Violation Witnesses and Result Validation for Multi-Threaded Programs
Implementation and Evaluation with CPAchecker

Dirk Beyer and Karlheinz Friedberger

LMU Munich, Munich, Germany

Abstract. Invariants and error traces are important results of a program analysis, and therefore, a standardized exchange format for verification witnesses is used by many program analyzers to store and share those results. This way, information about program traces and variable assignments can be shared across tools, e.g., to validate verification results, or provided to users, e.g., to visualize and explore the results in order to fix bugs or understand the reason for a program's correctness. The standard format for correctness and violation witnesses that was used by SV-COMP for several years was only applicable to sequential (single-threaded) programs. To enable the validation of results for multi-threaded programs, we extend the existing standard exchange format by adding information about thread management and thread interleaving. We contribute a reference implementation of a validator for violation witnesses in the new format, which we implemented as component of the software-verification framework CPAchecker. We experimentally evaluate the format and validator on a large set of violation witnesses. The outcome is promising: several verification tools already produce violation witnesses that help validating the verification results, and our witness validator can re-verify most of the produced witnesses.

Keywords: Verification witness · Result validation · Software verification · Proof format · Program analysis · Violation witness · Counterexample · CPAchecker

1 Introduction

Reliable and correct software is a basic dependency of today's society and industry. For proving programs correct as well as for finding errors in programs, formal verification is a powerful technique. Given a program and a specification, a software verifier either finds an error path through the program that exposes the specification violation or proves that the specification is satisfied by the program. In most cases, the analysis produces some kind of data that is valuable for the user and can

Replication package available on Zenodo [14].

Funded in part by Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY).

T. Margaria and B. Steffen (Eds.): ISoLA 2020, LNCS 12476, pp. 449–470, 2020.
https://doi.org/10.1007/978-3-030-61362-4_26

be used in further applications. Several tool chains support the direct reuse of verification results [5,6,25]. In general, information about the program analysis can be provided in form of a verification witness, either as correctness witness [10] (e.g., describing invariants from the correctness proof) or as violation witness [11,12] (e.g., representing an abstract counterexample towards a property violation).

The standard witness exchange format was specified and continuously improved by the verification community (especially SV-COMP) over the last years.[1] The specification was first supporting only sequential programs (since SV-COMP 2015 [4,11]), and we later extended it to multi-threaded programs as well (SV-COMP 2018–2020). In this paper, we describe the necessary extensions to the witness format and provide evidence that violation witnesses for concurrent tasks are not only *produced* by many verification tools (in SV-COMP 2020: Cbmc [29], CPAchecker [18], CPAlockator [1], Dartagnan [33], Divine [3], Esbmc [32], Lazy-CSeq [39], PeSCo [31], Ultimate Automizer [37], Ultimate Taipan [35], Yogar-Cbmc [40]), but that most of the violation witnesses for concurrent programs can also be *validated* by our implementation of a validation tool.

Contributions. The paper makes the following contributions:

- Extension of the existing violation witness format by additional hints on thread management: (i) thread interleavings are represented using thread-ids at all edges and (ii) thread creation is added to the witness.
- Implementation of an approach for validation of violation witnesses for multi-threaded programs in the verification framework CPAchecker and make the source code available as reference implementation for others.[2]
- Experimental evaluation of the new format and validator on a large number of verification tasks with violation witnesses from several verifiers to show that the approach is effective and helps validating the existence of error traces in multi-threaded programs.
- Availability of all experimental results, including raw data, tables, experiment setup, etc. (see Sect. 6).

Related Work. As we extend an existing standardized witness format and validation technology, this work is based on a number of existing ideas, which we outline in the following.

Verification Artifacts. Many program-analysis techniques are efficient at discovering proofs or failures. However, it is often difficult to evaluate results, such as program paths towards property violations. Artifacts [24] from verifier executions are valuable for users [2,28,36]. The standard exchange format for verification witnesses [11] is the basis of our work; we describe and extend it in this paper and apply it in our evaluation.

[1] https://github.com/sosy-lab/sv-witnesses
[2] https://cpachecker.sosy-lab.org

Test Execution and Harnesses. While it is comparatively simple to create an executable harness for a sequential program [12,27,30,34], the situation for multi-threaded programs is more complex. Simple test cases can not capture the difficulty of nondeterministically interleaved threads and can only be used to heuristically execute a sample of all possible program traces. The scheduling of threads needs to be encoded into the harness in such a way that all statements are interleaved in the correct ordering.

Sequentialization. Tools like Lazy-CSeq [38,39] apply sequentialization techniques before verification and can thus provide data about multi-threaded counterexample traces via a sequentialized program. However, the mapping from a sequentialized program (and the found counterexample path in it) back to its multi-threaded origin needs to be supported.

2 Background

We provide only a short overview of some basic concepts and definitions that we use to describe our approach, including the program representation, the format for violation witnesses, and the multi-threaded program analysis in CPAchecker.

2.1 Program Representation

For presentation, we restrict our programs to a simple imperative programming language that contains only assignments, assumptions, declarations, function calls, and function returns. The language supports simple thread management via the calls of *pthread_create* and *pthread_join*, and assumes that each statement in the code is atomic on its own, i.e., uses a strong memory model providing sequential consistency, such that an update of a shared variable is immediately visible to all threads and the verification approach does not need to care about asynchronous memory accesses like simultaneously updating the same memory cell from multiple threads or unit-local caching of values that might happen on hardware level. This is not a theoretical restriction, as each statement could be decomposed into a sequence of reading and writing statements, where each statement involves at most one shared variable. For simplicity and generality, the witnesses ignore further thread-management methods like *mutex locks*, *wait*, and *cancel* operations, as well as interrupts. In violation witnesses such operations do not need to be specified for the validation tool.

