SV-LIB 1.0

A Standard Exchange Format for Software-Verification Tasks

Marian Lingsch-Rosenfeld

Dirk Beyer Gidon Ernst Martin Jonáš

2026-04-11

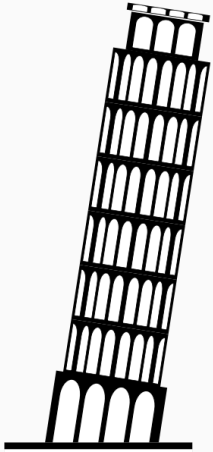
LMU Munich, Germany

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

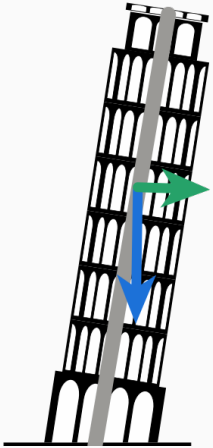
(SV-LIB)

SoSy-Lab Software Systems

Physical System



Abstract Model



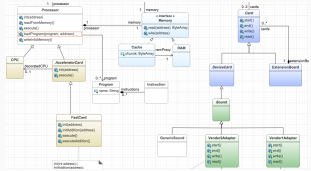
Software System

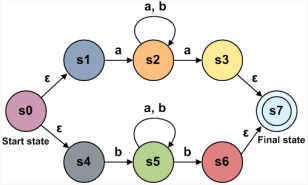
```
 int sqliteProcessJoin(Parse *pParse, Select *p){
ist *pSrc;                  /* All tables in the FROM clause *
i, j;                       /* Loop counters */
ct SrcList_item *pLeft;     /* Left table being joined */
ct SrcList_item *pRight;    /* Right table being joined */

 = p->pSrc;
t = &pSrc->a[0];
nt = &pLeft[1];
i=0; i<pSrc->nSrc-1; i++, pRight++, pLeft++){
ole *pRightTab = pRight->pTab;
t isOuter;

( NEVER(pLeft->pTab==0 || pRightTab==0) ) continue;
Outer = (pRight->fg.jointype & JT_OUTER)!=0;

 When the NATURAL keyword is present, add WHERE clause terms f
 every column that the two tables have in common.

( pRight->fg.jointype & JT_NATURAL ){
if( pRight->pOn || pRight->pUsing ){
  sqlite3ErrorMsg(pParse, "a NATURAL join may not have "
     "an ON or USING clause", 0);
  return 1;
}
for(j=0; j<pRightTab->nCol; j++){
  char *zName;   /* Name of column in the right table */
  int iLeft;     /* Matching left table */
  int iLeftCol;  /* Matching column in the left table */

  zName = pRightTab->aCol[j].zName;
  if( tableAndColumnIndex(pSrc, i+1, zName, &iLeft, &iLeftCol)
```

Images thanks to Gidon Ernst

Abstract Model

$$\forall\ x.\ \mathrm{read}(\mathrm{write}(x)) = x$$





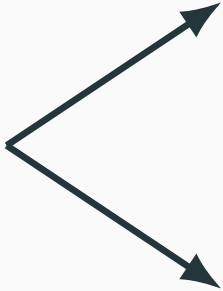
Example translation

```
int x = nondet();
int y = x + 1;
assert(y > x);
```

Is the program safe?

Example translation

```
int x = nondet();
int y = x + 1;
assert(y > x);
```



```
; Integer Encoding
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= y (+ x 1)))
(assert (not (> y x)))
(check-sat)
; unsat
```

```
; Bitvector Encoding
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(assert (= y (bvadd x (_ bv1 32))))
(assert (not (bvugt y x)))
(check-sat)
; sat
```

Program Verification

Automatic Verification

- Requires no user Input
- Modelling done by the tool
- Proof search done by the tool

Deductive Verification

- Requires abstractions from the user
- Modelling done by the tool
- Proof search done by the user

Interactive Theorem Provers

- Requires every proof step from the user
- Modelling done by the user
- Proof search done by the user

Let us take a look!

`https://gitlab.com/sosy-lab/research/data/sv-lib-demo`

What is the Problem?

