Automatic Verification of C and Java Programs: SV-COMP 2019

Dirk Beyer

LMU Munich, Munich, Germany

Abstract. This report describes the 2019 Competition on Software Verification (SV-COMP), the 8th edition of a series of comparative evaluations of fully automatic software verifiers for C programs, and now also for Java programs. The competition provides a snapshot of the current state of the art in the area, and has a strong focus on replicability of its results. The repository of benchmark verification tasks now supports a new, more flexible format for task definitions (based on YAML), which was a precondition for conveniently benchmarking Java programs in the same controlled competition setting that was successfully applied in the previous years. The competition was based on 10 522 verification tasks for C programs and 368 verification tasks for Java programs. Each verification task consisted of a program and a property (reachability, memory safety, overflows, termination). SV-COMP 2019 had 31 participating verification systems from 14 countries.

1 Introduction

Software verification is an increasingly important research area, and the annual Competition on Software Verification (SV-COMP)[1] is the showcase of the state of the art in the area, in particular, of the effectiveness and efficiency that is currently achieved by tool implementations of the most recent ideas, concepts, and algorithms for fully automatic verification. Every year, the SV-COMP project consists of two parts: (1) The collection of verification tasks and their partitioning into categories has to take place before the actual experiments start, and requires quality-assurance work on the source code in order to ensure a high-quality evaluation. It is important that the SV-COMP verification tasks reflect what the research and development community considers interesting and challenging for evaluating the effectivity (soundness and completeness) and efficiency (performance) of state-of-the-art verification tools. (2) The actual experiments of the comparative evaluation of the relevant tool implementations is performed by the organizer of SV-COMP. Since SV-COMP shall stimulate and showcase new technology, it is necessary to explore and define standards for a reliable and reproducible execution of such a competition: we use BenchExec [19], a modern framework for reliable benchmarking and resource measurement, to run the experiments, and verification witnesses [14,15] to validate the verification results.

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

https://doi.org/10.1007/978-3-030-17502-3_9

As for every edition, this SV-COMP report describes the (updated) rules and definitions, presents the competition results, and discusses other interesting facts about the execution of the competition experiments. Also, we need to measure the success of SV-COMP by evaluating whether the main objectives of the competition are achieved (cf. [10]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,
2. establish a repository of software-verification tasks that is publicly available for free use as standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results, and
4. accelerate the transfer of new verification technology to industrial practice.

As for (1), there were 31 participating software systems from 14 countries, representing a broad spectrum of technologies (cf. Table 5). SV-COMP is considered an important event in the research community, and increasingly also in industry. As for (2), the total set of verification tasks written in C increased in size from 8 908 tasks to 10 522 tasks from 2017 to 2019, and in addition, 368 tasks written in Java were added for 2019. Still, SV-COMP has an ongoing focus on collecting and constructing verification tasks to ensure even more diversity, as witnessed by the issue tracker[2] and by the pull requests[3] in the GitHub project. As for (3), the largest step forward was to establish a exchangeable standard format for verification witnesses. This means that verification results are fully counted only if they can be independently validated. As for (4), we continuously receive positive feedback from industry. Colleagues from industry reported to us that they observe SV-COMP in order to know about the newest and best available verification tools. Moreover, since SV-COMP 2017 there are also a few participating systems from industry.

Related Competitions. It is well-understood that competitions are an important evaluation method, and there are other competitions in the field of software verification: RERS[4] [40] and VerifyThis[5] [41]. While SV-COMP performs replicable experiments in a *controlled* environment (dedicated resources, resource limits), the RERS Challenges give more room for exploring combinations of interactive with automatic approaches without limits on the resources, and the VerifyThis Competition focuses on evaluating approaches and ideas rather than on *fully automatic* verification. The termination competition termCOMP[6] [33] concentrates on termination but considers a broader range of systems, including logic and functional programs. This year, SV-COMP is part of TOOLympics [6]. A more comprehensive list of other competitions is provided in the report on

[2] https://github.com/sosy-lab/sv-benchmarks/issues
[3] https://github.com/sosy-lab/sv-benchmarks/pulls
[4] http://rers-challenge.org
[5] http://etaps2016.verifythis.org
[6] http://termination-portal.org/wiki/Termination_Competition

SV-COMP 2014 [9]. There are other large benchmark collections as well (e.g., by SPEC[7]), but the `sv-benchmark` suite[8] is (a) free of charge and (b) tailored to the state of the art in software verification. There is a certain flow of benchmark sets between benchmark repositories: For example, the `sv-benchmark` suite contains programs that were used in RERS[9] or in termCOMP[10] before.

2 Procedure

The overall competition organization did not change in comparison to the past editions [7–12]. SV-COMP is an open competition, where all verification tasks are known before the submission of the participating verifiers, which is necessary due to the complexity of the C language. During the *benchmark submission* phase, new verification tasks were collected, classified, and added to the existing benchmark suite (i.e., SV-COMP uses an accumulating benchmark suite), during the *training* phase, the teams inspected the verification tasks and trained their verifiers (also, the verification tasks received fixes and quality improvement), and during the *evaluation* phase, verification runs were preformed with all competition candidates, and the system descriptions were reviewed by the competition jury. The participants received the results of their verifier directly via e-mail, and after a few days of inspection, the results were publicly announced on the competition web site. The *Competition Jury* consisted again of the chair and one member of each participating team. Team representatives of the jury are listed in Table 4.

3 Definitions, Formats, and Rules

License Requirements. Starting 2018, SV-COMP required that the verifier must be publicly available for download and has a license that

(i) allows replication and evaluation by anybody (including results publication),
(ii) does not restrict the usage of the verifier output (log files, witnesses), and
(iii) allows any kind of (re-)distribution of the unmodified verifier archive.

Verification Tasks. The definition of verification tasks was not changed and we refer to previous reports for more details [9,12]. The validation of the results based on verification witnesses [14,15] was done exactly as in the previous years (2017, 2018), mandatory for *both* answers True or False; the only change was that an additional new, execution-based witness validator [16] was used. A few categories were excluded from validation if the validators did not sufficiently support a certain kind of program or property.

