Proc. TACAS 2015, (c) Springer

# Software Verification and Verifiable Witnesses
(Report on SV-COMP 2015)

Dirk Beyer

University of Passau, Germany

**Abstract.** SV-COMP 2015 marks the start of a new epoch of software verification: In the 4th Competition on Software Verification, software verifiers produced for each reported property violation a machine-readable error witness in a common exchange format (so far restricted to reachability properties of sequential programs without recursion). Error paths were reported previously, but always in different, incompatible formats, often insufficient to reproduce the identified bug, and thus, useless to the user. The common exchange format and the support by a large set of verification tools that use the format will make a big difference: One verifier can re-verify the witnesses produced by another verifier, visual error-path navigation tools can be developed, and here in the competition, we use witness checking to make sure that a verifier that claimed a found bug, had really found a valid error path. The other two changes to SV-COMP that we made this time were (a) the addition of the new property, a set of verification tasks, and ranking category for termination verification, and (b) the addition of two new categories for reachability analysis: Arrays and Floats. SV-COMP 2015, the fourth edition of the thorough comparative evaluation of fully-automatic software verifiers, reports effectiveness and efficiency results of the state of the art in software verification. The competition used 5 803 verification tasks, more than double the number of SV-COMP'14. Most impressively, the number of participating verifiers increased from 15 to 22 verification systems, including 13 new entries.

## 1 Introduction

The Competition on Software Verification (SV-COMP) [1] is a service to the verification community that consists of two parts: (a) the collection of verification tasks that the community of researchers in the area of software verification finds interesting and challenging, and (b) the systematic comparative evaluation of the relevant state-of-the-art tool implementations for automatic software verification with respect to effectiveness and efficiency.

*Repository of Verification Tasks.* The benchmark repository of SV-COMP [2] serves as collection of verification tasks that represent the current interest and abilities of tools for software verification. For the purpose of the competition, all verification tasks that are suitable for the competition are arranged into categories, according

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

[2] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk`

to the characteristics of the programs and the properties to be verified. The assignment is discussed in the community, implemented by the competition chair, and finally approved by the competition jury. For the 2015 edition of SV-COMP, a total of 13 categories were defined, selected from verification tasks written in the programming language C. The SV-COMP repository also contains tasks written in Java[3] and as Horn clauses[4], but those were not used in SV-COMP.

*Comparative Experimental Evaluation.* This report concentrates on describing the rules, definitions, results, and on providing other interesting information about the setup and execution of the competition experiments. The main objectives that the community and organizer would like to achieve by running yearly competitions are the following:

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 can freely and publicly be used 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.

The competition serves Objective (1) very well, which is witnessed by the past competition sessions at TACAS being among the best-attended ETAPS sessions, and by the large number of participating verification teams. Objective (2) is also served well: the repository was rapidly growing in the last years and reached a considerable size; many publications on algorithms for software verification base the experimental evaluation on the established verification benchmarks from the SV-COMP repository, and thus, it becomes a standard for evaluating new algorithms to use the SV-COMP collection. SV-COMP 2015 was a big step forward with respect to Objective (3). It was requested since long that **verification witnesses** should be given in a common format and can be accepted only if re-validated automatically by an independent witness checker. We have worked towards verifiable witnesses with success, but there is a lot of work left to be done. Whether or not SV-COMP serves well towards Objective (4) cannot be evaluated here.

*Related Competitions.* There are two other competitions in the field of software verification in general: RERS[5] and VerifyThis[6]. In difference to the RERS Challenges, SV-COMP is an experimental evaluation that is performed on dedicated machines, which provide the same *limited* amount of resources to each verification tool. In difference to the VerifyThis Competitions, SV-COMP focuses on evaluating tools for *fully-automatic* verification of program *source code* in a standard programming language. A more comprehensive list of other competitions was given in the previous report [3].