(a) Translation not unique

(b) Back-and-forth translation

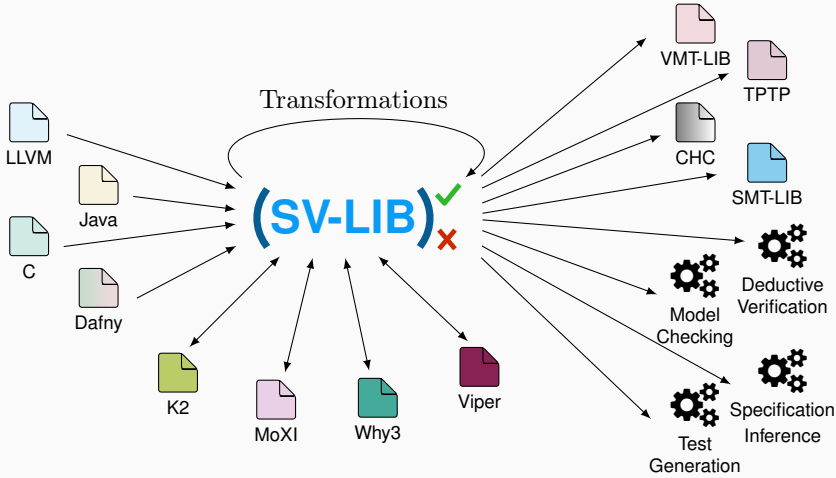
(c) Memory-model

(d) Re-implementation for each tool

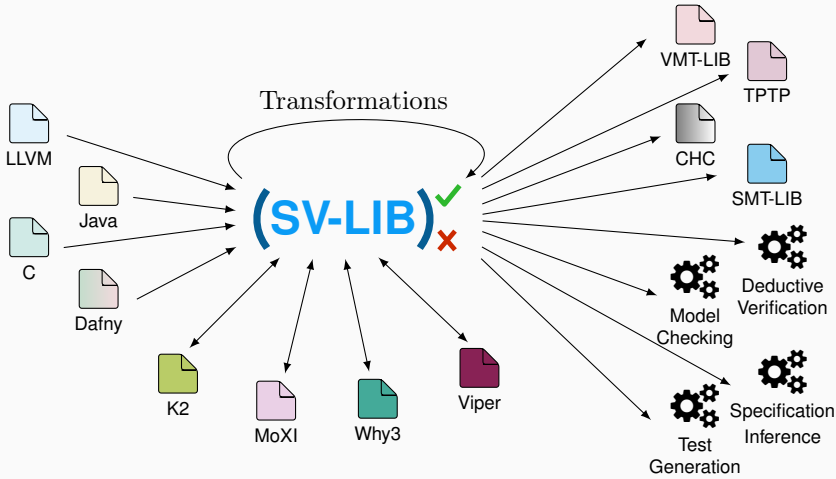
```
; Integer Encoding
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= y (+ x 1)))
(assert (not (> y x)))
(check-sat)
```

```
; Bitvector Encoding
(declare-fun x () (_ BitVec 32))
(declare-fun y () (_ BitVec 32))
(assert (= y (bvadd x (_ bv1 32))))
(assert (not (bvgt y x)))
(check-sat)
```

SV-LIB


Transformations
LLVM
Java
C
Dafny
K2
MoXI
Why3
Viper
(SV-LIB)
VMT-LIB
TPTP
CHC
SMT-LIB
Deductive Verification
Model Checking
Specification Inference
Test Generation

SV-LIB



(a) Inspired by k2 [3]

(b) Based on SMT-LIB [1]

(c) MoXI [5] is for transition systems

(d) Btor2 [4] is for hardware circuits

Motivation for another IVL

(a) Reduce engineering complexity (e.g., easy to parse and interpret)

(b) Complexities of the input language transfer to complexity in presenting verification results (e.g., witnesses)

(c) Information exchange between tools is difficult (e.g., witnesses, ...)

