SV-LIB 1.0: A Standard Exchange Format for Software-Verification Tasks

DIRK BEYER, LMU Munich, Germany
GIDON ERNST, LMU Munich, Germany
MARTIN JONÁŠ, Masaryk University, Czech Republic
MARIAN LINGSCH-ROSENFELD, LMU Munich, Germany

In the past two decades, significant research and development effort went into the development of verification tools for individual languages, such as C, C++, and Java. Many of the used verification approaches are in fact language-agnostic and it would be beneficial for the technology transfer to allow for using the implementations also for other programming and modeling languages. To address the problem, we propose SV-LIB, an exchange format and intermediate language for software-verification tasks, including programs, specifications, and verification witnesses. SV-LIB is based on well-known concepts from imperative programming languages and uses SMT-LIB to represent expressions and sorts used in the program. This makes it easy to parse and to build into existing infrastructure, since many verification tools are based on SMT solvers already. Furthermore, SV-LIB defines a witness format for both correct and incorrect SV-LIB programs, together with means for specifying witness-validation tasks. This makes it possible both to implement independent witness validators and to reuse some verifiers also as validators for witnesses. This paper presents version 1.0 of the SV-LIB format, including its design goals, the syntax, and informal semantics. Formal semantics and further extensions to concurrency are planned for future versions.

Additional Key Words and Phrases: Software verification, Program analysis, Exchange format, Witness, Certifying Algorithm, Intermediate language, SV-LIB, SMT-LIB

Version 1.0 (SV-LIB) 2025-11-26

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

Authors' Contact Information: Dirk Beyer, Institute for Informatics, LMU Munich, Munich, Germany, dirk.beyer@sosy.ifi.lmu.de; Gidon Ernst, Institute for Informatics, LMU Munich, Munich, Germany, gidon.ernst@sosy.ifi.lmu.de; Martin Jonáš, Faculty of Informatics, Masaryk University, Brno, Czech Republic, martin.jonas@mail.muni.cz; Marian Lingsch-Rosenfeld, Institute for Informatics, LMU Munich, Munich, Germany, marian.lingsch-rosenfeld@sosy.ifi.lmu.de.

Contents

1 Introduction

Software verification of a given input program p and input specification φ is the problem to determine whether the statement $p \models \varphi$ holds, i.e., whether p satisfies all properties in φ. The specification can describe safety and liveness properties, such as absence of runtime errors, functional correctness, and termination. If the verifier finds a violation to the specification, it provides a violation witness [12], which testifies the specification violation by describing a counterexample. If the verifier can prove that the specification holds, then it provides a correctness witness [11], which testifies that the specification holds. The precise form of the correctness witness depends on the property, and can for example consist of loop invariants for safety and of ranking functions for proving termination. Witness validation for a given input program p, input specification φ, and witness w is the problem of determining whether the statement $p \models_w \varphi$ holds, that is, whether the information in w is correct and can be used for the proof of $p \models \varphi$ or $p \not\models \varphi$.

A recent Dagstuhl seminar [15] noted that, while there are many state-of-the-art tools for software verification available, the coverage of the various programming languages is very diverse. For example, there are many verifiers for the languages C and Java, but not so many verifiers for the languages C++ and Rust. A recent overview [19] noted that there has been immense progress in research and technology in the past two decades, leading from a lack of verification tools to an abundance of them: the overview counts 76 automatic verifiers for the languages C and Java.

Most verification approaches are not specific to a particular programming language, but of general nature. For example, bounded model checking [25], k-induction [13], property-driven reachability [27], interpolation-based model checking (IMC) [45], and others [49, 50] are implemented in many verification tools (see [21], Table 8). Those approaches were usually first invented for hardware verification and later adopted to software verification (e.g., [8, 17]). It is worth noting that this can take 20 years, as for the original IMC algorithm [17].

The verification approaches and other new technology can be transferred much faster to other languages via *transformation* of the input program or model (for example, from Btor2 to C [9] and back [28, 35]). It is not necessary to re-implement each verification approach, but only a transformation needs to be implemented once and then existing tools can be applied. Similarly, many approaches are readily available for the language C but not for many other languages: An intermediate language for programs could help in this situation.

Furthermore, there is an acute lack of information exchange between verifiers form different communities working on software verification. In particular, it is difficult to exchange information between deductive verification tools (e.g., [5, 44, 47]) and automatic software verification tools (e.g., [4, 30, 37]), although preliminary studies [1, 20] showed that it can be helpful. This is partially due to the different input languages used and the differing goals and semantics of the tools. A common intermediate language for programs with a well-defined semantics could help here as well.

An intermediate language can be used also to transform programs from an imperative programming language A to an imperative programming language B (A → SV-LIB and SV-LIB → B), for example with the goal to use a verifier for B to verify programs written in A. This needs only $O(n)$ transformations for n source and target languages. Using direct transformations between n end-user programming languages, we would need to develop $O(n^2)$ transformations. This is illustrated in Fig. 1. In particular, through transformation, the different communities (hardware vs. software, deductive vs. automatic, models vs. programming languages) grow together. This can be observed by the extension of benchmark sets for competitions: For example, there is a benchmark set originating from the hardware-verification competition HWMC (which focusses on Btor2 and Aiger) that was contributed into the software-verification competition SV-COMP (which focussed on C and Java). In a similar vein, we translated C programs into SV-LIB programs

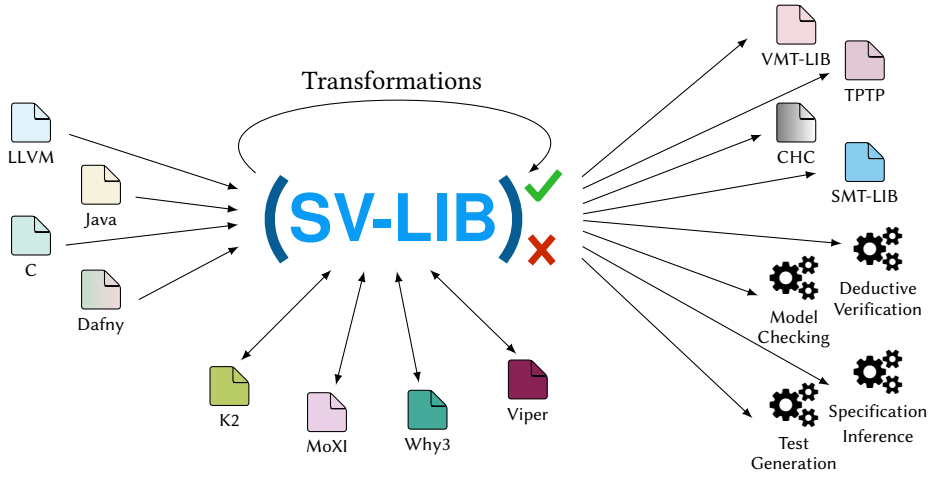

Fig. 1. Possible transformations and use-cases for SV-LIB as an intermediate language

(sv-benchmarks/.../sv-lib). Another example for inter-community exchange is using the exchange format for verification witnesses [3] for cooperation between automatic and interactive verifiers [1].

Multiple intermediate formats have been developed over the past years, and are in widespread use, prominently Boogie [31], Why3 [33], and Viper [47]. In contrast to SV-LIB, these are not directly based on SMT-LIB, making their parsing and support more difficult for tools, and they incorporate some higher-level features such as modules (Why3), expressive types (Why3, Boogie), or impose a heap model (Viper), which offers convenience in some situations but hinders adoption for other use-cases. The intermediate languages VMT [29] and MoXI [2, 40, 48] are based on SMT-LIB, but they are meant for models and represent transition systems, and do not support control flow. Furthermore, their support for witnesses is limited, and not expressible in the input format itself. Most closely related to SV-LIB is the language K2 [35], which is an intermediate language for imperative programs (supporting `if` and `goto`) and served as the major inspiration for developing SV-LIB. SV-LIB extends the spirit of K2 by adding commands to express witnesses (including invariants, statement contracts, and ranking functions for correctness witnesses, as well as concrete traces for violation witnesses), and structured control flow (`while`) directly to to the language. In contrast to K2, SV-LIB re-uses the SMT-LIB standard [6] verbatim, supporting its type and term language as well as many of the SMT-LIB top-level commands. In contrast to MoXI, we do not make a-priori restrictions on the allowed features, inheriting SMT-LIB's full expressiveness to cover use-cases in deductive verification. As an added benefit, we largely avoid the effort to document and maintain that aspect of the language.

Another popular choice for an intermediate solver-backend language is the lower-level format of Constrained Horn Clauses (CHC) [26, 38, 39]. CHCs are standardized as a subset of the SMT-LIB language, which means they offer a language- and tool-agnostic way to decouple front-ends of verification tools from back-end solvers. However, the encoding of a program-verification task into CHCs already makes some choices about the verification approach to be used and discards higher-level structure of the original task (e.g., invariants vs. loop contracts [32], large blocks vs. small blocks [10, 16]). Moreover, despite some efforts, the format of models and proofs for CHC problems is not standardized across verifiers and the information these artifacts convey is difficult to relate back to the original tasks.

Not only are intermediate languages useful for verification, they are also a popular choice for working with and compiling programming languages. For example, LLVM IR [42] is a widely-used intermediate representation for optimizing compilers and program analysis tools. Other frameworks, such as Java Bytecode and WebAssembly, serve as intermediate representations for their respective ecosystems. In such environments, having a flexible intermediate verification language that covers a large span of technologies may help with the challenges of multilingual verification tasks [34].

We developed the intermediate language for verification tasks SV-LIB to address these problems. Verification approaches that are implemented for SV-LIB programs can be used for various imperative languages by transforming the verification tasks to SV-LIB and then applying any SV-LIB verifier. Like CHC, we build on the SMT-LIB language and logic for the representation of terms of the language, which covers a rich spectrum from implementation-centric properties about arithmetic and arrays to behavioral specifications that make use of data abstractions. In contrast to CHC, we aim to *capture program-verification tasks declaratively* and to *enable justification of verdicts via a fully-specified witness format in the same format as the original verification task*. Thus, the format extends SMT-LIB with a syntax for imperative program statements and commands for the definition of procedures and correctness claims.

2 Language Design

This document introduces version 1.0 of SV-LIB, extending SMT-LIB version 2.7 [6] with constructs usually found in imperative programming languages, like (recursive) procedures, (global) variables, and (un-)structured control flow. In particular, extending SMT-LIB allows us to reuse most of its infrastructure like sorts, terms, functions, and some commands. To differentiate between SMT-LIB files and SV-LIB files, we propose to use the file ending `.svlib` for SV-LIB files, and its corresponding witnesses. In addition we recommend setting the option `:format-version` to the version of the language that is used in the program, such that it is possible to easily analyze the format version that the program corresponds to.

2.1 Community Call for a New Intermediate Language

The community proclaimed the need to exchange verification tasks and proposed to create a new intermediate language (IL) that is easy to support and process by tools [15]. In the following, we list goals and requirements for the new language, which were discussed in a working group at Dagstuhl [15]. The new language should:

(a) support the exchange of complete verification tasks, consisting of (1) programs, (2) specifications (safety and liveness), and (3) witnesses (for counterexamples and proofs),
(b) be based on a human-readable text format (instead of a binary format) and use a well-known and easy to parse base syntax (e.g., Lisp-based, as used by SMT-LIB),
(c) have a formal semantics without undefined behavior (except for cases akin to SMT-LIB where the notion of under-specification is logically well-understood (such as that division by zero yields some unknown natural number)), such that comparative evaluations of verification algorithms do not depend on interpretations, and the same should hold for witness validation,
(d) support interoperability, such that frontends and backends are exchangable,
(e) express verification and validation tasks in the same language,
(f) allow that witness validation is compilable to simple first-order checks (i.e., SMT queries) without further inference, for all properties, that is, witnesses should be informative enough to allow validation with SMT Solvers,

(g) be independent from the higher-level input language (such as C and Java) and independent from verification technology used in the backend,
(h) have first-class support for widely-used concepts like global variables and procedures,
(i) offer a way to keep the specification separate from the program,
(j) retain more structure than MoXI, VMT, and CHC (control flow, procedures, sequential composition, ...) to express tasks before encoding to verification conditions by a particular approach,
(k) be based on well-understood programming concepts and consolidate existing languages in that spectrum (e.g., Boogie, Why3, Viper), and
(l) be usable for use-cases in both automatic and deductive verification.

Furthermore, the community requested that the organizers of SV-COMP [21] include a track on SV-LIB in the Competition on Software Verification 2026[1].

