SvLibChecker: A Light-Weight Tool for Software Model Checking

Dirk Beyer and Marian Lingsch-Rosenfeld

LMU Munich, Munich, Germany
{dirk.beyer, marian.lingsch-rosenfeld}@sosy.ifi.lmu.de

Abstract. SvLibChecker is a small tool for software model checking. The goal of the tool is to provide a light-weight framework that makes it easy to implement and explore algorithms for verifying software. The input to SvLibChecker is given in SV-LIB, an intermediate language that relieves the developers from dealing with sophisticated language features and their semantics. Software verifiers can normally be complex software systems with hundreds of thousands of lines of code. Due to the simple input, algorithms in SvLibChecker can be written in a succinct way. Our tool currently provides six different model-checking algorithms. Each algorithm consists of about 100 lines of Python code. The full project has about 2500 LOC in total, which are well-documented and have a good test coverage ($> 90\,\%$). The simplicity, lean architecture, and modular design of SvLibChecker lends itself for education. It is much easier to understand the implementation of an algorithm implemented in SvLibChecker, compared to complex verifiers for languages like C. SvLibChecker's predicate abstraction with CEGAR has a performance comparable to CPAchecker, a mature state-of-the-art tool for software verification. The combination of simplicity and performance makes SvLibChecker a suitable tool for verification researchers and educators, for experimenting with new verification approaches and teaching students.

Keywords: Model Checking · Software Verification · Program Analysis · SV-LIB

1 Introduction

Software model checking has become a well-established and mature research area, with many tools implementing the research results [1]. The success of such tools has made their implementations complex and hard to maintain, usually due to (a) the inherent complexity of supporting the semantics of a programming language and (b) having many optimizations to improve performance. For example, most tools participating in SV-COMP [2] are usually complex, due to the need to support the features of C or Java. This has the effect that verification researchers often spend more time on dealing with complicated language features than with improving the verification algorithms themselves.

To improve this situation, in a recent Dagstuhl seminar, the community requested the development of a new intermediate language. Its goal is to facilitate

SvLibChecker is available at: gitlab.com/sosy-lab/software/svlibchecker

To appear in Proc. CAV 2026

Table 1: Model-checking algorithms implemented in SvLibChecker

Algorithm	Description	References
Predicate Abstraction with CEGAR	Abstracts the program state using boolean predicates, refining the abstraction using CEGAR	[6, 7]
Trace Abstraction with CEGAR	Uses an automaton describing syntactic traces as abstraction, refining it using CEGAR	[8, 9]
k-Induction	Uses induction to proof the absence of errors by checking base cases and inductive steps using BMC	[10]
Symbolic Execution	Tracks the program state using symbolic values and path conditions	[11]
Bounded Model Checking (BMC)	Explores the state space using an SMT solver to check path satisfiability	[12]
Value Analysis	Tracks constant values or unknown	[13]

easier implementation of verification algorithms, verification via transformations, exchange between verifiers, and a syntax that is easy to approach. The seminar report contains a list of requirements [3]. Furthermore, the community requested the new language to be included in the competition on software verification for annual evaluation of the state-of-the-art. Both has happened: the intermediate language SV-LIB [4] has been developed and a new track (including SvLibChecker) was added to SV-COMP [1].

SV-LIB extends SMT-LIB [5] by constructs typical of procedural programming languages, such as procedures, loops, and goto's. In addition, it has first-level support for (modular) specifications like assertions, invariants, statement contracts, among others. To fully make use of SV-LIB's capability to simplify model-checking tools, we implemented SvLibChecker, a simple tool for software model checking. Its goal is to enable researchers and educators to quickly implement and explore model-checking algorithms for software.

SvLibChecker is the first standalone model checker for SV-LIB programs and implements six different algorithms Table 1. Each algorithm is implemented in its own file, each being around 100 LOC. In total, the code-base has about 2500 LOC, including tests, translators from and to PySMT, among others. The whole code-base is well-documented and has a high code coverage ($> 90\,\%$ for statement and branch coverage). Therefore, the code is well-designed to teach and study model-checking algorithms. Even though the code-base is small, the predicate abstraction of SvLibChecker performs similar to CPAchecker, which is a mature state-of-the-art tool and the only other verifier supporting SV-LIB currently.

Contributions. In this paper, we provide

- the first light-weight tool for model checking of SV-LIB programs (Sect. 4),
- publicly available and easy-to-understand implementation of six model-checking algorithms in a well-documented and tested code base (Sect. 5), and
- an evaluation showing that SvLibChecker is precise and efficient (Sect. 5).