The violation witnesses for multi-threaded programs that are produced by the verifiers all support the C programming language as input language and may support a wider range of thread-management operations. We will analyze the quality of those witnesses in the evaluation (Sect. 5).

A program is represented by a *control-flow automaton* (CFA) (L, l_{init}, G), which consists of a set L of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ of control-flow edges (modeling the control flow with assignment and assumption operations as well as declarations and function calls

```
1  int NUM = 4, FIB = 55;
2  int i = 1, j = 1;
3
4  void *t1() {
5    for (int k = 0; k < NUM; k++) {
6      i += j;
7    }
8    pthread_exit(0);
9  }
10
11 void *t2() {
12   for (int k = 0; k < NUM; k++) {
13     j += i;
14   }
15   pthread_exit(0);
16 }
17
18 int main() {
19   pthread_t id1, id2;
20   pthread_create(&id1, 0, t1, 0);
21   pthread_create(&id2, 0, t2, 0);
22   if (i >= FIB || j >= FIB) {
23     __VERIFIER_error();
24   }
25   return 0;
26 }
```

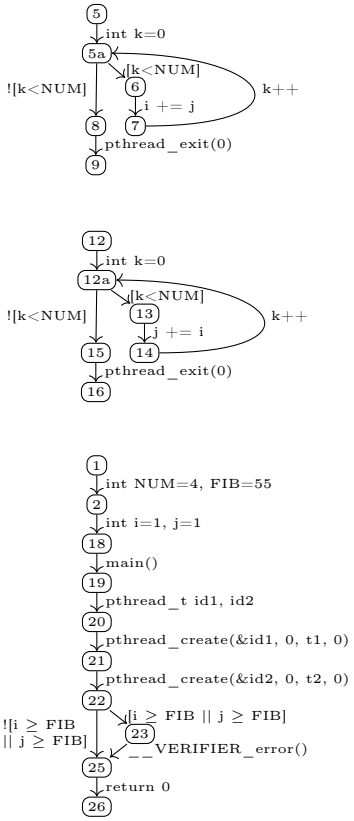


Fig. 1. Source code and CFAs for multi-threaded example program, adopted from program https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread/fib_bench-2.c

and returns from *Ops*), and a program-entry location $l_{init} \in L$. A sequence $<g_1, g_2, ..., g_n>$ of CFA edges from G is called *program path* if it starts from the program-entry location (i.e., $g_1 = (l_{init}, \cdot, \cdot)$). As we analyze multi-threaded programs, this sequence consists of potentially interleaved edges from different threads, e.g., there is no need that the end location l of a CFA edge $g_i = (\cdot, \cdot, l)$ is identical with the start location l' of its directly succeeding CFA edge $g_{i+1} = (l', \cdot, \cdot)$, but the next CFA edge along the sequence from the same thread must start with program location l. At the program entry and at each thread entry, there is no matching previous program location in a valid program path.

The example in Fig. 1 shows a short multi-threaded program and the corresponding CFAs. The program is build around the Fibonacci sequence, even if the source itself does not directly reveal this. We will later examine this example and find a sequence of operations such that $fib(10) = 55$ was computed (this is modeled as a violation of the specification `G ! call(__VERIFIER_error())` i.e., a call to function `__VERIFIER_error` is not reachable).

2.2 Violation Witnesses

Witnesses in software verification are based on the concept of protocol automata [11] that are matched against a CFA for validation. A protocol automaton consists of control states with invariants and edges between control states that represent program transitions. An edge contains a source guard, which restricts the transition to a specific set $S \subseteq G$ of edges from the CFA, and a state-space guard, which restricts the state space by giving additional constraints on variables.

For exporting a violation witness to a file, the protocol automaton is converted into *GraphML* [26], enriched with additional meta-data (like a hash of the analyzed program). When importing a violation witness from a file, the *GraphML* data structure is transformed into a protocol automaton, such that it can be used internally in parallel to any of CPAchecker's program analyses.

2.3 Analysis of Multi-Threaded Programs in CPAchecker

CPAchecker is based on the concept of configurable program analysis (CPA) [15,16]. Different concerns of a program are analyzed by different components (denoted as CPAs). To track variables and their assigned values, we can choose from a predicate-abstraction analysis [19], an explicit-value analysis [21], a BDD-based analysis [23], a symbolic execution [20], and several more. For the analysis of program locations in multi-threaded programs, the multi-threading analysis [13] explores the state space, computes possible thread interleavings on-the-fly, and maintains abstract states, where each abstract state consists of several program locations (one per thread) together with their call stacks (also one per thread). Additional optimizations like partial-order reduction are available in the implementation, but not considered here.

To avoid collisions of identifiers during a program analysis, e.g., as it might happen if the same function is called in two different threads at the same time, CPAchecker uses different function names for parallel running threads. If necessary, we use several copies of the CFA for a function of the program, using indexed names. For exporting a violation witness, the indexes are removed, because changed function names are not compatible across different tools. When using an existing violation witness for validating a multi-threaded program, we reintroduce a matching of available thread identifiers in the witness and indexed function copies of the CFAs.

3 Detailed Example

In the following, we explain an example step by step. First we start a verifier to verify an example program and produce a witness, and second we start a validator to validate the verification result using the produced witness.