[3] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/java`
[4] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses`
[5] `http://rers-challenge.org`
[6] `http://etaps2015.verifythis.org`

## 2 Procedure

The procedure for the competition organization was the same as in previous editions of SV-COMP [1, 2, 3], consisting of the three 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). Again, SV-COMP was an open competition, i.e., the verification tasks were known before the participating verifiers were submitted, such that there were no surprises and developers were able to train the verifiers. All systems and their descriptions have been archived on the SV-COMP web site and stamped for identification with SHA hash values. All teams received the preliminary results of their verifier for approval, before the results were publicly announced. This time, there was no demonstration category.

## 3 Definitions, Formats, and Rules

The specification of the various properties was streamlined last year, such that it was easy to extend the property language to express reachability using function calls instead of C labels in the source code of the verification tasks, which eliminates completely the need of C labels in the verification tasks. Most importantly, we introduced a *syntax for error witnesses* (more details are given below). The definition of verification task was not changed (taken from [3]).

**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:

**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).

WITNESS is explained below in an own sub-section. TIME is measured as consumed CPU time until the verifier terminates, including the consumed CPU time of all processes that the verifier started [4]. 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).

The verification tasks were partitioned into twelve 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.

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

| Formula | Interpretation |
|---|---|
| `G ! call(foo())` | A call to function `foo` 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. |

**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 as `.prp` files in the respective directories of the benchmark categories.

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 `call(foo)` is true if a call to the function `foo` is reached, and the proposition `end` is true if the program execution terminates (e.g., return of function `main`, program exit, abort).

*Call Unreachability.* The reachability property $p_{\text{error}}$ is encoded in the program source code using a call to function `__VERIFIER_error()`, expressed using the following specification (the interpretation of the LTL formula is given in Table 1):

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

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, functions `malloc` and `alloca` are 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. Further assumptions are explained under *Definitions and Rules* on the competition web site.

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

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

**Verifiable Witnesses.** For the first time in the history of software verification (of real-world, C programs),[7] we defined a formal, machine-readable format for error witnesses and required the verifiers to produce automatically-verifiable witnesses for the counterexample path that is part of the result triple as WITNESS. This new rule was applied to the categories with reachability properties and verification tasks of sequential programs without recursion. If an error path required recursive function calls or heap operations, the witness was not checked.

We represent witnesses as automata. Formally, a witness automaton consists of states and transitions, where each transition is annotated with data that can be used to match program executions. A data annotation can be (a) a token number (position in the token stream that the parser receives), (b) an assumption (for example, the assumption $a = 1$; means that program variable $a$ has value 1), (c) a line number and a file name, (d) a function call or return, and (e) a piece of source-code syntax. More details are given on a web page.[8]

A witness checker is a software verifier that analyzes the synchronized product of the program with the witness automaton, where transitions are synchronized using program operations and transition annotations. This means that the witness automaton observes the program paths that the verifier wants to explore: if the operation on the program path does not match the transition of the witness automaton, then the verifier is forbidden to explore that path further; if the operation on the program path matches, then the witness automaton and the program proceed to the next state, possibly restricting the program's state such that the assumptions given in the data annotation are satisfied.

In SV-COMP, the time limit for a validation run was set to 10 % of the CPU time for a verification run, i.e., the witness checker was limited to 90 s. If the witness checker did not succeed in the given amount of time, then most likely the witness was not concrete enough (time for validation can be a quality indicator).

[7] 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 [21], and CPACHECKER was used to re-check previously computed error paths by interpreting them as automata that control the state-space search [5]. The competition on termination uses CPF: `http://cl-informatik.uibk.ac.at/software/cpf`.