2.2 Design Principles for the New Intermediate Language SV-LIB

For ease of use, we build on a well-established and well-understood standard (SMT-LIB) to encode the logical language of expressions. We extend it with constructs native to an imperative language like (recursive) procedures, (global) variables, and (un-)structured control flow, remaining as close as possible to the existing syntax. To address a wide range of approaches, the language is built around a small core with several fragments, between which translations are possible. For example, structured and unstructured control-flow fragments.

In the same manner as SMT solvers keep an interactive connection with the solver, SV-LIB also keeps an interactive connection with the verifier. We describe the interaction from the point of view of the client of the verifier, where "<" is a response of the verifier and ">" is a follow-up query from the client.

> **(verify-call** ⟨*symbol*⟩ **(**⟨*term*⟩***))**
< **correct** or **incorrect**

> **(get-witness)**
< ⟨*witness*⟩

Verifiers should be able to read a list of commands from standard input and write the outputs to standard output. Similarly to SMT-LIB, verifiers can be controlled by setting options in the commands. All verifiers are encouraged to support the options **:regular-output-channel** and **:diagnostic-output-channel**.

Moreover, to make use of the compositional nature of the specification mechanism in terms of **:annotate-tag** etc., verifiers should also accept verification tasks given as a list of files on the command line, with the interpretation that their concatenation in the given order jointly defines a single verification script. For example, this allows users to use several specifications for the same program, alternatively or conjunctively. Also, a benchmark definition might be prepended with use-case specific options (e.g., experimental evaluations might define that the witness is printed to a specific file).

2.3 Tool Standardization

Apart from having a standard language to express verification tasks, it has been recognized as beneficial to standardize the way tools interact with each other, since this enables cooperative verification and makes it easier for users to try out how other tools perform for the same task [22, 23].

[1] https://sv-comp.sosy-lab.org/2026/demo.php

In contrast to SMT-LIB, where the interaction is based on commands sent via standard input and output, we propose to allow this and additionally standardize a set of command-line options which tools need to support. This approach is motivated by security concerns, since some options may result in security concerns if set via the input file, for example, if they influence the file system access of the tool. On the other hand, it is useful to be able to set some options via the input file to be able to document those settings together with the verification task.

In order to achieve tool standardization, SV-LIB defines the following set of command-line flags, which verification tools should support. Usually when a command-line flag is not security-relevant, it can also be set via an option inside the input file. Furthemore, the command-line flags have precedence over the options set inside the input file overriding whatever was set in the program. Currently tools supporting SV-LIB should support:

(a) "`--produce-witnesses`" which enables witness production, it can also be set via the option `:produce-witnesses` inside the input file,
(b) "`--witness-output-channel <channel>`" which specifies where to output the witness, where `<channel>` is either `stdout`, `stderr`, or a POSIX compliant file name, If not specified it defaults to `stdout`.

3 Full Examples

Before showing the detailed definition of the syntax in Sects. 4, 5, 6, 7, and 8 including the informal semantics of SV-LIB, we provide some full examples (based on the algorithm add from Fig. 2, whose variables are encoded as unbounded integers in the SV-LIB examples for simplicity), showcasing different scenarios and objects (commands, properties, witnesses) that users will encounter when working with SV-LIB.

We first show an example of how the output of the verifier looks when verifying a correct program simultaneously for safety and liveness properties in Sect. 3.1. Notable for this case, the verifier has to produce a witness for both properties at the same time, instead of a witness for safety and liveness separately. Afterwards, we extend this example to show an application to incremental verification in Sect. 3.2. In this case the verifier can choose to maintain an internal state across multiple **`verify-call`** commands, in order to be more efficient. This is not required, and the verifier can re-verify the full program if it cannot perform stateful verification.

Finally we show multiple examples of how the output of the verifier looks like when verifying a program with an incorrect safety property (Sect. 3.3), liveness property (Sect. 3.4), and invariants (Sect. 3.5). These examples illustrate how a violation witness looks for different types of properties. In particular, the violation witness for invariants in Sect. 3.5 illustrates how modular correctness annotations are refuted, which may make reference to paths which do not exist in the program but which arise only in the overapproximation expressed by the inductiveness of invariants and contracts and the well-foundedness conditions of ranking functions.

After showing the use-cases for verification, we briefly show in Sect. 3.6 how a witness and program can be linked together in order to generate a validation task. This is important, since both validation tasks and verification tasks are SV-LIB scripts, and therefore, one needs to be able to combine a verification task with a witness into a new SV-LIB script.

```
int add(int x, int y) {
  while (0 < y) {
    x = x + 1;
    y = y - 1;
  }
  return x;
}
```

Fig. 2. Procedure add written in C

```
1 (set-logic LIA)
2
3 (define-proc
4   add
5   ((x0 Int) (y0 Int))
6   ((x Int))
7   ((y Int))
8   (! (sequence
9        (assign (x x0) (y y0))
10       (! (while
11            (< 0 y)
12            (assign
13              (x (+ x 1))
14              (y (- y 1))))
15        :tag while-loop))
16    :tag proc-add))
17
18 (annotate-tag
19   proc-add
20   :requires (<= 0 y0)
21   :ensures  (= x (+ x0 y0)))
22
23 (annotate-tag while-loop :not-recurring)
24
25 (declare-const x1 Int)
26 (declare-const y1 Int)
27
28 (verify-call add (x1 y1))
```

(a) Correct SV-LIB script

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

(b) Possible witness for Fig. 3a

Fig. 3. Correct SV-LIB program that fulfills all its specifications (left) and a possible witness for it (right); scripts available at the SV-LIB repo as verification and validation tasks

3.1 Verifying a Correct Program

Consider the correct program in Fig. 3a, which uses the **`LIA`** SMT-LIB logic. It defines in line 3 the procedure **`add`** with two input constants **`x0`** and **`y0`**, an output variable **`x`**, and a local variable **`y`**. The procedure adds the two numbers **`x0`** and **`y0`** using the local variable **`y`**, and variable **`x`** returns the result. This program shows how SV-LIB can separate the program from its specification, since the specifications are given by the **`:annotate-tag`** commands, which refer to the **`:tag`** annotations and are therefore not intertwined with the program itself. This also allows the reuse of the same specifications at multiple program locations. Furthermore, the specification can also be added directly by inlining them at the relevant locations, if desired. The contract for the body of the procedure **`add`** is given by the **`:annotate-tag`** command in line 18. Furthermore, line 23 specifies that the loop in line 10 terminates. Finally, the **`verify-call`** command in line 28 asks the verifier to prove that the procedure **`add`** fulfills its specification for two constant inputs **`x1`** and **`y1`**, which are defined using the SMT-LIB command **`declare-const`**.

Figure 3b shows an example witness for the program in Fig. 3a. It has the same structure as a SV-LIB script itself, since it uses the same commands. Having a program and its witness in the same format allows tool developers to focus on a single format, which can be used for both input and output, and similarly for both verification and validation.

The goal of a correctness witness is to simplify the validation of correctness, when compared to verification from scratch. Therefore, the witness provides the necessary invariants, contracts, and ranking functions, to reduce the correctness proof into SMT queries.

```
1 (set-logic LIA)
2
3 (define-proc
4   add
5   ((x0 Int) (y0 Int))
6   ((x Int))
7   ((y Int))
8   (! (sequence
9        (assign (x x0) (y y0))
10       (! (while
11            (< 0 y)
12            (assign
13              (x (+ x 1))
14              (y (- y 1))))
15        :tag while-loop))
16    :tag proc-add))
17
18 (annotate-tag
19   proc-add
20   :requires (<= 0 y0)
21   :ensures  (= x (+ x0 y0)))
22
23 (annotate-tag while-loop :not-recurring)
24
25 (declare-const x1 Int)
26 (declare-const y1 Int)
27
28 (verify-call add (x1 y1))
```

(a) First part

```
29 (define-proc
30   mul
31   ((x2 Int) (y2 Int))
32   ((m Int))
33   ((l Int))
34   (! (sequence
35        (assign (m x2) (l y2))
36        (! (while
37             (< 0 l)
38             (sequence
39               (call add (m x2) (m))
40               (assign (l (- l 1)))))
41         :tag while-mul))
42   :tag proc-mul))
43
44 (annotate-tag
45   proc-mul
46   :requires (<= 0 y2)
47   :ensures  (= m (* x2 y2)))
48
49 (verify-call mul (x1 y1))
50 (get-witness)
```

(b) Second part

Fig. 4. Two parts of an SV-LIB script showing incremental solver interaction; the solver may choose to maintain a state across multiple **verify-call** commands to be more efficient; script available at the SV-LIB repo as verification task

3.2 Incremental Solver Interaction

SV-LIB verifiers maintain a dialogue with the user, similar to SMT solvers. The user can send multiple commands to the verifier, and the verifier responds to each command. This interaction benefits from the solver maintaining its state across commands, allowing for more efficient and context-aware responses. Nonetheless, even though it may be useful to retain the state between verification calls, this is left to the discretion of the solver, which has the freedom to decide if it wants to keep parts of the state or recompute everything from scratch.

Figure 4 shows an example in which a solver could benefit from incremental verification. The first **verify-call** command at line 28 proves the procedure **add** correct. Therefore the contract of **add** can be used to prove the correctness of the procedure **mul** in the second **verify-call** command at line 49. While a solver can reuse invariants, contracts counterexamples which it has previously found, this may not always be useful. Sometimes, a previous abstraction may not be strong enough to prove the new property correct. For example, the contract **:ensures true** for the procedure **add** may not be sufficient to prove the correctness of the procedure **mul**.

The reuse of the internal state of the verifier is not only possible for finding invariants and contracts, but also for finding counterexamples. For example one could verify from the start of the **verify-call** command untill a state implying a previously found counterexample is reached or reuse previously found abstractions to speed up the search for a new counterexample [18]. But this should be done with care, since the previously found abstractions may not be useful in the new context.

```
1 (set-logic LIA)
2
3 (define-proc
4   add
5   ((x0 Int) (y0 Int))
6   ((x Int))
7   ((y Int))
8   (! (sequence
9        (assign (x x0) (y y0))
10       (! (while
11            (<= 0 y)
12            (assign
13              (x (+ x 1))
14              (y (- y 1))))
15         :tag while-loop))
16     :tag proc-add))
17
18 (annotate-tag
19   proc-add
20   :requires (<= 0 y0)
21   :ensures  (= x (+ x0 y0)))
22
23 (annotate-tag while-loop :not-recurring)
24
25 (declare-const x1 Int)
26 (declare-const y1 Int)
27
28 (verify-call add (x1 y1))
```

(a) SV-LIB script violating a safety property

```
((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)))))
```

(b) Example witness for Fig. 5a

Fig. 5. Incorrect SV-LIB program violating a safety property (left) and a possible witness for it (right), which provides the initial values of the global constants x1 and y1 and claims that no further choices need to be resolved to follow the execution to a violation of the postcondition. Scripts available at the SV-LIB repo as verification and validation tasks

3.3 Verifying an Incorrect Program for a Safety Property

Finding bugs in programs is also an important application of verifiers. In the same manner in which we want to simplify the validation of correctness proofs when compared to verification from scratch, we also want to simplify the validation of counterexamples compared to verification from scratch. Therefore, counterexamples fully describe a single path through the program which leads to the violation of the property. In particular, they resolve all types of non-determinism relevant to uniquely characterize the path to the violation, and provide the SMT model, such that the user can execute the resulting trace through the program and see the property violation. This ensures that the validation can be done through an interpreter-like execution of the program.