3.1 Producing a Violation Witness

The program from Fig. 1 creates two threads `id1` and `id2`, which run in parallel and increase the value of the variables `i` and `j`. If any of the variables `i`

Program Path	Operation Scheduling			Variable Values				Line
	main	id1	id2	i	j	k_{t1}	k_{t2}	
(1, ., 2),(2, ., 18),(18, ., 19),(19, ., 20)	i=1, j=1			1	1			2
(20, ., 21),(5, ., 5a)		$k_{t1} = 0$				0		5
(5a, ., 6),(6, ., 7)		i+=j		2				6
(7, ., 5a)		k_{t1}++				1		5
(21, ., 22),(12, ., 12a)			$k_{t2} = 0$				0	12
(12a, ., 13),(13, ., 14)			j+=i		3			13
(14, ., 12a)			k_{t2}++				1	12
(5a, ., 6),(6, ., 7)		i+=j		5				6
(12a, ., 13),(13, ., 14)			j+=i		8			13
(14, ., 12a)			k_{t2}++				2	12
(7, ., 5a)		k_{t1}++				2		5
(5a, ., 6),(6, ., 7)		i+=j		13				6
(12a, ., 13),(13, ., 14)			j+=i		21			13
(14, ., 12a)			k_{t2}++				3	12
(7, ., 5a)		k_{t1}++				3		5
(5a, ., 6),(6, ., 7)		i+=j		34				6
(12a, ., 13),(13, ., 14)			j+=i		55			13
(22, ., 23)	j >= FIB							22

Fig. 2. Counterexample trace represented by program path, scheduling of operations, data state as variable assignment, and line number as reference

or `j` reaches their limit (which is *fib*(10)), then function `__VERIFIER_error` is reached and a standard verifier can check this by using the specification `G ! call(__VERIFIER_error())` and let it produce a counterexample path. This case can happen if the assignments `i+=j` and `j+=i` in the two threads `id1` and `id2` are executed in alternating order for all iterations of the loops. The rest of the loop statements in both threads, i. e., checking the loop bound, can be executed in arbitrary ordering here and allows a wide range of possible thread interleaving.

The following command line runs CPAchecker as a verifier, configured to use an explicit-value-based analysis for verifying multi-threaded programs:

```
scripts/cpa.sh \
    -outputpath verification \
    -setprop counterexample.export.graphml=witness.graphml \
    -setprop counterexample.export.compressWitness=false \
    -spec config/properties/unreach-call.prp \
    -valueAnalysis-concurrency \
    fib.c
```

This command specifies the directory for all output (including the witness file), the name of the witness file (without compressing it), the specification (which searches for the function call `__VERIFIER_error`), the domain-specific analysis for the verification process, and the subject program.

Fig. 3. Graphical representation of a violation witness and the available data

The verification process starts the analysis at the program entry and explores the reachable state space. In this example, it finds and reports an error trace as a program path (first column of Fig. 2) and provides the violation witness in Fig. 3, which is written into the file `verification/witness.graphml.gz`. Both, the counterexample trace and the violation witness specify the interleaved thread execution and variable assignments, such that a user or a witness validator can directly follow the path until reaching the property violation in the program. We highlight the information that is relevant for the thread interleaving. The violation witness uses sink nodes for branches or thread interleavings that do not follow the counterexample path. For simplicity, we avoid them in the graphical representation.

Using the explicit-value domain allows us to export detailed data about the counterexample trace, such as assignments for all variables at many program locations. The verification witness is enriched with these assignments, such that the validator can use them as additional constraints.

The information about which thread is executed, and how the interleaving looks like, is important for the user (and also for the validator). In a program with threads created from the same function (that is, with identical line numbers), the thread identifier is the only way to distinguish between different contexts. Therefore, a witness must contain a thread identifier for every transition (edge) in the witness. In this example, the executed threads have different function scopes (`t1` and `t2`) which makes it easier for the reader to find the correct trace towards the property violation.

3.2 Validating Results Based on a Violation Witness

In order to validate the information from the witness, the violation witness is matched against the program source code. As the violation witness describes a limited set of paths (best case: exactly one path), the validation process is expected to be efficient and to only analyze a small portion of the reachable state space of the whole program.

The following command line runs CPAchecker as a validator based on the provided violation witness for the multi-threaded program:

```
scripts/cpa.sh \
    -outputpath validation \
    -spec config/properties/unreach-call.prp \
    -witnessValidation \
    -witness verification/witness.graphml \
    fib.c
```

This command specifies the directory for all output (including the newly generated witness file), the specification (which searches for the function call `__VERIFIER_error`), the validation analysis that will select the strategy to analyze multi-threaded programs, the witness that will be used for the validation (as second, parallel specification), and the subject program. Figure 4

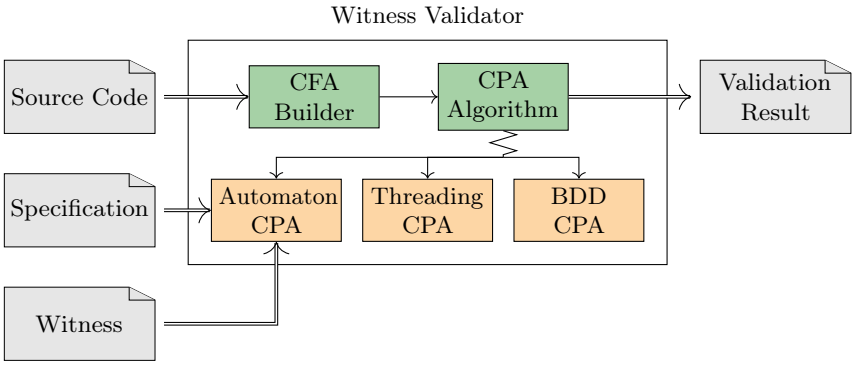


Fig. 4. Overview of CPAchecker's control flow for violation witness validation for multi-threaded programs

shows the architecture of CPAchecker for the witness validation for multi-threaded programs. The program is parsed into a CFA and then given to an analysis based on the CPA concept [17]. The property specification and the violation witness are used as protocol automata.

The validation proceeds with the following steps: The witness validator CPAchecker converts the GraphML file (from Fig. 3) into its internal protocol automaton [11], which includes the constraints of the witness. The analysis then runs this automaton in parallel to the default analysis (reduced product) and strengthens the transition relation of the analysis with the additional constraints from the witness. The analysis starts with an initial abstract state built from the program-entry location in the CFA and the entry node in the witness automaton. Then it computes successors for each state and follows a strategy that aims at getting as deep as possible into the witness automaton. This corresponds with strict guidance from the protocol automaton.

