Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016)

Dirk Beyer(✉)

University of Passau, Passau, Germany
dirk.beyer@sosy-lab.org

Abstract. The 5th Competition on Software Verification (SV-COMP 2016) continues the tradition of a thorough comparative evaluation of fully-automatic software verifiers. This report presents the results of the competition and includes a special section that describes how SV-COMP ensures that the experiments are reliably executed, precisely measured, and organized such that the results can be reproduced later. SV-COMP uses BenchExec for controlling and measuring the verification runs, and requires violation witnesses in an exchangeable format, whenever a verifier reports that a property is violated. Each witness was validated by two independent and publicly-available witness validators. The tables report the state of the art in software verification in terms of effectiveness and efficiency. The competition used 6 661 verification tasks that each consisted of a C program and a property (reachability, memory safety, termination). SV-COMP 2016 had 35 participating verification systems (22 in 2015) from 16 countries.

1 Introduction

The annual Competition on Software Verification (SV-COMP)[1] is a continuous effort by the software-verification community. The effort consists of the following two parts: (1) The SV-COMP community defines and collects verification tasks that the researchers and developers of software verifiers find interesting and challenging; these verification problems should be used to evaluate the effectivity (soundness and completeness) and efficiency (performance) of modern verification tools. (2) The organizer of SV-COMP performs a systematic comparative evaluation of the relevant state-of-the-art tool implementations for automatic software verification with respect to effectiveness and efficiency; part of this is to define and explore standards for a reliable and reproducible execution of such a competition. This paper describes the rules, definitions, results, and other interesting facts about the execution of the competition experiments, in particular how to make the experiments reproducible. The main objectives that the community aims at by running yearly competitions are the following (taken from [5]):

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

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 887–904, 2016.
DOI: 10.1007/978-3-662-49674-9_55

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.

There is consensus that (1) and (2) are already achieved, but need continuous improvement: the community of research groups and verifiers that participate in SV-COMP is increasing, and the set of verification tasks needs even more diversity, growing, and quality assurance. The repository and the issue tracker show that there was considerable effort spent on consolidating the verification tasks, in terms of consistency and quality. Regarding (3), the simple syntax of the property language works well for SV-COMP, while it would be great to increase the supported fragment of LTL. The standard witness language as a common, exchangeable format was a big step forward in terms of standardization. The requirement in SV-COMP that bug reports are counted only if the bug is reproducible, i.e., the witness can be re-played on a different machine with a different validation tool, makes it easier to understand problems. We received positive feedback in terms of Objective (4), but we cannot evaluate this here.

Related Competitions. SV-COMP is complemented by two other competitions in the field of software verification: RERS[2] and VerifyThis[3]. While SV-COMP performs reproducible experiments in a *controlled* environment (dedicated resources, resource limits), the RERS Challenges gives 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 report on SV-COMP 2014 provides a more comprehensive list of other competitions [4].

2 Procedure

The procedure for the competition organization did not change in comparison to the past SV-COMP editions [2–5]. SV-COMP was again an open competition where all verification tasks were known before the submission of the participating verifiers, such that there were no surprises and developers were able to train the verifiers. In the *benchmark submission* phase, we collected and classified new verification tasks, in the *training* phase, the teams inspected verification tasks and trained their verifiers, and in the *evaluation* phase, verification runs were preformed with all competition candidates and the system descriptions were reviewed by the competition jury. As in the last years, the participants received the preliminary results of their verifier per e-mail for inspection, after which the results were publicly announced.

[2] http://rers-challenge.org
[3] http://etaps2015.verifythis.org

3 Definitions, Formats, and Rules

Verification Task. The definition of verification task was not changed (taken from [4]). A verification task consists of a C program and a property. A verification run is a non-interactive execution of a competition candidate on a single verification task, in order to check whether the following statement is correct: "The program satisfies the property." The result of a verification run is a triple (ANSWER, WITNESS, TIME). ANSWER is one of the following outcomes:

TRUE: The property is satisfied (i.e., no path that violates the property exists).

FALSE: The property is violated (i.e., there exists a path that violates the property) and a counterexample path is produced and reported as WITNESS.

UNKNOWN: The tool cannot decide the problem, or terminates abnormally, or exhausts the computing resources time or memory (i.e., the competition candidate does not succeed in computing an answer TRUE or FALSE).

The component WITNESS [6] was this year mandatory only for FALSE answers; in the future, witnesses are also required for TRUE answers. SV-COMP was supported by the two witness validators CPACHECKER and UAUTOMIZER. TIME is measured as consumed CPU time until the verifier terminates, including the consumed CPU time of all processes that the verifier started [8]. If the wall time was larger than the CPU time, then the TIME is set to the wall time. If TIME is equal to or larger than the time limit (15 min), then the verifier is terminated and the ANSWER is set to 'timeout' (and interpreted as UNKNOWN).