To show this in practice, let us consider Fig. 5a, which is a slight modification of the program in Fig. 3a, where we introduce an extra iteration of the loop inside the function **add**, which results in a violation of the **:ensures** annotation of **proc-add**. An example witness showing a counterexample is shown in Fig. 5b. The witness presents a single path through the program, by resolving all (implicit) non-determinism, which in this case is only the non-deterministic initialization of the global variables **x1** and **y1**. Note that even though the local variable **y** and output variable **x** are also non-deterministically initialized by the procedure **add**, the witness does not need to resolve this non-determinism, since the values of these variables are fully determined by the inputs **x1** and **y1** and the assign statement at line 9. The trace ends with the full description of the violated tag, encompassing its name and if applicable the violated term.

```
(set-logic LIA)

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

(a) Program with incorrect liveness property

```
((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
     while-loop
     :not-recurring)
   (using-annotation
     while-loop
     :invariant (= y 0)
   )))
```

(b) Example witness for Fig. 6a

Fig. 6. SV-LIB program violating a liveness property (left) and a possible witness for it (right), expressing a lasso that stays forever in the loop with `(= y 0)`. Scripts available at SV-LIB repo as verification and validation tasks

3.4 Verifying an Incorrect Program for a Liveness Property

In contrast to safety counterexamples, which can be validated by executing the program along the described path, for the validation of liveness violations, one requires reasoning about infinite traces. This requires using abstractions like invariants to describe the infinite part of the trace. We express the infinite trace which violates the liveness property by a finite prefix followed by modular abstractions that guarantee that the property of the annotated tag is violated.

As an example, consider the program in Fig. 6a, which modifies the program in Fig. 3a such that **`y`** is decremented only if it is larger than **`1`**. This results in an infinite loop, for any input where **`y0`** is larger than **`1`**, violating the liveness property expressed by the **`:not-recurring`** tag.

The corresponding witness is shown in Fig. 6b, which describes a finite trace by resolving all (implicit) non-determinism which cannot be inferred from the program structure. In this case, the only non-determinism is the initialization of the global constants **`x1`** and **`y1`**, since the local variable **`y`** and output variable **`x`** are fully determined by the inputs and the program structure. After the resolution of all non-determinism, the witness ends with the violated tag. Finally the infinite part of the trace is described by a sequence of **`using-annotation`** commands which express the inductive abstractions required to proof that all paths from that state fulfilling the annotations violate the tag. For the validation of such a witness, only the tag annotations inside of the **`select-trace`** command

```
(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)

(annotate-tag
  while-loop
  :invariant (= (+ x y) (+ x0 y0)))

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

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

(a) Program with an insufficient invariant

```
((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
       )
     (leap
       while-loop
       (x 3) (y -1)))
   (incorrect-annotation
     proc-add
     :ensures
     (= x (+ x0 y0)))))
```

(b) Insufficient invariant witness

Fig. 7. SV-LIB script with insufficient invariant (left) and a possible witness for it (right). The missing part is `(<= y 0)` so that y might still contribute to the sum of the invariant and thus the postcondition does not follow. Note that the `leap` step encodes a transition that was not part of the original program but rather reflects the overapproximation expressed by this imprecise invariant. Scripts available at the SV-LIB repo as verification and validation tasks

can be used when traversing the infinite part of the trace. In particular, this means that if existing annotations are relevant, they also need to be repeated inside of the **`select-trace`** command.

3.5 Verifying a Program for an Insufficient Invariant

It can occur that an invariant holds for all program executions, but is not sufficient to prove the correctness of the program. This might be the case for states that fulfill the invariant but no path exists in an actual program execution that reaches such a state. Then the invariant might be either not inductive or not safe. Therefore, a counterexample for an invariant needs to be able to enter an arbitrary state satisfying the invariant and then provide a counterexample either to the inductiveness of the invariant or to the specification. The same holds also for contracts, since they are required to be modular, and for ranking functions, since they are required to decrease in each iteration.