[8] `http://sv-comp.sosy-lab.org/2015/witnesses`

**Table 2.** Scoring schema for SV-COMP 2015 (penalties increased, cf. [3])

| Reported result | Points | Description |
|---|---|---|
| UNKNOWN | 0 | Failure to compute verification result |
| FALSE correct | $+1$ | Violation of property in program was correctly found |
| FALSE incorrect | $-6$ | Violation reported but property holds (false alarm) |
| TRUE correct | $+2$ | Correct program reported to satisfy property |
| TRUE incorrect | $-12$ | Incorrect program reported as correct (wrong proofs) |

Machine-readable witnesses in a common exchange format have the following advantages for the competition:

– Witness Validation: The answer FALSE is only accepted if the witness can be validated by an automatic witness checker.
– Witness Inspection: If a verifier found an error in a verification task that was previously assumed to have expected outcome TRUE, the witness that was produced could immediately be validated with two different verifiers (one explicit-value-based and one SAT-based).

Outside the competition, the following examples are among the many useful applications of witnesses in a common format:

– Witness Database: Witnesses can be stored in databases as later source of information.
– Bug Report: Witnesses can be a useful attachment for bug reports, in order to precisely report to the developers what the erroneous behavior is.
– Bug Confirmation: To gain more confidence that a bug is indeed present, the error witness can be re-confirmed with a different verifier, perhaps using a completely different technology.
– Re-Verification: If the result FALSE was established, the error witness can later be reused to re-establish the verification result with much less resources, for example, if the program source code is slightly changed and the developer is interested if the same bug still exists in a later version of the program [5].

**Evaluation by Scores and Run Time.** The scoring schema was changed in order to increase the penalty for wrong results (in comparison to the previous edition of the competition by a factor of 1.5). The overview 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. *Opting-out from Categories* 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 of the jury are listed in Table 3.

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

| Competition candidate | Ref. | Jury member | Affiliation |
|---|---|---|---|
| APROVE | [23] | Thomas Ströder | RWTH Aachen, Germany |
| BEAGLE | | Dexi Wang | Tsinghua U, China |
| BLAST | [22] | Vadim Mutilin | ISP RAS, Russia |
| CASCADE | [26] | Wei Wang | New York U, USA |
| CBMC | [15] | Michael Tautschnig | Queen Mary U London, UK |
| CPACHECKER | [8] | Matthias Dangl | U Passau, Germany |
| CPAREC | [7] | Ming-Hsien Tsai | Academia Sinica, Taiwan |
| ESBMC | [17] | Jeremy Morse | U Bristol, UK |
| FOREST | [9] | Pablo Sanchez | U Cantabria, Spain |
| FORESTER | [13] | Ondřej Lengál | Brno UT, Czech Republic |
| FUNCTION | [25] | Caterina Urban | ENS Paris, France |
| HIPTNT+ | [16] | Ton-Chanh Le | NUS, Singapore |
| LAZY-CSEQ | [14] | Gennaro Parlato | U Southampton, UK |
| MAP2CHECK | | Herbert O. Rocha | FUA, Brazil |
| MU-CSEQ | [24] | Bernd Fischer | Stellenbosch U, South Africa |
| PERENTIE | [6] | Franck Cassez | Macquarie U/NICTA, Australia |
| PREDATORHP | [18] | Tomáš Vojnar | Brno UT, Czech Republic |
| SEAHORN | [10] | Arie Gurfinkel | SEI, USA |
| SMACK+CORRAL | [11] | Zvonimir Rakamarić | U Utah, USA |
| ULTIAUTOMIZER | [12] | Matthias Heizmann | U Freiburg, Germany |
| ULTIKOJAK | [20] | Alexander Nutz | U Freiburg, Germany |
| UNB-LAZY-CSEQ | [19] | Salvatore La Torre | U Salerno, Italy |

## 4 Results and Discussion

The results of 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.

**Technical Resources.** The technical setup for running the experiments was similar to last year [3], except that we used eight, newer machines. All verification runs were natively executed on dedicated unloaded compute servers with a 3.4 GHz 64-bit Quad-Core CPU (Intel i7-4770) and a GNU/Linux operating system (x86_64-linux). The machines had 33 GB of RAM, of which exactly 15 GB (memory limit) were made available to the verifiers. The run-time limit for each verification run was 15 min of CPU time. The run-time limit for each witness check was set to 1.5 min of CPU time. The tables report the run time in seconds of CPU time; all measured values are rounded to two significant digits.