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

```
1 (set-logic LIA)                ; SMT-LIB command to set the logic
2
3 (define-proc add               ; declaration of the procedure add
4   ((x0 Int) (y0 Int))          ; input variables
5   ((x Int)) ((y Int))          ; output and local variables
6   (! (sequence
7        (assign (x x0) (y y0)) ; assignment of all terms simultaneously
8        (! (while               ; While loop
9             (< 0 y)            ; condition of the while loop
10            (assign            ; body of the loop is an assign statement
11              (x (+ x 1))
12              (y (- y 1))))
13          :tag while-1))       ; tags are used to denote program locations
14     :tag proc-add))
15
16 (annotate-tag                 ; command to annotate existing tags
17   proc-add
18   :requires (<= 0 y0)         ; pre-condition of add
19   :ensures  (= x (+ x0 y0)))  ; post-condition of add
20
21 (declare-const xC Int)        ; SMT-LIB constants to call add with
22 (declare-const yC Int)
23
24 (assert (<= 0 yC))            ; SMT-LIB assert to satisfy the pre-condition
25
26 (verify-call add (xC yC))     ; verification call
```

Fig. 1: `loop-add-verification.safe.svlib` (slightly adapted) example SV-LIB program which can be verified correctly using SvLibChecker; it specifies a procedure `add` (line 3) which returns `x`, the result of adding `x0` and `y0` together

2 Related Work

Frameworks for Software Verification. Some frameworks for software verification provide the connection to abstract domains and the interaction with them, for example PySMT [14], and JavaSMT [15] provide the interface to SMT solvers, while Apron [16] provides an interface to numerical abstract domains. Other frameworks try to make the implementation of new algorithms easier by providing common infrastructure, but are usually very large and complex. For example, CPAchecker [17] provides a framework to implement new algorithms based on the configurable program analysis concept [18], but is a large software project with hundreds of thousands LOC. Related frameworks like Ultimate [19], SeaHorn [20], and Kratos [21], suffer from similar problems. Other approaches like MoXIchecker [22] try to keep the implementation light-weight, but focus on transition systems and not on software.

Intermediate Languages. Intermediate languages are often used to simplify the implementation of software verification tools. Since the complexity of supporting a feature rich input language is reduced. Many such languages exist, like Boogie [23], Why3 [24], MoXI [25], CHCs [26], SMT-LIB [5], among many others [27, 28, 29, 30, 31]. In contrast to them, SV-LIB [4] is designed specifically for both software model-checking and deductive verification. In addition, it has a simple syntax and clear semantics, which makes it a good candidate for a light-weight model-checker like SvLibChecker.

Program Transformations. In addition to using SV-LIB as input language, SvLibChecker simplifies its code-base by transforming specifications and proce-

dure calls/returns into new CFA edges during the state-space exploration. Therefore, algorithms only need to handle statement, without procedure calls/returns, and assume edges. Program transformations have been studied extensively in the literature [32, 33, 34, 35, 36, 37, 38], with some works focusing specifically on transforming specifications [39, 40, 41, 42, 43]. Nonetheless, these ideas are usually applied to the input program in advance, and not during the state-space exploration.

3 Input Language SV-LIB

SV-LIB [4] is an intermediate verification language designed for both software model-checking and deductive verification. It is based on SMT-LIB [5] and extends it with constructs typical of procedural programming languages, like procedures, loops, and gotos. In addition, it has first-level support for (modular) specifications, such as assertions (`check-true`), invariants (`invariant`), and statement contracts (`requires`, `ensures`).

Figure 1 shows an example SV-LIB program which can be verified correctly using SvLibChecker. It has a procedure `add` (line 3) which returns `x`, the result of adding `x0` and `y0` together, using `y` as a local variable. The specification of `add` is given as a pre- and post-condition (lines 18 and 19). The pre-condition is implied by the SMT-LIB `assert` (line 24), leaving the post-condition to be shown. The verification task is given by the command `verify-call` (line 26).

4 Software Verifier SvLibChecker

SvLibChecker is implemented in python using PySMT [14] to interact with SMT solvers, PySvLib [4] to work with SV-LIB programs, and AutomataLib [44] to work with finite automata, which is required for trace abstraction.