Categories, Properties, Scoring Schema, and Ranking. The categories are listed in Tables 6 and 7 and described in detail on the competition web site.[11]

[7] https://www.spec.org
[8] https://github.com/sosy-lab/sv-benchmarks
[9] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/eca-rers2012/README.txt
[10] https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/termination-restricted-15/README.txt
[11] https://sv-comp.sosy-lab.org/2019/benchmarks.php

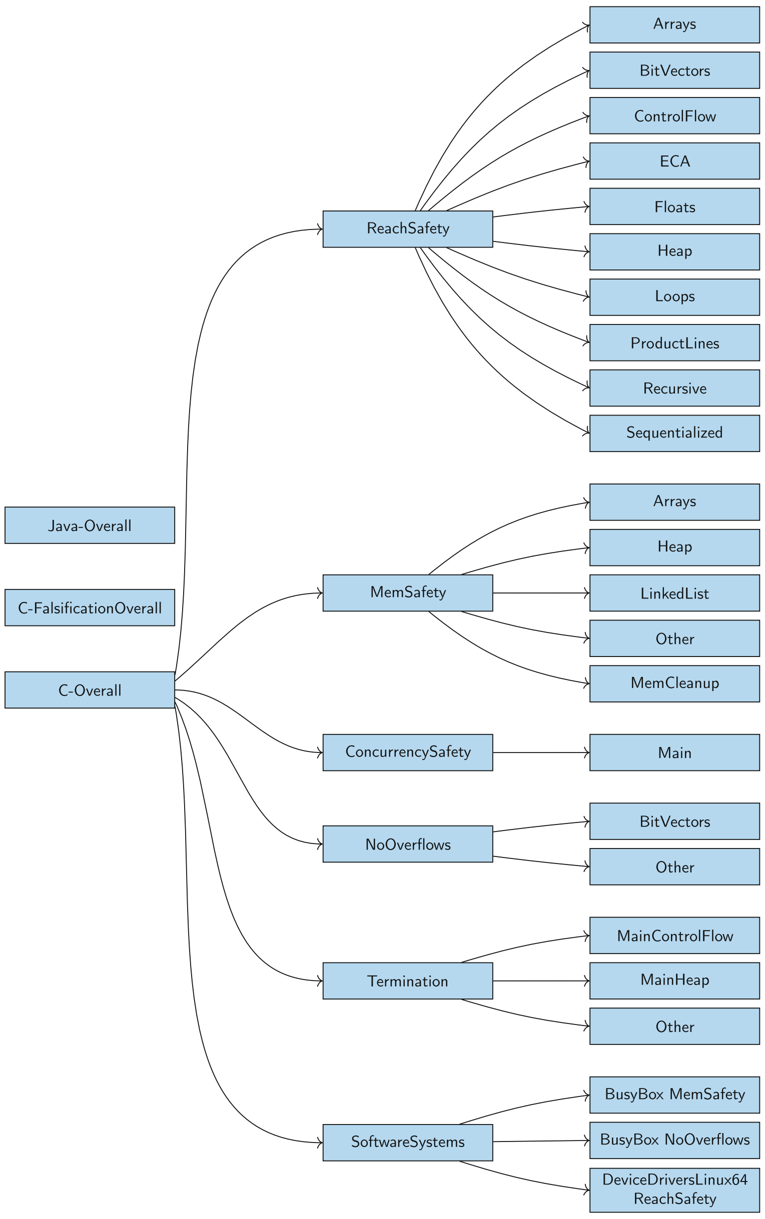

Fig. 1. Category structure for SV-COMP 2019; category *C-FalsificationOverall* contains all verification tasks of *C-Overall* without *Termination*; *Java-Overall* contains all Java verification tasks

Table 1. Properties used in SV-COMP 2019 (`G valid-memcleanup` is new)

Formula	Interpretation
`G ! call(foo())`	A call to function `foo` is not reachable on any finite execution.
`G valid-free`	All memory deallocations are valid (counterexample: invalid free). More precisely: There exists no finite execution of the program on which an invalid memory deallocation occurs.
`G valid-deref`	All pointer dereferences are valid (counterexample: invalid dereference). More precisely: There exists no finite execution of the program on which an invalid pointer dereference occurs.
`G valid-memtrack`	All allocated memory is tracked, i.e., pointed to or deallocated (counterexample: memory leak). More precisely: There exists no finite execution of the program on which the program lost track of some previously allocated memory.
`G valid-memcleanup`	All allocated memory is deallocated before the program terminates. In addition to valid-memtrack: There exists no finite execution of the program on which the program terminates but still points to allocated memory. (Comparison to Valgrind: This property can be violated even if Valgrind reports 'still reachable'.)
`F end`	All program executions are finite and end on proposition `end`, which marks all program exits (counterexample: infinite loop). More precisely: There exists no execution of the program on which the program never terminates.

Table 2. Scoring schema for SV-COMP 2019 (unchanged since 2017 [12])

Reported result	Points	Description
Unknown	0	Failure to compute verification result
False correct	+1	Violation of property in program was correctly found and a validator confirmed the result based on a witness
False incorrect	−16	Violation reported but property holds (false alarm)
True correct	+2	Program correctly reported to satisfy property and a validator confirmed the result based on a witness
True correct unconfirmed	+1	Program correctly reported to satisfy property, but the witness was not confirmed by a validator
True incorrect	−32	Incorrect program reported as correct (wrong proof)

Figure 1 shows the category composition. For the definition of the properties and the property format, we refer to the 2015 competition report [10]. All specifications are available in the directory `c/properties/` of the benchmark repository. Table 1 lists the properties and their syntactical representation as overview. Property `G valid-memcleanup` was used for the first time in SV-COMP 2019.

The scoring schema is identical for SV-COMP 2017–2019: Table 2 provides the overview and Fig. 2 visually illustrates the score assignment for one property. The scoring schema still contains the special rule for unconfirmed correct results for

