strong enough to establish the specified safety property. We show two such examples in Fig. 2.

Upon failure, existing validators would at most output information about whether a specific invariant was confirmed or not. LIV can give more fine-grained reports, as we can distinguish between whether the invariant holds after the initialization, whether it is inductive, or whether the invariant does not ensure the safety property (assertion after the loop). The two programs in Figs. 2a and 2b are examples in which LIV tells us why the proof of correctness cannot be established.

In sum, LIV's feedback is for each proof step by construction.

RQ 3. The results for the second experiment are shown in Table II. We show only the results for verifiers that created non-trivial correctness witnesses in SV-COMP 2022, i.e., witnesses that actually contain at least one syntactically valid invariant.

We can observe that a significant fraction of the invariants reported by the verifiers are actually sufficient for an inductive proof by LIV. Upon closer inspection of the output of the two CPACHECKER-based variants, we can observe that it often happens that they ignore the provided invariants, leading to confirmed results even if the provided invariants are not sufficient. Also, neither of the CPACHECKER-based approaches rejected any witness due to a wrong invariant.

For RQ3, while there are significantly many witnesses that help proving correctness, we also report that there is still room for improvement in the invariants provided from the verifiers.

IV. Related Work

Witness Validation. There are only few approaches for validating correctness witnesses [9, 10]. The closest to our approach is METAVAL [9]. METAVAL encodes additional proof goals for the invariants provided via the witness into a C program, but does not split programs up into multiple, simple sub-programs. The C program is constructed as automaton product of the

TABLE II: Results for LIV and CPAchecker-based witness validation on the SV-COMP witnesses from each verifier

Verifier	# Tasks		LIV				CPA-kind		CPA-1ind	
	total	non-trivial	confirmed	rejected	unknown	error	confirmed	unknown	confirmed	unknown
2LS	13	12	6	7	0	0	11	2	7	6
CBMC	7	7	1	5	1	0	7	0	3	4
CVT-AlgoSel	16	11	2	13	1	0	9	7	5	11
CVT-ParPort	19	5	4	15	0	0	14	5	9	10
CPAchecker	21	6	5	14	0	2	15	6	10	11
Graves	22	9	5	14	2	1	12	10	10	12
PeSCo	21	16	11	5	1	4	15	6	15	6
UAutomizer	22	22	9	12	1	0	11	11	7	15
UKojak	21	21	10	10	1	0	10	11	7	14
UTaipan	22	22	6	16	0	0	11	11	7	15

witness automaton and the CFA of the original program, and as such it is not immediately clear whether this construction is sound, and less likely to be faster to validate. In case the validation fails, it is not clear to the user where exactly the proof goes wrong, and loops are still present.

Verification-Condition Generation. Our design can also be seen as a variation of deductive verification, where verification conditions are generated from the (user-)provided invariants, which are then typically handed off to a backend solver. For example, Dafny [11] translates to Boogie [12, 13] as an intermediate representation. Other examples include VeriFast [14] and Frama-C [15]. The automatic verifier Korn [16] translates to constrained Horn clauses. While for existing deductive and automatic verifiers, the verification conditions usually use a very specific representation that is bound to the particular backend and cannot easily be reused, LIV uses the original programming language to encode the verification conditions and off-the-shelf verifiers as backend to solve them. Our approach is closely motivated by VST-A [2, 17], which is an annotation verifier for C programs. The main difference is that our transformation is purely AST-based, while for VST-A the actual splitting is defined over the CFA of the program. In general, algorithms for verification-condition generation are tool-specific and not described in detail in literature, with few exceptions [18].

V. Conclusion

There is a demand for invariant-validation tools in order to ensure the correctness of the results of verification tools. LIV is a new validator for correctness witnesses. The unique feature of this tool is that it reads the invariants from correctness witnesses, and reduces the validation problem to the verification of a set of straight-line programs. This enables the application of arbitrary off-the-shelf verification tools for C programs, including bounded verifiers, for establishing a proof of correctness.

Acknowledgement. This work was inspired by a discussion with Andrew Appel at the Isaac Newton Institute, Cambridge, in July 2022.

Data-Availability Statement. The source code of LIV is available at https://gitlab.com/sosy-lab/software/liv and a reproduction package [19] includes the tool and experimental data. The results of our experiments are available in interactive BenchExec tables at https://www.sosy-lab.org/research/liv/.

Funding Statement. This project was funded in part by the Deutsche Forschungsgem. (DFG) – 378803395 (ConVeY).

References

[1] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). doi:10.1145/3477579

[2] Zhou, L.: Foundationally sound annotation verifier via control flow splitting. In: Proc. SPLASH. pp. 69–71. ACM (2022). doi:10.1145/3563768.3563956

[3] Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2022 and Test-Comp 2022). Zenodo (2022). doi:10.5281/zenodo.5831003

[4] Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). doi:10.1007/978-3-030-99524-9_31

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

[6] Beyer, D.: Verification witnesses from verification tools (SV-COMP 2022). Zenodo (2022). doi:10.5281/zenodo.5838498

[7] Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). doi:10.1007/978-3-540-24730-2_15

[8] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). doi:10.1007/978-3-642-22110-1_16

[9] Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). doi:10.1007/978-3-030-53291-8_10

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

[11] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proc. LPAR. pp. 348–370. LNCS 6355, Springer (2010). doi:10.1007/978-3-642-17511-4_20

[12] DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)

[13] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Proc. FMCO. pp. 364–387. LNCS 4111, Springer (2005). doi:10.1007/11804192_17

[14] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. NFM. pp. 41–55. LNCS 6617, Springer (2011). doi:10.1007/978-3-642-20398-5_4

[15] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Proc. SEFM. pp. 233–247. Springer (2012). doi:10.1007/978-3-642-33826-7_16

[16] Ernst, G.: Korn: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). doi:10.1007/978-3-031-30820-8_36

[17] Wang, Q., Cao, Q.: VST-A: A foundationally sound annotation verifier. arXiv/CoRR **1909**(00097) (August 2019). doi:10.48550/arXiv.1909.00097

[18] Homeier, P.V., Martin, D.F.: A mechanically verified verification condition generator. Comput. J. **38**(2), 131–141 (1995). doi:10.1093/comjnl/38.2.131

[19] Beyer, D., Spiessl, M.: Reproduction package for ASE 2023 article 'LIV: Invariant validation using straight-line programs'. Zenodo (2023). doi:10.5281/zenodo.8289101