Categories. The collection of verification tasks, which represents the current interest and abilities of tools for software verification, is arranged into categories, according to the characteristics of the programs and the properties to be verified. The assignment was proposed and implemented by the competition chair, and approved by the competition jury.

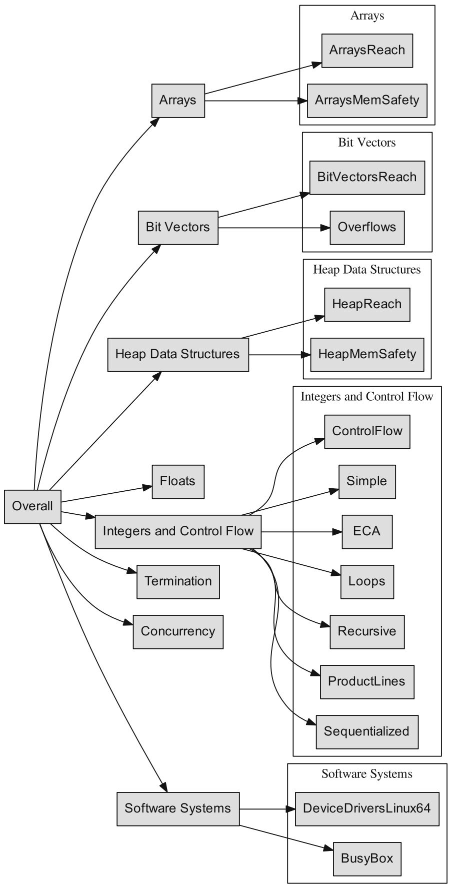

Fig. 1. Categories (generated by GRAPHVIZ)

Table 1. Properties used in the competition (cf. [5] for more details)

Formula	Interpretation / Syntax of property
`G ! call(foo())`	A call to function `foo` is not reachable on any finite execution. `CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )`
`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. `CHECK( init(main()), LTL(G valid-free) )`
`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. `CHECK( init(main()), LTL(G valid-deref) )`
`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. `CHECK( init(main()), LTL(G valid-memtrack) )`
`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. `CHECK( init(main()), LTL(F end) )`

Table 2. Scoring schema for SV-COMP 2016

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result
FALSE correct	+1	Violation of property in program was correctly found
FALSE incorrect	−16	Violation reported but property holds (false alarm)
TRUE correct	+2	Correct program reported to satisfy property
TRUE incorrect	−32	Incorrect program reported as correct (wrong proof)

For the 2016 edition of SV-COMP, a total of 10 categories were defined. The structure of categories is illustrated in Fig. 1 and described in more detail on the competition web site[4]. As a new feature of the competition, a new (meta) category *Falsification* was defined, which was meant to explore bug hunting capabilities of verifiers that are not able to construct correctness proofs. The new category consisted of all verification tasks with safety properties, and any answers TRUE were ignored. The categories, their defining category-set files, and the contained programs are explained in more detail under *Verification Tasks* on the competition web site.

Properties and Their Format. For the definition of the properties and the property format we refer to the previous competition report [5]. All specifications

[4] http://sv-comp.sosy-lab.org/2016/benchmarks.php

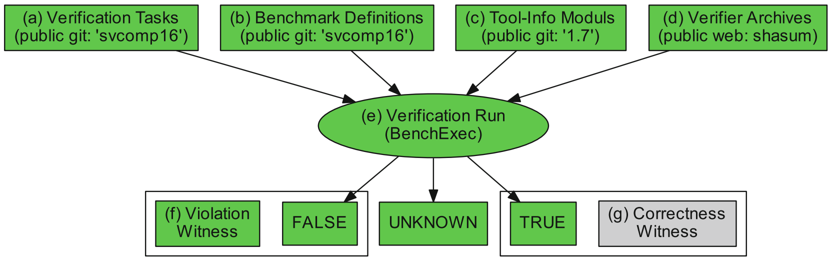
(a) Verification Tasks (public git: 'svcomp16')
(b) Benchmark Definitions (public git: 'svcomp16')
(c) Tool-Info Moduls (public git: '1.7')
(d) Verifier Archives (public web: shasum)
(e) Verification Run (BenchExec)
(f) Violation Witness
FALSE
UNKNOWN
TRUE
(g) Correctness Witness

Fig. 2. Setup: components that support reproducibility are highlighted in green

are available as `.prp` files in the respective directories of the benchmark categories in the repository. Table 1 lists the properties and their syntax as overview.