**Table 4.** Technologies and features that the verification tools offer

| Verifier | CEGAR | Predicate Abstraction | Symbolic Execution | Bounded Model Checking | k-Induction | Property-Directed Reachability | Explicit-Value Analysis | Numerical Interval Analysis | Shape Analysis | Separation Logic | Bit-Precise Analysis | ARG-Based Analysis | Lazy Abstraction | Interpolation | Automata-Based Analysis | Concurrency Support | Ranking Functions |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APROVE | | | ✓ | | | | ✓ | ✓ | | | ✓ | | | | | | ✓ |
| BEAGLE | ✓ | ✓ | | ✓ | | | | | | | | | | | | | |
| BLAST | ✓ | ✓ | | | | | ✓ | | | | | ✓ | ✓ | ✓ | | | |
| CASCADE | | | ✓ | ✓ | | | | | | | ✓ | | | | | | |
| CBMC | | | | ✓ | ✓ | | | | | | ✓ | | | | | ✓ | |
| CPACHECKER | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | | | |
| CPAREC | ✓ | ✓ | | | | | | | | | | ✓ | ✓ | ✓ | | | |
| ESBMC | | | | ✓ | ✓ | | | | | | ✓ | | | | | ✓ | |
| FOREST | | | ✓ | ✓ | | | | | | | ✓ | | | | | | |
| FORESTER | | | | | | | | | ✓ | | | | | | ✓ | | |
| FUNCTION | | | | | | | | ✓ | | | | | | | | | ✓ |
| HIPTNT+ | | | | | | | | | | ✓ | | | | | | | ✓ |
| LAZY-CSEQ | | | | ✓ | | | | | | | ✓ | | | | | ✓ | |
| MAP2CHECK | | | | ✓ | | | | | | | ✓ | | | | | | |
| MU-CSEQ | | | | ✓ | | | | | | | ✓ | | | | | ✓ | |
| PERENTIE | ✓ | | ✓ | | | | ✓ | | | | | | | ✓ | ✓ | | |
| PREDATORHP | | | | | | | | | ✓ | | | | | | | | |
| SEAHORN | | | | ✓ | | ✓ | | ✓ | | | | ✓ | ✓ | ✓ | | | |
| SMACK+CORRAL | ✓ | | | ✓ | | | | | | | | | ✓ | | | | |
| ULTIAUTOMIZER | ✓ | ✓ | | | | | | | | | | | ✓ | ✓ | ✓ | | ✓ |
| ULTIKOJAK | ✓ | ✓ | | | | | | | | | | | ✓ | ✓ | ✓ | | |
| UNB-LAZY-CSEQ | ✓ | ✓ | | | | | | | | | | ✓ | ✓ | ✓ | | ✓ | |