By the definition of the witness and the CFA, it is guaranteed that each step through the violation witness matches one or more edges in the program's CFA. The witness structure guides the search towards the property violation in the program. The validator only confirms a property violation from a violation witness, if both the witness automaton and the program location refer to a property violation according to the specification.

For the example, the validation process reports a property violation and confirms the violation witness. The framework reports the validated counterexample trace in form of a new violation witness, which looks quite similar to the existing one. As our validation process uses the BDD-based domain, intermediate steps can be different and more precise than with the previously used explicit-value analysis. However, exporting data from BDDs is more difficult and CPAchecker does not (yet) support it for the witness export.

4 Violation Witnesses for Multi-threaded Programs

This section gives some details about the extension of the witness format to multi-threaded programs and the implementation of a validator. We used the most obvious way to model traces in multi-threaded programs: specify which thread executes which statement at which point in the trace.

4.1 Extending the Existing Format

A violation witness should contain sufficient information about the verification task, such that a validator can efficiently replay the property violation, that is, without re-analyzing the whole state space of the program. This means that for guiding the validator towards a certain property violation, the witness needs to contain sufficient information about all branching choices. While branching points are obvious in sequential programs —just mark all if-then-else statements—, the situation in a multi-threaded program is more complex. The difficulty is to determine the correct ordering of thread interleavings along the counterexample trace. A detailed look provides us insights about the encoding of thread interleavings in CPAchecker: Each program state represents multiple program counters (i.e., one program location per thread) and thus allows the execution of the follower statement from any available thread. We identified only one single information that is critical for the validator to successfully validate a violation witness for a multi-threaded program: a unique *thread identifier* to identify the actual thread that executes a statement given in the witness. Along a violation witness, the thread identifier is required for two different steps:

- Whenever a new thread is started via a control-flow edge calling `pthread_create`, we insert the information `createThread=<ID>`, where `ID` is a new thread identifier for the new thread. Using these hints on *thread creation*, the validator can register a new thread and follow its control flow.
- The thread interleaving is encoded with the *thread identifier* that is given for each statement in the witness. The information `threadId=<ID>` is added to all control-flow edges in the witness, where the thread `<ID>` executes the statement along the control-flow edge.

To keep the witness format as simple as possible, our extension of the witness format consists of only the two above pieces of information (and even those two are optional, i.e., just act as hints for the validator to find the property violation faster). Overall, this allows verification tools that already have support for exporting violation witness and can analyze concurrent programs to directly export violation witnesses for concurrent programs without larger changes to their code base. We considered to include an explicit notion of thread exit or thread join into the set of critical information, but it turned out that none of these actually helped or improved the performance of the validator. In other words, terminating a single thread is unimportant and the validator can automatically infer such information, whenever a single thread reaches the end of its control flow.

Limitations of the Format. The current witness format does not support assumptions using thread-local scopes of identifiers, such as *x from thread 1 is larger than x from thread 2*. The validator could in principle overcome this limitation by heuristically choosing which thread is responsible for which identifier. This could make validation slow due to a potentially large overhead. Alternatively, we could extend the assumption format, which are currently plain C statements, with fully qualified names. However, that requires several changes to the syntax (parser and exporter) of the assumptions in both producer and consumer of the witness format. Thus, our validator currently ignores assumptions for which it can not deterministically assign the corresponding thread.

The current witness format does not support quantifiers. For a possibly unbounded number of threads in the program, a correctness witness has to provide information (invariants) over all threads, i.e., uses quantifiers such as *forall threads: property violation can not happen.*

4.2 Implementation of the Validator in CPAchecker

CPAchecker transforms a given GraphML-based witness into its internal automaton format, which is then applied along the program analysis to restrict the reachable state space. Additional assumptions over program variables that are given in the witness can be used to strengthen domain-specific transfer relations or cut off the state-space exploration, e. g., if an assumption about a program variable does not satisfy its current assignment.

The validator uses the information from the violation witness for two different features: (1) The state-space exploration is configured to prioritize the search in the direction of the violation witness, i.e., as soon as any control-flow edge from the witness is matching, CPAchecker directly follows that direction. This does not exclude other traces of the program, as they will just be scheduled later in the exploration algorithm. (2) If an assumption is available in the witness, the validator applies *strengthening* and allows to exchange of information between CPAs.

Matching Thread Identifiers. The validator needs to combine the information provided in the witness automaton with its own thread model. The important information for multi-threading is provided as an (optional) thread identifier for each single control-flow edge. The validator assumes that the identifier is unique for any particular state in the witness, and we allow to reuse a thread identifier if its previous usage is out of scope, i.e., the corresponding thread has already exited and was joined.

Our internal thread model uses indices to refer to threads in an abstract state. When validating the violation witness, we create a mapping of the thread identifier from the witness to a possible thread index of our own thread model. This allows the validator to be independent from any concrete representation of a thread identifier in the witness.

Analyses in CPAchecker with Support for Multi-threaded Programs. The validation for violation witnesses uses the default CPA algorithm [17], which provides an efficient state-space exploration and can be combined with CEGAR.

With the CPA concept, we combine independent analyses (CPAs) that work for different aspects of the program analysis. The automaton analysis handles the matching against the specification automaton and the witness automaton, The threading analysis [13] manages the thread scheduling and interleaving. Additional CPAs like an explicit-value analysis [21], a BDD-based [23], or interval-based analysis allow to reason about assignments of variables.

For validation of violation witnesses, we additionally *strengthen* the abstract threading state with information provided in the witness automaton, in order to track the mapping of thread identifiers and thread indices, and to cut off irrelevant branches in the state space eagerly.

Limitations of the Validator. There are some conceptional or implementation-defined limitations of the current implementation of the validator. We discuss these limitations to encourage developers of future validators for multi-threaded programs to improve the approach in our tool, to extend other validators for sequential programs by support for multi-threaded tasks, or to provide new validators.