Evaluation by Scores and Run Time. In order to reflect the steady progress towards completeness and soundness of verification tools, the scoring schema was again adjusted in order to increase the penalty for wrong results. Table 2 provides the overview. The ranking is 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 [3] (page 597). The *Competition Jury* consists again of the chair and one member of each participating team. Team representatives of the jury are listed in Table 3.

4 Reproducibility

One of the main goals of SV-COMP is to make the competition as transparent and reproducible as possible. To achieve this goal, it is necessary to control as many as possible of the variables that might influence the results. Figure 2 gives an overview over the components that contribute to the reproducible setup of SV-COMP.

BenchExec: Precise Controlling and Measurement of Resources (e). For scientifically valid experiments, we require for each verification run a reliable assignment and controlling of computing resources (cores, memory, CPU time), and a precise measurement. There are several requirements that experiments of a competition such as SV-COMP have to fulfill [8]: (i) accurate measurement and reliable enforcement of limits for CPU time and memory, (ii) reliable termination of processes (including all child processes), and (iii) correct assignment of local memory (for NUMA architectures). We use BenchExec[5] to perform all SV-COMP experiments, because this benchmarking framework lets us conveniently benefit from the modern resource control and measurement mechanisms that the Linux kernel offers.

[5] https://github.com/sosy-lab/benchexec

Repository of Verification Tasks (a). The verification tasks are organized in a public repository[6]. The repository was moved to GitHub in order to support an issue tracker and to efficiently handle contributions from the community via pull requests. The more appropriate logging of change history and issues gives credit to people that contribute. Furthermore, the continuous-integration system TravisCI is used to ensure that the verification tasks are compilable by Gcc and Clang. The move to GitHub also had a positive effect on the activity on the benchmark suite: more people are involved, and more fixes to verification tasks were contributed. For reproducing the results of SV-COMP, the exact versions of the verification tasks as used for SV-COMP 2016 are available via the PGP-signed tag 'svcomp16' in the git repository.

Benchmark Definitions (b). For executing verification runs, we need to know for each verifier, (i) which verification tasks need to be given to the verifier (derived from participation declaration) and (ii) which parameters need to be passed to the verifier (there are global parameters that are specified for all categories, and there are specific parameters such as the bit architecture and memory model). The benchmark definitions are XML files in the format that BenchExec expects; they are collected in a specific repository for SV-COMP[7], in which the PGP-signed tag 'svcomp16' points to the exact versions that were used in SV-COMP 2016.

Tool-Specific Information (c). In order to successfully execute a verifier and correctly interpret its results, a tool-info module needs to be provided to BenchExec. First, the command-line to properly invoke the verifier (including source and property file as well as the options) is assembled from the parts specified in the benchmark definition (b). Second, the (tool-specific) information that the verifier produces needs to be interpreted and translated into the uniform SV-COMP result (True, False(p), Unknown). The tool-info modules that were used in SV-COMP 2016 are available in BenchExec release 1.7.

Verifier Archive (d). The verifiers are provided in an archive containing a license (that permits academic use, use in SV-COMP, and reproducing the results) and all parts that are needed to execute the verifier (statically-linked executables, all components that are required in a certain version, or for which no standard Ubuntu package is available, are included). The verifiers and the above-mentioned components are provided on the systems-description page of the SV-COMP web site[8], together with the SHA1 hashes for verification of consistency.

Violation Witnesses (f). SV-COMP counts answers False only if a valid witness according to an exchangeable, machine-readable format is part of the result triple as witness. This means that each verification run must be followed by a validation run that checks if the witness adheres to the exchange format

[6] https://github.com/sosy-lab/sv-benchmarks
[7] https://github.com/sosy-lab/sv-comp
[8] http://sv-comp.sosy-lab.org/2016/systems.php

and can be reproduced. The time limit for a validation run was set to 10 % of the CPU time for a verification run, i.e., the witness validation was limited to 90 s. The purpose of the tighter resource limit is to avoid delegating verification work to the validator. This ensures a high quality of assignment of scores: if a verifier claims a found bug but is not able to provide a witness, then no score is assigned. The witness format and the validation process is explained on the web page[9]. More details on witness validation is given in a related research article [6].

Correctness Witnesses (g). Although SV-COMP requires since its second edition (2013) that each result must be accompanied by a witness, this requirement was not enforced for the answer TRUE, mainly due to the lack of validators for correctness witnesses. This year, there was a demonstration category on validation of correctness witnesses, with the purpose to get prepared for witness validation for correctness results in the future.

5 Results and Discussion

For the fifth time, the competition experiments represent the state of the art in fully-automatic and publicly-available software-verification tools. The report shows the improvements of the 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 approved by the participating teams.