Figure 7a shows a program which is similar to the one in Fig. 3a, but annotated with an invariant which is not strong enough to prove the post-condition **`:ensures`** of procedure **`add`**. This is because to proof the post-condition, one needs to know that **`y`** is always larger than or equal to **`0`**. In the modular setting required for invariants, where every variable modified by the loop is non-deterministically chosen to fulfill the invariant at the loop-head, we cannot prove the post-condition in this case.

```
(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)

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

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

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

Fig. 8. Validation task produced by combining the SV-LIB script from Fig. 3a with the witness from Fig. 3b; script available at SV-LIB repo as validation task

Figure 7b shows a possible witness for this. It first resolves the (implicit) non-determinism of the global constants and then calls the procedure to be verified. Once it reaches the tag **`while-loop`**, it **`leap`**s into a state in which the variables modified in the loop body are chosen arbitrarily but satisfying the conjunction of all **`:invariant`**s at that tag. In this case a possible choice for the modified variables is $x \mapsto 3$ and $y \mapsto -1$. Note that it is possible to **`leap`** to this state, since it fulfills the conjunction of all **`:invariant`**s even though it will never occur in an actual execution of the program. After **`leap`**ing, the witness provides the violated tag, i.e., **`:ensures`** for the tag **`proc-add`**.

3.6 Witness Validation

Once we have obtained a witness for a verification task, we can construct a validation task from the verification task and the witness by inserting all commands of the witness (i.e., the complete witness without the wrapping parentheses) just before the **`verify-call`** which produced it. For example, for the correct program in Fig. 3a and the corresponding witness in Fig. 3b, this results in the SV-LIB script in Fig. 8, which represents a new verification task. This is one of the major advantages of having both verification tasks and witnesses in the same format of SV-LIB scripts, since witness validation can be trivially reduced to verification. In general, the concatenation of two SV-LIB scripts is also a SV-LIB script.

$$
\begin{aligned}
\langle \mathit{command} \rangle ::= \ & \texttt{(declare-var } \langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)} \\
| \ & \texttt{(define-proc } \langle \mathit{symbol} \rangle \ \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{)} \\
& \qquad \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{)} \\
& \qquad \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{)} \\
& \quad \langle \mathit{statement} \rangle \texttt{)} \\
| \ & \texttt{(define-procs-rec (}\langle \mathit{symbol} \rangle \ \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{)} \\
& \qquad \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{)} \\
& \qquad \texttt{((}\langle \mathit{symbol} \rangle \ \langle \mathit{sort} \rangle \texttt{)}^{*}\texttt{))}^{n+1} \\
& \quad \texttt{(}\langle \mathit{statement} \rangle \texttt{)}^{n+1}\texttt{)} \\
| \ & \texttt{(annotate-tag } \langle \mathit{symbol} \rangle \ \langle \mathit{attribute} \rangle^{+}\texttt{)} \\
| \ & \texttt{(select-trace } \langle \mathit{trace} \rangle \texttt{)} \\
| \ & \texttt{(verify-call } \langle \mathit{symbol} \rangle \ \texttt{(}\langle \mathit{term} \rangle^{*}\texttt{))} \\
| \ & \texttt{(get-witness)} \\
| \ & \texttt{(}\langle \mathit{tool\text{-}specific} \rangle\texttt{-call } \langle \mathit{symbol} \rangle \ \texttt{(}\langle \mathit{term} \rangle^{*}\texttt{))} \\
| \ & \cdots \qquad \textit{(subset of SMT-LIB commands, see Sect. 4.9)}
\end{aligned}
$$

Fig. 9. Syntax of top-level commands in SV-LIB

4 SV-LIB Commands

An SMT-LIB script consists of a sequence of top-level commands that are processed sequentially by the solver. Commands encompass declaration, definitions, and axioms to describe the problem, as well as queries that probe for solutions to the problem, such as satisfiability, models, and proofs. This principle allows for incremental sessions in which a series of queries is posed by the client and answered by the solver in a dialogue. This execution mode is a significant difference to model checking, which processes a whole verification task at once, not having an active connection where commands can be added after a response.

The design of SV-LIB follows the same principle: It extends SMT-LIB by top-level commands to declare global variables, to define (recursive) procedures and desired correctness properties. In particular, it therefore inherits sorts, identifiers/symbols, and terms from SMT-LIB. The verification tool can then be queried for whether a specific procedure call is correct. If a verdict has been reached, the verifier can be instructed to return a witness (either for correctness or for violation). Being an extension, SV-LIB therefore inherits many elements from SMT-LIB, for example some commands, terms, and sorts.

Figure 9 shows the syntax of all top-level commands in SV-LIB. The list of SMT-LIB commands which need to be supported is given in Sect. 4.9. Apart from these commands, all lines starting with a semicolon are treated as comments and ignored by the verifier.

Whenever the verifier finished processing a command, or an external interruption of the verifier was triggered, like receiving a “SIGTERM” it should provide a response. In particular, the verifier should always formulate a response to the last command before terminating, in case it is interrupted externally. For each command, except explicitly stated otherwise like for **`verify-call`**, the appropriate response is **`success`** if the option **`:print-success`** is enabled as in SMT-LIB or an empty

response if it is not enabled. In case of an error, for example if the program is not well-formed or some features are not supported, the verifier should respond with **`error`**.

4.1 Global Variables

The command **`(declare-var`** x σ**`)`** introduces a global program variable x of sort σ. It is analogous to SMT-LIB's **`declare-const`** but global program variables may occur on the left-hand side of assignments and thus may change their respective value throughout computations. Global variables implicitly come in scope of all subsequent procedure declarations — the use of global variables inside a procedure body does not need to be declared explicitly.

The declaration of a global variable immediately initializes the variable to an arbitrary value of the corresponding sort. Therefore, reading a declared variable which has not yet been assigned any value is defined behavior.

One important purpose of global variables is to model state parts that are implicit in high-level languages but explicit in logical encodings, such as the state of the heap. Additionally, global variables are a concept that is heavily relied on by typical verification approaches and therefore we support them first-class.

4.2 Procedure Declarations

The command **`(define-proc`** ρ **`(`***in***`) (`***out***`) (`***local***`)`** s**`)`** introduces a procedure with name ρ, a list of formal input parameters *in*, a list of formal output parameters *out* (return values), and a list of local variables *local*. The parameters and variables are available in the body statement s of the procedure. The syntax of statements is explained in Sect. 5. Each output parameter and local variable has an arbitrary initial value of its corresponding sort whenever the procedure is called, but they can be assigned to in the body of the procedure. There is no explicit return of output values; the last value assigned to an output parameter is considered the return value of the procedure for that output parameter.

In their body statements, procedures are allowed to use only variables and procedure calls that have been defined before. Additionally, the procedure body cannot modify the values of its input parameters.

4.3 Recursive Procedures

Since commands are processed sequentially, normally all symbols used in a procedure declaration need to be known at the time of its declaration. This does not work for recursive procedures, since they need to reference the procedure symbol introduced by the procedure declaration before the command has been processed.

The command **`define-procs-rec`** defines a set of recursive procedures. It takes a non-empty list of procedure declarations (see Sect. 4.2) which can reference each other. It first defines the signature of each procedure (i.e., name, input parameters, output parameters, local variables) and only once this is known, it introduces the bodies of the procedures, which can reference each other. This makes the parsing easier, since all signatures are known before parsing the bodies.

For now, we do not actually require the procedures to be recursive, therefore it is fine to just embed all procedures in the task into a single such command. However, both verifiers as well as benchmark generators are encouraged to either recognize or limit such cases.

4.4 Annotating Tags

The command **`(annotate-tag`** τ $a_1, \ldots, a_n$**`)`** links program locations τ to their specification, which are expressed by attributes $a_1, \ldots, a_n$. Here, τ is a *tag* potentially mentioned as part of the statements in the procedure bodies, identifying locations of interest. If the tag τ does not exist, the annotation

command is ignored. The syntax of these attributes follows the generic annotation mechanism of SMT-LIB with keywords **`:kw`** or keyword-value pairs **`:kw`** *e* for some S-expression *e*. The properties supported by SV-LIB are described in Sect. 6.

Program variables appearing in an attribute are interpreted in the context of the beginning of the statement that is tagged (not in the context of the **`annotate-tag`** command). This means that the sorts of the variables, and the variable referenced (input/output/local of the procedure or global) by the variable identifiers in the property must be resolved when interpreting the **`annotate-tag`** command. See also the discussion on well-formedness of programs in Sect. 8.

SMT-LIB sorts, constants, and function symbols appearing in an attribute are interpreted at the location of the **`annotate-tag`** command. This implies, for example, that witnesses can introduce new definitions that were not available in the original verification task. It also means that some care has to be taken when moving such annotations between the global command level and inline attributes of statements. This condition can be detected syntactically and if needed the corresponding function definitions can be moved to an earlier location in the script.

4.5 Selecting Traces

The command **`(select-trace`** π**`)`** selects a concrete trace π through the program for the (prefix of) possible executions. The trace resolves initialization of global variables, specifies the entry-point into the execution, and resolves all non-deterministic choices including initialization of local variables, as described in Sect. 7. Furthermore, it can represent jumps into arbitrary states, and set tags to represent infinite loops. The full description of traces is given in Sect. 7.2 for safety violations, in Sect. 7.3 for violations of modular specifications, and in Sect. 7.4 for violations of liveness properties.

The primary goal of this command is to restrict the verification to a single trace for witness validation. Therefore, only **`verify-call`** commands come after the command **`select-trace`**; any other command is ignored.

4.6 Verification Queries

The command **`(verify-call`** ρ **`(`**$t_1, \ldots, t_n$**`))`** is the analogue of SMT-LIB's **`(check-sat)`**. It instructs the verifier to check that all executions starting at the specified entry procedure ρ satisfy the properties annotated as part of the program, i.e., specified by prior **`annotate-tag`** commands. The initial states are given by the interpretation of the terms $t_1, \ldots, t_n$. They denote the (possibly symbolic) values of input parameters of ρ, possibly using global variables, all of which are described by prior definitions and constraints from **`(assert`** ϕ**`)`** commands. In case a **`select-trace`** command is present as the last command before the **`verify-call`** command, the verification is restricted to the single trace specified by the **`select-trace`** command.

The **`verify-call`** command poses the question: *Is the SV-LIB script correct with respect to its specifications, when starting all executions at the entry procedure* ρ *with the given arguments?"* The response to a **`verify-call`** command is one of **`correct`**, **`incorrect`**, **`unknown`**, **`unsupported`**, and **`error`**. The verifier produces **`correct`** if it was able to verify that the program satisfies *all* specified properties, **`incorrect`** if it found a counterexample to *at least one* of the specified properties, **`unknown`** if it was unable to reach a conclusion, **`unsupported`** if it does not support some feature of the SV-LIB script, and **`(error ...)`** if an error was encountered during the verification process (same format as in SMT-LIB).

If the verification task contains one or multiple **`select-trace`** commands for that procedure, the verifier may make use of them to see if they do in fact violate the respective property. This involves checking which of these apply to a specific call site, wether the trace exists in the program, and whether the indicated property violation can be confirmed. This is of interest particularly for the construction of tasks for validation of violation witnesses. In that case, the verdict should

still be **incorrect**. If some of those traces are erroneous, this can be reflected in the response as well. More details on this are in Sect. 7.

4.7 Witness Retrieval

Whenever a verification query is answered with **correct** or **incorrect**, the client may request a witness as evidence in form of a correctness or a violation witness, with the command **(get-witness)**. The format of the witnesses is described in Sect. 7: it consists of structured data. Since the response may span multiple lines, it is enclosed by a pair of parentheses.

The command **(get-witness)** requests the verifier to return a witness for the last verification query. If any other command was executed between the last verification query and the **get-witness** command, the command should produce an error, since the previous witness may be invalid, for example, if the inserted command is an **annotate-tag** command.

In case witness production is not enabled, the **get-witness** command should respond with an error. Additionally the output channel to write the witness to defaults to `stdout` if not specified otherwise.

4.8 Tool-Specific Calls

Tools may introduce further specific queries, for example to run a concrete or symbolic simulation, to generate best-effort invariants independently of a specification, or to generate test suites. We suggest such extensions to follow the same scheme as the function **verify-call**. Namely, they should specify the procedure to call and the arguments to pass. If more is necessary, it should be encoded using the already known commands.

While not strictly required, we recommend that responses to tool-specific calls adhere to the design principles of SV-LIB. In particular, we recommend that the response be either a single symbol like **correct** and **incorrect** or a sequence of SV-LIB commands like the result of the **get-witness** command. This ensures easier reuse of the commands, and may allow them to be added to the SV-LIB standard in the future.

4.9 SMT-LIB Commands

While many SMT-LIB commands make sense in the context of SV-LIB, others do not, like **check-sat**. Figure 10 lists all SMT-LIB commands supported by SV-LIB. Their usage is defined over syntactic elements, which are part of SMT-LIB; please refer to the SMT-LIB standard [6] for their syntax and semantics.

The purpose of including definitions is to model the application domain or the semantics of a higher-level language from which the verification task has been compiled, for example to model heap data structures or similar. We have deliberately included the **assert** command as well, since not all features of such background theories are easily described by proper definitions, rather one may want to resort to axiomatization of uninterpreted functions instead. One example is the axiom $a \notin s \implies |s \cup \{a\}| = |s| + 1$, for a finite set s. Because sets are not freely generated (they admit non-unique syntactic presentations of semantically equivalent values), there is no obvious definitional principle that is recursive over the elements. Another example is the axiomatic under-specification of trigonometric functions, for which precise reasoning is difficult for an SMT solver. In general, front-end tools like Dafny or Frama-C often ship with large preludes encoded in this way, which we want to accommodate as well.