(d) Have formal semantics without undefined behavior

(e) many more! [2]

SV-LIB: Goals

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

SV-LIB: Goals

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

Syntax of SV-LIB

```
(set-logic LIA)

(define-proc
  add
  ((x0 Int) (y0 Int))
  ((x Int))
  ((y Int))
  (! (sequence
       (assign (x x0) (y y0))
       (! (while
            (< 0 y)
            (assign
              (x (+ x 1))
              (y (- y 1))))
        :tag while-loop))
    :tag proc-add))
```

```
(annotate-tag
  proc-add
  :requires (<= 0 y0)
  :ensures  (= x (+ x0 y0)))

(annotate-tag while-loop :not-recurring)

(declare-const x1 Int)
(declare-const y1 Int)

(verify-call add (x1 y1))
```

```
((annotate-tag
   while-loop
   :invariant
     (and
       (<= 0 y)
       (= (+ x y) (+ x0 y0)))
   :decreases y))
```

Syntax of SV-LIB

```
(set-logic LIA)

(define-proc
  add
  ((x0 Int) (y0 Int))
  ((x Int))
  ((y Int))
  (! (sequence
       (assign (x x0) (y y0))
       (! (while
            (<= 0 y)
            (assign
              (x (+ x 1))
              (y (- y 1))))
        :tag while-loop))
    :tag proc-add))
[...]
```

```
((select-trace
   (model
     (define-fun x1 () Int 1)
     (define-fun y1 () Int 1))
   (init-global-vars)
   (entry-proc add)
   (steps
     (init-proc-vars add
        ; initialization of x and y
        ; not necessary
        ))
   (incorrect-annotation
     proc-add
     :ensures
     (= x (+ x0 y0)))))
```

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

Keep Programs and Specifications Separate

```
(set-logic LIA)

(define-proc
  add
  ((x0 Int) (y0 Int))
  ((x Int))
  ((y Int))
  (! (sequence
       (assign (x x0) (y y0))
       (! (while
            (< 0 y)
            (assign
              (x (+ x 1))
              (y (- y 1))))
        :tag while-loop))
    :tag proc-add))
```

```
(annotate-tag
  proc-add
  :requires (<= 0 y0)
  :ensures  (= x (+ x0 y0)))

(annotate-tag while-loop :not-recurring)

(declare-const x1 Int)
(declare-const y1 Int)

(verify-call add (x1 y1))
```

```
((annotate-tag
   while-loop
   :invariant
     (and
       (<= 0 y)
       (= (+ x y) (+ x0 y0)))
   :decreases y))
```

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

Witnesses as first-class citizens

```
(set-logic LIA)

(define-proc
  add
  ((x0 Int) (y0 Int))
  ((x Int))
  ((y Int))
  (! (sequence
       (assign (x x0) (y y0))
       (! (while
            (< 0 y)
            (assign
              (x (+ x 1))
              (y (- y 1))))
        :tag while-loop))
    :tag proc-add))
```

```
(annotate-tag
  proc-add
  :requires (<= 0 y0)
  :ensures  (= x (+ x0 y0)))

(annotate-tag while-loop :not-recurring)

(declare-const x1 Int)
(declare-const y1 Int)

(verify-call add (x1 y1))
```

```
((annotate-tag
   while-loop
   :invariant
     (and
       (<= 0 y)
       (= (+ x y) (+ x0 y0)))
   :decreases y))
```

Witnesses as first-class citizens

```
(set-logic LIA)

(define-proc
  add
  ((x0 Int) (y0 Int))
  ((x Int))
  ((y Int))
  (! (sequence
       (assign (x x0) (y y0))
       (! (while
            (< 0 y)
            (assign
              (x (+ x 1))
              (y (- y 1))))
        :tag while-loop))
    :tag proc-add))
```

```
(annotate-tag
  proc-add
  :requires (<= 0 y0)
  :ensures  (= x (+ x0 y0))
  :not-recurring)

(declare-const x1 Int)
(declare-const y1 Int)

; Command taken from the witness
(annotate-tag
  while-loop
  :invariant
    (and
      (<= 0 y)
      (= (+ x y) (+ x0 y0)))
  :decreases y)

(verify-call add (x1 y1))
```

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

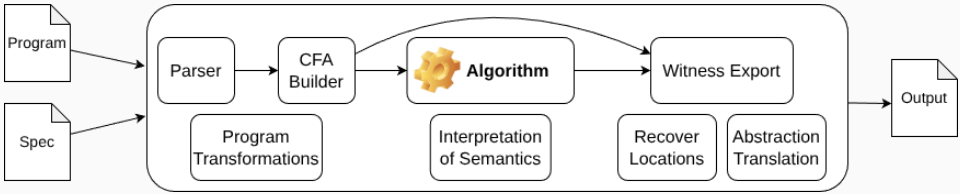
(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