Importantly, we use PySvLib to transform an SV-LIB program into a CFA, which is the usual data-structure used by model-checking algorithms. It implements an usual translation to a CFA, creating a separate CFA for each function. The translation adds to each CFA node $\mathcal{N}$ an attribute $\mathcal{N}$.`properties` containing all specifications belonging to it, which aids in tracking them. Furthermore, it also adds to each CFA node the CFA node $\mathcal{N}_{end}$ which corresponds to the node at the end of the statement starting at this node, if it exists. Knowing the end node of the statement is useful to identify the end position for post-conditions.

Currently, SvLibChecker implements six different model checking algorithms as shown in Table 1. Each algorithm is implemented in its own file and is around 100 LOC. The total code-base has around 2597 LOC, including tests, translators to and from PySMT, and CLI utilities (see Item Claim 3 for a detailed analysis). The different algorithms are based on a common interface (`Algorithm`), and some make use of the state-space exploration typical of model-checking algorithms (`State Space Exploration`). The precise way the algorithms inherit, and make use of each other is shown in Fig. 2. All the implemented algorithms are well-known, therefore we refer to their implementation in SvLibChecker and relevant references for details, see Table 1.

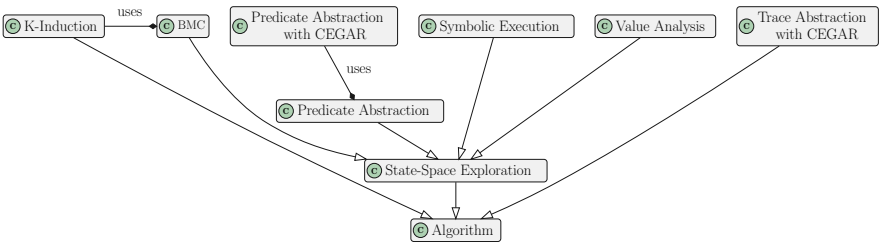

Fig. 2: Architecture of SvLibChecker, showing all algorithms; we refer to *predicate abstraction with CEGAR* as *predicate abstraction* in the text for brevity

To make understanding and debugging the implemented algorithms easier, SvLibChecker represents the abstract states of each algorithm, as close to SV-LIB data structures as possible. For example, all predicates in predicate abstraction are represented as SV-LIB terms, and paths are represented as sequences of CFA edges. This removes the need to understand additional intermediate layers when analyzing the intermediate state and output of an algorithm.

The main reason for the small size of SvLibChecker, is it using SV-LIB as input language. The second and third main reasons, are: (a) the *State Space Exploration* component, which handles the state-space exploration, (b) and the handling of specifications through CFA rewriting during the state-space exploration.

4.1 State-Space Exploration

The full state-space exploration algorithm is shown in Appendix A – Alg. 1. It consists of a three-fold process: (a) exploring the state space through the CFA edges, (b) handling the call-stack due to procedure calls/returns by creating new CFA edges, which are then explored, and (c) transforming specifications into CFA edges, and exploring them. The first one is standard in model checking algorithms, while the other two are not usual ways of handling the call-stack and specifications. Nonetheless, both simplify the implementation massively, since all algorithms inheriting from the state-space exploration component only need to know how to: (a) create an initial state of the abstract state tracked by the algorithm ($\mathcal{A}_0$), and (b) how to handle a CFA assume or statement edge without it being a procedure call or return (`execute_edge`).

4.2 Specifications as on-the-fly CFA rewriting

It is usual for an analysis to use information from the CFA in order to compute relevant information for it. For example, predicate abstraction needs to know at which locations to abstract. Therefore, modification of the CFA structure in advance can affect the behavior of the analyses. That is why, in order to handle all specifications uniformly, while not modifying the underlying CFA structure, SvLibChecker rewrites specifications into CFA edge sequences during the state-space exploration.

The rules for rewriting the supported specifications are shown in Fig. 3. They make use of the current CFA node ($\mathcal{N}$), the CFA node where the statement ends

$$\mathcal{N}, \mathcal{N}_{end}, \texttt{check-true}\ \phi \Rightarrow ([\texttt{assume}\ \neg\phi], \text{true}, \{\})$$
$$\mathcal{N}, \mathcal{N}_{end}, \quad \texttt{requires}\ \phi \Rightarrow ([\texttt{assume}\ \neg\phi], \text{true}, \{\})$$
$$\mathcal{N}, \mathcal{N}_{end}, \quad \texttt{invariant}\ \phi \Rightarrow ([\texttt{assume}\ \neg\phi], \text{true}, \{\}),$$
$$([\texttt{havoc}, \texttt{assume}\ \phi], \text{false}, \{\mathcal{N} \mapsto \phi\})$$
$$\mathcal{N}, \mathcal{N}_{end}, \quad \texttt{ensures}\ \phi \Rightarrow ([\texttt{havoc}, \texttt{assume}\ \wedge_{(\texttt{requires},\psi)\in\mathcal{N}.\texttt{properties}}\ \psi], \text{false}, \{\mathcal{N}_{end} \mapsto \phi\})$$