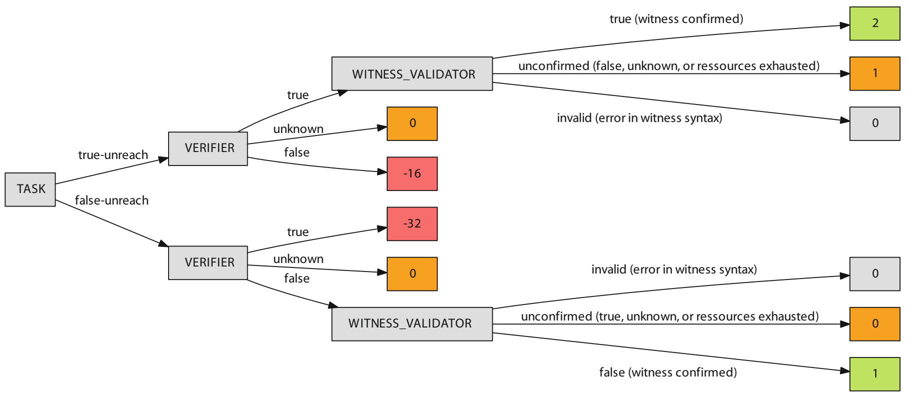
TASK
true-unreach
false-unreach
VERIFIER
true
unknown
false
WITNESS_VALIDATOR
true (witness confirmed)
unconfirmed (false, unknown, or ressources exhausted)
invalid (error in witness syntax)
2
1
0
0
-16
VERIFIER
true
unknown
false
-32
0
WITNESS_VALIDATOR
invalid (error in witness syntax)
unconfirmed (true, unknown, or ressources exhausted)
false (witness confirmed)
0
0
1

Fig. 2. Visualization of the scoring schema for the reachability property

expected result TRUE that was introduced in the transitioning phase: one point is assigned if the answer matches the expected result but the witness was not confirmed. Starting with SV-COMP 2020, the single-point rule will be dropped, i.e., points are only assigned if the results got validated or no validator was available.

The ranking was again decided based on the sum of points (normalized for meta categories) and for equal sum of points according to success run time, which is the total CPU time over all verification tasks for which the verifier reported a correct verification result. *Opt-out from Categories* and *Score Normalization for Meta Categories* was done as described previously [8] (page 597).

4 New Format for Task Definitions

Technically, we need a verification task (a pair of a program and a specification to verify) to feed as input to the verifier, and an expected result against which we check the answer that the verifier returns. We changed the format of how these tasks are specified for SV-COMP 2019: The C track is still based on the old format, while the Java track already uses the new format.

Recap: Old Format. Previously, the above-mentioned three components were specified in the file name of the program. For example, consider the file name `c/ntdrivers/floppy_true-unreach-call_true-valid-memsafety.i.cil.c`, which encodes the program, the specification (consisting of two properties), and two expected results (one for each property) in the following way:

- **Program:** The program file is identified using the file name `floppy_true-unreach-call_true-valid-memsafety.i.cil.c` in directory `c/ntdrivers/`. The original program was named as `floppy` (see [17]).
- **Specification:** The program comes with a specification that consists of two properties `unreach-call` and `valid-memsafety` thus, the two verification tasks (`floppy`, `unreach-call`) and (`floppy`, `valid-memsafety`) are defined.
- **Expected results:** The expected result for both verification tasks is `true`.

```
 1 format_version: '1.0'
 2
 3 # old file name: floppy_true-unreach-call_true-valid-memsafety.i.cil.c
 4 input_files: 'floppy.i.cil-3.c'
 5
 6 properties:
 7   - property_file: ../properties/unreach-call.prp
 8     expected_verdict: true
 9   - property_file: ../properties/valid-memsafety.prp
10     expected_verdict: true
```

Fig. 3. Example task definition for program `floppy.i.cil-3.c`

This format was used for eight years of SV-COMP, because it is easy to understand and use. However, whenever a new property should be added to the specification of a given program, the program's file name needs to be changed, which has negative impact on traceability and maintenance. From SV-COMP 2020 onwards, the repository will use the following new format for all tracks.

Explicit Task-Definition Files. All the above-discussed information is stored in an extra file that contains a structured definition of the verification tasks for a program. For each program, the repository contains the program file and a task-definition file. The above program is available under the name `floppy.i.cil-3.c` and comes with its task-definition file `floppy.i.cil-3.yml`. Figure 3 shows this task definition.

The task definition uses the YAML format as underlying structured data format. It contains a version id of the format (line 1) and can contain comments (line 3). The field `input_files` specifies the input program (example: '`floppy.i.cil-3.c`'), which is either one file or a list of files. The field `properties` lists all properties of the specification for this program. Each property has a field `property_file` that specifies the property file (example: `../properties/unreach-call.prp`) and a field `expected_verdict` that specifies the expected result (example: `true`).

5 Including Java Programs

The first seven editions of SV-COMP considered only programs written in C. In 2019, the competition was extended to include a Java track. Some of the Java programs existed already in the repository, and many other Java programs were contributed by the community [29]. Currently, most of the programs are from the regression-test suites from the verifiers that participate in the Java track; the goal is to substantially increase the benchmark set over the next years.

In principle, the same definitions and rules as for the C track apply, but some technical details need to be slightly adapted for Java programs. Most prominently, the classes of a Java program cannot be inlined into one Java file, which is solved by using the new task-definition format, which allows lists of input files. This required an extension of BenchExec that is present in version 1.17[12] and higher.

[12] https://github.com/sosy-lab/benchexec/releases/tag/1.17

```
CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )
```

(a) Property **c/properties/unreach-call.prp**

```
CHECK( init(Main.main()), LTL(G assert) )
```

(b) Property **java/properties/assert.prp**

Fig. 4. Standard reachability property in comparison for C and for Java

The property for reachability is also slightly different, as shown in Fig. 4: The function call to the start of the program is `Main.main()` instead of `main()`, and the verifiers check that proposition `assert` is always true, instead of checking that `__VERIFIER_error()` is never called. The new proposition `assert` is false where a Java assert statement fails, i.e., the exception `AssertionError` is thrown.

The rules for the C track specify a function `__VERIFIER_nondet_X()` for each type `X` from the set {`bool`, `char`, `int`, `float`, `double`, `loff_t`, `long`, `pchar`, `pointer`, `pthread_t`, `sector_t`, `short`, `size_t`, `u32`, `uchar`, `uint`, `ulong`, `unsigned`, `ushort`} (no side effects, `pointer` for `void *`, etc.) that all return an arbitrary, nondeterministic value ('input' value) of the respective type that may differ for different invocations. Similarly for the Java track: we use a Java class named `org.sosy_lab.sv_benchmarks.Verifier` with the following parameterless static methods: `nondetBoolean`, `nondetByte`, `nondetChar`, `nondetShort`, `nondetInt`, `nondetLong`, `nondetFloat`, `nondetDouble`, and `nondetString`. Each of those methods creates a value of the respective type using functionality from `java.util.Random`. The earlier proposal [29] to use the array of arguments that is passed to the main method to obtain nondeterministic values was not followed. The SV-COMP community found that the explicitly defined methods are better for the competition, and also closer to practice.