Note that axioms can render the verification meaningless: for example, adding **(assert false)** to the script should make the program vacuously correct. We consider this as an orthogonal problem that is not in scope for SV-LIB verifiers.

⟨*smt-lib-command*⟩ ::= **(assert** ⟨*term*⟩**)**
| **(declare-const** ⟨*symbol*⟩ ⟨*sort*⟩**)**
| **(declare-datatype** ⟨*symbol*⟩ ⟨*datatype_dec*⟩**)**
| **(declare-datatypes (**⟨*sort_dec*⟩$^{n+1}$**) (**⟨*datatype_dec*⟩$^{n+1}$**))**
| **(declare-fun** ⟨*symbol*⟩ **(**⟨*sort*⟩***))**
| **(declare-sort** ⟨*symbol*⟩ ⟨*numeral*⟩**)**
| **(define-const** ⟨*symbol*⟩ ⟨*sort*⟩ ⟨*term*⟩**)**
| **(define-fun** ⟨*function_def*⟩**)**
| **(define-fun-rec** ⟨*function_def*⟩**)**
| **(define-funs-rec (**⟨*function_dec*⟩$^{n+1}$**) (**⟨*term*⟩$^{n+1}$**))**
| **(define-sort** ⟨*symbol*⟩ **(**⟨*symbol*⟩***)** ⟨*sort*⟩**)**
| **(get-assertions)**
| **(get-info** ⟨*info_flag*⟩**)**
| **(get-option** ⟨*keyword*⟩**)**
| **(set-info** ⟨*attribute*⟩**)**
| **(set-logic** ⟨*symbol*⟩**)**
| **(set-option** ⟨*option*⟩**)**

Fig. 10. SMT-LIB commands allowed in SV-LIB scripts (taken from [6])

However, there is an important restriction: Formulas ϕ asserted by **(assert** ϕ**)** can only constrain the SMT model but *not* the dynamic execution state of the program. Thus, ϕ must not contain references to global variables declared with **declare-var**. This implies that constraints on the initial global state at **verify-call** must be specified by other means, such as **assume** statements inside the procedures.

5 SV-LIB Statements

Statements occur as part of procedures (see Sect. 4) to define computational steps. In the design of SV-LIB, we aim to standardize constructs that are well-understood and commonly used, without imposing a particular encoding to the program. For instance, the language has unstructured control flow but also loops, acknowledging that verification tools and approaches may prefer one over the other. We express this by using *fragments*, akin to SMT-LIB logics, for different feature sets. In some cases it is possible to translate back and forth between fragments, for example by replacing loops via gotos, or by inlining non-recursive procedures. Therefore, in the absence of support for certain features, verifiers may choose to translate to a supported fragment.

Figure 11 shows the syntax of statements with the respective fragments being indicated on the right. The intended semantics of programs are finite and infinite sequences of states, which in turn are comprised of global and local variables. For safety proofs, binary relations between states that collapse the (in)finite sequences are an alternative view. We do not define the semantics here but we will present verification conditions in a future version of this document, which explain the logical properties of statements precisely.

$$
\begin{aligned}
\langle \textit{statement} \rangle ::= \ & \texttt{(assume } \langle \textit{term} \rangle \texttt{)} && \text{(B)} \\
\mid \ & \texttt{(assign (} \langle \textit{symbol} \rangle \ \langle \textit{term} \rangle \texttt{)}^{+} \texttt{)} && \text{(B)} \\
\mid \ & \texttt{(sequence } \langle \textit{statement} \rangle^{*} \texttt{)} && \text{(B)} \\
\mid \ & \texttt{(! } \langle \textit{statement} \rangle \ \langle \textit{attribute} \rangle^{+} \texttt{)} && \text{(B)} \\
\mid \ & \texttt{(call } \langle \textit{symbol} \rangle \texttt{ (} \langle \textit{term} \rangle^{*} \texttt{) (} \langle \textit{symbol} \rangle^{*} \texttt{))} && \text{(P)} \\
\mid \ & \texttt{(return)} && \text{(P)} \\
\mid \ & \texttt{(label } \langle \textit{symbol} \rangle \texttt{)} && \text{(U)} \\
\mid \ & \texttt{(goto } \langle \textit{symbol} \rangle \texttt{)} && \text{(U)} \\
\mid \ & \texttt{(if } \langle \textit{term} \rangle \texttt{ (goto } \langle \textit{symbol} \rangle \texttt{))} && \text{(U}^{*}\text{)} \\
\mid \ & \texttt{(if } \langle \textit{term} \rangle \ \langle \textit{statement} \rangle \ \langle \textit{statement} \rangle^{?} \texttt{)} && \text{(S)} \\
\mid \ & \texttt{(while } \langle \textit{term} \rangle \ \langle \textit{statement} \rangle \texttt{)} && \text{(S)} \\
\mid \ & \texttt{(break)} \mid \texttt{(continue)} && \text{(S)} \\
\mid \ & \texttt{(havoc } \langle \textit{symbol} \rangle^{+} \texttt{)} && \text{(N)} \\
\mid \ & \texttt{(choice } \langle \textit{statement} \rangle^{+} \texttt{)} && \text{(N)}
\end{aligned}
$$

Fig. 11. Syntax of statements. Each colored letter represents a distinct fragment of SV-LIB; B corresponds to basic statements, P to procedure calls and returns, U to unstructured control flow, S to structured commands, and N to nondeterminism; the superscript * indicates an extension of the respective fragment only under certain conditions

Many statements make use of symbols, sorts and/or terms in their definition. These are SMT-LIB symbols, sorts and terms, we therefore delegate the description of their syntax and semantics to the SMT-LIB standard [6]. In practice whoever SMT-based verifiers will not need to understand them, since they can be delegated to the underlying SMT solver in most cases. One exception to this is that symbols starting with # cannot appear in SV-LIB scripts, even though they are allowed in SMT-LIB. This allows verifiers to easily be able to create new variables by pre-fixing # to an existing variable. These symbols, are also not allowed to appear in the witness, so verifiers need to take care to rename such symbols when generating witnesses.

5.1 Basic Language (B)

The basic language fragment (B) encompasses assumptions, assignments, as well as sequential composition and annotations.

Assumptions. Assumptions `(assume` ϕ`)` restrict the possible executions to those satisfying the boolean term ϕ. By convention, `(assume true)` is used to encode the statement that takes a step but does nothing, `skip`. Analogously, `(assume false)` denotes that the current program execution should no longer be considered at all. In particular, this means that `(assume false)` always terminates, but has no final state [46].

Of course, `assume` is a intended modeling construct that cannot be executed in general. Moreover, even though the intention of such assumptions is to constrain the program's execution, it may have some effect on the interaction of the underlying SMT model with possible states of the program. This implies that the models that initialize the state as part of counterexample traces are

implicitly required to reflect such assumptions. This is intended, for example, when the execution depends on an uninterpreted global constant, such as the length of some array data structure or the assumption of bounded resources.

Assignments. Parallel assignments **(assign** $(x_1\ t_1)$ ... $(x_m\ t_m)$**)**. evaluate the right-hand side terms $t_1, \ldots, t_m$ first in that order and then simultaneously and atomically replace the values of $x_1, \ldots, x_m$ with the results. An assignment is well-formed if the variables are distinct, and the right-hand side types match the corresponding variable types.

Note that neither constants nor input parameters of procedures can be assigned to, and therefore cannot appear on the left-hand side of assignments.

Furthermore, note that SMT-LIB functions can be assigned to variables, but procedures cannot, since we want all their calls to be explicit, as a statement and not occur inside of terms, as in traditional programming languages. Since allowing procedure calls in terms introduces a lot more complexity for verification tools, since they need to somehow decide in what order to handle the calls, and the evaluation of terms would no longer be side-effect free.

Sequential Composition. Sequential composition **(sequence** s_1 ... s_n**)** executes the respective constituents in sequence in the given order. Another way of writing **skip** is to take an empty list of statements in a **sequence**. This is desired, in order to simplify the translation of programs into SV-LIB.

Annotations. The basic language furthermore supports the annotation mechanism from SMT-LIB at the level of statements: **(!** s ... **)** denotes that the command s has additional meta-data attached. There are two-predefined annotations types, tags and properties. For an explanation about allowed properties seeSect. 6 Note that annotations are compositional, i.e., **(! (!** s a**)** b**)** is the same as **(!** s a b**)**.

Tags **:tag** ⟨*symbol*⟩, allow users and front-end tools to mark the (entire) statement s as being of interest to a concern related to the given symbol. It is allowed that the same tag can be attached to multiple places in the program, for example to mark groups of locations that should satisfy a joint property. The primary use case of tags is to link program locations to properties via the **annotate-tag** command. We emphasize that the **:named** annotation from SMT-LIB or other conventions can be used to (uniquely) identify program locations for other purposes, such as the encoding of structured loops into explicit control flow (see below).

Properties are used to express properties of the program. For example, **(!** s **:check-true** ϕ**)** specifies that whenever s is about to execute, formula ϕ should hold, i.e., ϕ is a location invariant at the start of s. With the use of tags, we can liberally and losslessly move such property annotations in and out of programs. Note that when ϕ is moved out of the program by use of tags, and a **annotate-tag** command, all symbols are resolved in the context at the beginning of the tagged statement s.

The distinction between what information is part of statements and what is part of the annotation/property language is driven by the principle that annotations should never be used to describe the semantics of the statements. In particular, tags are orthogonal to labels, where the sole purpose of the latter is to identify jump targets. This also explains, why assumptions are realized by **assume** statements, as they are an essential construct to constrain executions in relation to nondeterministic assignments and choice, whereas assertions are realized separately as **:check-true** annotations, as these reflect properties to be proven in relation to the executions.

In order to export witnesses correctly, it is important that certain crucial statements are marked with script unique tags. These are loops, labels and the outermost statement in a procedure body. Ensuring this is the responsibility of the script generator, not of the solver.

5.2 Procedures (P)

The procedural fragment (P) of SV-LIB comprises statements for procedure calls and for returning from them.

Procedure Call. A procedure call can be made using `(call` ρ `(`t_1 ... t_n`) (`y_1 ... y_m`) )`. This statement calls the procedure ρ, which has to have been introduced by the `define-proc` command previously. The terms $t_1, \ldots, t_n$ denote the values of the input parameters, and the variables $y_1, \ldots, y_m$ receive the output values. A call is well-defined if the respective number and types of input/output arguments match the procedure definition. Moreover, similarly to the assignment statement, the variables $y_1, \ldots, y_m$ need to be distinct.

The semantics of calls is stack-oriented as expected, i.e., each invocation of a (recursive) procedure gets its own copies of the input/output/local variables declared by the procedure. Furthermore, only values are passed from the input arguments to the procedure, and from the procedure back to the output arguments. This means that even if a constant is returned from the function, the caller can assign its value to a variable which can be modified. This means that a symbol being constant or not does not affect its assignability with respect to procedure calls.

A procedure returns either when a `return` statement is executed, or when the end of the procedure body is reached. When returning, the current values of the output variables are passed back to the caller.

Return. The second part of the procedural fragment is the `return` statement. Note that this statement itself does not denote the values of the output parameters of the surrounding procedure, rather, these are given by ordinary assignments, so that the outputs produced are whatever values are currently stored inside the output variables.

5.3 Unstructured Control Flow (U)

The fragment of unstructured control flow (U) is comprised of `label`s and `goto`s with the standard meaning. In case the unstructured control flow fragment is used without the structured control flow fragment, a restricted form of conditional jumps is automatically included into the unstructured control flow fragment.

Labels. Labels `(label` ℓ`)` mark positions in the program that can be jumped to by `goto` statements. Labels ℓ must be unique within each procedure, such that the target of a jump is always unambiguous. Labels are separate constructs from tags, therefore it is possible to have tags and labels with the same name in the same procedure. This may be even desirable to link properties to the label locations.

Goto. Goto statements `(goto` ℓ`)` jump to the position marked by label ℓ. Jumps can only be made within the same procedure, this guarantees that all variables which were in scope at the call-site are still in scope at the target location. In case a label is not found in the current procedure matching the target of a `goto`, the program is considered ill-formed.

Conditional Jumps. In case the unstructured control flow fragment is used without the structured control flow fragment, conditional jumps `(if` ϕ `(goto` ℓ`))` are supported as well. These statements behave the same way as an `if` statement. They are required whenever the structured control flow fragment is not used, to allow for branching behavior.

5.4 Structured Commands (S)

The fragment of structured control flow (S) includes the typical programming constructs `if`, and `while`. In order to support a well-defined translation of `if` and `while` statements into unstructured control flow, a useful convention is to annotate these with `:named` attributes, which gives

them a (unique) name in addition to tags and labels present, from which the labels of entry and exit locations can be generated.

Conditions. For `if` statements, the absence of the second statement in a conditional defaults to `(assume true)` as usual.

Loops. Only `while` loops are supported, since they are sufficient for all cases requiring other types of loops. Tags resp. properties attached as annotations to loops specifically can identify high-level proof principles such as loop invariants for safety properties and well-founded orders for termination properties.

Break and Continue. Statements `break` and `continue` are only allowed to occur inside the scope of a loop. They have the standard meaning of exiting the innermost surrounding loop immediately, and skipping to the next iteration of the innermost surrounding loop, respectively.

5.5 Nondeterminism (N)

For modeling purposes, nondeterminism (N) is a key feature of intermediate languages. We consider two types of nondeterminism, data nondeterminism through `havoc`, and control-flow nondeterminism through `choice`. Note that control-flow nondeterminism can be encoded via data nondeterminism whenever the structured control-flow fragment is available, by using a nondeterministic variable in combination with `if` statements.

All nondeterminism in SV-LIB should be interpreted as potential program behaviors which might occur during execution and hence should be considered during verification. In that regard, all choice is "demonic". Albeit the distinction between demonic and angelic choice only makes sense with respect to a game-like semantics, which definitely goes beyond the scope of SV-LIB. In concrete counterexample traces, choices for `havoc` are resolved to values, similarly, the branch taken by `choices` must be made explicit.

Havoc. The atomic statement `(havoc` $x_1 \ \ldots \ x_n$`)` overwrites the values of (distinct) variables $x_1, \ldots, x_n$ by fresh values. This is done atomically and simultaneously.

Choice. The statement `(choice` $s_1 \ \ldots \ s_n$`)` selects one sub-statements among the $s_1, \ldots, s_n$ to execute. Note that Dijkstra's guarded commands can be written by a combination of `choice` and `assume` statements.

6 SV-LIB Properties