Fig. 3: Rules for rewriting a CFA node $\mathcal{N}$, whose statement in the program ends in the CFA node $\mathcal{N}_{end}$, and a specification (`name` ϕ), into a list of (a) a sequence of CFA edges, (b) a boolean flag indicating an error state in the successor, and (c) a mapping of nodes to terms which need to be satisfied at the given node

in the program input program ($\mathcal{N}_{end}$), and the specification (`name` ϕ) rewriting them into a list of (a) a sequence of CFA edges, (b) a boolean flag indicating if the end state is considered an error state or not, and (c) a mapping of nodes to terms which need to be satisfied at the given node the next time it is visited, which behaves the same as adding a one-time extra `check-true` ϕ to the specifications of the CFA node for that path being explored. The returned edge sequences are all explored immediately always starting from the current abstract state.

Note that for modular abstractions, like *invariants*, the successor state from the leaving edges of the CFA will usually be subsumed, when using abstraction like in predicate abstraction, by the successors created from the specification edges. This is in line with the SV-LIB semantics, where specifications can only add more traces to the system. Therefore, handling specifications through dynamic CFA rewrites, results in little overhead from keeping the original CFA edges.

5 Evaluation

SvLibChecker has two main goals: (a) allow researchers to easily implement new algorithms, and (b) make model-checking algorithms accessible to newcomers of the field. To show that SvLibChecker is a suitable tool for *implementing* new model-checking algorithms, we compare it to CPAchecker [17], a mature state-of-the-art model-checker (Claim 1 and Claim 2). We show that it performs similarly to CPAchecker using the same algorithm and SMT-solver back-end. In addition, we show that it is suitable for education and quickly implementing new algorithms, since it has a small and well-documented code-base (Claim 3).

Claim 1 (Effectiveness): SvLibChecker's predicate abstraction is comparable to CPAchecker in terms of solved verification tasks.

Claim 2 (Efficiency): SvLibChecker's predicate abstraction is comparable to CPAchecker in terms of consumed run time.

Claim 3 (Code Size): SvLibChecker's implementations of the algorithms are short, concise, and documented.

Benchmark Set. In our evaluation, we use SV-Benchmarks, which includes the first benchmark set of SV-LIB programs with known verdicts, in the current version

Table 2: Correctly solved verification tasks (*Solved*) with percentage of solved tasks from the total 262 tasks in gray; also given the amount of solved tasks which do not contain a specification violation (*True*), those which do (*False*), and amount of tasks solved only by a single analysis (*Unique*); rows are sorted alphabetically by tool-name, and by the amount of correctly solved tasks

Verifier	Analysis	Solver	Solved	True	False	Unique
CPAchecker	Predicate Abstraction	MathSAT	161 (61.5%)	126	35	7
SvLibChecker	Predicate Abstraction	MathSAT	**167** (63.7%)	126	41	3
	Trace Abstraction	MathSAT	132 (50.4%)	93	39	1
	k-Induction	Z3	111 (42.4%)	69	42	0
	k-Induction	MathSAT	110 (42.0%)	68	42	0
	Symbolic Execution	MathSAT	86 (32.8%)	42	44	0
	Symbolic Execution	Z3	86 (32.8%)	42	44	0
	BMC	MathSAT	76 (29.0%)	31	45	0
	BMC	Z3	76 (29.0%)	31	45	0
	Value Analysis	–	31 (11.8%)	23	8	0

for SV-LIB programs 6a7152b4. The dataset contains manually written programs (core-validation and core-verification) as well as programs translated from the C category *C.ReachSafety.Loops* of SV-COMP [2] using Cuvee [45] (c-translated).

Benchmark Environment. We use BenchExec [46] to ensure reliable benchmarking. All benchmarks are performed on machines with an Intel Xeon E5-1230 CPU (4 physical cores with 2 processing units each), 33 GB of RAM, and running the Ubuntu 24.04 operating system. For all experiments we use a cpu-time timeout of 900 s per tool invocation and restrict it to 2 processing units and 15 GB of RAM.

