Status Report on Software Verification
(Competition Summary SV-COMP 2014)

Dirk Beyer

University of Passau, Germany

Abstract. This report describes the 3rd International Competition on Software Verification (SV-COMP 2014), which is the third edition of a thorough comparative evaluation of fully automatic software verifiers. The reported results represent the state of the art in automatic software verification, in terms of effectiveness and efficiency. The verification tasks of the competition consist of nine categories containing a total of 2 868 C programs, covering bit-vector operations, concurrent execution, control-flow and integer data-flow, device-drivers, heap data structures, memory manipulation via pointers, recursive functions, and sequentialized concurrency. The specifications include reachability of program labels and memory safety. The competition is organized as a satellite event at TACAS 2014 in Grenoble, France.

1 Introduction

Software verification is an important part of software engineering, which is responsible for guaranteeing safe and reliable performance of the software systems that our economy and society relies on. The latest research results need to be implemented in verification tools, in order to transfer the theoretical knowledge to engineering practice. The Competition on Software Verification (SV-COMP) [1] is a systematic comparative evaluation of the effectiveness and efficiency of the state of the art in software verification. The benchmark repository of SV-COMP [2] is a collection of verification tasks that represent the current interest and abilities of tools for software verification. For the purpose of this competition, the verification tasks are arranged in nine categories, according to the characteristics of the programs and the properties to verify. Besides the verification tasks that are used in this competition and written in the programming language C, the SV-COMP repository also contains tasks written in Java [3] and as Horn clauses [4].

The main objectives of the Competition on Software Verification are to:

1. provide an overview of the state of the art in software-verification technology,
2. establish a repository of software-verification tasks that is widely used,
3. increase visibility of the most recent software verifiers, and
4. accelerate the transfer of new verification technology to industrial practice.

[1] `http://sv-comp.sosy-lab.org`
[2] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk`
[3] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/java`
[4] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses`

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 373–388, 2014.

The large attendance at the past competition sessions at TACAS witnesses that the community is interested in the topic and that the competition really helps achieving the above-mentioned objectives (1) and (3). Also, objective (2) is achieved: an inspection of recent publications on algorithms for software verification reveals that it becomes a standard for evaluating new algorithms to use the established verification benchmarks from the SV-COMP repository.

The difference of SV-COMP to other competitions [5] [6] [7] [8] [9] [10] [11] [12] [13] is that we focus on evaluating tools for *fully automatic* verification of program *source code* in a standard programming language [1, 2]. The experimental evaluation is performed on dedicated machines that provide the same *limited* amount of resources to each verification tool.

2 Procedure

The procedure for the competition was not changed in comparison to the previous editions [1,2], and consisted of the phases (1) *benchmark submission* (collect and classify new verification tasks), (2) *training* (teams inspect verification tasks and train their verifiers), and (3) *evaluation* (verification runs with all competition candidates and review of the system descriptions by the competition jury). All systems and their descriptions were again archived and stamped for identification with SHA hash values. Also, before public announcement of the results, all teams received the preliminary results of their verifier for approval. After the competition experiments for the 'official' categories were finished, some teams participated in demonstration categories, in order to experiment with new categories and new rules for future editions of the competition.

3 Definitions and Rules

As a new feature of the competition and to streamline the specification of the various properties, we introduced a syntax for properties (described below). The definition of verification tasks was not changed (taken from [2]).

Verification Tasks. 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:

[5] `http://www.satcompetition.org`
[6] `http://www.smtcomp.org`
[7] `http://ipc.icaps-conference.org`
[8] `http://www.qbflib.org/competition.html`
[9] `http://fmv.jku.at/hwmcc12`
[10] `http://www.cs.miami.edu/~tptp/CASC`
[11] `http://termination-portal.org`
[12] `http://fm2012.verifythis.org`
[13] `http://rers-challenge.org`

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 by a tool crash, or exhausts the computing resources time or memory (i.e., the competition candidate does not succeed in computing an answer TRUE or FALSE).

For the counterexample path that must be produced as WITNESS for the result FALSE, we did not require a particular fixed format. (Future editions of SV-COMP will support machine-readable error witnesses, such that error witnesses can be automatically validated by a verifier.) The TIME is measured as consumed CPU time until the verifier terminates, including the consumed CPU time of all processes that the verifier started. If TIME is equal to or larger than the time limit, then the verifier is terminated and the ANSWER is set to 'timeout' (and interpreted as UNKNOWN). The verification tasks are partitioned into nine separate categories and one category *Overall* that contains all verification tasks. The categories, their defining category-set files, and the contained programs are explained under Verification Tasks on the competition web site.