Participating Verifiers. Table 3 provides an overview of the participating competition candidates and Table 4 lists the features and technologies that are used in the verification tools.

Computing Resources. The resource limits were the same as last year [5]: 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. The machines for running the experiments were different from last year, because we had to use 24 machines instead of eight. Each machine had two Intel Xeon E5-2650 v2 CPUs, with 16 processing units each, a frequency of 3.4 GHz, 135 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 14.04 with Linux kernel 4.2). All verification runs were executed on a dedicated CPU, i.e., 8 processing units were assigned to the verification run, while the other 8 processing units were reserved and left idle.

One complete verification execution of the competition consisted of 313 benchmarks (each verifier on each selected category according to the opt-outs), summing up to 115 761 verification runs. Witness validation required 524 benchmarks (combinations of verifier, category with witness validation, and two validators) summing up to 50 249 validation runs. The consumed total CPU time for one competition run for verification only required a total of 319 days of CPU

[9] http://sv-comp.sosy-lab.org/2016/witnesses/

Table 3. Competition candidates with their system-description references and representing jury members

Participant	Ref.	Jury member	Affiliation
2LS	[31]	Peter Schrammel	U Oxford, UK
APROVE	[33]	Jera Hensel	RWTH Aachen, Germany
BLAST	[32]	Vadim Mutilin	ISPRAS, Russia
CASCADE	[35]	Wei Wang	New York U, USA
CBMC	[22]	Michael Tautschnig	Queen Mary U London, UK
CEAGLE		Dexi Wang	Tsinghua U, China
CEAGLE-ABSREF		Guang Chen	Tsinghua U, China
CIVL	[36]	Stephen Siegel	U Delaware, USA
CPA-BAM	[14]	Karlheinz Friedberger	U Passau, Germany
CPA-KIND	[7]	Matthias Dangl	U Passau, Germany
CPA-REFSEL	[9]	Stefan Löwe	U Passau, Germany
CPA-SEQ	[12]	—	U Passau, Germany
DIVINE	[37]	Vladimír Štill	Masaryk U, Czech Republic
ESBMC	[24]	Mikhail Ramalho	U Southampton, UK
ESBMC+DEPTHK	[28]	Lucas Cordeiro	Federal U Amazonas, Brazil
FOREST	[13]	Pablo Sanchez	U Cantabria, Spain
FORESTER	[18]	Ondřej Lengál	Brno UT, Czech Republic
HIPREC	[23]	Quang Loc Le	National U, Singapore
IMPARA		Björn Wachter	U Oxford, UK
LAZY-CSEQ	[19]	Omar Inverso	Gran Sasso Sc. Inst., Italy
LCTD	[30]	Keijo Heljanko	Aalto U, Finland
LPI	[20]	George Karpenkov	VERIMAG, France
MAP2CHECK	[29]	Herbert Rocha	Federal U Roraima, Brazil
MU-CSEQ	[34]	Gennaro Parlato	U Southampton, UK
PAC-MAN	[11]	Ming-Hsien Tsai	Academia Sinica, Taiwan
PREDATORHP	[21]	Tomas Vojnar	Brno UT, Czech Republic
SEAHORN	[15]	Jorge Navas	NASA Ames, USA
SKINK		Franck Cassez	Macquarie U, Australia
SMACK+CORRAL	[27]	Zvonimir Rakamaric	U Utah, USA
SYMBIOTIC	[10]	Jan Strejček	Masaryk U, Czech Republic
SYMDIVINE	[1]	Jiří Barnat	Masaryk U, Czech Republic
UAUTOMIZER	[17]	Matthias Heizmann	U Freiburg, Germany
UKOJAK	[26]	Daniel Dietsch	U Freiburg, Germany
UL-CSEQ	[25]	Bernd Fischer	Stellenbosch U, ZA
VVT	[16]	Alfons Laarman	TU Vienna, Austria