Finally, the static method `assume(boolean)` in the same class can be used to assume a certain value range. The implementation halts using `Runtime.getRuntime().halt(1)`. It was proposed [29] to omit this method but in the end the community decided to include it.

6 Reproducibility

It is important that all SV-COMP experiments can be independently replicated, and that the results can be reproduced. Therefore, all major components that are used for the competition need to be publicly available. Figure 5 gives an overview over the components that contribute to the reproducible setup of SV-COMP, and Table 3 provides the details. We refer to a previous report [11] for a description of all components of the SV-COMP organization and how it is ensured that all parts are publicly available for maximal replicability.

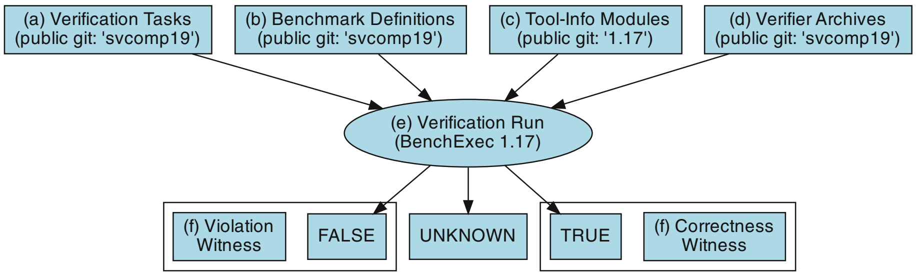
(a) Verification Tasks (public git: 'svcomp19')
(b) Benchmark Definitions (public git: 'svcomp19')
(c) Tool-Info Modules (public git: '1.17')
(d) Verifier Archives (public git: 'svcomp19')
(e) Verification Run (BenchExec 1.17)
(f) Violation Witness
FALSE
UNKNOWN
TRUE
(f) Correctness Witness

Fig. 5. Setup: SV-COMP components and the execution flow

Table 3. Publicly available components for replicating SV-COMP 2019

Component	Fig. 5	Repository	Version
Verification Tasks	(a)	`github.com/sosy-lab/sv-benchmarks`	`svcomp19`
Benchmark Definitions	(b)	`github.com/sosy-lab/sv-comp`	`svcomp19`
Tool-Info Modules	(c)	`github.com/sosy-lab/benchexec`	`1.17`
Verifier Archives	(d)	`gitlab.com/sosy-lab/sv-comp/archives-2019`	`svcomp19`
Benchmarking	(e)	`github.com/sosy-lab/benchexec`	`1.17`
Witness Format	(f)	`github.com/sosy-lab/sv-witnesses`	`svcomp19`

Since SV-COMP 2018, we use a more transparent way of making the verifier archives publicly available. All verifier archives are now stored in a public Git repository. We chose GitLab to host the repository for the verifier archives due to its generous repository size limit of 10 GB (we could not use GitHub, because it has a strict size limit of 100 MB per file, and recommends to keep the repository under 1 GB). An overview table with information about all participating systems is provided in Table 4 and on the competition web site[13].

In addition to providing the components to replicate the experiments, SV-COMP also makes the raw results available in the XML-based exchange format in which BenchExec [20] delivers the data, and also publishes all verification witnesses [13].

7 Results and Discussion

For the eighth time, the competition experiments represent the state of the art in fully automatic software-verification tools. The report shows the improvements compared to last year, in terms of effectiveness (number of verification tasks that can be solved, correctness of the results, as accumulated in the score) and efficiency (resource consumption in terms of CPU time). The results that are presented in this article were inspected and approved by the participating teams.

[13] https://sv-comp.sosy-lab.org/2019/systems.php

Table 4. Competition candidates with tool references and representing jury members

Participant	Ref.	Jury member	Affiliation
2LS	[49,61]	Peter Schrammel	U. of Sussex, UK
AProVE	[34,38]	Jera Hensel	RWTH Aachen, Germany
CBMC	[46]	Michael Tautschnig	Amazon Web Services, UK
CBMC-Path	[44]	Kareem Khazem	U. College London, UK
CPA-BAM-BnB	[1,64]	Vadim Mutilin	ISP RAS, Russia
CPA-Lockator	[2]	Pavel Andrianov	ISP RAS, Russia
CPA-Seq	[18,30]	Marie-Christine Jakobs	LMU Munich, Germany
DepthK	[58,60]	Omar Alhawi	U. of Manchester, UK
DIVINE-explicit	[5,62]	Vladimír Štill	Masaryk U., Czechia
DIVINE-SMT	[47,48]	Henrich Lauko	Masaryk U., Czechia
ESBMC-kind	[31,32]	Mikhail R. Gadelha	U. of Southampton, UK
JayHorn	[42,43]	Philipp Rümmer	Uppsala U., Sweden
JBMC	[27,28]	Lucas Cordeiro	U. of Manchester, UK
JPF	[3,63]	Cyrille Artho	KTH, Sweden
Lazy-CSeq	[50]	Omar Inverso	Gran Sasso Science Inst., Italy
Map2Check	[57,59]	Herbert Rocha	Federal U. of Roraima, Brazil
PeSCo	[56]	Cedric Richter	U. of Paderborn, Germany
Pinaka	[24]	Eti Chaudhary	IIT Hyderabad, India
PredatorHP	[39,45]	Veronika Šoková	BUT, Brno, Czechia
Skink	[21]	Franck Cassez	Macquarie U., Australia
Smack	[36,55]	Zvonimir Rakamaric	U. of Utah, USA
SPF	[51,53]	Willem Visser	Stellenbosch U., South Africa
Symbiotic	[22,23]	Marek Chalupa	Masaryk U., Czechia
UAutomizer	[37]	Matthias Heizmann	U. of Freiburg, Germany
UKojak	[52]	Alexander Nutz	U. of Freiburg, Germany
UTaipan	[35]	Daniel Dietsch	U. of Freiburg, Germany
VeriAbs	[25]	Priyanka Darke	Tata Consultancy Services, India
VeriFuzz	[26]	R. K. Medicherla	Tata Consultancy Services, India
VIAP	[54]	Pritom Rajkhowa	Hong Kong UST, China
Yogar-CBMC	[65,66]	Liangze Yin	Nat. U. of Defense Techn., China
Yogar-CBMC-Par.	[67]	Haining Feng	Nat. U. of Defense Techn., China

Participating Verifiers. Table 4 provides an overview of the participating verification systems and Table 5 lists the features and technologies that are used in the verification tools.