Based on the requirement to prepare a CFA for each thread of a multi-threaded program, there is a fixed upper bound in the number of threads (default value is 5). The validator ignores traces that use more than the given number of threads, which is an unsound approximation. Note that this is no general limitation of the witness format or the validator. Each concrete error trace for a violation witness has a bounded length and thus can only use a bounded number of thread interleavings. (For example, the number of threads could be added to the metadata of the violation witness and the limit value could be set appropriately.) For the evaluated verification tasks, the default limit was sufficient. If the violation witness is to imprecise and the program allows to create more threads than given in the violation witness, the validator can of course also apply the analysis for more threads. Due to our simple threading analysis, we can only track threads with simple thread-identifier assignments, i.e., where the thread itself is not assigned to an array element or complex pointer structure.

CPAchecker currently supports two domains concretely for analyzing multi-threaded programs, which are explicit-value analysis and BDD-based analysis. The default is to use the BDD-based approach, as it can also handle symbolic values. The validator inherits the limitations of those domains, e.g., it has only limited support for heap-related data structures, such that we need to ignore most array- or pointer-related operations, which can make the validation process imprecise and in some cases even unsound (in case of pointer assignments). This leads to two general cases in which a validator can be wrong: (a) there could be a perfectly valid violation witness but the validator cannot replay it and rejects it due to missing feature support and (b) there could be an invalid violation witness (does not describe a feasible error path) but the validator still finds a different feasible counterexample itself and accepts it due to imprecise information in the witness. There were a few such cases in SV-COMP 2020. The following examples are extracted from the results of SV-COMP 2020 by manual investigation, to give an impression for unsupported features and how they show up in the results[3]:

[3] SV-COMP published all referenced witnesses [9] and verification tasks [8].

- CBMC provides a valid (rather short) violation witness[4] for the task `tls_destructor_worker.yml`[5]. The validator with BDD-based analysis can not confirm this witness due to missing support for `pthread_create_key` and pointer operations.
- ESBMC provides a valid violation witness[6] for the task `race-2_3-container_of.yml`[7], which the validator with BDD-based analysis can not confirm due to missing support for structs.
- YOGAR-CBMC provides a valid violation witness[8] for the task `bigshot_p.yml`[9], for which the validator aborts due to an unexpected assignment of a thread identifier into an array element.

So far, the presented validator is the only validator for multi-threaded programs, and it participated already three years in the competition of software verification (since SV-COMP 2018).

5 Experimental Evaluation

We perform an experimental evaluation on violation witnesses for multi-threaded programs to provide qualitative and quantitative insights on how well the result validation based on violation witnesses for multi-threaded programs works.

5.1 Evaluation Questions

We split our experimental evaluation into the following evaluation questions:

Q1: Which verifiers already support the export of violation witnesses for multi-threaded programs after a successful verification run and what kind of information about the counterexample trace is provided within the witness.
Q2: Is the format sufficient and concrete enough for the validator to re-verify the counterexample trace?
Q3: Is the validation process faster than the verification process?

[4] https://sv-comp.sosy-lab.org/2020/results/fileByHash/c4a519d36a719304f05e0af3675a0bcf40a7ce4d5000fba784365eed63105ee0.graphml
[5] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread-divine/tls_destructor_worker.yml
[6] https://sv-comp.sosy-lab.org/2020/results/fileByHash/784befbee140f91b180268f489a6cdce2471ffc6f8578fb0e361c3d2953313d1.graphml
[7] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/ldv-races/race-2_3-container_of.yml
[8] https://sv-comp.sosy-lab.org/2020/results/fileByHash/f197f473759cc28e4845bcfc6f92af00c0d3ad27e020ee9db1029bfd7c854dba.graphml
[9] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp20/c/pthread/bigshot_p.yml

5.2 Benchmark Set

We evaluate the witness format and the validator on a large set of verification tasks, which is taken from the SV-Benchmarks collection [8][10], in the same version as used for SV-COMP 2020. We limit the benchmark set to the subset of verification tasks that exactly matches the category *ConcurrencySafety* in SV-COMP 2020, i.e., multi-threaded programs with a reachability property as specification.

5.3 Setup

Our experiments were executed on computers with Intel Xeon E3-1230 v5 CPUs, 3.40 GHz CPU frequency, and 33 GB of RAM. We limited the CPU time to 15 min and the memory to 15 GB.

We evaluated our validator on violation witnesses from SV-COMP [9] that were produced by several different software verifiers. We selected those verifiers that participated in SV-COMP 2020 [7], support violation witnesses (produced more than 100 such witnesses that were confirmed), and have publicly available archives on GitLab[11]. Those verifiers are the following seven: Cbmc, CPA-Seq, Divine, Esbmc, Lazy-CSeq, PeSCo, and Yogar-CBMC. In addition to the witnesses that we took from SV-COMP [9], we also used an updated version of CPAchecker (revision `r33531`) to produce witnesses, where a small extension for the export of violation witnesses was applied (add all beneficial information about thread identifiers to the violation witness and consider more thread interleavings). We include this additional verifier to show that a small and inexpensive extension can lead to a significant improvement of the validation results. The CPU time and memory consumption for each verification run was measured by SV-COMP using BenchExec [22], and the number of nodes and transitions was counted using the GraphML witness files.

Currently, there is only one validator available for violation witnesses of multi-threaded programs, which is the validator explained in Sect. 4.2 and implemented in the CPAchecker framework[2]. We use revision `r33531` for the experiments.

5.4 Results and Discussion

Q1: Verifier Support and Available Information. All verifiers that we considered in our experiments support (1) the verification of multi-threaded programs and (2) the export of violation witnesses. Some tools include the beneficial information about thread interleaving in the violation witness. That specific feature was already requested in SV-COMP 2018, when the organizers extended the validation of violation witnesses to the category of concurrent tasks. This shows that our extension was already adopted to other verification tools. However, the availability and the quality of the integration differs between the tools.