Properties. The specification to be verified is stored in a file that is given as parameter to the verifier. In the repository, the specifications are available in `.prp` files in the main directory.

The definition `init(main())` gives the initial states of the program by a call of function `main` (with no parameters). The definition `LTL(f)` specifies that formula `f` holds at every initial state of the program. The LTL (linear-time temporal logic) operator `G f` means that `f` globally holds (i.e., everywhere during the program execution), and the operator `F f` means that `f` eventually holds (i.e., at some point during the program execution). The proposition `label(ERROR)` is true if the C label `ERROR` is reached, and the proposition `end` is true if the program execution terminates (e.g., return of function `main`, program exit, abort).

Label Unreachability. The reachability property p_{error} is encoded in the program source code using a C label and expressed using the following specification (the interpretation of the LTL formula is given in Table 1):

```
CHECK( init(main()), LTL(G ! label(ERROR)) )
```

The new syntax (in comparison to previous SV-COMP editions) allows a more general specification of the reachability property, by decoupling the specification from the program source code, and thus, not requiring the label to be named `ERROR`.

Memory Safety. The memory-safety property $p_{\text{memsafety}}$ (only used in one category) consists of three partial properties and is expressed using the following specification (interpretation of formulas given in Table 1):

```
CHECK( init(main()), LTL(G valid-free) )
CHECK( init(main()), LTL(G valid-deref) )
CHECK( init(main()), LTL(G valid-memtrack) )
```

Table 1. Formulas used in the competition, together with their interpretation

Formula	Interpretation
`G ! label(ERROR)`	The C label `ERROR` is not reachable on any finite execution of the program.
`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.
`F end`	All program executions are finite and end on proposition `end` (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 2013 and 2014 (taken from [2])

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result
FALSE correct	+1	Violation of property in program was correctly found
FALSE incorrect	−4	Violation reported but property holds (false alarm)
TRUE correct	+2	Correct program reported to satisfy property
TRUE incorrect	−8	Incorrect program reported as correct (missed bug)

The verification result FALSE for the property $p_{\text{memsafety}}$ is required to include the violated partial property: FALSE(p), with $p \in \{p_{\text{valid-free}}, p_{\text{valid-deref}}, p_{\text{valid-memtrack}}\}$, means that the (partial) property p is violated. According to the requirements for verification tasks, all programs in category *MemorySafety* violate at most one (partial) property $p \in \{p_{\text{valid-free}}, p_{\text{valid-deref}}, p_{\text{valid-memtrack}}\}$. Per convention, function `malloc` is assumed to always return a valid pointer, i.e., the memory allocation never fails, and function `free` always deallocates the memory and makes the pointer invalid for further dereferences.

Program Termination. The termination property $p_{\text{termination}}$ (only used in a demonstration category) is based on the proposition `end` and expressed using the following specification (interpretation in Table 1):

```
CHECK( init(main()), LTL(F end) )
```

Evaluation by Scores and Run Time. The scoring schema was not changed from SV-COMP 2013 to 2014 and is given in Table 2. The ranking is decided based on the sum of points 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. Sanity tests on obfuscated versions of verification tasks (renaming of variable and function names; renaming of file)

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

Competition candidate	Ref.	Jury member	Affiliation
Blast 2.7.2	[31]	Vadim Mutilin	ISP RAS, Moscow, Russia
Cbmc	[24]	Michael Tautschnig	Queen Mary U, London, UK
CPAchecker	[25]	Stefan Löwe	U Passau, Germany
CPAlien	[27]	Petr Muller	TU Brno, Czech Republic
CSeq-Lazy	[20]	Bernd Fischer	Stellenbosch U, South Africa
CSeq-Mu	[33]	Gennaro Parlato	U Southampton, UK
Esbmc 1.22	[26]	Lucas Cordeiro	FUA, Manaus, Brazil
FrankenBit	[15]	Arie Gurfinkel	SEI, Pittsburgh, USA
Llbmc	[11]	Stephan Falke	KIT, Karlsruhe, Germany
Predator	[9]	Tomas Vojnar	TU Brno, Czech Republic
Symbiotic 2	[32]	Jiri Slaby	Masaryk U, Brno, Czech Rep.
Threader	[29]	Corneliu Popeea	TU Munich, Germany
Ufo	[14]	Aws Albarghouthi	U Toronto, Canada
Ultimate Automizer	[16]	Matthias Heizmann	U Freiburg, Germany
Ultimate Kojak	[10]	Alexander Nutz	U Freiburg, Germany