As indicated earlier, properties are kept distinct from the definition of the execution steps. The intention is to clearly separate the two concerns of (a) modeling a system and (b) specifying its desired behaviors. Specifically, for SV-LIB, we aim to avoid encoding specification concerns into the program definition, neither in the form of annotations like assertions, contracts, or invariants [36], nor in the form of, e.g., automata-theoretic constructions like monitoring automata that are composed with the program [14]. The view is that such constructions reflect details of a specific verification approach and therefore such artifacts should be regarded as *output* of a proof procedure instead of being its input.

The design of linking specifications into programs is guided by this view. As shown earlier, specifications are attached as properties to statements via the annotation mechanism borrowed from SMT-LIB. This connection can either be drawn explicitly or alternatively, indirectly by `:tag` annotations, which are used to identify locations in the program that are linked to properties by `annotate-tag` top-level commands (or further mechanisms).

⟨*attribute*⟩ ::=	**:check-true**	⟨*relational-term*⟩	(GB)
|	**:recurring**		(FB)
|	**:not-recurring**		(FB)
|	**:requires**	⟨*relational-term*⟩	(GP)
|	**:ensures**	⟨*relational-term*⟩	(GP)
|	**:invariant**	⟨*relational-term*⟩	(GS)
|	**:decreases**	⟨*relational-term*⟩	(FS)
|	**:decreases-lex**	**(**⟨*relational-term*⟩$^+$**)**	(FS)

Fig. 12. Syntax of property declarations; the fragment which introduces the need for this property, together with their classification as liveness F or safety G properties is indicated on the right-hand side

In its current format, the properties which can be specified about SV-LIB scripts are only LTL properties. Therefore, they are partitioned into safety (G) and liveness (F) properties. Safety properties encode that something bad never happens, while liveness properties ensure that something good eventually happens. In this section, we describe the properties, shown in Fig. 12. In general, we make use of relational terms, which extend SMT-LIB terms by allowing the use of certain constructs at each location a variable symbol is allowed.

Relational Terms. Relational terms are constructed in the same manner as terms, with the addition of **(at** ⟨*symbol*⟩ ⟨*symbol*⟩**)**, which can be used in every location a symbol is allowed. It references the value of a symbol at the beginning of the last statement visited which is marked with the tag referenced by the **at**. The first parameter corresponds to the variable whose value is referenced, and the second parameter corresponds to the tag. This feature can be used to encode for example the specification construct **old** in languages like ACSL [7], JML [43], or Dafny [44] by referencing the tag at the beginning of the procedure body. This tag must always exist in well-formed programs, see Sect. 8.2.

The **at** construct may be applied to symbols representing program variables only, but not to general terms. This avoids assigning meaning to nested **at** constructs and avoids the need to keep track of bound variables in terms, for example **(forall ((k Int)) (at (select a k)** τ**))**, where bound variable **k** in the scope of **at** should *not* be resolved with respect to the state at the earlier occurrence of tag τ. Front-end tools for SV-LIB are encouraged to support more general compositions using **at** by compiling them into this lower-level form.

6.1 Location Invariants (G)

The basic safety property consists of a location invariant, denoted by the attribute **:check-true**. While in most programming languages location invariants are characterized by assertions, we choose to use **:check-true** instead in order to make the distinction between the SMT-LIB **assert** and location invariants clearer. A location invariant is always evaluated in the state before executing the statement it is attached to, with the exception of **while** loops, and **label** statements. Counterexamples to location invariant violations are expressed by counterexamples to safety, see Sect. 7.2.

Some clarification should be made about the interpretation when s is a **(while** ... **)** statement. Should ϕ be checked only once or on every iteration? Arguments can be made in favor of both views, but we find it more pragmatic to repeat the check each time when the loop condition is evaluated. The reasoning why this is the better option is as follows: First, it gives a convenient way to specify

semantic loop invariants (without requiring them to be inductive). Second, when translating loops into goto programs or CFA representation, the annotation can simply be attached to the loop head without the need to distinguish the first visit from later ones. The same arguments apply for **`label`** statements, where we decide that the assertion should be checked each time the label is jumped to.

6.2 Invariants (G)

For (un)structured control-flow, we support invariants using the attribute **`:invariant`**. Invariants are required to be fulfilled, the first time the loop is entered, inductive, and be strong enough to prove the desired properties if the loop is abstracted by the invariant. This applies to both structured **`while`** loops and unstructured loops using labels and gotos.

Similarly to loops, annotating loop-heads, i.e., labels with **`:invariant`**s, can help break cycles in such unstructured programs to achieve modular verification. There are multiple interpretations for where the invariants should be in order to break cycles, at least they should be at one node inside each cycle, and at most at each label. Due to these interpretations not providing clear advantages over each other, we leave the precise choice to the tools generating or processing SV-LIB scripts, and aim to provide a more formal definition in a future version of the format. In particular, if a verifier does not produce invariants for labels not considered loop-heads/contributing to cycles, invariants can be generated for them from other existing annotations using strongest post-conditions and weakest pre-conditions.

We use a weak notion of inductivity, which is with respect to all the variables being modified within the loop. This means, that if we start in the state where the loop is entered the first time, then **`havoc`** all modified variables inside the loop, execute the body of the loop, then the loop invariant must hold. In particular, this means that variables which are not modified inside the loop, will retain the same value as before the loops execution even when abstracting the loop using the invariant. This includes, but is not limited to, global variables, and constants.

Note that finding out which variables are modified inside the loop is purely a syntactic task, since simply all variables on the left side of an assignment, or as output of a procedure call inside the loop body, are considered to be modified. This is possible since SV-LIB does not have a heap, for which alias analysis would be required. In particular, for arrays, we consider the whole array to be modified, regardless the precise locations written to. In this case, the invariant must express through the use of quantifiers and relational terms that certain elements of the array retain their value.

Whenever the over-approximation given by the invariant is applied, for example, during inductiveness constraint, **`continue`** and **`break`** statements need to be handled correctly. When a **`continue`** statement is executed, then the invariant must hold again, producing no further verification conditions for the continued execution after the loop. In contrast, when a **`break`** statement is executed, then the invariant does not need to hold anymore, and the execution continues after the loop body. In particular, this means that abstracting a loop using its invariants can generate multiple successor states, depending on the number of **`break`** statements inside the loop body.

Having three properties which need to be proven correct, for an invariant to be valid, there are also three types of counterexamples which are possible. If the invariant does not hold when the loop is entered the first time, a counterexample to safety, see Sect. 7.2 must be exported, like for assertions. If the invariant is not inductive, a counterexample to modular specifications must be exported, see Sect. 7.3. Finally, if the invariant is not strong enough to proof the rest of the program correct, the same type of counterexample as for inductivity must be exported.

6.3 Statement Contracts (G)

Procedure contracts are represented in terms of more general statement contracts, defined by the pair of attributes **`:requires`** and **`:ensures`**. The **`:requires`** attribute denotes the pre-condition

of the statement, while the **:ensures** attribute denotes the post-condition of the statement. The **:requires** attribute is evaluated in the state directly *before* executing the statement it is attached to, while the **:ensures** attribute is evaluated in the state directly *after* executing the statement.

Statement contracts are checked inductively similarly to invariants:

First, the pre-condition given by the **:requires** attribute must hold in the initial state before the statement is executed. Such preconditions can be violated in the same way as **:invariant**s by a trace potentially containing **leap** steps to abstract an arbitrary number of recursive calls towards a the violation Sect. 7.3 when the contract is attached to the top-level statement of a procedure.[2]

Second, the contract must have a modular proof with respect to the surrounding context and the statically-determined set of local/global variables modified within the statement. This means that whenever the statement is executed with all variables, which the statement modifies, being nondeterministic and the chosen variables fulfill the **:requires** attribute, then at the end of the statements execution, the **:ensures** attribute must hold. Moreover, in the context of recursive procedures, the statement contract can make use of inductive hypotheses about recursive calls as usual.

Third the post-condition given by the **:ensures** attribute must be strong enough to prove the remainder of the program correct. Whenever this does not hold a counterexample to modular specifications must be exported, see Sect. 7.3.

6.4 Ranking Functions (F)

Ranking functions, expressed by the attribute **:decreases**, can be used to prove termination of loops or procedures. In the simple form, the term of the attribute must be a term of SMT-LIB **Int** sort, Whenever the statement is executed, the value of the term must decrease strictly according to the built-in order **<** but remain non-zero. We do not aim to support further primitive well-founded orders, rather other orders should be mapped to integers explicitly, for example by converting bit-vectors to their numeric values, or by taking the cardinality of a finite set.

A second form with attribute **:decreases-lex** is supported to denote compound well-founded orders by lexicographic combination of integer terms. The leftmost entry is the most significant one. The order is defined by a (possibly-empty) prefix of the entries that remain unchanged, followed by an entry that decreases strictly.

Whenever the ranking function does not decrease on some execution, or falls outside the designated range of values such as non-negative integers, a counterexample to this must be exported, see Sect. 7.3.

6.5 (Not-)Recurring Locations (F)

We have two main liveness properties, **:recurring** and **:not-recurring**. They are used to annotate tags, denoting locations which must be visited infinitely often (recurring) or only finitely many times (not-recurring) on all infinite paths. For finite paths, both properties are considered to be trivially satisfied, which will become clear from the formalization of the semantics in a future version of the format. For recurring locations, we can decide which location of the tag group to visit in each iteration, i.e., not all of them need to be visited infinitely often. In particular, these liveness properties are implicitly satisfied by finite runs.

Guaranteed termination of a program can be encoded by annotating each loop and function head as **:not-recurring**. An alternative formalization is to assume an implicit self-loop at the

[2]We leave the use of these annotations for loop contracts [32] for future work, since these are not widely-used. The rationale is that within loop contracts, referencing variables should be consistent across pre- and postcondition, i.e., it would be preferential to let x in **:ensures** annotations to refer to the arbitrary intermediate execution, not the final one state, which is incompatible with the established convention for procedure contracts. Unfortunately, **(at** x ℓ**)** does not have the right semantics to encode the reference the intermediate state when ℓ is the tag annotated to the loop.

end of the program which does nothing but has the implicit tag **`exit`** that occurs only there, in which case termination is encoded by this tag being **`:recurring`**. We claim that both approaches are equivalent, also with respect to validation.

In contrast to safety counterexamples, liveness counterexamples are oftentimes more involved, since they require an argument that the system always makes progress. This usually requires a combination of invariants and ranking functions. Counterexamples for liveness are discussed in Sect. 7.4.

7 SV-LIB Witnesses

A witness is a sequence of commands, which can be combined with the original script to generate a new valid SV-LIB script. This guarantees that witness validation is the same task as verification, but aided by the witness, making it possible to reuse verification tools for validation. We distinguish between correctness, and violation witnesses. Correctness witnesses certify that a given property holds, whereas violation witnesses certify that a given property is violated.

Since both correctness and violation witnesses are comprised of several components, we delimit the response of the verifier inside a pair of parentheses. This allows clients to reliably detect the end of a multi-line response without accidentally blocking on the communication channel and it allows tools in general to detect incomplete witnesses stored in files. We emphasize that witnesses are part of the *response* of the verifier to a script and as such they are not valid as commands in a SV-LIB script.

$$\langle \textit{witness} \rangle ::= \langle \textit{correctness-witness} \rangle \mid \langle \textit{violation-witness} \rangle$$

The goal of a witness is to provide sufficient information to make the validation task inherently simpler than the original verification task. For correctness witnesses, this is achieved by providing inductive invariants, modular contracts, and ranking functions. These allow the direct generation of verification conditions in first-order logic, which can be checked by an SMT solver. For violation witnesses, this is achieved by providing a concrete execution trace, whenever possible, that leads to a property violation. This allows the validation to be done by concrete execution instead of symbolic reasoning. The only exception is when a liveness property is violated, where a lasso-shaped execution is required to witness the violation, which again can be checked by an SMT solver.

In addition to the information to replay the verification verdict, a witness may also contain metadata, like the tool that generated it, the time of generation, and other useful information for debugging and tracking purposes. For this, we use the SMT-LIB **`set-info`** command to set arbitrary attributes. We recommend setting at least the **`:producer`** and, **`:input-files`** attributes to indicate the tool that generated the witness, and the input files which composed the verification task for which the witness was created.

$$\langle \textit{metadata} \rangle ::= \texttt{(set-info } \langle \textit{attribute} \rangle \texttt{)}^*$$

This section presents mostly the syntax of witnesses, their semantics is kept informal for now, but will be made precise in a future version of this document. Section 7.1 describes correctness witnesses, Sect. 7.2 describes counterexamples to safety, and Sect. 7.3 describes counterexamples to modular specifications, like inductiveness of invariants, modularity of contracts, and the decreasing property of ranking functions, and finally Sect. 7.4 describes counterexamples to liveness.

Whenever a violation witness is validated, it contains a trace which needs to be fulfilled. This requires the trace to not only be syntactically valid w.r.t. the program, but also terminate, and reach the violation. For correctness witnesses, the annotations provided could contain unsound information like **`(assert true)`**. Invalid witnesses are discussed in Sect. 7.5, for both correctness, and violation witnesses. For most of these cases, exporting a witness showing the invalidity of the program composed of the original script and the witness is left for future work. Therefore, it is not possible to export a witness for all scripts describing exactly what went wrong during