Computing Resources. The resource limits were the same as in the previous competitions [11]: Each verification run was limited to 8 processing units (cores), 15 GB of memory, and 15 min of CPU time. The witness validation was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 15 min of CPU time for correctness witnesses. The machines for running the experiments are part of a compute cluster that consists of 168 machines;

Table 5. Technologies and features that the competition candidates offer

Participant	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✓	✓			✓			✓						✓	
AProVE			✓				✓	✓		✓	✓						✓	
CBMC				✓							✓					✓		
CBMC-Path				✓							✓					✓		
CPA-BAM-BnB	✓	✓					✓				✓	✓	✓	✓				
CPA-Lockator	✓	✓					✓				✓	✓	✓	✓		✓		
CPA-Seq	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
DepthK				✓	✓						✓					✓		
DIVINE-explicit							✓				✓					✓		
DIVINE-SMT							✓				✓					✓		
ESBMC-kind				✓	✓						✓					✓		
JayHorn	✓	✓				✓		✓					✓	✓				
JBMC				✓							✓					✓		
JPF				✓			✓	✓			✓					✓		
Lazy-CSeq				✓							✓					✓		
Map2Check				✓							✓							
PeSCo	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
Pinaka			✓	✓							✓							
PredatorHP									✓									
Skink	✓						✓							✓	✓			
Smack	✓			✓		✓					✓		✓			✓		
SPF			✓						✓							✓		
Symbiotic			✓					✓			✓							
UAutomizer	✓	✓									✓		✓	✓	✓		✓	
UKojak	✓	✓									✓		✓	✓				
UTaipan	✓	✓									✓		✓	✓	✓			
VeriAbs	✓			✓	✓		✓	✓										
VeriFuzz				✓				✓										✓
VIAP																		
Yogar-CBMC	✓			✓							✓		✓			✓		
Yogar-CBMC-Par.	✓			✓							✓		✓			✓		

each verification run was executed on an otherwise completely unloaded, dedicated machine, in order to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04 with Linux kernel 4.15). We used BenchExec [19] to measure and control computing resources (CPU time, memory, CPU energy) and VerifierCloud[14] to distribute, install, run, and clean-up verification runs, and to collect the results.

One complete verification execution of the competition consisted of 418 benchmarks (each verifier on each selected category according to the opt-outs), summing up to 178 674 verification runs. The total consumed CPU time was 461 days for one complete competition run for verification (without validation). Witness-based result validation required 2 645 benchmarks (combinations of verifier, category with witness validation, and a set of validators) summing up to 517 175 validation runs. Each tool was executed several times, in order to make sure no installation issues occur during the execution. Including pre-runs, the infrastructure managed a total of 5 880 071 runs and consumed 15 years and 182 days of CPU time.

Quantitative Results. Table 6 presents the quantitative overview over all tools and all categories. The head row mentions the category, the maximal score for the category, and the number of verification tasks. The tools are listed in alphabetical order; every table row lists the scores of one verifier. We indicate the top three candidates by formatting their scores in bold face and in larger font size. An empty table cell means that the verifier opted-out from the respective main category (perhaps participating in subcategories only, restricting the evaluation to a specific topic). More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web site.[15]

Table 7 reports the top three verifiers for each category. The run time (column 'CPU Time') refers to successfully solved verification tasks (column 'Solved Tasks'). The columns 'False Alarms' and 'Wrong Proofs' report the number of verification tasks for which the verifier reported wrong results: reporting an error path when the property holds (incorrect False) and claiming that the program fulfills the property although it actually contains a bug (incorrect True), respectively.

Discussion of Scoring Schema. The verification community considers computation of correctness proofs to be more difficult than computing error paths: according to Table 2, an answer True yields 2 points (confirmed witness) and 1 point (unconfirmed witness), while an answer False yields 1 point (confirmed witness). This can have consequences for the final ranking, as discussed in the report of SV-COMP 2016 [11]. The data from SV-COMP 2019 draw a different picture.

Table 8 shows the mean and median values for resource consumption regarding CPU time and energy consumption: the first column lists the five best verifiers of category *C-Overall*, the second to fifth columns report the CPU time and CPU energy (mean and median) for results True, and the sixth to ninth

[14] https://vcloud.sosy-lab.org
[15] https://sv-comp.sosy-lab.org/2019/results

Table 6. Quantitative overview over all results; empty cells mark opt-outs

Verifier	**ReachSafety** 6296 points 3831 tasks	**MemSafety** 649 points 434 tasks	**ConcurrencySafety** 1344 points 1082 tasks	**NoOverflows** 574 points 359 tasks	**Termination** 3529 points 2007 tasks	**SoftwareSystems** 4955 points 2809 tasks	**C-FalsificationOverall** 3843 points 8515 tasks	**C-Overall** 16663 points 10522 tasks	**Java-Overall** 532 points 368 tasks
2LS	2397	129	0	280	1279	119	733	4174	
AProVE					**2476**				
CBMC	2781	60	613	227	827	0	1432	4341	
CBMC-Path	1657	-59	-150	192	535	-151	81	1587	
CPA-BAM-BnB						**1185**			
CPA-Lockator			-441						
CPA-Seq	**4299**	**349**	**996**	**431**	**1785**	**1073**	**2823**	**9329**	
DepthK	986	-113	420	39	37	-1182	129	159	
DIVINE-explicit	1413	25	493	0	0	2	200	1547	
DIVINE-SMT	1778	-158	339	0	0	0	-339	726	
ESBMC-kind	3404	-208	404	224	826	714	**1916**	3636	
JayHorn									247
JBMC									**470**
JPF									**290**
Lazy-CSeq			**1245**						
Map2Check		38		8					
PeSCo	**4239**						**2313**	**8466**	
Pinaka				218	561				
PredatorHP		**416**							
Skink									
Smack									
SPF									**365**
Symbiotic	3143	**426**	0	331	1153	555	1828	6129	
UAutomizer	3264	-163	270	**449**	**3001**	1020	1050	**6727**	
UKojak	2195	-211	0	396	0	818	1060	2595	
UTaipan	3012	-91	271	**438**	0	962	1024	4188	
VeriAbs	**4638**					**1061**			
VeriFuzz	1132			123					
VIAP									
Yogar-CBMC			**1277**						
Yogar-CBMC-Par									