did not reveal any discrepancy of the results. *Opting-out from Categories* and and *Computation of Score for Meta Categories* were defined as in SV-COMP 2013 [2]. The *Competition Jury* consists again of the chair and one member of each participating team. Team representatives are indicated in Table 3.

4 Participating Teams

Table 3 provides an overview of the participating competition candidates. The detailed summary of the achievements for each verifier is presented in Sect. 5. A total of 15 competition candidates participated in SV-COMP 2014: Blast 2.7.2 [14], Cbmc [15], CPAchecker [16], CPAlien [17], CSeq-Lazy [18], CSeq-Mu, Esbmc 1.22 [19], FrankenBit [20], Llbmc [21], Predator [22], Symbiotic 2 [23], Threader [24], Ufo [25], Ultimate Automizer [26], and Ultimate Kojak [27].

[14] http://forge.ispras.ru/projects/blast
[15] http://www.cprover.org/cbmc
[16] http://cpachecker.sosy-lab.org
[17] http://www.fit.vutbr.cz/~imuller/cpalien
[18] http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html
[19] http://www.esbmc.org
[20] http://bitbucket.org/arieg/fbit
[21] http://llbmc.org
[22] http://www.fit.vutbr.cz/research/groups/verifit/tools/predator
[23] https://sf.net/projects/symbiotic
[24] http://www7.in.tum.de/tools/threader
[25] http://bitbucket.org/arieg/ufo
[26] http://ultimate.informatik.uni-freiburg.de/automizer
[27] http://ultimate.informatik.uni-freiburg.de/kojak

Table 4. Technologies and features that the verification tools offer (*incl. demo track*)

Verification tool (incl. demo track)	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Check.	Explicit-Value Analysis	Interval Analysis	Shape Analysis	Bit-precise Analysis	ARG-based Analysis	Lazy Abstraction	Interpolation	Concurrency Support	Ranking Functions
APROVE			✓										✓
BLAST 2.7.2	✓	✓							✓	✓	✓		
CBMC				✓				✓				✓	
CPALIEN					✓		✓						
CPACHECKER	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		
CSEQ-LAZY				✓								✓	
CSEQ-MU				✓								✓	
ESBMC 1.22				✓				✓				✓	
FUNCTION						✓							✓
FRANKENBIT				✓				✓			✓		
LLBMC				✓									
PREDATOR							✓						
SYMBIOTIC 2			✓										
T2	✓	✓				✓			✓	✓	✓		✓
TAN	✓	✓	✓	✓		✓		✓		✓			✓
THREADER	✓	✓							✓		✓	✓	
UFO	✓	✓		✓		✓			✓	✓	✓		
ULTIMATE AUTOMIZER	✓	✓								✓	✓		
ULTIMATE KOJAK	✓	✓								✓	✓		
ULTIMATE BÜCHI	✓	✓								✓	✓		✓

Table 4 lists the features and technologies that are used in the verification tools. Counterexample-guided abstraction refinement (CEGAR) [8], predicate abstraction [13], bounded model checking [6], lazy abstraction [19], and interpolation for predicate refinement [18] are implemented in many verifiers. Other features that were implemented include symbolic execution [22], the construction of an abstract reachability graph (ARG) as proof of correctness [3], and shape analysis [21]. Only a few tools support the verification of concurrent programs. Computing ranking functions [28] for proving termination is a feature that is implemented in tools that participated in the demo category on termination.