$$\langle \textit{correctness-witness} \rangle ::= \texttt{(} \langle \textit{metadata} \rangle^* \; \langle \textit{smt-lib-command} \rangle^* \; \texttt{(annotate-tag } \tau \; a_1, \ldots, a_n \texttt{)}^* \texttt{)}$$

Fig. 13. Syntax of correctness witnesses

validation. A special case, where the witness can no longer be followed, since some step through the program cannot be realized, e.g., choosing a value outside the allowed range of a **choice** statement, is discussed in Sect. 7.6.

7.1 Correctness Witnesses

For both safety and liveness properties, a correctness witness is a sequence of ⟨*smt-lib-command*⟩s followed by a sequence of **annotate-tag** commands, as shown in Fig. 13. The SMT-LIB commands can be used to provide auxiliary definitions, e.g., abbreviations for predicates used in invariants, frame conditions, or theory extensions, e.g., to express summations over array ranges. These may be needed to express facts mathematical facts to be used in the annotations proving the program correct. Of the commands shown in Fig. 10, SV-LIB accepts all those except for those starting with **get-** and **set-**.

We point out that the **assert** command is to be considered "dangerous", insofar that only *conservative* extensions to the theory should be allowed here, e.g., postulating **(assert false)** reduce the potential models to the empty set and thus vacuously make the specification true. We leave this problem for the future and suggest for the use of SV-LIB in competitions to simply forbid the **assert** command here. However, note that SMT-LIB has other ways to introduce inconsistency into the specification, for example by non-terminating functions declared with **define-fun-rec**.

Afterwards, the sequence of **annotate-tag** commands, provides a modular abstraction for loops via invariants, procedures via contracts, and ranking functions for liveness properties.

To be able to annotate the required program locations, the program is required to have tags at (a) the top-level statement in each procedure, (b) each loop head, and (c) each label. These conditions can be checked by a linter, as discussed in Sect. 8.2.

7.2 Safety Violations

A violation of a safety property is witnessed by a collection of concrete counterexample traces. Each trace indications one way to violate some property of the specification. The grammar is shown in Fig. 14. Each trace is a sequence of entries that describe the initial conditions at the falsified **verify-call** followed by a detailed description of how to resolve the non-determism towards the indicated property violation.

Specifically, the first part of a trace is a trace-specific SMT-LIB model, comprised of several SMT-LIB definitions (see ⟨*model-response*⟩ in [6, Sec. 3.11.1]), covering notably all global constant symbols. The intention of including the model is to allow validation by concrete execution. This model is paired with an initialization of some of global variables by name, where **(init-global-vars** $(x_1\ v_1)\ \cdots\ (x_n\ v_n)$**)** can leave out those global variables whose initial value is irrelevant for the concrete execution, like those global variables which are guaranteed to be assigned or havoced explicitly before being read, on that execution path.

For good measure, the trace indicates the entry-point into the program in terms of the procedure name of the corresponding **verify-call**. We emphasize that listing the concrete values of input parameters here is not necessary, as we have full information to determine these from the model

$$
\begin{aligned}
\langle \textit{violation-witness} \rangle &::= \texttt{(}\langle \textit{metadata} \rangle\ \texttt{(select-trace}\ \langle \textit{trace} \rangle\texttt{)}^*\texttt{)} \\
\langle \textit{trace} \rangle &::= \texttt{(model}\ \langle \textit{model-response} \rangle^*\texttt{)} \\
&\quad\ \texttt{(init-global-vars (}\langle \textit{symbol} \rangle\ \langle \textit{value} \rangle\texttt{)}^*\texttt{)} \\
&\quad\ \texttt{(entry-proc}\ \langle \textit{symbol} \rangle\texttt{)} \\
&\quad\ \texttt{(steps}\ \langle \textit{step} \rangle^*\texttt{)} \\
&\quad\ \langle \textit{violated-property} \rangle \\
&\quad\ \texttt{(using-annotation}\ \langle \textit{symbol} \rangle\ \langle \textit{attribute} \rangle^+\texttt{)}^* \\
\langle \textit{step} \rangle &::= \texttt{(init-proc-vars}\ \langle \textit{symbol} \rangle\ \texttt{(}\langle \textit{symbol} \rangle\ \langle \textit{value} \rangle\texttt{)}^*\texttt{)} \\
&\quad\ |\ \ \texttt{(havoc (}\langle \textit{symbol} \rangle\ \langle \textit{value} \rangle\texttt{)}^*\texttt{)} \\
&\quad\ |\ \ \texttt{(choice}\ k\texttt{)} \\
&\quad\ |\ \ \texttt{(leap}\ \langle \textit{symbol} \rangle\ \texttt{(}\langle \textit{symbol} \rangle\ \langle \textit{value} \rangle\texttt{)}^*\texttt{)} \\
\langle \textit{violated-property} \rangle &::= \texttt{(incorrect-annotation}\ \langle \textit{symbol} \rangle\ \langle \textit{attribute} \rangle^+\texttt{)} \\
&\quad\ |\ \ \texttt{(invalid-step}\ \langle \textit{step} \rangle\texttt{)}
\end{aligned}
$$

Fig. 14. Syntax of concrete traces for counterexamples

and global variables already. This idea is applied to representation of subsequent execution steps, defined by ⟨*step*⟩, insofar that only additional information is listed that is required to resolve any type of (relevant) non-determinism. The listed ⟨*step*⟩s are thus matched eagerly to the statements occuring in the execution trace as followed.

A **(init-proc-vars** ρ **(**x_1 v_1**)** $\cdots$ **(**x_n v_n**))** step is applied right after a call to procedure with name ρ, initializing some of the local variables as well as output variables. Similarly to the **init-global-vars** trace entry, we may omit those variables whose initial value is irrelevant. The rationale is that often, local variables and outputs are written explicitly, thereby avoiding the need that a verifier invents arbitrary initial values in such cases, just to have these overwritten immediately. Note, we including the procedure name ρ for debugging purposes, it should be recoverable during execution, and the trace should be rejected if a call with a mismatching name is encountered.

The same idea is applied to **havoc** step, which are matched by **havoc** statments but are presented here rather as an assignment to those variables for which the initialization matters. A **choice** step selects the branch leading to the violation from a **choice** statement by index (starting at 0).

A violation is claimed by trace elements **(incorrect-annotation** τ $a_1 \dots a_n$**)** where τ is a tag uniquely identifying a position in the program and $a_1 \dots a_n$ should all be violated by this trace. Checking whether that annotation occurs syntactically exactly as part of the original annotation is an aspect of witness validation. It may be possible to relax this check in the future and to permit some reasoning, for example to allow signalling a violation of ϕ for the specification **:check-true (**and ϕ ψ**)**, as the latter is semantically equivalent to the annotations **:check-true** ϕ **:check-true** ψ.

Steps **leap** are only allowed to occur in counterexamples to inductiveness, they will be described in Sect. 7.3. Similarly, **using-annotation** at the end of a trace is only allowed to occur in counterexamples to liveness as explained in Sect. 7.4. Finally, **invalid-step**s refers to previously-claimed violation traces that were found to be spurious as explained in Sects. 7.5 and 7.6.

Confirming that traces lead to a violation should by design be possible with concrete execution. However, a bad witness could deliberately lead such an approach into an infinite loop. This problem is left for the future, e.g., by allowing for some approximation of loop and recursion counters as part of the trace. Another future extension to SV-LIB would be a symbolic alternative to describe counterexample traces, for example, that describe the program's unrolling towards the violation.

7.3 Violations of Modular Specifications

In contrast to counterexamples for **`:check-true`** annotations, **`:invariant`**s and statement contracts are intended not just to express true properties of all executions but rather admit an inductive proof. This is a stronger property that is required to hold potentially on executions that do not exists in reality but which are implicitly included from the overapproximation inherent to such constructs. The same is true for ranking functions, which are supposed to decrease in each iteration.

However, one should not be forced to include as part of invariants those properties which are obviously stable across loop iterations, notably when they are over variables not modified by the loop. This leads to a somewhat subtle notion of inductiveness with respect to the given context. As explained more detail in Sect. 6.2.

As a consequence, counterexamples to inductiveness, and the decreasing property of ranking functions cannot just be execution traces. Instead, one needs to be able to over-approximate the effect of loops and statement contracts in a way that is consistent with the given annotations. Through this, one ends up in a state from which a violation of the respective property can be observed. This is achieved by the step inside a trace **`(leap`** ⟨*symbol*⟩ **`(`**⟨*symbol*⟩ ⟨*value*⟩**`)`*****`)`** which once the provided tag is reached, transitions into an arbitrary state at that tag, where the values of the given symbols are set to the given values. This allows the counterexample to go into a hypothetical state that may not be reachable by a concrete execution but which is consistent with the over-approximation inherent to loops and statement contracts. This requires the state given by the **`leap`** step to satisfy all annotations at the given tag, notably the invariants at loop heads, and the pre-conditions of statement contracts. After executing a **`leap`** step, the execution continues as normal. The **`leap`** command must be used as soon as the corresponding tag is reached, and before executing any statement at that tag.

The grammar of counterexamples to modular specifications, is given in Fig. 14, but in this case the **`leap`** step is allowed as a step inside the trace, to over-approximate the effect of loops and contracts. Nonetheless, **`using-annotation`** constructs are still not allowed here, and the violated property must still always be **`incorrect-annotation`**.

Note that the **`leap`** step should contain all, and only those variables which are modified by the loop or statement inside the contract. According to our understanding of inductiveness, all of these variables may take arbitrary values.

7.4 Liveness Violations

A violation of a liveness property is witnessed by a lasso, which has a concrete trace as a stem, just as for counterexamples to safety, that navigates to the head of a loop that, if it has infinite executions, fails to validate the property. Their syntax is shown in Fig. 14, where all the constructs in the grammar are allowed, including **`leap`** and **`using-annotation`**, and the violated property must always be **`incorrect-annotation`**.

The stem of the lasso is given by a **`(select-trace`** ⟨*trace*⟩**`)`** command, which navigates to the tag which corresponds to the head of the loop resulting in the lasso which witnesses the violation of the liveness property. For the stem to be valid, the trace must result in a state satisfying all annotations set by the **`using-annotation`** elements for the tag at the end of the trace. In case no such tag exists, **`true`** is assumed.

The lasso itself is described by a sequence of **`(using-annotation ...)`** elements. These elements describe how to reach the head of the loop again, by using the annotations to inductively abstract everything in between, and using ranking functions to ensure progress towards visiting the head of the lasso. For the liveness property to be violated, all paths starting in an arbitrary state at the head of the lasso fulfilling all annotations set by the **`using-annotation`** elements there, must always eventually return to the head of the lasso by abstracting everything in between using the annotations, fulfilling all annotations there again.

Decoupling the original annotations from those needed to witness a liveness violation is crucial to allow for the verifier to detect liveness bugs even if the original safety annotations are incorrect, too. In particular, this requires the solver to ignore already existing annotations, and to only consider those set by the **`using-annotation`** elements inside the **`select-trace`** command.

7.5 Invalid Witnesses

Buggy verifiers could produce witnesses that do not make much sense, for example, when annotations do not refer to existing locations, which is easy enough to detect by syntactic checks. Moreover, all correctness witnesses only extend the set of annotations, which means that a problematic witness can be identified and flagged by the standard means of verification, and a counterexample to a supposed correctness witness can be represented and re-checked. However, we currently do not have a certifiable way to refute that counterexample traces exist in the program. The problem is on one hand more subtle (lack of monotony when extending the verification task) but on the other hand simpler, because detecting the non-existence of traces requires simple techniques in theory (i.e., concrete execution). To showcase how such cases could be handled, the case of not being able to execute a ⟨*step*⟩ in a trace is discussed in Sect. 7.6.

We therefore reserve the standard response **`error(cannot export witness)`** to denote invalid programs where the reason for the invalidity cannot be expressed using SV-LIB. Verifiers are encouraged to include a human-readable error message.

7.6 Violations to Step Execution

Whenever a SV-LIB script contains a **`select-trace`** command, it is possible that the trace cannot be executed because some step cannot be fulfilled. For example, a **`choice`** step selects a value outside the allowed range. Some of these constraints cannot be checked syntactically, but require some semantic reasoning, by executing the program along the trace. When such a situation occurs, the verifier should return **`incorrect`**, and export a violation witness that indicates that the trace cannot be executed. The violation witness should correspond to a safety violation witness, see Sect. 7.2, where the violated property at the end of the trace is **`invalid-step`** indicating the step that could not be executed.

8 Well-Formed SV-LIB Programs

The syntax presented in this document defines a large set of possible SV-LIB programs. However, not all syntactically valid programs are appropriate as a verification task. For example, a verifier may want to export an invariant, but have no tag to attach it to, leaving it unable to export a correctness witness. Therefore, we define a set of conditions that a SV-LIB program should satisfy in order to be well-formed. These rules will also be implemented in a linter that can be used to check SV-LIB programs automatically.

The basis for SV-LIB programs are S-expressions as defined in [6, Sects. 3.1–3.4]. For SV-LIB scripts, we distinguish between two levels of *well-formedness* conditions. Structural and typing conditions (Sect. 8.1) which ensure that the program is executable, and conditions for *full well-formedness*

(Sect. 8.2) which in addition ensure that all annotations can be meaningfully interpreted by a verifier or validator, ensuring that the program represents a meaningful verification task.

8.1 Structural Conditions and Typing

The conditions outlined in this section aim to ensure that each term, statement, and command makes sense semantically and thus guarantee that the program to be executed.

Here, we deliberately exclude checking any property specifications annotated inline in the program or using the top-level **`annotate-tag`** command. The first (minor) difficulty with the latter is that the terms don't occur within their respective lexical scopes. More importantly, however, these terms can contain syntactic constructs beyond those in SMT-LIB, specifically references to certain execution states via **`at`**, which may need some further context to be resolved or encoded and that context may be specific to the verification tool or methodology.

All of these conditions should be checked by a compliant parser during the parsing process. The selection is such that this is relatively straight-forward, both conceptually as well as from an implementation view.

(a) it is built according to the syntax defined in this document,
(b) all requirements inherited from SMT-LIB are respected,