[10] https://github.com/sosy-lab/sv-benchmarks/tree/svcomp20
[11] https://gitlab.com/sosy-lab/sv-comp/archives-2020/-/tree/svcomp20/2020

Table 1. Statistical description of the generated witnesses for the verifiers

Verifier	Number of states				Number of transitions			
	Median	Mean	Max	Sum	Median	Mean	Max	Sum
Cbmc	6.00	6.05	10	4 790	4.00	4.05	8	3 210
CPA-Seq	48.0	48.7	662	38 700	82.0	84.4	744	67 000
CPAchecker (r33531)	141	140	1 480	112 000	207	202	1 620	161 000
Divine	3.00	3.05	5	1 820	2.00	2.05	4	1 220
Esbmc	3.00	5.19	30	4 140	2.00	4.19	29	3 340
Lazy-CSeq	66.0	64.1	156	52 100	64.0	62.1	154	50 500
PeSCo	51.0	49.5	662	38 600	83.0	85.9	744	67 000
Yogar-CBMC	86.0	84.7	188	68 300	84.0	82.7	186	66 700

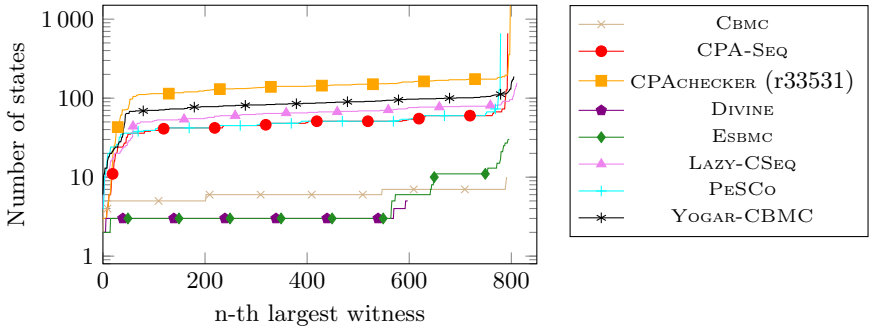
Table 2. Properties of the exported violation witnesses

Verifier	Thread id	Thread creation	All thread interleavings
Cbmc		✓	
CPA-Seq	✓	✓	
CPAchecker (r33531)	✓	✓	✓
Divine			
Esbmc			
Lazy-CSeq	✓	✓	✓
PeSCo	✓	✓	
Yogar-CBMC	✓	✓	✓

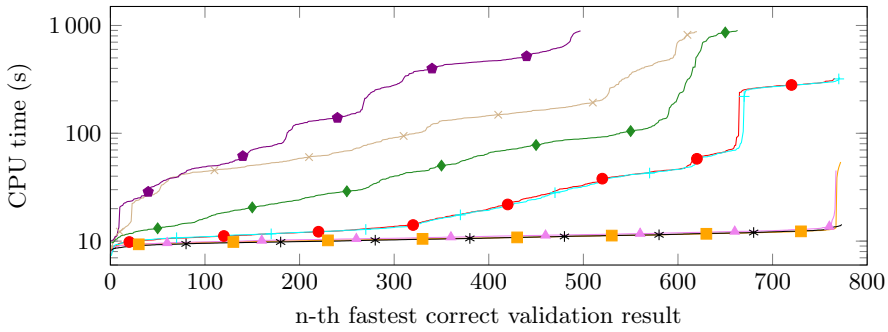
Table 1 gives a statistical overview of the provided violation witnesses and shows how many states and transitions are available in the violation witnesses. Figure 5a shows the distribution of sizes of the violation witnesses for different verifiers. As most tasks have roughly equal difficulty and length of the counterexample trace, the sizes of the violation witnesses are in a certain range. The noticeable difference comes with the tools themselves, i.e., some tools export more details than others.

We also inspected the witnesses for the kind of information they contain. Table 2 shows the different kinds of information available in the witnesses produced by the verifiers. We analyzed whether the violation witnesses contain the *thread id* for every transition, information about *thread creation* for newly started threads during the counterexample trace, and information about thread interleaving. Cbmc, Divine, and Esbmc only export the main thread of the multi-threaded program, which is not suitable for a counterexample trace with interleaving thread statements, because all information about other threads is missing.

Q2: Validation Results. The validation results for the produced violation witnesses show whether the information from the violation witness was sufficient to guide the validator towards confirming the given counterexample trace. Overall, the performance of the validation run is determined by two factors: first, how well the violation witness itself guides the state-space exploration and defines the thread



(a) Size of witnesses



(b) Runtime of validation of witnesses

Fig. 5. Quantile plots for violation witnesses from different verifiers

scheduling, and second, how precise the data in the violation witness are. The less information is provided in the violation witness, the more work is left to the validator with its heuristics to recover the error trace. In other words, more precise violation witnesses are often validated faster than less precise witnesses. Figure 5b shows the CPU time of the validator for violation witnesses from different verification tools. Comparing the results with the annotations exported by the tools (Table 2) leads us to a first conclusion: exporting thread interleavings is critical for finding a concrete counterexample path through the program during validation.

As Cbmc, Divine, and Esbmc produce violation witnesses that contain only a minimal set of nodes and transitions, especially only consisting of the main function of the program and ignoring any additional threads, the validation can not follow the given trace sufficiently and performs worse than for other violation witnesses. CPA-Seq and PeSCo use the same underlying analysis, i.e., both tools apply CPAchecker's BDD-based concurrency analyzer with nearly identical options. Thus, they produce nearly identical violation witnesses which also results in similar validation performance.