Table 5. Quantitative overview over all results — Part 1 (score / CPU time)

Competition candidate Representing jury member	**BitVectors** 86 points max. 49 verif. tasks	**Concurrency** 136 points max. 78 verif. tasks	**ControlFlow** 1 261 points max. 843 verif. tasks	**DeviceDrivers** 2 766 points max. 1 428 verif. tasks	**HeapManip.** 135 points max. 80 verif. tasks
Blast 2.7.2 V. Mutilin, Moscow, Russia	—	—	508 32 000 s	**2 682** 13 000 s	—
Cbmc M. Tautschnig, London, UK	**86** 2 300 s	**128** 29 000 s	397 42 000 s	2 463 390 000 s	**132** 12 000 s
CPAchecker S. Löwe, Passau, Germany	**78** 690 s	0 0.0 s	**1009** 9 000 s	2 613 28 000 s	107 210 s
CPAlien P. Muller, Brno, Czech Republic	—	—	455 6 500 s	—	71 70 s
CSeq-Lazy B. Fischer, Stellenbosch, ZA	—	**136** 1 000 s	—	—	—
CSeq-Mu G. Parlato, Southampton, UK	—	**136** 1 200 s	—	—	—
Esbmc 1.22 L. Cordeiro, Manaus, Brazil	77 1 500 s	32 30 000 s	949 35 000 s	2 358 140 000 s	97 970 s
FrankenBit A. Gurfinkel, Pittsburgh, USA	—	—	**986** 6 300 s	**2 639** 3 000 s	—
Llbmc S. Falke, Karlsruhe, Germany	**86** 39 s	0 0.0 s	**961** 13 000 s	0 0.0 s	**107** 130 s
Predator T. Vojnar, Brno, Czech Republic	-92 28 s	0 0.0 s	511 3 400 s	50 9.9 s	**111** 9.5 s
Symbiotic 2 J. Slaby, Brno, Czech Republic	39 220 s	-82 5.7 s	41 39 000 s	980 2 200 s	105 15 s
Threader C. Popeea, Munich, Germany	—	100 3 000 s	—	—	—
Ufo A. Albarghouthi, Toronto, Canada	—	—	912 14 000 s	**2 642** 5 700 s	—
Ultimate Automizer M. Heizmann, Freiburg, Germany	—	—	164 6 000 s	—	—
Ultimate Kojak A. Nutz, Freiburg, Germany	-23 1 100 s	0 0.0 s	214 5 100 s	0 0.0 s	18 35 s

Table 6. Quantitative overview over all results — Part 2 (score / CPU time)

Competition candidate Representing jury member	**MemorySafety** 98 points max. 61 verif. tasks	**Recursive** 39 points max. 23 verif. tasks	**Sequentialized** 364 points max. 261 verif. tasks	**Simple** 67 points max. 45 verif. tasks	**Overall** 4 718 points max. 2 868 verif. tasks
Blast 2.7.2 V. Mutilin, Moscow, Russia	— 	— 	— 	30 5 400 s	—
Cbmc M. Tautschnig, London, UK	4 11 000 s	**30** 11 000 s	**237** 47 000 s	**66** 15 000 s	**3 501** 560 000 s
CPAchecker S. Löwe, Passau, Germany	**95** 460 s	0 0.0 s	97 9 200 s	**67** 430 s	**2 987** 48 000 s
CPAlien P. Muller, Brno, Czech Republic	9 690 s	— 	— 	— 	—
CSeq-Lazy B. Fischer, Stellenbosch, ZA	— 	— 	— 	— 	—
CSeq-Mu G. Parlato, Southampton, UK	— 	— 	— 	— 	—
Esbmc 1.22 L. Cordeiro, Manaus, Brazil	-136 1 500 s	-53 4 900 s	**244** 38 000 s	31 27 000 s	975 280 000 s
FrankenBit A. Gurfinkel, Pittsburgh, USA	— 	— 	— 	37 830 s	—
Llbmc S. Falke, Karlsruhe, Germany	**38** 170 s	3 0.38 s	**208** 11 000 s	0 0.0 s	**1 843** 24 000 s
Predator T. Vojnar, Brno, Czech Republic	**14** 39 s	-18 0.12 s	-46 7 700 s	0 0.0 s	-184 11 000 s
Symbiotic 2 J. Slaby, Brno, Czech Republic	-130 7.5 s	6 0.93 s	-32 770 s	-22 13 s	-220 42 000 s
Threader C. Popeea, Munich, Germany	— 	— 	— 	— 	—
Ufo A. Albarghouthi, Toronto, Canada	— 	— 	83 4 800 s	**67** 480 s	—
Ultimate Automizer M. Heizmann, Freiburg, Germany	— 	**12** 850 s	49 3 000 s	— 	399 10 000 s
Ultimate Kojak A. Nutz, Freiburg, Germany	0 0.0 s	**9** 54 s	9 1 200 s	0 0.0 s	139 7 600 s