Tools and Configurations. We use CPAchecker in version 4.2.2 [47] in its SV-COMP'26 configuration, which uses predicate abstraction, with the backend SMT-Solver MathSAT [48] in version 5.6.15. For SvLibChecker, we use the commit c719332d with a separate configuration for each algorithm. SvLibChecker uses MathSAT [48] in version 5.6.15 and Z3 [49] in version 4.15.4. This version of Z3 does not support interpolation, hence we use only MathSAT for predicate abstraction and trace abstraction which both require interpolation.

5.1 Claim 1 Effectiveness of SvLibChecker

Table 2 shows the amount of correctly solved verification tasks by each analysis, in SvLibChecker and CPAchecker. There were no incorrect results produced by any verifier. Shown are the total number of correctly solved tasks (*Solved*), tasks which do not contain a specification violation (*True*), those which do (*False*), and those which could only be solved by a single analysis (*Unique*).

The results show that SvLibChecker's predicate abstraction is the most effective analysis, solving 167 (63.7%) verification tasks correctly, followed by

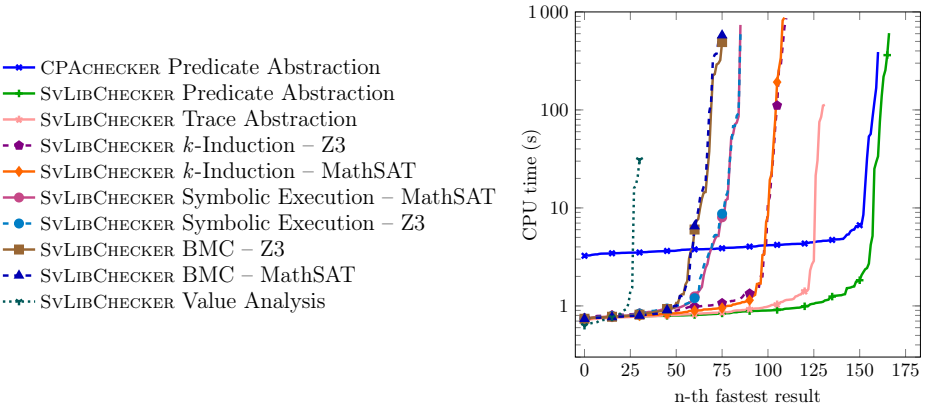

Fig. 4: Time taken by SvLibChecker and CPAchecker to obtain the n-th fastest result across all 262 verification tasks; legend is sorted alphabetically by tool-name, followed by the amount of correctly solved tasks

CPAchecker's predicate abstraction, which solved 161 (61.5%) verification tasks correctly. This is due to SvLibChecker supporting modular specifications, which are not supported by CPAchecker. When not considering the tasks with modular abstractions (invariants, and pre- and post-conditions, i.e., the c-translated tasks), CPAchecker's predicate abstraction and SvLibChecker's predicate abstraction solve the same amount of tasks correctly. This is interesting, since CPAchecker is a mature state-of-the-art model-checker, while SvLibChecker follows a light-weight implementation. Showing that SvLibChecker can perform at a comparable level to state-of-the-art tools using the same algorithm and SMT-solver back-end.

Notably, SvLibChecker's predicate abstraction and CPAchecker's predicate abstraction each solve some tasks which no other analysis could solve. This shows that both implementations have some unique strengths. Note that these tasks do not contain modular specifications, since those can also be solved by trace abstraction or k-Induction.

Looking at the different SMT solvers used by SvLibChecker, we see that MathSAT performs similarly to Z3. The only exception is k-Induction, where Z3 solves 1 more task correctly than MathSAT. This indicates that the choice of SMT solver has only a minor influence on the effectiveness of SvLibChecker.

> SvLibChecker solves a similar amount of tasks as CPAchecker using the same algorithm and SMT-solver back-end. It outperforms CPAchecker, since it handles modular specifications like invariants.

5.2 Claim 2 Efficiency of SvLibChecker

Figure 4 shows the time taken by each analysis in SvLibChecker and CPAchecker to obtain the n-th fastest result across all 262 verification tasks. Overall, we see that for both tools and all analyses most tasks are solved in less than 10 sec-

Table 3: Lines of code (LOC) and comments for each file containing the corresponding algorithm (*stand-alone*), and including all dependency files according to Fig. 2 (*cumulative*); additionally the total size for all algorithm files together, and for the entire repository are shown; rows sorted according to stand-alone size

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

onds, with only a few tasks taking between 10 s and 900 s. This shows that SvLibChecker has a similar efficiency as CPAchecker.