Table 4. Technologies and features that the verification tools offer

Verifier	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
2LS				✓	✓						✓						
AProVE			✓				✓	✓			✓						✓
Blast	✓	✓					✓					✓	✓	✓			
Cascade			✓	✓							✓						
CBMC				✓	✓						✓					✓	
Ceagle				✓							✓						
Ceagle-Absref	✓	✓		✓							✓		✓	✓			
CIVL			✓	✓				✓								✓	
CPA-BAM	✓	✓					✓				✓	✓	✓	✓			
CPA-kInd	✓	✓		✓	✓		✓	✓			✓	✓	✓	✓			
CPA-RefSel	✓	✓					✓				✓	✓	✓	✓			
CPA-Seq	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	
DIVINE							✓				✓				✓	✓	
ESBMC				✓							✓					✓	
ESBMC+DepthK				✓	✓						✓					✓	
Forest			✓	✓							✓						
Forester	✓								✓						✓		
HIPrec									✓	✓							
Impara			✓				✓	✓			✓	✓	✓	✓		✓	
Lazy-CSeq				✓							✓					✓	
LCTD	✓	✓															
LPI	✓				✓			✓				✓	✓				
Map2Check				✓							✓						
MU-CSeq				✓							✓					✓	
PAC-MAN			✓												✓		
PredatorHP									✓								
SeaHorn				✓		✓		✓				✓	✓	✓			✓
Skink	✓		✓				✓							✓	✓		
SMACK+Corral	✓			✓		✓					✓		✓			✓	
Symbiotic			✓														
SymDIVINE			✓				✓				✓				✓	✓	
UAutomizer	✓	✓									✓		✓	✓	✓		✓
UKojak	✓	✓									✓		✓	✓			
UL-CSeq	✓	✓										✓	✓	✓		✓	
VVT	✓	✓		✓		✓	✓							✓		✓	

Table 5. Quantitative overview over all results

Verifier	**Arrays** 316 points max. 183 tasks	**BitVectors** 92 points max. 60 tasks	**Heap** 382 points max. 239 tasks	**Floats** 140 points max. 81 tasks	**IntegersControlFlow** 3629 points max. 2331 tasks	**Termination** 1129 points max. 631 tasks	**Concurrency** 1240 points max. 1016 tasks	**DeviceDriversLinux64** 3977 points max. 2120 tasks	**FalsificationOverall** 2371 points max. 6030 tasks	**Overall** 10855 points max. 6661 tasks
2LS				**136**	1196				-2438	-38205
AProVE						**909**				
Blast					-1653			**2704**		
Cascade			**197**							
CBMC	62	46	8	**134**	-1239		882	1972	391	3386
Ceagle				**136**						
Ceagle-Absref				124						
CIVL							**1240**			
CPA-BAM	-57	28	-80	42	1822	0	0	2550	-1218	1939
CPA-kInd	3	**77**	161	76	**2095**	0	0	2350	**707**	4094
CPA-RefSel				35	1539	0	0	**3177**	36	2157
CPA-Seq	-61	**87**	**234**	75	**2652**	0	282	**2801**	496	**4794**
DIVINE							951			
ESBMC	**190**	**84**	163	-15	1217	0	742	1688	248	4145
esbmc+depthk	62	47	58	7	1111	0	877	2009	495	3110
Forest	-970		-1263				-20613			
Forester			86							
HIPrec										
Impara		-592		132	-1524		42			
Lazy-CSeq							**1240**			
LCTD										
LPI					1804			2107		
Map2Check			-121							
MU-CSeq							**1240**			
PAC-MAN				-449						
PredatorHP			**298**							
SeaHorn	-301	-131	-257	0	1572	**504**	-24659	1694	-4333	-22393
Skink					113					
smack+corral	**146**	44	155	0	**2013**	0	999	2206	**800**	**4223**
Symbiotic	**101**	-2	105	-18	633	0	0	980	-370	1223
SymDIVINE							-135			
UAutomizer	83	69	169	2	1865	**895**		2686	**823**	**4843**
UKojak	60	19	31	0	1096			937	339	1407
UL-CSeq							856			
VVT					421		1029			

Table 6. 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
Arrays						
1	ESBMC	**190**	3.2	131	2	
2	SMACK+CORRAL	146	2.5	111		
3	SYMBIOTIC	101	.61	77		
Bit Vectors						
1	CPA-SEQ	**87**	1.1	55		
2	ESBMC	84	.61	51		
3	CPA-KIND	77	.67	47		
Heap						
1	PREDATORHP	**298**	.31	211	2	
2	CPA-SEQ	234	1.1	188	4	
3	CASCADE	197	2.7	140	2	
Floats						
1	2LS	**136**	.98	79		
2	CEAGLE	136	1.0	77		
3	CBMC	134	5.0	78		
Integers Control Flow						
1	CPA-SEQ	**2652**	35	1625	1	
2	CPA-KIND	2095	35	1278		
3	SMACK+CORRAL	2013	97	978		**4**
Termination						
1	APROVE	**909**	4.8	500		
2	UAUTOMIZER	895	3.2	503		
3	SEAHORN	504	.97	323		**2**
Concurrency						
1	MU-CSEQ	**1240**	.93	1016		
2	LAZY-CSEQ	1240	2.7	1016		
3	CIVL	1240	7.8	1016		
Device Drivers Linux64						
1	CPA-REFSEL	**3177**	24	1646	2	
2	CPA-SEQ	2801	23	1458	4	
3	BLAST	2704	5.9	1547	13	**5**
Falsification Overall						
1	UAUTOMIZER	**823**	7.0	381	1	
2	SMACK+CORRAL	800	17	1140	26	
3	CPA-KIND	707	14	479	2	
Overall						
1	UAUTOMIZER	**4843**	44	3138	1	**5**
2	CPA-SEQ	4794	65	3535	16	
3	SMACK+CORRAL	4223	160	3464	26	**9**

time. Each tool was executed several times, in order to make sure no installation issues occur during the execution.