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

| **Verifier**<br>Repr. jury member | **Arrays**<br>145 points<br>86 tasks | **BitVectors**<br>83 points<br>47 tasks | **Concurrent**<br>1 222 points<br>1 003 tasks | **ContrFlow**<br>3 122 points<br>1 927 tasks | **DeviceDriv.**<br>3 097 points<br>1 650 tasks | **Floats**<br>140 points<br>81 tasks | **HeapManip.**<br>135 points<br>80 tasks |
|---|---|---|---|---|---|---|---|
| **APROVE**<br>T. Ströder, Germany | | | | | | | |
| **BEAGLE**<br>D. Wang, China | | 4<br>58 s | | | | | |
| **BLAST**<br>V. Mutilin, Russia | | | | 983<br>33 000 s | **2 736**<br>11 000 s | | |
| **CASCADE**<br>W. Wang, USA | | 52<br>16 000 s | | 537<br>43 000 s | | | 70<br>6 000 s |
| **CBMC**<br>M. Tautschnig, UK | -134<br>2 500 s | **68**<br>1 800 s | **1 039**<br>78 000 s | 158<br>570 000 s | 2 293<br>380 000 s | **129**<br>15 000 s | **100**<br>13 000 s |
| **CPACHECKER**<br>M. Dangl, Germany | 2<br>62 s | **58**<br>870 s | 0<br>0 s | **2 317**<br>47 000 s | **2 572**<br>39 000 s | **78**<br>5 000 s | 96<br>930 s |
| **CPAREC**<br>M.-H. Tsai, Taiwan | | | | | | | |
| **ESBMC**<br>J. Morse, UK | -206<br>5.5 s | **69**<br>470 s | 1 014<br>13 000 s | **1 968**<br>59 000 s | 2 281<br>36 000 s | **-12**<br>5 300 s | 79<br>37 s |
| **FOREST**<br>P. Sanchez, Spain | | | | | | | |
| **FORESTER**<br>O. Lengál, Czechia | | | | | | | 32<br>1.8 s |
| **FUNCTION**<br>C. Urban, France | | | | | | | |
| **HIPTNT+**<br>T.-C. Le, Singapore | | | | | | | |
| **LAZY-CSEQ**<br>G. Parlato, UK | | | **1 222**<br>5 600 s | | | | |
| **MAP2CHECK**<br>H. O. Rocha, Brazil | | | | | | | |
| **MU-CSEQ**<br>B. Fischer, ZA | | | **1 222**<br>16 000 s | | | | |
| **PERENTIE**<br>F. Cassez, Australia | | | | | | | |
| **PREDATORHP**<br>T. Vojnar, Czechia | | | | | | | **111**<br>140 s |
| **SEAHORN**<br>A. Gurfinkel, USA | 0<br>0.61 s | -80<br>550 s | -8 973<br>42 s | **2 169**<br>30 000 s | **2 657**<br>16 000 s | -164<br>5.9 s | -37<br>14 s |
| **SMACK+CORRAL**<br>Z. Rakamarić, USA | **48**<br>400 s | | | 1 691<br>78 000 s | 2 507<br>72 000 s | | **109**<br>820 s |
| **ULTIAUTOMIZER**<br>M. Heizmann, Germany | **2**<br>6.4 s | 5<br>170 s | | 1 887<br>54 000 s | 274<br>850 s | | 84<br>460 s |
| **ULTIKOJAK**<br>A. Nutz, Germany | **2**<br>5.9 s | -62<br>120 s | | 872<br>10 000 s | 82<br>270 s | | 84<br>420 s |
| **UNB-LAZY-CSEQ**<br>S. La Torre, Italy | | | 984<br>36 000 s | | | | |