For the three tools that export thread interleavings in the violation witness, the validation is fast and precise for most of the available verification tasks. Apart from the startup time of the validator (due to starting the Java VM), the

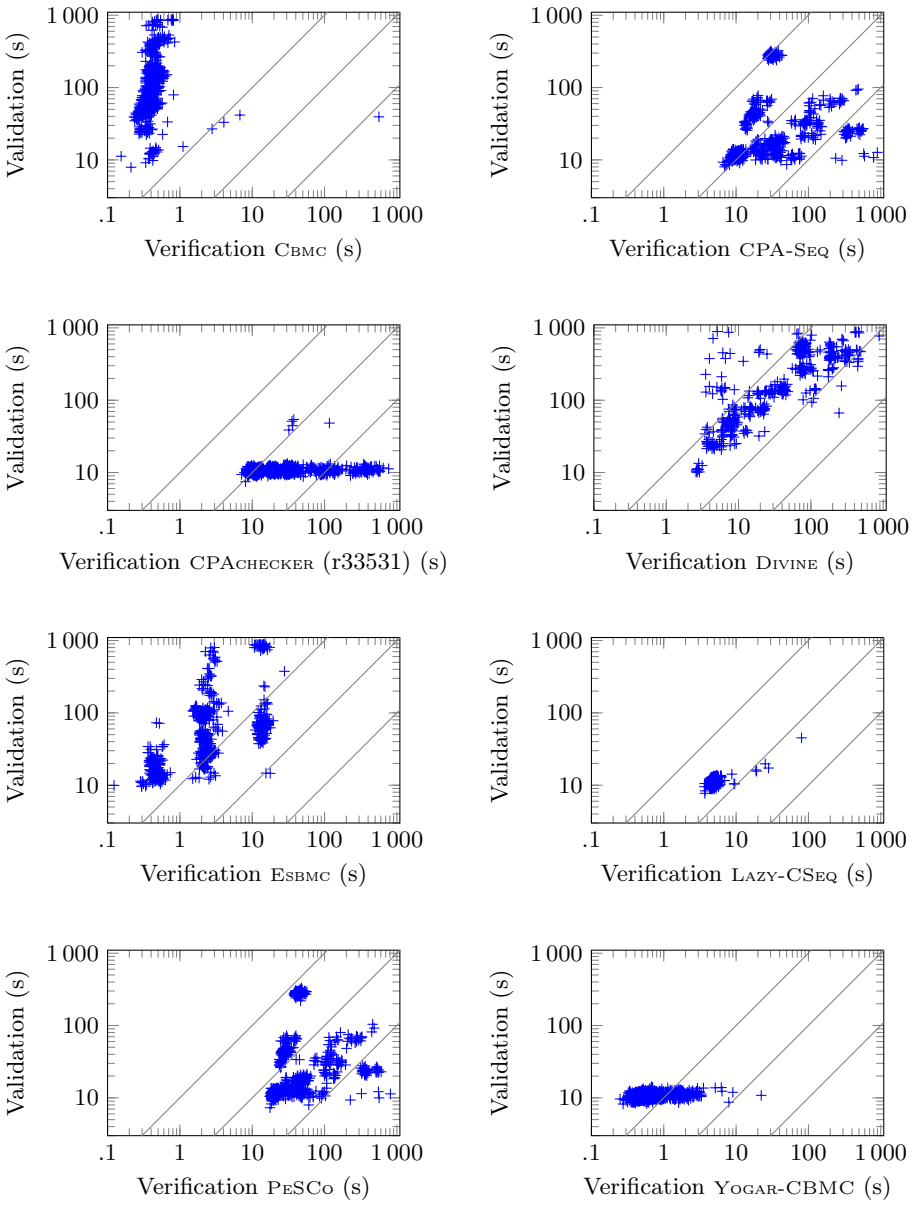


Fig. 6. Scatter plot showing the CPU time of the verification process of several tools against the CPU time of the validation process

runtime of the validation itself is negligible. The violation witness guides the validator in the right direction, i.e., all scheduling information is available and nearly no overhead from unimportant program traces appears in the validation process. Only some validation tasks suffer from a high runtime, but these cases also suffer from a rather long and complex to find counterexample trace, such that the violation witness itself contains several thousands of nodes. Note that depending on the verifier, a different path might have been determined, resulting in violation witnesses of different length for each verification task.

Q3: Performance of Validation Compared to Verification. Based on the CPU time consumed by the verifiers and the CPU time consumed by the validator, we can compare the performance of the validation with the performance of the verification per verifier. Figure 6 shows several scatter plots, each comparing for a given verifier the CPU times for successful validation runs against the corresponding verification runs. Each data point in the scatter plots represents a verification task that was verified and the resulting violation witness was then successfully validated. The three diagonal lines indicate the factors of 0.1, 1, and 10 between the coordinates.

The overall picture for all scatter plots is as follows: The validator (as part of CPAchecker) is written in Java and has a large startup overhead. This makes it difficult to see a clear performance difference for the small and fast verification tasks. For Cbmc and Esbmc, which are tools with only very imprecise witnesses, the validation usually needs much more CPU time than the actual verification took, i.e., the validator needed much more time to find a counterexample trace matching the rudimentary information in the violation witness. Divine not only has quite imprecise witnesses, but also requires more CPU time for the verification process; thus the difference to the time required for the validation is smaller. For more precise witnesses, as produced by Lazy-CSeq, Yogar-CBMC, and CPAchecker (r33531), the validation process is often faster, or at least requires mostly a nearly constant time (about 10 s).

5.5 Threats to Validity

The validity of our experiments is limited by certain choices that we made in the experiment setup.

External Validity. The verifiers are all state-of-the-art and seven of them are taken from SV-COMP 2020. We applied the same options and a similar environment that was used in the competition execution, and collected the violation witnesses from the selected verifiers.