Table 7. Overview of the top-three verifiers for each category (CPU time in h, rounded to two significant digits)

Rank	Verifier	Score	CPU Time	Solved Tasks	False Alarms	Wrong Proofs
ReachSafety						
1	**VeriAbs**	**4638**	110	2 811		
2	CPA-Seq	4299	60	2 519		
3	PeSCo	4239	58	2 431	2	
MemSafety						
1	**Symbiotic**	**426**	.030	299		
2	PredatorHP	416	.61	296		
3	CPA-Seq	349	.55	256		
ConcurrencySafety						
1	**Yogar-CBMC**	**1277**	.31	1 026		
2	Lazy-CSeq	1245	3.0	1 006		
3	CPA-Seq	996	13	830		
NoOverflows						
1	**UAutomizer**	**449**	.94	306		
2	UTaipan	438	.96	302		
3	CPA-Seq	431	.59	283		
Termination						
1	**UAutomizer**	**3001**	13	1 662		
2	AProVE	2476	33	1 004		
3	CPA-Seq	1785	15	1 319		
SoftwareSystems						
1	**CPA-BAM-BnB**	**1185**	9.1	1 572		**7**
2	CPA-Seq	1073	28	1 447		
3	VeriAbs	1061	24	1 407		
C-FalsificationOverall						
1	**CPA-Seq**	**2823**	40	2 129		
2	PeSCo	2313	53	2 105	8	
3	ESBMC-kind	1916	15	1 753	14	
C-Overall						
1	**CPA-Seq**	**9329**	120	6 654		
2	PeSCo	8466	120	6 466	8	**1**
3	UAutomizer	6727	85	5 454	5	**10**
Java-Overall						
1	**JBMC**	**470**	2.7	331		
2	SPF	365	.27	337	4	**2**
3	JPF	290	.15	331		**6**

Table 8. Necessary effort to compute results FALSE versus TRUE (measurement values rounded to two significant digits)

Result	TRUE				FALSE			
	CPU Time (s)		CPU Energy (J)		CPU Time (s)		CPU Energy (J)	
	mean	median	mean	median	mean	median	mean	median
CPA-SEQ	67	9.5	690	82	58	14	560	120
PESCO	56	19	540	160	77	26	680	220
UAUTOMIZER	56	17	540	140	58	19	570	180
SYMBIOTIC	4.8	.25	57	2.9	19	.45	210	5.5
CBMC	8.6	.20	91	2.3	21	.24	180	2.8

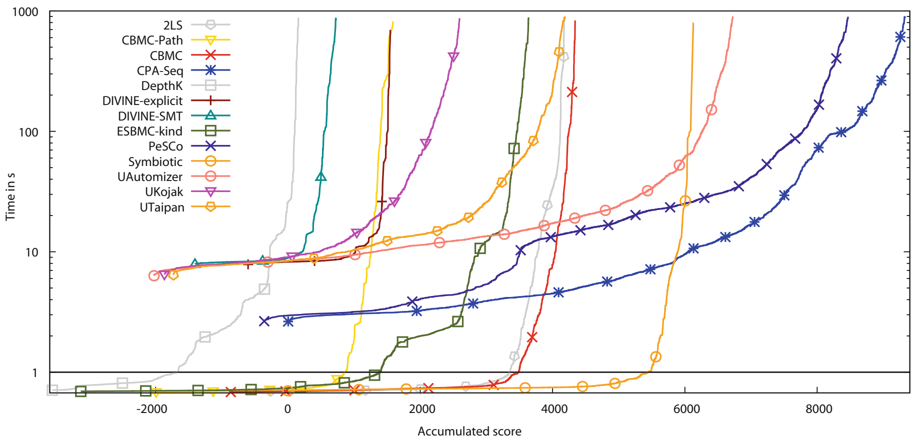

Fig. 6. Quantile functions for category *C-Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by correct verification runs below a certain run time (y-coordinate). More details were given previously [8]. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

columns for results FALSE. The mean and median are taken over successfully solved verification tasks; the CPU time is reported in seconds and the CPU energy in joule (BENCHEXEC reads and accumulates the energy measurements of Intel CPUs using the tool CPU Energy Meter[16]).

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [8] because these visualizations make it easier to understand the results of the comparative evaluation. The web site[15] includes such a plot for each category; as example, we show the plot for category *C-Overall*

[16] https://github.com/sosy-lab/cpu-energy-meter

Table 9. Alternative rankings; quality is given in score points (sp), CPU time in hours (h), energy in kilojoule (kJ), wrong results in errors (E), rank measures in errors per score point (E/sp), joule per score point (J/sp), and score points (sp)

Rank	**Verifier**	**Quality (sp)**	**CPU Time (h)**	**CPU Energy (kJ)**	**Solved Tasks**	**Wrong Results (E)**	**Rank Measure**
Correct Verifiers							(E/sp)
1	**CPA-Seq**	9 329	120	4 300	2 811	0	.0000
2	Symbiotic	6 129	9.7	390	2 519	0	.0000
3	PeSCo	8 466	120	3 900	2 431	9	.0011
worst							.3836
Green Verifiers							(J/sp)
1	**Symbiotic**	6 129	9.7	390	299	0	64
2	CBMC	4 341	11	380	296	14	88
3	DIVINE-explicit	1 547	4.4	180	256	10	120
worst							4 200
New Verifiers							(sp)
1	**PeSCo**	8 466	120	3 900	1 026	9	8 466
2	CBMC-Path	1 587	8.9	380	1 006	69	1 587

(all verification tasks) in Fig. 6. A total of 13 verifiers participated in category *C-Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [8]). A more detailed discussion of score-based quantile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [8,11].

Alternative Rankings. The community suggested to report a couple of alternative rankings that honor different aspects of the verification process as complement to the official SV-COMP ranking. Table 9 is similar to Table 7, but contains the alternative ranking categories *Correct*, *Green*, and *New Verifiers*. Column 'Quality' gives the score in score points, column 'CPU Time' the CPU usage of successful runs in hours, column 'CPU Energy' the CPU usage of successful runs in kilojoule, column 'Solved Tasks' the number of correct results, column 'Wrong results' the sum of false alarms and wrong proofs in number of errors, and column 'Rank Measure' gives the measure to determine the alternative rank.