Quantitative Results. Table 5 presents the quantitative overview over all tools and all categories (HIPrec participated only in subcategory *Recursive* and LCTD only in subcategory *BitVectorsReach*). The format of the table is similar to those of previous SV-COMP editions [5], with the exception that due to the volume we now omit the CPU times. The tools are listed in alphabetical order; every table row lists the scores of one verifier for each category. 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 category. For the calculation of the score and for the ranking, the scoring schema in Table 2 was applied, the scores for the meta categories were computed using normalized scores as defined in the report for SV-COMP'13 [3]. There were two categories for which the winner was decided based on the run time: in category *Concurrency*, all top-three verifiers achieved the maximum score of 1240 points, but the run time differed considerably; in category *Floats* the first and second both achieved a score of 136 points. More information (including formatted interactive tables, quantile plots for every category, and also the raw data in XML format) is available on the competition web-site.[10]

Table 6 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 tool reported wrong results: reporting an error path but 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 and Normalization. The SV-COMP community considers it more difficult to compute correctness proofs compared to computing error paths (cf. Table 2: True yields 2 points, False yields 1 point) [2]. This has consequences on the final ranking: For example, AProVE won the category *Termination* although UAutomizer solved more verification tasks: AProVE solved 500, UAutomizer solved 503 verification tasks. Both verifiers did not report any wrong results in this category. So the higher score of AProVE (score: 909) is due to its ability to compute more proofs than UAutomizer (score: 895), while UAutomizer found more violations. AProVE computed 409 proofs and found 91 property violations, while UAutomizer computed 392 proofs and found 111 property violations. So in this case, the scoring schema provides a good mapping from the community's intuition to the ranking.

A similar observation can be made on the score normalization. The community considers the value of solving a verification task in a large category (many verification tasks) less than the value of solving a verification task in a small category (only a few verification tasks) [3]. The values for

[10] http://sv-comp.sosy-lab.org/2016/results/

category *Overall* in Table 6 illustrate the purpose of the score normalization: CPA-SEQ solved 3 535 tasks, which is about 400 solved tasks more than the winner UAUTOMIZER could solve (3 138). So why did CPA-SEQ not win the category? Because UAUTOMIZER is better in the intuitive sense of 'overall': UAUTOMIZER solved tasks more diversely, the 'overall' value of the verification work is higher. Most prominently, UAUTOMIZER solved many tasks in category *Termination* which is not supported by CPA-SEQ. Similarly, in category *FalsificationOverall*, SMACK+CORRAL solved more tasks than UAUTOMIZER, but produced also a lot of false alarms and the tasks that SMACK+CORRAL solved were considered of less value (i.e., from large categories with many tasks). In these cases, the score normalization correctly maps the community's intuition.

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [3] because these visualizations make it easier to understand the results of the comparative evaluation. The competition web-site[10] includes such a plot for each category; as example, we illustrate the category *Overall* (all verification tasks) in Fig. 3 and discuss the results below. A total of 13 verifiers participated in category *Overall* (only 6 the year before), for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [3]).

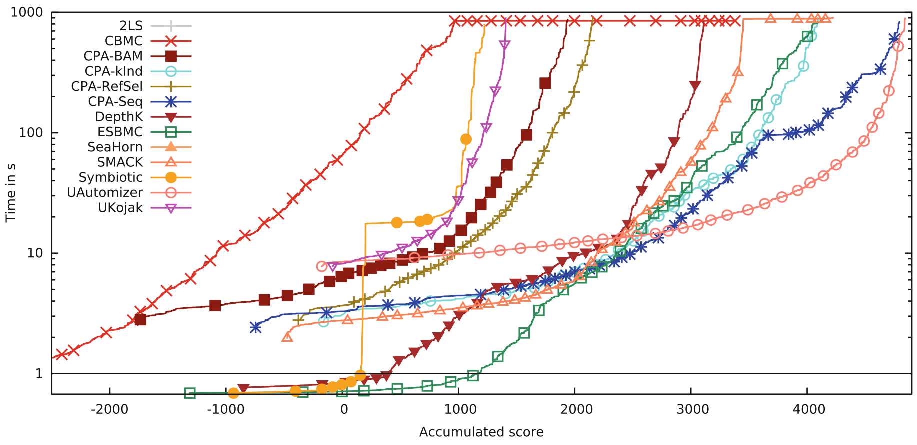

Fig. 3. Quantile functions for category *Overall*. Each quantile function illustrates the quantile (x-coordinate) of the scores obtained by correct verification runs below a certain run time (y-coordinate). More details are given in a previous report [3]. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