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

| **Verifier**<br>Repr. jury member | **MemSafety**<br>361 points<br>205 tasks | **Recursive**<br>40 points<br>24 tasks | **Sequential**<br>364 points<br>261 tasks | **Simple**<br>68 points<br>46 tasks | **Termination**<br>742 points<br>393 tasks | **Overall**<br>9 562 points<br>5 803 tasks |
|---|---|---|---|---|---|---|
| **APROVE**<br>T. Ströder, Germany | | | | | **610**<br>5 400 s | |
| **BEAGLE**<br>D. Wang, China | | 6<br>22 s | | | | |
| **BLAST**<br>V. Mutilin, Russia | | | | 32<br>4 200 s | | |
| **CASCADE**<br>W. Wang, USA | **200**<br>82 000 s | | | | | |
| **CBMC**<br>M. Tautschnig, UK | -433<br>14 000 s | 0<br>10 000 s | -171<br>39 000 s | 51<br>16 000 s | | **1 731**<br>1 100 000 s |
| **CPACHECKER**<br>M. Dangl, Germany | **326**<br>5 700 s | 16<br>31 s | **130**<br>11 000 s | **54**<br>4 000 s | 0<br>0 s | **4 889**<br>110 000 s |
| **CPAREC**<br>M.-H. Tsai, Taiwan | | **18**<br>140 s | | | | |
| **ESBMC**<br>J. Morse, UK | | | **193**<br>9 600 s | 29<br>990 s | | -2 161<br>130 000 s |
| **FOREST**<br>P. Sanchez, Spain | | | | | | |
| **FORESTER**<br>O. Lengál, Czechia | 22<br>25 s | | | | | |
| **FUNCTION**<br>C. Urban, France | | | | | 350<br>61 s | |
| **HIPTNT+**<br>T.-C. Le, Singapore | | | | | **545**<br>300 s | |
| **LAZY-CSEQ**<br>G. Parlato, UK | | | | | | |
| **MAP2CHECK**<br>H. O. Rocha, Brazil | 28<br>2 100 s | | | | | |
| **MU-CSEQ**<br>B. Fischer, ZA | | | | | | |
| **PERENTIE**<br>F. Cassez, Australia | | | | | | |
| **PREDATORHP**<br>T. Vojnar, Czechia | **221**<br>460 s | | | | | |
| **SEAHORN**<br>A. Gurfinkel, USA | 0<br>0 s | -88<br>2.3 s | -59<br>5 800 s | **65**<br>1 400 s | 0<br>0 s | -6 228<br>53 000 s |
| **SMACK+CORRAL**<br>Z. Rakamarić, USA | | **27**<br>2 300 s | | **51**<br>5 100 s | | |
| **ULTIAUTOMIZER**<br>M. Heizmann, Germany | 95<br>13 000 s | **25**<br>310 s | **15**<br>8 600 s | 0<br>1 800 s | **565**<br>8 600 s | **2 301**<br>87 000 s |
| **ULTIKOJAK**<br>A. Nutz, Germany | 66<br>4 800 s | 10<br>220 s | -10<br>7 000 s | 3<br>140 s | | 231<br>23 000 s |
| **UNB-LAZY-CSEQ**<br>S. La Torre, Italy | | | | | | |