Correct Verifiers—Low Failure Rate. The right-most columns of Table 7 report that the verifiers achieve a high degree of correctness (all top three verifiers in the C track have less than 1% wrong results). The winners of category *C-Overall* and *Java-Overall* produced not a single wrong answer.

The first category in Table 9 uses a failure rate as rank measure: $\frac{\text{number of incorrect results}}{\text{total score}}$, the number of errors per score point (E/sp). We use E as unit for number of incorrect results and sp as unit for total score. The total score is used as tie-breaker to distinguish the rank of error-free verifiers.

Table 10. Confirmation rate of verification witnesses in SV-COMP 2019

Result	TRUE				FALSE			
	Total	Confirmed		Unconf.	Total	Confirmed		Unconf.
CPA-SEQ	4 417	3 968	90 %	449	2 859	2 686	94 %	173
PESCO	4 176	3 814	91 %	362	2 823	2 652	94 %	171
UAUTOMIZER	4 244	4 199	99 %	45	1 523	1 255	82 %	268
SYMBIOTIC	2 430	2 381	98 %	49	1 451	1 214	84 %	237
CBMC	1 813	1 702	94 %	111	1 975	1 248	63 %	727
UTAIPAN	3 015	2 936	97 %	79	915	653	71 %	262
2LS	2 072	2 045	99 %	27	1 419	945	67 %	474
ESBMC-KIND	3 679	3 556	97 %	123	2 141	1 753	82 %	388
UKOJAK	2 070	2 038	98 %	32	553	548	99 %	5
CBMC-PATH	1 206	1 162	96 %	44	897	670	75 %	727
DIVINE-EXPLICIT	693	673	97 %	20	768	353	46 %	415
DIVINE-SMT	645	626	97 %	19	943	601	64 %	342
DEPTHK	612	602	98 %	10	1 938	1 370	71 %	568

Green Verifiers—Low Energy Consumption. Since a large part of the cost of verification is given by the energy consumption, it might be important to also consider the energy efficiency. The second category in Table 9 uses the energy consumption per score point as rank measure: $\frac{\text{total CPU energy}}{\text{total score}}$, with the unit J/sp.

New Verifiers—High Quality. To acknowledge the achievements of verifiers that participate for the first time in SV-COMP, the third category in Table 9 uses the quality in score points as rank measure, that is, the official SV-COMP rank measure, but the subject systems reduced to verifiers that participate for the first time. The Java track consists exclusively of new verifiers, so the new-verifiers ranking is the same as the official ranking.

Verifiable Witnesses. For SV-COMP, it is not sufficient to answer with just TRUE or FALSE: each answer should be accompanied by a verification witness. All verifiers in categories that required witness validation support the common exchange format for violation and correctness witnesses. We used four independently developed witness-based result validators [14–16].

The majority of witnesses that the verifiers produced can be confirmed by the results-validation process. Interestingly, the confirmation rate for the TRUE results is significantly higher than for the FALSE results. Table 10 shows the confirmed versus unconfirmed results: the first column lists the verifiers of category *C-Overall*, the three columns for result TRUE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with TRUE, respectively, and the three columns for result FALSE reports the total, confirmed, and unconfirmed number of verification tasks for which the verifier answered with FALSE, respectively. More information (for all verifiers) is given in

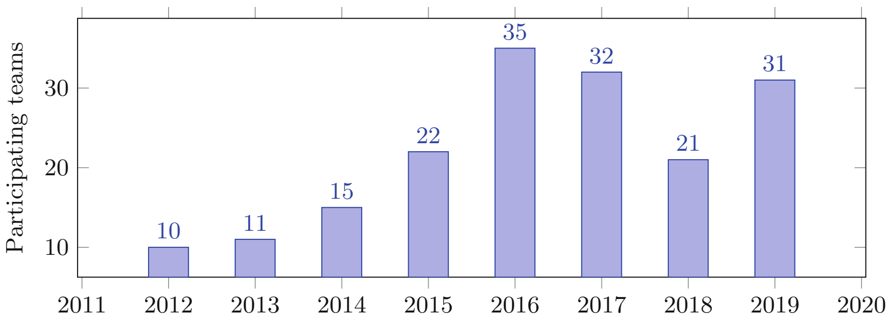

Fig. 7. Number of participating teams for each year

the detailed tables on the competition web site[15], cf. also the report on the demo category for correctness witnesses from SV-COMP 2016 [11]. Result validation is an important topic also in other competitions (e.g., in the SAT competition [4]).

8 Conclusion

SV-COMP 2019, the 8th edition of the Competition on Software Verification, attracted 31 participating teams from 14 countries (see Fig. 7 for the development). SV-COMP continues to offer the broadest overview of the state of the art in automatic software verification. For the first time, the competition included Java verification; this track had four participating verifiers. The competition does not only execute the verifiers and collect results, but also tries to validate the verification results, based on the latest versions of four independently developed results validators. The number of verification tasks was increased to 10 522 in C and to 368 in Java. As before, the large jury and the organizer made sure that the competition follows the high quality standards of the TACAS conference, in particular with respect to the important principles of fairness, community support, and transparency.

References

1. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 355–359. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_22
2. Andrianov, P., Mutilin, V., Khoroshilov, A.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Proc. TMPA, CCIS, vol. 779, pp. 11–23. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-71734-0_2

3. Artho, C., Visser, W.: Java Pathfinder at SV-COMP 2019 (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 224–228. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_18
4. Balyo, T., Heule, M.J.H., Järvisalo, M.: SAT Competition 2016: Recent developments. In: Proc. AI, pp. 5061–5063. AAAI Press (2017)
5. 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, Cham (2017)
6. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 3–24. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_1
7. Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS, LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_38
8. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS, LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_43
9. Beyer, D.: Status report on software verification (Competition summary SV-COMP 2014). In: Proc. TACAS, LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_25
10. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS, LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_31
11. 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, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_55
12. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS, LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
13. Beyer, D.: Verification witnesses from SV-COMP 2019 verification tools. Zenodo (2019). https://doi.org/10.5281/zenodo.2559175
14. 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
15. 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
16. 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, Cham (2018). https://doi.org/10.1007/978-3-319-92994-1_1
17. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transfer **9**(5–6), 505–525 (2007). https://doi.org/10.1007/s10009-007-0044-z
18. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
19. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN, LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12