Table 7. Overview of the top-three verifiers for each category (CPU time in s)

Rank	Candidate	Score	CPU Time	Solved Tasks	False Alarms	Missed Bugs
BitVectors						
1	**Llbmc**	**86**	39	49		
2	Cbmc	86	2 300	49		
3	CPAchecker	78	690	45		
Concurrency						
1	**CSeq-Lazy**	**136**	1 000	78		
2	CSeq-Mu	136	1 200	78		
3	Cbmc	128	29 000	76	1	
ControlFlow						
1	**CPAchecker**	**1009**	9 000	764	2	
2	FrankenBit	986	6 300	752		**2**
3	Llbmc	961	13 000	783		**14**
DeviceDrivers						
1	**Blast 2.7.2**	**2 682**	13 000	1 386		**2**
2	Ufo	2 642	5 700	1 354	2	**3**
3	FrankenBit	2 639	3 000	1 383	5	**5**
HeapManipulation						
1	**Cbmc**	**132**	12 000	78		
2	Predator	111	9.5	68		
3	Llbmc	107	130	66		
MemorySafety						
1	**CPAchecker**	**95**	460	59		
2	Llbmc	38	170	31		
3	Predator	14	39	43	12	
Recursive						
1	**Cbmc**	**30**	11 000	22		**1**
2	Ultimate Automizer	12	850	9		
3	Ultimate Kojak	9	54	7		
SequentializedConcurrency						
1	**Esbmc 1.22**	**244**	38 000	187	2	
2	Cbmc	237	47 000	225		**10**
3	Llbmc	208	11 000	191	3	**3**
Simple						
1	**CPAchecker**	**67**	430	45		
2	Ufo	67	480	45		
3	Cbmc	66	15 000	44		
Overall						
1	**Cbmc**	**3 501**	560 000	2 597	3	**90**
2	CPAchecker	2 987	48 000	2 421	12	
3	Llbmc	1 843	24 000	1 123	3	**17**

5 Results and Discussion

The results that we obtained in the competition experiments and reported in this article represent the state of the art in fully automatic and publicly available software-verification tools. The results show achievements in effectiveness (number of verification tasks that can be solved, correctness of the results) and efficiency (resource consumption in terms of CPU time). All reported results were approved by the participating teams.

The verification runs were natively executed on dedicated unloaded compute servers with a 3.4 GHz 64-bit Quad Core CPU (Intel i7-2600) and a GNU/Linux operating system (x86_64-linux). The machines had (at least) 16 GB of RAM, of which exactly 15 GB were made available to the verification tools. The run-time limit for each verification run was 15 min of CPU time. The tables report the run time in seconds of CPU time; all measured values are rounded to two significant digits. One complete competition run with all candidates on all verification tasks required a total of 51 days of CPU time.

Tables 5 and 6 present a quantitative overview over all tools and all categories. The tools are listed in alphabetical order; every table cell for competition results lists the score in the first row and the CPU time for successful runs in the second row. We indicated the top-three candidates by formatting their score in bold face and in larger font size. The entry '—' 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 meta categories (*Overall* and *ControlFlow*; consisting of several sub-categories) were computed using normalized scores as defined in last year's report [2].

Table 7 reports the top-three verifiers for each category. The run time refers to successfully solved verification tasks. The columns 'False Alarms' and 'Missed Bugs' report the number of verification tasks for which the tool reported wrong results: reporting a counterexample path but the property holds (false positive) and claiming that the program fulfills the property although it actually contains a bug (false negative), respectively.

Score-Based Quantile Functions for Quality Assessment. As described in the previous competition report [2], score-based quantile functions are a helpful visualization of the results. The competition web page[28] presents such a plot for each category, while we illustrate in Fig. 1 only the category *Overall* (all verification tasks). A total of eight verifiers participated in category *Overall*, for which the quantile plot shows the overall performance over all categories. (Note that the scores are normalized as described last year [2].)