Looking to the left of the quantile plot, we see that SvLibChecker solves its first task around 2 seconds faster than CPAchecker. This is likely due to SvLibChecker being a python based tool, which has no startup overhead, while CPAchecker being a Java based tool requires the startup of the JVM.

When comparing algorithms which use different SMT-solvers, we see no significant difference in time taken. With both Z3 and MathSAT showing similar performance. This indicates that the choice of SMT-solver has only a minor influence on the efficiency of SvLibChecker.

> Both SvLibChecker and CPAchecker solve most verification tasks within 10 s with only a few taking longer. Overall, SvLibChecker is slightly faster than CPAchecker, mainly due to the difference in startup time (JVM).

5.3 Claim 3: Code Size of Implemented Algorithms

Table 3 shows the amount of lines of code (LOC) and amount of comments for each file containing the corresponding algorithm implemented in SvLibChecker, as well as the size for all algorithm files together and the entire repository. The count was performed using CLOC[2] a state-of-the art tool for such purposes. We distinguish between two ways of counting: (a) *stand-alone*, which considers only the algorithm file itself, mimicking the complexity of understanding a new algorithm

[2] https://github.com/AlDanial/cloc

integrated into SvLibChecker, and (b) *cumulative*, which includes all files corresponding containing one of the algorithm being used or inherited from according to Fig. 2, mimicking the complexity of understanding the algorithm from scratch.

From Table 3 it follows that to understand each algorithm implemented in SvLibChecker as a *stand-alone* component, one needs to read only around 100 LOC. In case one also wants to understand the algorithm dependencies, one needs to read around 500 LOC. The main contribution to the dependency size comes from the state-space exploration (195 LOC) and the base algorithm interface (83 LOC). Since most dependencies are shared, one needs to only read 896 LOC to understand all algorithms together with their dependencies. In particular, the code is well-documented, since the average ratio of comments to code is around 0.46 for all algorithm files. This means that on average each second line of code is accompanied by a line with a comment.

If one wants to understand the entire SvLibChecker code-base, including tests, translators to and from PySMT, CLI utilities, among others, one has to read around 2500 LOC. This is very little compared to existing tools for software model-checking, which usually contain tens if not hundreds of thousands of LOC.

In addition to its small code-size and good documentation, the SvLibChecker code-base has a high test coverage. It has 93 % statement coverage, and 92 % branch coverage, reported by its CI. The test-coverage is not higher, mainly due to defensive programming in some modules, making it hard, or in some cases impossible to cover all branches.

> SvLibChecker has a small, well-documented code-base with a high test coverage. This makes it easy to understand existing algorithms, and implement and experiment with new algorithms inside it.

6 Conclusion

SvLibChecker is a light-weight tool for software model checking, aiming to make implementing and exploring (new) algorithms for verifying software easier. Currently, it implements six different well-known model-checking algorithms. Each algorithm is implemented in its own file in around 100 lines of code. Its total code-base is only around 2500 lines of code, with a code coverage above 90 %. The code is also well-documented, having an average 0.46 comment to code ratio, for algorithm files. It achieves its simplicity by using SV-LIB as input language, having a lean and modular architecture, and using program transformations during the state-space exploration to handle procedure calls/returns and specifications. Despite its simplicity, SvLibChecker's predicate abstraction achieves a comparable performance to CPAchecker's, a mature state-of-the-art tool for software verification.

The combination of simplicity and performance makes SvLibChecker a suitable tool for both software verification researchers and educators, for experimenting with new verification approaches, and teaching students.

Data-Availability Statement. A reproduction package, which includes all software and data that we used for our experiments, is available on Zenodo [50].

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

Acknowledgement. SV-LIB is the result of an initiative that started at the Dagstuhl Seminar 25172 [3, Sect. 4.3], in which the community requested such an intermediate language for software verification. We thank especially Gidon Ernst, Albergo Griggio, and Martin Jonáš for their contributions to SV-LIB [51]. Furthermore, we thank the SV-COMP community for including an SV-LIB track in the competition on software verification (as demo track already in SV-COMP 2026). Finally, we thank the CAV reviewers for suggesting to add PDR to SvLibChecker.