Complexity of Implementing a Verifier



Algorithmic vs. Engineering complexity

Making Implementation easier: SvLibChecker

	Stand-Alone		Cumulative	
Algorithm	**LOC**	**Comments**	**LOC**	**Comments**
Trace Abstraction	113	62	196	131
Symbolic Execution	101	27	379	223
BMC	87	19	365	215
Predicate Abstraction	86	50	461	276
Value Analysis	75	25	353	221
k-Induction	59	22	424	237
All Algorithms	604	274	896	431
Total Code Base	2597	722	2597	722

(a) Consolidate existing IVLs from an engineering perspective by taking well-understood constructs, and avoiding semantic complexity

(b) Keep programs and specifications separable and the latter extensible

(c) Make witnesses first-class citizens by providing a standard format for them

(d) Make implementation of verification tools easier

(e) Strive for a software ecosystem by providing building blocks (parser, linter, default validator, transformations, benchmarks, ...)

Goal: Software Ecosystem

Aim to provide building blocks to facilitate adoption:

(a) ANTLR Grammar for parsing SV-LIB

(b) PySvLib: Python library for working with SV-LIB programs

(c) Integration of SV-LIB into CPAchecker

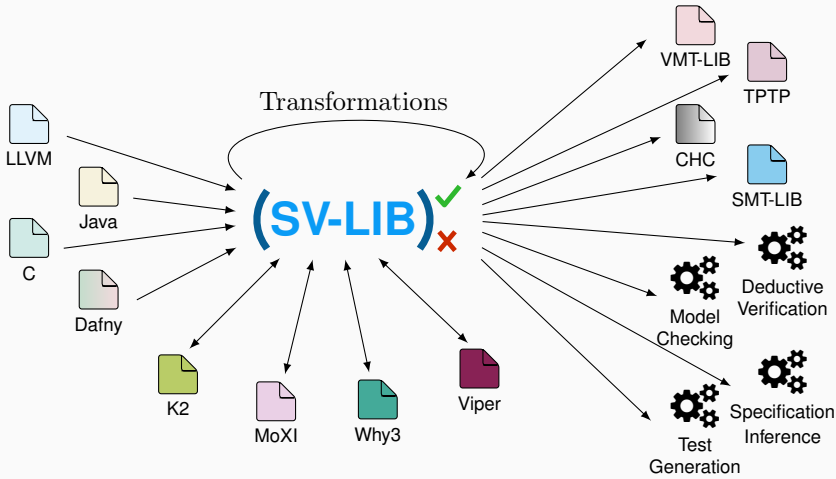
(d) Lightweight Model-Checker SvLibChecker

SV-LIB Software Ecosystem

Let us take a look!

`https://gitlab.com/sosy-lab/research/data/sv-lib-demo`

Future Work


Transformations
(SV-LIB)
LLVM
Java
C
Dafny
K2
MoXI
Why3
Viper
VMT-LIB
TPTP
CHC
SMT-LIB
Deductive Verification
Model Checking
Specification Inference
Test Generation

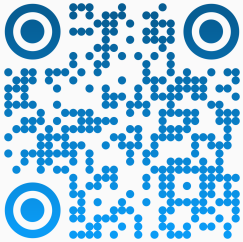
Conclusion

(a) SV-LIB 1.0 format available to read [2]

(b) Reduces engineering complexity for static analysis tools

(c) First-class support for witnesses and validation

(d) Tool support available



https://gitlab.com/sosy-lab/benchmarking/sv-lib

References i

[1] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., University of Iowa (2010), https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf

[2] Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks. arXiv/CoRR **2511**(21509) (December 2025). https://doi.org/10.48550/arXiv.2511.21509

[3] Griggio, A., Jonáš, M.: KRATOS2: An SMT-based model checker for imperative programs. In: Proc. CAV. pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20

References ii

[4] Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32

[5] Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: MoXI: An intermediate language for symbolic model checking. In: Proc. SPIN. pp. 26–46. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_2