Overall Quality Measured in Scores (Right End of Graph). Cbmc is the winner of this category, because 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 7; 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

[28] `http://sv-comp.sosy-lab.org/2014/results`

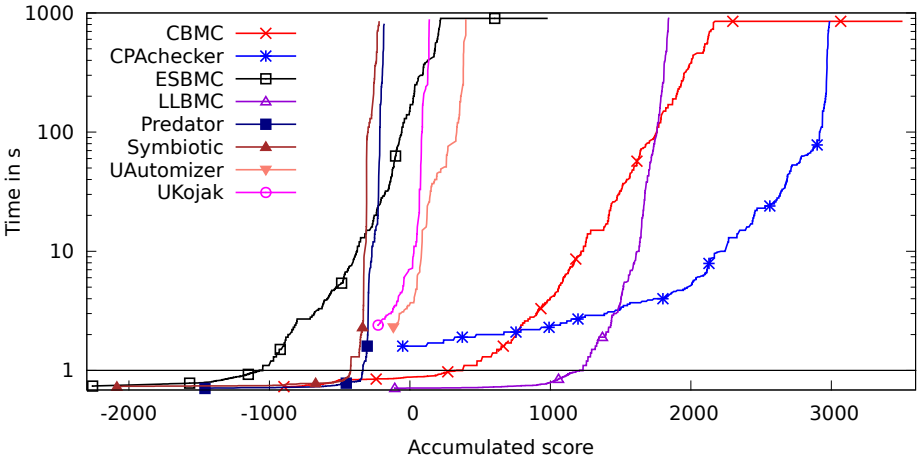

Fig. 1. Quantile functions: For each competition candidate, we plot all data points (x, y) such that the maximum run time of the n fastest correct verification runs is y and x is the accumulated score of all incorrect results and those n correct results. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s. The graphs are decorated with symbols at every 15-th data point.

(x-coordinate), i.e., amount of incorrect verification work. Verifiers should start with a score close to zero; CPAchecker is best in this aspect (also the right-most columns of category *Overall* in Table 7 report this: only 12 false alarms and no missed bug for all 2 868 verification tasks).

Characteristics of the Verification Tools. The plot visualizations also help understanding how the verifiers work internally: (1) The y-coordinate of the left-most data point refers to the 'easiest' verification task for the verifier. We can see that verifiers that are based on a Java virtual machine need some start-up time (CPAchecker, Ultimate). (2) The y-coordinate of the right-most data point refers to the successfully solved verification task that the verifier spent most time on (this is mostly just below the time limit). We can read the ranking of verifiers in this category from right to left. (3) The area below a graph is proportional to the accumulated CPU time for successfully solved tasks. We can identify the most resource-efficient verifiers by looking at the graphs close to the x-axis. (4) Also the shape of the graph can give interesting insights: From the two horizontal lines just below the time limit (at 850 s and 895 s, resp.), we can see that two of the bounded model checkers (Cbmc, Esbmc 1.22) return a result just before the time limit is reached. The quantile plot for category *DeviceDrivers64* (not available here, but on the competition web page) shows an interesting bend at about 20 s of run time for verifier CPAchecker: the verifier gives up with one strategy (without abstraction) and performs an internal restart for using another strategy (with abstraction and CEGAR-based refinement).

Table 8. Quantitative overview over results in category *Termination*

Competition candidate Representing team member	**Termination-crafted** 89 points max. 47 verif. tasks	**Termination-ext** 265 points max. 199 verif. tasks	**Errors** false alarms missed bugs
AProVE [12] J. Giesl, Aachen, Germany	**58** 360 s	**0** 0 s	
FuncTion [34] C. Urban, Paris, France	**20** 220 s	**0** 0 s	
T2 [7] M. Brockschmidt, Cambridge, UK	**46** 80 s	**50** 64 s	
Tan [23] C. Wintersteiger, Oxford, UK	**12** 33 s	**23** 590 s	2 1
Ultimate Büchi [17] M. Heizmann, Freiburg, Germany	**57** 250 s	**117** 4 800 s	

Robustness, Soundness, and Completeness. The best tools of each category show that state-of-the-art verification technology significantly progressed in terms of wrong verification results. Table 7 reports, in its last two columns, the number of false alarms and missed bugs, respectively, for the best verifiers in each category: There is a low number of false alarms (wrong bug reports), which witnesses that verification technology can avoid wasted developer time being spent on investigation of spurious bug reports. Also in terms of soundness, the results look promising, considering that the most missed bugs (wrong safety claims) were reported by bounded model checkers. In three categories, the top-three verifiers did not report any wrong result.