(c) all functions and procedures are declared, and/or defined before they are used, except for groups of explicit mutual recursion like those marked with **`define-procs-rec`**,
(d) no symbol int he script, i.e., variable, function, tags, or procedure name, starts with the reserved symbol prefix #,
(e) all terms representing a condition of a statement, like **`if`**, **`assume`**, and **`while`** must be of the boolean sort,
(f) the same procedure name is not used for two **`define-proc`** commands,
(g) labels of **`label`** statements need to be uninque within a procedure (but not necessarily globally unique),
(h) labels of **`goto`** statements must exist within the enclosing procedure,
(i) all variables used in terms, statements, and top-level commands are in scope,
(j) all terms, statements, and top-level commands are type-correct,
(k) all procedure calls provide the correct number of arguments,
(l) variables occurring as left-hand sides of assignments or recipients of procedure returns are allowed to be modified: global, outputs, and local variables, of the enclosing procedure, but *not* procedure inputs

8.2 Fully Well-formed SV-LIB Programs

The additional constraints outlined below are concerned with characterizing "useful" verification tasks. In particular, these conditions guarantee that all annotations can be meaningfully interpreted by a verifier or validator. The requirements on the presence of certain tags ensures that important program locations can be referenced by witnesses.

(a) the program is structurally valid and type correct as defined in Sect. 8.1
(b) program variables occurring in annotations must be in scope of all locations in the program they refer to, via the **`:tag`**-based linking mechanism
(c) function symbols occurring in inline statement annotations must have been declared before the respective statement
(d) function symbols occurring in **`annotate-tag`** commands must be declared before that command, regardless of where the respective tag may occur (see Sect. 4.4).

(e) all terms for the property annotations **`:invariant`**, **`:check-true`**, **`:requires`**, and **`:ensures`** must be type-correct and of the boolean sort,
(f) all terms in a **`:decreases`** and **`:decreases-lex`** annotation must be type-correct and of the integer sort,
(g) all variables used in annotations are in scope at the respective tag,
(h) **`at`** references can be statically resolved,
(i) the top-level statement in every procedure, each loop, and every label has a script-unique tag, and
(j) each statement with an inline annotation has at least one script-unique tag.

8.3 Warnings and Code Smells

Furthermore, there are some SV-LIB scripts which while well-formed present common pitfalls, which may indicate a problem in the tool creating or processing the SV-LIB scripts. In particular, linters are encouraged to warn in case:

(a) some local variable is read before it has been explicitly initialized or havoced,
(b) some output variables are not initialized at procedure return,
(c) the version of the format is missing using a **`set-info`** command,
(d) the export of witnesses is not explicitlyy enabled or disabled using a **`set-info`** command.

9 Current Tools and Libraries

One of the primary goals of SV-LIB is to facilitate interoperability between different verification tools. To this end, we provide some tools and libraries that support working with SV-LIB programs. We envision that these can be used and extended by the community to build a rich ecosystem around SV-LIB aiming to fulfill the vision presented in Fig. 1. We recommend looking at the official website of SV-LIB https://gitlab.com/sosy-lab/benchmarking/sv-lib for the latest information on available tools and libraries.

9.1 Python Library: PySvLib

For working with SV-LIB programs in Python, we provide the PySvLib library[3] as a python package. This package provides some common functionality for parsing, analyzing, and manipulating SV-LIB programs. Currently it contains a parser that translates SV-LIB scripts into an abstract syntax tree (AST). In addition, it contains a printer which serializes the AST back into a string. Furthermore, it includes a verification condition generator which can be used to implement a validator for SV-LIB programs and witnesses. We defer further details to the documentation of the library itself.

9.2 ANTLR Grammar

While python is a popular programming language, other languages are also used in the verification community. To facilitate working with SV-LIB programs in other programming languages, we provide an ANTLR grammar from which a parser for most languages can be easily generated. The ANTLR grammar for SV-LIB programs[4] is based on an existing SMT-LIB grammar[5], which it extends by adding the additional commands, statements, and properties defined in this document.

[3] https://gitlab.com/sosy-lab/benchmarking/sv-lib/-/blob/format-1.0/pysvlib
[4] https://gitlab.com/sosy-lab/benchmarking/sv-lib/-/blob/format-1.0/grammar/SvLib.g4
[5] https://github.com/antlr/grammars-v4/blob/5660ba571209e7c28c0e36c38414729e5b6db087/smtlibv2/SMTLIBv2.g4

9.3 SV-LIB in CPAchecker

The software verification tool CPAchecker [4] is a mature software verification framework which allows developers to easily implement new analyses and verification techniques. Furthermore, its architecture allows developers to integrate new input languages while making it possible to reuse existing analyses and components with only little extra effort. Therefore, we have integrated SV-LIB as an input language into CPAchecker to enable development of further analyses on a mature code-base.

This integration includes a front-end that translates SV-LIB programs first into an AST representation of the program, which is then translated into CPAchecker's internal CFA representation. The integration also includes a translation to and from SV-LIB terms into SMT-LIB formulas using the JavaSMT [41] interface. Finally, we include support for exporting witnesses by translating CPAchecker's analysis result in form of an abstract reachability graph (ARG) or a counterexample into a SV-LIB correctness or violation witness. With these components we implemented support for SV-LIB in CPAchecker's predicate analysis. Our implementation was integrated into the release 4.2 of CPAchecker [24].

10 Conclusion

We introduce the intermediate language SV-LIB as a standard exchange format for software-verification tasks. The language covers commands (Sect. 4), statements (Sect. 5), properties (Sect. 6), and witnesses (Sect. 7). The goal of SV-LIB is to be an intermediate language for imperative programs with the goal of serving as exchange format between verification frontends and backends. This allows to decouple the support of more programming languages (and their features) from the implementation of verification algorithms, working on the mathematical structure of programs. Furthermore, we aim to provide a standard format for different communities, focusing on the automatic software verification and deductive verification communities. We hope that SV-LIB will be adopted by the community and will foster the exchange of verification tasks between different tools and communities.

Data-Availability Statement. The latest information on SV-LIB can be found online at the official website https://gitlab.com/sosy-lab/benchmarking/sv-lib. This includes an `ANTLR` grammar for SV-LIB and a `pip` package with useful functionality to work with SV-LIB scripts.

Funding Statement. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and the Free State of Bavaria.

Acknowledgments. SV-LIB was designed, and this article was written, as direct response to a proposal made during a recent Dagstuhl seminar [15], in which a group of participants has proposed to design an exchange format like SV-LIB. We thank the community for the ideas, requirements, and support towards this proposal. We are grateful for the inspiration that we received from K2 [35]. The development name for our language was K3, before we finalized it to SV-LIB.

References

[1] L. Armborst, D. Beyer, M. Huisman, and M. Lingsch-Rosenfeld. 2025. AutoSV-Annotator: Integrating Deductive and Automatic Software Verification. In *Proc. FMICS 2025 (LNCS 16040)*. Springer, 59–77. doi:10.1007/978-3-032-00942-5_4

[2] S. Ates, D. Beyer, P.-C. Chien, and N.-Z. Lee. 2025. MoXIchecker: An Extensible Model Checker for MoXI. In *Proc. VSTTE 2024 (LNCS 15525)*. Springer, 1–14. doi:10.1007/978-3-031-86695-1_1

[3] P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček. 2024. Software Verification Witnesses 2.0. In *Proc. SPIN (LNCS 14624)*. Springer, 184–203. doi:10.1007/978-3-031-66149-5_11

[4] D. Baier, D. Beyer, P.-C. Chien, M.-C. Jakobs, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, H. Wachowitz, and P. Wendler. 2024. Software Verification with CPAchecker 3.0: Tutorial and User Guide. In *Proc. FM (LNCS 14934)*. Springer, 543–570. doi:10.1007/978-3-031-71177-0_30

[5] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. FMCO (LNCS 4111)*. Springer, 364–387. doi:10.1007/11804192_17

[6] C. Barrett, P. Fontaine, and C. Tinelli. 2024. *The SMT-LIB Standard: Version 2.7*. Technical Report. University of Iowa. https://smt-lib.org/papers/smt-lib-reference-v2.7-r2025-07-07.pdf.

[7] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. 2021. ACSL: ANSI/ISO C Specification Language Version 1.17. Available at https://frama-c.com/download/acsl-1.17.pdf.

[8] D. Beyer, P.-C. Chien, M. Jankola, and N.-Z. Lee. 2024. A Transferability Study of Interpolation-Based Hardware Model Checking for Software Verification. *Proc. ACM Softw. Eng.* 1, FSE, Article 90 (2024), 23 pages. doi:10.1145/3660797

[9] D. Beyer, P.-C. Chien, and N.-Z. Lee. 2023. Bridging Hardware and Software Analysis with Btor2C: A Word-Level-Circuit-to-C Translator. In *Proc. TACAS (2) (LNCS 13994)*. Springer, 152–172. doi:10.1007/978-3-031-30820-8_12

[10] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. 2009. Software Model Checking via Large-Block Encoding. In *Proc. FMCAD*. IEEE, 25–32. doi:10.1109/FMCAD.2009.5351147

[11] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. doi:10.1145/2950290.2950351

[12] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. doi:10.1145/2786805.2786867

[13] D. Beyer, M. Dangl, and P. Wendler. 2015. Boosting k-Induction with Continuously-Refined Invariants. In *Proc. CAV (LNCS 9206)*. Springer, 622–640. doi:10.1007/978-3-319-21690-4_42

[14] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast. *Int. J. Softw. Tools Technol. Transfer* 9, 5-6 (2007), 505–525. doi:10.1007/s10009-007-0044-z

[15] D. Beyer, M. Huisman, J. Strejček, and H. Wehrheim. 2025. Report from Dagstuhl Seminar 25172: Information Exchange in Software Verification. *Dagstuhl Reports* (2025).

[16] D. Beyer, M. E. Keremoglu, and P. Wendler. 2010. Predicate Abstraction with Adjustable-Block Encoding. In *Proc. FMCAD*. FMCAD, 189–197. https://dl.acm.org/doi/10.5555/1998496.1998532

[17] D. Beyer, N.-Z. Lee, and P. Wendler. 2025. Interpolation and SAT-Based Model Checking Revisited: Adoption to Software Verification. *J. Autom. Reasoning* 69, 1 (2025), 5. doi:10.1007/s10817-024-09702-9

[18] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. 2013. Precision reuse for efficient regression verification. In *Proc. FSE*. ACM, 389–399. doi:10.1145/2491411.2491429

[19] Dirk Beyer and Andreas Podelski. 2022. Software Model Checking: 20 Years and Beyond. In *Principles of Systems Design (LNCS 13660)*. Springer, 554–582. doi:10.1007/978-3-031-22337-2_27

[20] D. Beyer, M. Spiessl, and S. Umbricht. 2022. Cooperation Between Automatic and Interactive Software Verifiers. In *Proc. SEFM (LNCS 13550)*. Springer, 111–128. doi:10.1007/978-3-031-17108-6_7

[21] D. Beyer and J. Strejček. 2025. Improvements in Software Verification and Witness Validation: SV-COMP 2025. In *Proc. TACAS (3) (LNCS 15698)*. Springer, 151–186. doi:10.1007/978-3-031-90660-2_9

[22] D. Beyer and H. Wachowitz. 2024. FM-Weck: Containerized Execution of Formal-Methods Tools. In *Proc. FM (LNCS 14934)*. Springer, 39–47. doi:10.1007/978-3-031-71177-0_3

[23] D. Beyer and H. Wehrheim. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In *Proc. ISoLA (1) (LNCS 12476)*. Springer, 143–167. doi:10.1007/978-3-030-61362-4_8

[24] D. Beyer and P. Wendler. 2025. *CPAchecker Release 4.2 (unix)*. doi:10.5281/zenodo.17698608

[25] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. 1999. Symbolic Model Checking without BDDs. In *Proc. TACAS (LNCS 1579)*. Springer, 193–207. doi:10.1007/3-540-49059-0_14

[26] N. S. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (LNCS 9300)*. Springer, 24–51. doi:10.1007/978-3-319-23534-9_2

[27] A. R. Bradley. 2011. SAT-Based model checking without unrolling. In *Proc. VMCAI (LNCS 6538)*. Springer, 70–87. doi:10.1007/978-3-642-18275-4_7

[28] P.-C. Chien and N.-Z. Lee. 2024. CPV: A Circuit-Based Program Verifier (Competition Contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 365–370. doi:10.1007/978-3-031-57256-2_22

[29] A. Cimatti, A. Griggio, and S. Tonetta. 2022. The VMT-LIB Language and Tools. In *Proc. SMT (CEUR Workshop Proceedings, Vol. 3185)*. CEUR-WS.org, 80–89. https://ceur-ws.org/Vol-3185/extended9547.pdf

[30] E. M. Clarke, D. Kröning, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. TACAS (LNCS 2988)*. Springer, 168–176. doi:10.1007/978-3-540-24730-2_15

[31] Rob DeLine and Rustan Leino. 2005. *BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs*. Technical Report MSR-TR-2005-70. Microsoft Research. https://www.microsoft.com/en-us/research/publication/boogiepl-a-typed-procedural-language-for-checking-object-oriented-programs/

[32] G. Ernst. 2022. Loop Verification with Invariants and Contracts. In *Proc. VMCAI (LNCS, Vol. 13182)*. 69–92. doi:10.1007/978-3-030-94583-1_4

[33] J.-C. Filliâtre and A. Paskevich. 2013. Why3: Where Programs Meet Provers. In *Programming Languages and Systems*. Springer, 125–128. doi:10.1007/978-3-642-37036-6_8

[34] Carlo A Furia and Abhishek Tiwari. 2024. Challenges of multilingual program specification and analysis. In *Proc. ISoLA (LNCS 15221)*. Springer, 124–143. doi:10.1007/978-3-031-75380-0_8

[35] A. Griggio and M. Jonáš. 2023. Kratos2: An SMT-Based Model Checker for Imperative Programs. In *Proc. CAV*. Springer, 423–436. doi:10.1007/978-3-031-37709-9_20

[36] R. Hähnle and M. Huisman. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science - State of the Art and Perspectives*. Springer, 345–373. doi:10.1007/978-3-319-91908-9_18

[37] M. Heizmann, M. Bentele, D. Dietsch, X. Jiang, D. Klumpp, F. Schüssele, and A. Podelski. 2024. Ultimate Automizer and the Abstraction of Bitwise Operations (Competition Contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 418–423. doi:10.1007/978-3-031-57256-2_31

[38] J. Jaffar and M. J. Maher. 1994. Constraint Logic Programming: A Survey. *J. Log. Program.* 19/20 (1994), 503–581. doi:10.1016/0743-1066(94)90033-7

[39] J. Jaffar, M. J. Maher, K. Marriott, and P. J. Stuckey. 1998. The Semantics of Constraint Logic Programs. *J. Log. Program.* 37, 1-3 (1998), 1–46. doi:10.1016/S0743-1066(98)10002-X

[40] C. Johannsen, K. Nukala, R. Dureja, A. Irfan, N. Shankar, C. Tinelli, M. Y. Vardi, and K. Y. Rozier. 2024. The MoXI Model Exchange Tool Suite. In *Proc. CAV (LNCS 14681)*. Springer, 203–218. doi:10.1007/978-3-031-65627-9_10

[41] E. G. Karpenkov, K. Friedberger, and D. Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *Proc. VSTTE (LNCS 9971)*. Springer, 139–148. doi:10.1007/978-3-319-48869-1_11

[42] C. Lattner and V. S. Adve. 2004. Llvm: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. CGO*. IEEE, 75–88. doi:10.1109/CGO.2004.1281665

[43] G. T. Leavens, C. Ruby, K. R. M. Leino, E. Poll, and B. Jacobs. 2000. JML: Notations and tools supporting detailed design in Java. In *Addendum Proc. OOPSLA*. ACM, 105–106. doi:10.1145/367845.367996

[44] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. LPAR (LNCS 6355)*. Springer, 348–370. doi:10.1007/978-3-642-17511-4_20

[45] K. L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Proc. CAV (LNCS 2725)*. Springer, 1–13. doi:10.1007/978-3-540-45069-6_1

[46] C. Morgan. 1988. Data Refinement by Miracles. *Inf. Process. Lett.* 26, 5 (1988), 243–246. doi:10.1016/0020-0190(88)90147-0

[47] P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proc. VMCAI (LNCS 9583)*. Springer, 41–62. doi:10.1007/978-3-662-49122-5_2

[48] K. Y. Rozier, R. Dureja, A. Irfan, C. Johannsen, K. Nukala, N. Shankar, C. Tinelli, and M. Y. Vardi. 2024. MoXI: An Intermediate Language for Symbolic Model Checking. In *Proc. SPIN (LNCS 14624)*. Springer, 26–46. doi:10.1007/978-3-031-66149-5_2

[49] Y. Vizel and O. Grumberg. 2009. Interpolation-Sequence Based Model Checking. In *Proc. FMCAD*. IEEE, 1–8. doi:10.1109/FMCAD.2009.5351148

[50] Y. Vizel, O. Grumberg, and S. Shoham. 2013. Intertwined Forward-Backward Reachability Analysis Using Interpolants. In *Proc. TACAS (LNCS 7795)*. Springer, 308–323. doi:10.1007/978-3-642-36742-7_22