Overall Quality Measured in Scores (Right End of Graph). UAUTOMIZER is the winner of this category: the x-coordinate of the right-most data point represents the highest total score (and thus, the total value) of the completed verification work (cf. Table 6; right-most x-coordinates match the score values in the table).

Amount of Incorrect Verification Work (Left End of Graph). The left-most data points of the quantile functions represent the total negative score of a verifier (x-coordinate), i.e., the amount of incorrect and misleading verification work. Verifiers should start with a score close to zero; the winner UAutomizer is very good in this aspect, together with the second place CPA-kInd (the two right-most columns of category *Overall* in Table 6 report the concrete numbers: only 1 and 16 false alarms, respectively, and 5 and 0 wrong proofs, for a total of 6 661 verification tasks).

Characteristics of the Verification Tools. Quantile plots also give hints on how a verification strategy works. For example, the horizontal lines show that some verifiers 'solve' a large quantity of verification tasks in the same run time, suggesting that an answer is given without the result being actually computed. A quick look at the wrapping execution scripts reveals that indeed a pre-mature answer is returned after 850s or 880s, respectively. This insight is one of the arguments for the community's goal to have each result supported by evidence, e.g., in the form of a verification witness.

Robustness, Soundness, and Completeness. Table 6 shows in the last two columns that the best verifiers of each category report a low number of wrong verification results (compared to the large number of verification tasks), indicating the advancement of the state-of-the-art verification technology. In the three categories *BitVectors*, *Floats*, and *Concurrency*, the top-three verifiers did not report any wrong results.

Verifiable Witnesses. SV-COMP counts answers False (bug reports) only if the result contains a violation witness, which represents directions through the state space to easily recover an error path. All verifiers in categories that required witness validation supported the common exchange format for error witnesses, and produced error paths in that format. For SV-COMP 2016, we used two completely different witness validators: CPAchecker and UAutomizer.

Table 7. Validation of Correctness Witnesses

	Verification	Validation CPAchecker	Validation UAutomizer
Total tasks	3171	1574	1574
Results True	1574	1295	956
Confirmed witnesses		82 %	61 %

Demonstration on Correctness Witnesses. The validation of the results for answers True was not yet considered, but is identified as the next open problem that the community should solve. As part of SV-COMP 2016, a demonstration category (i.e., without ranking and scores) was announced to explore the possibilities of validating correctness witnesses. Two teams participated, and the results are reported in Table 7. The table lists the results of a verification with CPAchecker (k-induction-based configuration) and the validation results of the correctness witnesses using the validators CPAchecker and UAutomizer. The first row reports the total number of verification tasks that were given as input. The verification was performed on an SV-COMP subset of 3 171 verification tasks from the categories *IntegersControlFlow* and *DeviceDriversLinux64*. The second row reports that for 1 574 verification tasks the expected and computed verification result was True. Those 1 574 verification tasks were given as input to the two validators, together with the correctness witness that the verification produced. CPAchecker was able to validate (i.e., re-verify with the given invariants from the witness) 1 295 verification tasks (82 %) and UAutomizer was able to validate 956 verification tasks (61 %). More information is given on the detailed table on the web page.[11]

6 Conclusion

SV-COMP 2016, the 5th edition of the Competition on Software Verification, attracted *35 participating teams* from 16 countries, which is so far the largest number of participants (2012: 10, 2013: 11, 2014: 15, 2015: 22). The repository of verification tasks was consolidated and the number of verification tasks was increased (from 5 803) to *6 661 verification tasks*. We used *verifiable witnesses* again to validate the bug reports, and the results False were counted towards the score only if the witness was confirmed. The number of witness validators was increased from one to two, which contributed to the trust and neutrality of SV-COMP's evaluation. SV-COMP 2016 is the so-far broadest overview of the state of the art in software verification. 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. Technical accuracy was ensured by using the benchmarking framework BenchExec.