Demonstration Categories. For the first time in SV-COMP, we performed experiments in demonstration categories, i.e., categories for which we wanted to try out new applications of verification, new properties, or new rules. For the demonstration categories, we neither rank the results nor assign awards.

Termination. Checking program termination is also an important objective of software verification. We started with two sets of verification tasks: category *Termination-crafted* is a community-contributed set of verification tasks that were designed by verification researchers for the purpose of evaluating termination checkers (programs were collected from well-known papers in the area), and category *Termination-ext* is a selection of verification tasks from existing categories for which the result was determined during the demonstration runs.

Table 9. Re-verification of verification results using error witnesses; verification time in s of CPU time; path length in number of edges; expected result is 'false' in all cases

Verification task	CBMC verification	Path length	CPACHECKER re-verification
parport_false	37	179	11
eureka_01_false	0.36	42	64
Tripl.2.ufo.BOUNDED-10.pals.c	0.81	356	53
Tripl.2.ufo.UNBOUNDED.pals.c	0.80	355	44
gigaset.ko_false	44	140	120
tcm_vhost-ko–32_7a	26	197	62
vhost_net-ko–32_7a	21	89	72
si4713-i2c-ko–111_1a	430	75	12

Table 8 shows the results, which are promising: five teams participated, namely APROVE [29], FUNCTION [30], T2 [31], TAN [32], and ULTIMATE BÜCHI [33]. Also, the quality of the termination checkers was extremely good: almost all tools had no false positive ('false alarms', the verifier reported the program would not terminate although it does) and no false negative ('missed bug', the verifier reported termination but infinite looping is possible).

Device-Driver Challenge. Competitions are always looking for hard problems. We received some unsolved problems from the LDV project [34]. Three teams participated and could compute answers to 6 of the 15 problems: CBMC found 3, CPACHECKER found 4, and ESBMC found 2 solutions to the problems.

Error-Witnesses. One of the objectives of program verification is to provide a witness for the verification result. This is an open problem of verification technology: there is no commonly supported witness format yet, and the verifiers are not producing accurate witnesses that can be automatically assessed for validity [35]. The goal of this demonstration category is to change this (restricted to error witnesses for now): in cooperation with interested groups we defined a format for error witnesses and the verifiers were asked to produce error paths in that format, in order to validate their error paths with *another* verification tool.

Three tools participated in this category: CBMC, CPACHECKER, and ESBMC. The demo revealed many interesting insights on practical issues of using a common witness format, serving as a test before introducing it as a requirement to

[29] `http://aprove.informatik.rwth-aachen.de`
[30] `http://www.di.ens.fr/~urban/FuncTion.html`
[31] `http://research.microsoft.com/en-us/projects/t2`
[32] `http://www.cprover.org/termination/cta/index.shtml`
[33] `http://ultimate.informatik.uni-freiburg.de/BuchiAutomizer`
[34] `http://linuxtesting.org/project/ldv`
[35] There was research already on reusing previously computed error paths, but by the same tool and in particular, using tool-specific formats: for example, ESBMC was extended to reproduce errors via instantiated code [30], and CPACHECKER was used to re-check previously computed error paths by interpreting them as automata that control the state-space search [5].

the next edition of the competition. We will report here only a few cases to show how this technique can help. We selected a group of verification tasks (with expected verification result 'false') that CBMC could solve, but CPACHECKER was not able to compute a verification result. We started CPACHECKER again on the verification task, now together with CBMC's error witness. Table 9 reports the details of eight such runs: CPACHECKER can prove the error witnesses of CBMC valid, although it could not find the bug in the program without the hints from the witness. In some cases this is efficient (first and last row) and sometimes it is quite inefficient: the matching algorithm needs improvement. The matching is based purely on syntactical hints (sequence of tokens of the source program). This technique of re-verifying a program with a different verification tool significantly increases the confidence in the verification result (and makes false-alarms unnecessary).

6 Conclusion

The third edition of the Competition on Software Verification had more participants than before: the participation in the 'official' categories increased from eleven to fifteen teams, and five teams took part in the demonstration on termination checking. The number of benchmark problems increased to a total of 2 868 verification tasks (excluding demonstration categories). The organizer and the jury made sure that the competition follows the high quality standards of the TACAS conference, in particular to respect the important principles of fairness, community support, transparency, and technical accuracy.