References

1. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
2. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
3. Beyer, D., Huisman, M., Strejček, J., Wehrheim, H.: Information exchange in software verification (Dagstuhl Seminar 25172). Dagstuhl Reports **15**(4), 92–111 (2025). https://doi.org/10.4230/DAGREP.15.4.92
4. Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks. arXiv/CoRR **2511**(21509) (December 2025). https://doi.org/10.48550/arXiv.2511.21509
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.5. Tech. rep., University of Iowa (2015), http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV. pp. 154–169. LNCS 1855, Springer (2000). https://doi.org/10.1007/10722167_15
7. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Proc. POPL. pp. 191–202. ACM (2002). https://doi.org/10.1145/503272.503291
8. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Proc. SAS. pp. 69–85. LNCS 5673, Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
9. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
10. Donaldson, A.F., Haller, L., Kröning, D., Rümmer, P.: Software verification using k-induction. In: Proc. SAS. pp. 351–368. LNCS 6887, Springer (2011). https://doi.org/10.1007/978-3-642-23702-7_26
11. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
12. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14

13. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
14. Gario, M., Micheli, A.: PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In: Proc. SMT (2015)
15. Baier, D., Beyer, D., Friedberger, K.: JavaSMT 3: Interacting with SMT solvers in Java. In: Proc. CAV. Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_9
16. Bertrand, J., Antoine, M.: Apron: A library of numerical abstract domains for static analysis. In: Proc. CAV. pp. 661–667. LNCS 5643, Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52
17. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
18. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
19. Heizmann, M., Bentele, M., Dietsch, D., Jiang, X., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Automizer and the abstraction of bitwise operations (competition contribution). In: Proc. TACAS (3). pp. 418–423. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_31
20. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Proc. CAV. pp. 343–361. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_20
21. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: Kratos: A software model checker for SystemC. In: Proc. CAV. pp. 310–316. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_24
22. Ates, S., Beyer, D., Chien, P.C., Lee, N.Z.: MoXIchecker: An extensible model checker for MoXI. In: Proc. VSTTE 2024. pp. 1–14. LNCS 15525, Springer (2025). https://doi.org/10.1007/978-3-031-86695-1_1
23. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
24. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: Programming Languages and Systems. pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
25. Rozier, K.Y., Dureja, R., Irfan, A., Johannsen, C., Nukala, K., Shankar, N., Tinelli, C., Vardi, M.Y.: MoXI: An intermediate language for symbolic model checking. In: Proc. SPIN. pp. 26–46. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_2
26. Angelis, E.D., K, H.G.V.: CHC-COMP 2022: Competition report. Electronic Proceedings in Theoretical Computer Science **373**, 44–62 (11 2022). https://doi.org/10.4204/eptcs.373.5
27. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of C programs. In: Proc. CC. pp. 213–228. LNCS 2304, Springer (2002). https://doi.org/10.1007/3-540-45937-5_16
28. Lattner, C., Adve, V.S.: Llvm: A compilation framework for lifelong program analysis and transformation. In: Proc. CGO. pp. 75–88. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665

29. Schuiki, F., Kurth, A., Grosser, T., Benini, L.: LLHD: A multi-level intermediate representation for hardware description languages. In: Proc. PLDI. pp. 258–271. ACM (2020). https://doi.org/10.1145/3385412.3386024
30. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools. In: Proc. SMT. CEUR Workshop Proceedings, vol. 3185, pp. 80–89. CEUR-WS.org (2022)
31. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Proc. VMCAI. pp. 41–62. LNCS 9583, Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_2
32. Visser, E.: A survey of strategies in program-transformation systems. In: Proc. WRS. pp. 109–143. ENTCS 57, Elsevier (2001). https://doi.org/10.1016/S1571-0661(04)00270-1
33. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: CEGAR-PT: A tool for abstraction by program transformation. In: Proc. ASE. pp. 2078–2081. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00215
34. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. LNCS 11187, Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
35. Lerner, S., Grove, D., Chambers, C.: Composing data-flow analyses and transformations. In: Proc. POPL. pp. 270–282. ACM (2002). https://doi.org/10.1145/503272.503298
36. Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M.: A unifying approach for control-flow-based loop abstraction. In: Proc. SEFM. pp. 3–19. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_1
37. Alglave, J., Kröning, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Proc. ESOP. pp. 512–532. LNCS 7792, Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_28
38. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Proc. APLAS. pp. 311–319. LNCS 12470, Springer (2020). https://doi.org/10.1007/978-3-030-64437-6_16
39. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
40. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Xia, T., Zheng, X.: TransVer: A modular program-transformation framework for reduction to reachability. In: Proc. SPIN. pp. 1–24. LNCS 15945, Springer (2025). https://doi.org/10.1007/978-3-032-06847-7_1
41. Julien, S.: E-ACSL: Executable ANSI/ISO C specification language (2022), available at http://frama-c.com/download/e-acsl/e-acsl.pdf
42. Amilon, J., Esen, Z., Gurov, D., Lidström, C., Rümmer, P.: Automatic program instrumentation for automatic verification. In: Proc. CAV. pp. 281–304 (2023). https://doi.org/10.1007/978-3-031-37709-9_14
43. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214
44. Evans, C., Robson, E.W.: automata: A Python package for simulating and manipulating automata. Journal of Open Source Software **8**(90), 5759 (2023). https://doi.org/10.21105/joss.05759
45. Ernst, G.: Cuvée: Blending SMT-LIB with programs and weakest preconditions. arXiv/CoRR **2511**(21509) (October 2020). https://doi.org/10.48550/arXiv.2010.05023

46. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
47. Beyer, D., Wendler, P.: Cpachecker release 4.2.2 (12 2025). https://doi.org/10.5281/zenodo.17777566
48. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
49. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS. pp. 337–340. LNCS 4963, Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
50. Beyer, D., Lingsch-Rosenfeld, M.: Reproduction package for CAV 2026 submission 'SvLibChecker: A Light-Weight Tool for Software Model Checking'. Zenodo (2026). https://doi.org/10.5281/zenodo.18380924
51. Beyer, D., Strejček, J.: Evaluating software verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In: Proc. TACAS (2). pp. 461–501. LNCS 16506, Springer (2026). https://doi.org/10.1007/978-3-032-22749-2_23

A State-Space Exploration Algorithm

Algorithm 1 shows the complete state-space exploration algorithm which is the backbone of most model-checking algorithms implemented in SvLibChecker. It consists of three main parts: (a) exploring the state space through the CFA edges, (b) handling the call-stack due to procedure calls/returns by creating new CFA edges on-the-fly, and then exploring them, and (c) transforming specifications into CFA edges on-the-fly, and exploring these new edges.

Note that for the state-space exploration to work correctly, the `is_subsumed` function should only return `true` for two nodes if they are both errors or both non-errors, and both have the same delegated specification checks.

Algorithm 1 State-Space Exploration Algorithm common to most Model Checking Algorithms implemented in SvLibChecker

Require: Start node $\mathcal{A}_0$ representing the initial abstract state
Ensure: Verdict (`Correct`/`Incorrect`) and the reachability graph $\mathcal{R}$ of explored states

▷ Helper Function to handle a single edge ◁
1: **function** HANDLE_EDGE($\mathcal{A}_i$, e, $\mathcal{R}$, `waitlist`)
2: $\mathcal{A}_{i+1} \leftarrow$ `execute_edge`($\mathcal{A}_i$, e)
3: **if** $\neg$`is_subsumed`($\mathcal{A}_{i+1}$, $\mathcal{R}$) **then**
4: `waitlist.push`($\mathcal{A}_{i+1}$)
5: $\mathcal{R}$.`add`($\mathcal{A}_i$, e, $\mathcal{A}_{i+1}$)
6: **end if**
7: **end function**

▷ Main State-Space Exploration ◁
8: **function** STATE_SPACE_EXPLORATION($\mathcal{A}_0$)
9: `waitlist` $\leftarrow [\mathcal{A}_0]$; $\mathcal{R} \leftarrow (\{\mathcal{A}_0\}, \emptyset)$
10: **while** |`waitlist`| > 0 **do**
11: $\mathcal{A}_i \leftarrow$ `waitlist.pop`()
▷ Return if an error state is reached
12: **if** $\mathcal{A}_i$.`hasError` **then return** `Incorrect`, $\mathcal{R}$
13: **end if**
▷ Explore all normal CFA edges
14: **for all** $e \in \mathcal{A}_i$.`cfaNode.leavingEdges` **do**
15: `handle_edge`($\mathcal{A}_i$, e, $\mathcal{R}$, `waitlist`)
16: **end for**
▷ Handle the callstack by creating CFA edges
17: **for all** $e \in$ `handle_callstack`($\mathcal{A}_i$) **do**
18: `handle_edge`($\mathcal{A}_i$, e, $\mathcal{R}$, `waitlist`)
19: **end for**
▷ Transform specifications into CFA edges
20: **for all** $E \in$ `transform_properties`($\mathcal{A}_i$.`cfaNode`) **do**
21: $\mathcal{A}_{\text{prop}} \leftarrow \mathcal{A}_i$
22: **for all** $e \in E$ **do**
23: `handle_edge`($\mathcal{A}_{\text{prop}}$, e, $\mathcal{R}$, `waitlist`)
24: **end for**
25: **end for**
26: **end while**
return `Correct`, $\mathcal{R}$
27: **end function**