References

1. Bauch, P., Havel, V., Barnat, J.: LTL model checking of LLVM bitcode with symbolic data. In: Hliněný, P., Dvořák, Z., Jaroš, J., Kofroň, J., Kořenek, J., Matula, P., Pala, K. (eds.) MEMICS 2014. LNCS, vol. 8934, pp. 47–59. Springer, Heidelberg (2014)
2. Beyer, D.: Competition on software verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-28756-5_38
3. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-36742-7_43

[11] http://sv-comp.sosy-lab.org/2016/witnesses/correctness-demo.html

4. Beyer, D.: Status report on software verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 373–388. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/978-3-642-54862-8_25
5. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-662-46681-0_31
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of FSE, pp. 721–733. ACM (2015). http://dx.org/10.1145/2786805.2786867
7. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-21690-4_42
8. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 160–178. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-23404-5_12
9. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 20–38. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-23404-5_3
10. Chalupa, M., Jonáš, M., Slaby, J., Strejček, J., Vitovská, M.: Symbiotic 3: New slicer and error-witness generation (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 946–949. Springer, Heidelberg (2016)
11. Chen, Y.-F., Hsieh, C., Lengál, O., Lii, T.-J., Tsai, M.-H., Wang, B.-Y., Wang, F.: Learning-based verification and model synthesis. In: Proceedings of ICSE (2016)
12. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 423–425. Springer, Heidelberg (2015)
13. Gonzalez-de-Aledo, P., Sanchez, P.: FramewORk for embedded system verification (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 429–431. Springer, Heidelberg (2015)
14. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 912–915. Springer, Heidelberg (2016)
15. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: a framework for verifying C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 447–450. Springer, Heidelberg (2015)
16. Günther, H., Laarman, A., Weissenbacher, G.: Vienna verification tool: IC3 for parallel software (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 954–957. Springer, Heidelberg (2016)
17. Heizmann, M., Dietsch, D., Greitschus, M., Leike, J., Musa, B., Schätzle, C., Podelski, A.: Ultimate automizer with two-track proofs (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 950–953. Springer, Heidelberg (2016)
18. Hruška, M., Holí, L., Lengál, O., Rogalewicz, A., Šimácek, J., Vojnar, T.: Run forester, run backwards! (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 923–926. Springer, Heidelberg (2016)
19. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 585–602. Springer, Heidelberg (2014)

20. Karpenkov, E.G., Monniaux, D., Wendler, P.: Program analysis with local policy iteration. In: Jobstmann, B., et al. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 127–146. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49122-5_6
21. Kotoun, M., Peringer, P., Šoková, V., Vojnar, T.: Optimized Predators and the SV-COMP heap and memory safety benchmark (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 942–945. Springer, Heidelberg (2016)
22. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
23. Le, Q.L., Tran, M., Chin, W.-N.: HIPrec: Verifying recursive programs with a satisfiability solver. Technical report (2016)
24. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22 (competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014)
25. Nguyen, T.L., Fischer, B., La Torre, S., Parlato, G.: Unbounded lazy-CSeq: a lazy sequentialization tool for C programs with unbounded context switches (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 461–463. Springer, Heidelberg (2015)
26. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015)
27. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Heidelberg (2014)
28. Rocha, H., Ismail, H.I., Cordeiro, L.C., Barreto, R.S.: Model checking embedded C software using k-induction and invariants. In: Proceedings of SBESC. IEEE (2015)
29. Rocha, H.O., Barreto, R., Cordeiro, L.: Hunting memory bugs in c programs with Map2Check (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 934–937. Springer, Heidelberg (2016)
30. Saarikivi, O., Heljanko, K.: LCTD: Tests-guided proofs for C programs on LLVM (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 927–929. Springer, Heidelberg (2016)
31. Schrammel, P., Kröning, D.: 2LS for program analysis (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016)
32. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7 (competition contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
33. Ströder, T., Aschermann, C., Frohn, F., Hensel, J., Giesl, J.: AProVE: termination and memory safety of C programs (competition contribution). In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 417–419. Springer, Heidelberg (2015)
34. Tomasco, E., Lam, T.N., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: MU-CSeq 0.4: Individual memory location unwindings (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 938–941. Springer, Heidelberg (2016)
35. Wang, W., Barrett, C.: Cascade. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 420–422. Springer (competition contribution), Heidelberg (2015)

36. Zheng, M., Edenhofner, J.G., Luo, Z., Gerrard, M.J., Dwyer, M.B., Siegel, S.F.: CIVL: applying a general concurrency verification framework to C/Pthreads programs (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 908–911. Springer, Heidelberg (2016)
37. Štill, V., Ročkai, P., Barnat, J.: DIVINE: Explicit-state LTL model checker (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 920–922. Springer, Heidelberg (2016)