The results showcase the progress in developing new algorithms and data structures for software verification, and in implementing efficient tools for fully-automatic program verification. The best verifiers have shown good quality in the categories that they focus on, in terms of robustness, soundness, and completeness. The participants represent a variety of general approaches — SMT-based model checking, bounded model checking, symbolic execution, and program analysis showed their different, complementing strengths. Also, the SV-COMP repository of verification tasks has grown considerably: it now contains termination problems and problems for regression verification [4], but also Horn clauses and some Java programs in addition to C programs.

Acknowledgement. We thank K. Friedberger for his support during the evaluation phase and for his work on the benchmarking infrastructure, the competition jury for making sure that the competition is well-grounded in the community, and the teams for making SV-COMP possible through their participation.

References

1. Beyer, D.: Competition on software verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
2. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)

3. 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)
4. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. ESEC/FSE, pp. 389–399. ACM (2013)
5. Beyer, D., Wendler, P.: Reuse of verification results - conditional model checking, precision reuse, and verification witnesses. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 1–17. Springer, Heidelberg (2013)
6. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
7. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 413–429. Springer, Heidelberg (2013)
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
9. Dudka, K., Peringer, P., Vojnar, T.: Predator: A shape analyzer based on symbolic memory graphs (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 412–414. Springer, Heidelberg (2014)
10. Ermis, E., Nutz, A., Dietsch, D., Hoenicke, J., Podelski, A.: Ultimate Kojak (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 421–423. Springer, Heidelberg (2014)
11. Falke, S., Merz, F., Sinz, C.: LLBMC: Improved bounded model checking of C programs using LLVM (Competition contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 623–626. Springer, Heidelberg (2013)
12. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
13. Graf, S., Saïdi, H.: Construction of abstract state graphs with Pvs. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with interpolants and abstract interpretation. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
15. Gurfinkel, A., Belov, A.: FrankenBit: Bit-precise verification with many bits (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 408–411. Springer, Heidelberg (2014)
16. Heizmann, M., Christ, J., Dietsch, D., Hoenicke, J., Lindenmann, M., Musa, B., Schilling, C., Wissert, S., Podelski, A.: Ultimate automizer with unsatisfiable cores (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 418–420. Springer, Heidelberg (2014)
17. Heizmann, M., Hoenicke, J., Leike, J., Podelski, A.: Linear ranking for linear lasso programs. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 365–380. Springer, Heidelberg (2013)
18. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
20. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Lazy-CSeq: A lazy sequentialization tool for C (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 398–401. Springer, Heidelberg (2014)

21. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data-flow analysis and programs with recursive data structures. In: POPL, pp. 66–74 (1982)
22. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
23. Kröning, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
24. Kröning, D., Tautschnig, M.: CBMC – C bounded model checker (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014)
25. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 392–394. Springer, Heidelberg (2014)
26. Morse, J., Ramalho, M., Cordeiro, L., Nicole, D., Fischer, B.: ESBMC 1.22 (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 405–407. Springer, Heidelberg (2014)
27. Muller, P., Vojnar, T.: CPAlien: Shape analyzer for CPAChecker (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 395–397. Springer, Heidelberg (2014)
28. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
29. Popeea, C., Rybalchenko, A.: Threader: A verifier for multi-threaded programs (Competition contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 633–636. Springer, Heidelberg (2013)
30. Rocha, H., Barreto, R., Cordeiro, L., Neto, A.D.: Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 128–142. Springer, Heidelberg (2012)
31. Shved, P., Mandrykin, M., Mutilin, V.: Predicate analysis with BLAST 2.7. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 525–527. Springer, Heidelberg (2012)
32. Slaby, J., Strejček, J.: Symbiotic 2: More precise slicing (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 415–417. Springer, Heidelberg (2014)
33. Tomasco, E., Inverso, O., Fischer, B., La Torre, S., Parlato, G.: MU-CSeq: Sequentialization of C programs by shared memory unwindings (Competition contribution). In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 402–404. Springer, Heidelberg (2014)
34. Urban, C., Miné, A.: An abstract domain to infer ordinal-valued ranking functions. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 412–431. Springer, Heidelberg (2014)