There exists only a single validator for multi-threaded violation witnesses, and it might be possible that our results (sufficient information in the witness format) does not apply to other, future validators for multi-threaded programs. We based our validator on the verification framework CPAchecker, because mechanisms for witness export and validation was already integrated. The configuration using a BDD-based analysis is currently the most performant approach for multi-threaded

programs in the framework. The heuristics for exploring the abstract state space are tuned to match witnesses from a broad range of verifiers.

The community-based SV-Benchmarks repository is a largest and most divers collection of verification tasks for the language C. We used all verification tasks that were also used by the most recent competition: category *ConcurrencySafety*.

Internal Validity. The validator might contain programming bugs, but we based our validator on the infrastructure that is used by the verifier CPA-Seq, which performed extremely well in the recent competitions. Thus, we believe that the implementation has a high quality. Also, previous versions of our validator participated in the competition since SV-COMP 2018. Limitations of the validator were discussed in depth in Sect. 4.2.

The execution of the verification and validation runs was done with BenchExec [22], the (only available) state-of-the-art benchmarking tool, which is also used by the StarExec competition infrastructure and competitions like SV-COMP and Test-Comp. BenchExec is used to enforce the limits and collect measurements for the consumed resources (CPU time and memory).

6 Conclusion

While validation of verification results for sequential programs has been thoroughly described in 2015, validation support for multi-threaded programs was not yet described in the literature. This paper closes this gap by describing the (only available) validator for multi-threaded programs, which was already used three times as validator in the competition on software verification (SV-COMP 2018-2020).

In our evaluation, we report the available features that the witnesses produced by several verifiers expose to the validator, and we report the performance. The results are promising, but it would be better for the verification community to have more such validators available: There are six validators for violation witnesses for sequential programs, but only one for multi-threaded programs.

Data Availability Statement. We make the violation witnesses, a ready-to-run archive of CPAchecker, and all experimental results (including raw data, tables, and plots) available on a supplementary web site[12] and in a Zenodo archive [14]. The verifiers that participated in SV-COMP 2020 have publicly available archives in a GitLab repository.[11] More witnesses and results from SV-COMP can be found in the archives mentioned in the report [7] (Table 4).

References

1. Andrianov, P., Mutilin, V., Khoroshilov, A.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proc. TMPA, CCIS, vol. 779. Springer (2018). https://doi.org/10.1007/978-3-319-71734-0_2

[12] https://www.sosy-lab.org/research/witnesses-concurrency

2. Artho, C., Havelund, K., Honiden, S.: Visualization of concurrent program executions. In: Proc. COMPSAC, pp. 541–546. IEEE (2007). https://doi.org/10.1109/COMPSAC.2007.236
3. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with Divine 4. In: Proc. ATVA, LNCS, vol. 10482, pp. 201–207. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
4. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS, LNCS, vol. 9035, pp. 401–416. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
5. Beyer, D.: Reliable and reproducible competition results with BenchExec and witnesses (Report on SV-COMP 2016). In: Proc. TACAS, LNCS, vol. 9636, pp. 887–904. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
6. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS, LNCS, vol. 10206, pp. 331–349. Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
7. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2), LNCS, vol. 12079, pp. 347–367. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
8. Beyer, D.: SV-Benchmarks: Benchmark set of 9th Intl. Competition on Software Verification (SV-COMP 2020). Zenodo (2020). https://doi.org/10.5281/zenodo.3633334
9. Beyer, D.: Verification witnesses from SV-COMP 2020 verification tools. Zenodo (2020). https://doi.org/10.5281/zenodo.3630188
10. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
11. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
12. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP, LNCS, vol. 10889, pp. 3–23. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
13. Beyer, D., Friedberger, K.: A light-weight approach for verifying multi-threaded programs with CPAchecker. In: Proc. MEMICS, EPTCS, vol. 233, pp. 61–71 (2016). https://doi.org/10.4204/EPTCS.233.6
14. Beyer, D., Friedberger, K.: Replication package for article 'Violation witnesses and result validation for multi-threaded programs'. Zenodo (2020). https://doi.org/10.5281/zenodo.3885694
15. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
16. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV, LNCS, vol. 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
17. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13

18. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
19. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
20. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE, pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
21. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE, LNCS, vol. 7793, pp. 146–162. Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11
22. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2017). https://doi.org/10.1007/s10009-017-0469-y
23. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
24. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR 1905(08505) (May 2019). https://arxiv.org/abs/1905.08505
25. Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Proc. SPIN, LNCS, vol. 7976, pp. 1–17. Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_1
26. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing, LNCS, vol. 2265, pp. 501–512. Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
27. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: Automatically generating inputs of death. In: Proc. CCS, pp. 322–335. ACM (2006). https://doi.org/10.1145/1180405.1180445
28. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE, pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
29. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS, LNCS, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
30. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE, pp. 422–431. ACM (2005). https://doi.org/10.1145/1062455.1062533
31. Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Predicting rankings of software verification tools. In: Proc. SWAN, pp. 23–26. ACM (2017). https://doi.org/10.1145/3121257.3121262
32. Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transfer **19**(1), 97–114 (2017). https://doi.org/10.1007/s10009-015-0407-9
33. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV, LNCS, vol. 11561, pp. 355–365. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
34. Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A., Schwartz, E.J.: Executable counterexamples in software model checking. In: Proc. VSTTE, LNCS, vol. 11294, pp. 17–37. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_2

35. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS, LNCS, vol. 10422, pp. 128–147. Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
36. Gunter, E.L., Peled, D.A.: Path exploration tool. In: Proc. TACAS, LNCS, vol. 1579, pp. 405–419. Springer (1999). https://doi.org/10.1007/3-540-49059-0_28
37. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV, LNCS, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
38. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS, LNCS, vol. 8413, pp. 398–401. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
39. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPoPP. ACM (2020)
40. Yin, L., Dong, W., Liu, W., Wang, J.: On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. Softw. Eng. (2018). https://doi.org/10.1109/TSE.2018.2864122