20. 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
21. Cassez, F., Sloane, A.M., Roberts, M., Pigram, M., Suvanpong, P., de Aledo Marugán, P.G.: Skink: Static analysis of programs in LLVM intermediate representation (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 380–384. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_27
22. Chalupa, M., Strejcek, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN, LNCS, vol. 10869, pp. 115–132. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94111-0_7
23. Chalupa, M., Vitovská, M., Strejcek, J.: Symbiotic 5: Boosted instrumentation (competition contribution). In: Proc. TACAS, LNCS, vol. 10806, pp. 442–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_29
24. Chaudhary, E., Joshi, S.: Pinaka: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 234–238. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_20
25. Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 404–408. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_32
26. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program aware fuzzing (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 244–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_22
27. Cordeiro, L.C., Kesseli, P., Kröning, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Proc. CAV, LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10
28. Cordeiro, L., Kröning, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 219–223. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_17
29. Cordeiro, L.C., Kröning, D., Schrammel, P.: Benchmarking of Java verification tools at the software verification competition (SV-COMP). CoRR abs/1809.03739 (2018)
30. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS, LNCS, vol. 9035, pp. 423–425. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_34
31. Gadelha, M.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp.209–213. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_15
32. Gadelha, M.Y., 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. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE, LNCS, vol. 9195, pp. 105–108. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_6
34. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity automatically with aprove. J. Autom. Reason. **58**(1), 3–31 (2017)

35. Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace abstraction and abstract interpretation (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 399–403. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_31
36. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: A modular verifier (competition contribution). In: Proc. TACAS, LNCS, vol. 9035, pp. 451–454. Springer, Heidelberg (2015)
37. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Automizer with an on-demand construction of Floyd-Hoare automata (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 394–398. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_30
38. Hensel, J., Emrich, F., Frohn, F., Ströder, T., Giesl, J.: AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 350–354. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_21
39. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Hardware and Software: Verification and Testing, LNCS, vol. 10028, pp. 202–209. Springer, Cham (2016) https://doi.org/10.1007/978-3-319-49052-6
40. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: Proc. ISoLA, LNCS, vol. 7609, pp. 608–614. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_45
41. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012 - A program verification competition. STTT **17**(6), 647–657 (2015). https://doi.org/10.1007/s10009-015-0396-8
42. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: Proc. CAV, LNCS, vol. 9779, pp. 352–358. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_19
43. Kahsai, T., Rümmer, P., Schäf, M.: JayHorn: A Java model checker (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 214–218. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_16
44. Khazem, K., Tautschnig, M.: CBMC Path: A symbolic execution retrofit of the C bounded model checker (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 199–203. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_13
45. Kotoun, M., Peringer, P., Soková, V., Vojnar, T.: Optimized Predators and the SV-COMP heap and memory safety benchmark (competition contribution). In: Proc. TACAS, LNCS, vol. 9636, pp. 942–945. Springer, Heidelberg (2016)
46. Kröning, D., Tautschnig, M.: Cbmc: C bounded model checker (competition contribution). In: Proc. TACAS, LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
47. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC, LNCS, vol. 11187, pp. 313–332. Springer, Cham (2018)

48. Lauko, H., Štill, V., Ročkai, P., Barnat, J.: Extending DIVINE with symbolic verification using SMT (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 204–208. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_14
49. Malik, V., Hruska, M., Schrammel, P., Vojnar, T.: 2LS: Heap analysis and memory safety (competition contribution). Tech. Rep. abs/1903.00712, CoRR (2019)
50. Nguyen, T.L., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq 2.0: Combining lazy sequentialization with abstract interpretation (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 375–379. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_26
51. Noller, Y., Pasareanu, C., Le, B.D., Visser, W., Fromherz, A.: Symbolic Pathfinder for SV-COMP (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 239–243. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_21
52. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks (competition contribution). In: Proc. TACAS, LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_44
53. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013)
54. Rajkhowa, P., Lin, F.: VIAP 1.1: Automated system for verifying integer assignment programs with loops (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 250–255. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_23
55. Rakamarić, Z., Emmi, M.: SMACK: Decoupling source language details from verifier implementations. In: Proc. CAV, LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
56. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS, LNCS, vol. 11429, pp. 229–233. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_19
57. Rocha, H., Barreto, R.S., Cordeiro, L.C.: Memory management test-case generation of C programs using bounded model checking. In: Proc. SEFM, LNCS, vol. 9276, pp. 251–267. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_18
58. Rocha, H., Ismail, H., Cordeiro, L.C., Barreto, R.S.: Model checking embedded C software using k-induction and invariants. In: Proc. SBESC, pp. 90–95. IEEE (2015). https://doi.org/10.1109/SBESC.2015.24
59. Rocha, H.O., Barreto, R.S., Cordeiro, L.C.: Hunting memory bugs in C programs with Map2Check (competition contribution). In: Proc. TACAS, LNCS, vol. 9636, pp. 934–937. Springer, Heidelberg (2016)
60. Rocha, W., Rocha, H., Ismail, H., Cordeiro, L.C., Fischer, B.: DepthK: A k-induction verifier based on invariant inference for C programs (competition contribution). In: Proc. TACAS, LNCS, vol. 10206, pp. 360–364. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_23
61. Schrammel, P., Kröning, D.: 2LS for program analysis (competition contribution). In: Proc. TACAS, LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_56
62. Štill, V., Ročkai, P., Barnat, J.: DIVINE: Explicit-state LTL model checker (competition contribution). In: Proc. TACAS, LNCS, vol. 9636, pp. 920–922. Springer, Heidelberg (2016)

63. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Autom. Softw. Eng. **10**(2), 203–232 (2003)
64. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proc. Inst. Syst. Program. (ISPRAS) **29**, 203–216 (2017). https://doi.org/10.15514/ISPRAS-2017-29(4)-13
65. Yin, L., Dong, W., Liu, W., Li, Y., Wang, J.: YOGAR-CBMC: CBMC with scheduling constraint based abstraction refinement (competition contribution). In: Proc. TACAS, LNCS, vol. 10806, pp. 422–426. Springer, Cham (2018)
66. Yin, L., Dong, W., Liu, W., Wang, J.: On scheduling constraint abstraction for multi-threaded program verification. IEEE Trans. Softw. Eng. https://doi.org/10.1109/TSE.2018.2864122
67. Yin, L., Dong, W., Liu, W., Wang, J.: Parallel refinement for multi-threaded program verification. In: Proc. ICSE. IEEE (2019)