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

| Rank | Candidate | Score | CPU Time | Solved Tasks | False Alarms | Wrong Proofs |
|---|---|---|---|---|---|---|
| *Arrays* | | | | | | |
| 1 | **SMACK+CORRAL** | **48** | 400 | 51 | 7 | **1** |
| 2 | ULTIKOJAK | 2 | 5.9 | 1 | | |
| 3 | ULTIAUTOMIZER | 2 | 6.4 | 1 | | |
| *BitVectors* | | | | | | |
| 1 | **ESBMC** | **69** | 470 | 45 | | **1** |
| 2 | CBMC | 68 | 1 800 | 44 | | **1** |
| 3 | CPACHECKER | 58 | 870 | 40 | | **1** |
| *Concurrency* | | | | | | |
| 1 | **LAZY-CSEQ** | **1 222** | 5 600 | 1003 | | |
| 2 | MU-CSEQ | 1 222 | 16 000 | 1003 | | |
| 3 | CBMC | 1 039 | 78 000 | 848 | 1 | **1** |
| *ControlFlow* | | | | | | |
| 1 | **CPACHECKER** | **2 317** | 47 000 | 1302 | 2 | **2** |
| 2 | SEAHORN | 2 169 | 30 000 | 1014 | 5 | **2** |
| 3 | ESBMC | 1 968 | 59 000 | 1212 | | **36** |
| *DeviceDrivers64* | | | | | | |
| 1 | **BLAST** | **2 736** | 11 000 | 1 481 | 5 | **9** |
| 2 | SEAHORN | 2 657 | 16 000 | 1 440 | 3 | **12** |
| 3 | CPACHECKER | 2 572 | 39 000 | 1 390 | 17 | **4** |
| *Floats* | | | | | | |
| 1 | **CBMC** | **129** | 15 000 | 74 | | |
| 2 | CPACHECKER | 78 | 5 100 | 54 | 2 | |
| 3 | ESBMC | −12 | 5 300 | 27 | 7 | **2** |
| *HeapManipulation* | | | | | | |
| 1 | **PREDATORHP** | **111** | 140 | 68 | | |
| 2 | SMACK+CORRAL | 109 | 820 | 76 | 3 | |
| 3 | CBMC | 100 | 13 000 | 69 | | **2** |
| *MemorySafety* | | | | | | |
| 1 | **CPACHECKER** | **326** | 5 700 | 199 | 4 | |
| 2 | PREDATORHP | 221 | 460 | 134 | 1 | |
| 3 | CASCADE | 200 | 82 000 | 154 | 2 | **5** |
| *Recursive* | | | | | | |
| 1 | **SMACK+CORRAL** | **27** | 2 300 | 23 | | **1** |
| 2 | ULTIAUTOMIZER | 25 | 310 | 16 | | |
| 3 | CPAREC | 18 | 140 | 12 | | |
| *SequentializedConcurrency* | | | | | | |
| 1 | **ESBMC** | **193** | 9 600 | 144 | 2 | |
| 2 | CPACHECKER | 130 | 11 000 | 113 | 1 | |
| 3 | ULTIAUTOMIZER | 15 | 8 600 | 51 | 9 | |
| *Simple* | | | | | | |
| 1 | **SEAHORN** | **65** | 1 400 | 44 | | |
| 2 | CPACHECKER | 54 | 4 000 | 32 | | |
| 3 | SMACK+CORRAL | 51 | 5 100 | 43 | 2 | |
| *Termination* | | | | | | |
| 1 | **APROVE** | **610** | 5 400 | 305 | | |
| 2 | ULTIAUTOMIZER | 565 | 8 600 | 304 | 1 | |
| 3 | HIPTNT+ | 545 | 300 | 290 | | |
| *Overall* | | | | | | |
| 1 | **CPACHECKER** | **4 889** | 110 000 | 3 211 | 29 | **7** |
| 2 | ULTIAUTOMIZER | 2 301 | 87 000 | 1 453 | 21 | **3** |
| 3 | CBMC | 1 731 | 1 100 000 | 4 056 | 77 | **453** |

One complete competition run (each candidate on all selected categories according to the opt-outs) consisted of 49 855 verification runs and 4 151 witness checks. The consumed total CPU time for one competition run required a total of 119 days of CPU time for the verifiers and 1 day for the witness checker. Each tool was executed at least twice, in order to make sure the results are accurate and not contradicting in any sense. Not counted in the above measures on the dedicated competition machines are the preparation runs that were required to find out if the verifiers are successfully installed and running. Other machines with a slightly different specification were used for those test runs while the eight dedicated machines were occupied by the official competition runs.

**Quantitative Results.** Tables 5 and 6 present a quantitative overview over all tools and all categories (Forest and Perentie participated only in subcategory *Loops*). The format of the table is similar to those of previous SV-COMP editions [3]: 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. 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 *Overall* and *ControlFlow* (consisting of several sub-categories) were computed using normalized scores as defined in the report for SV-COMP'13 [2].

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 tool reported wrong results: reporting an error 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.** Score-based quantile functions [2] are helpful for visualizing results of comparative evaluations. The competition web page [9] includes such a plot for each category; Fig. 1 illustrates only the category *Overall* (all verification tasks). Six verifiers participated in category *Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [2]).

*Overall Quality Measured in Scores (Right End of Graph).* CPAchecker 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 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 ($x$-coordinate), i.e., amount of incorrect and misleading verification work. Verifiers should start with a score close to zero; UltiAutomizer and CPAchecker are best in this aspect (also the right-most columns of category *Overall* in Table 7 report this: only 21 and 29 false alarms, respectively, and only 3 and 7 wrong proofs, for a total of 5 803 verification tasks).

[9] `http://sv-comp.sosy-lab.org/2015/results`

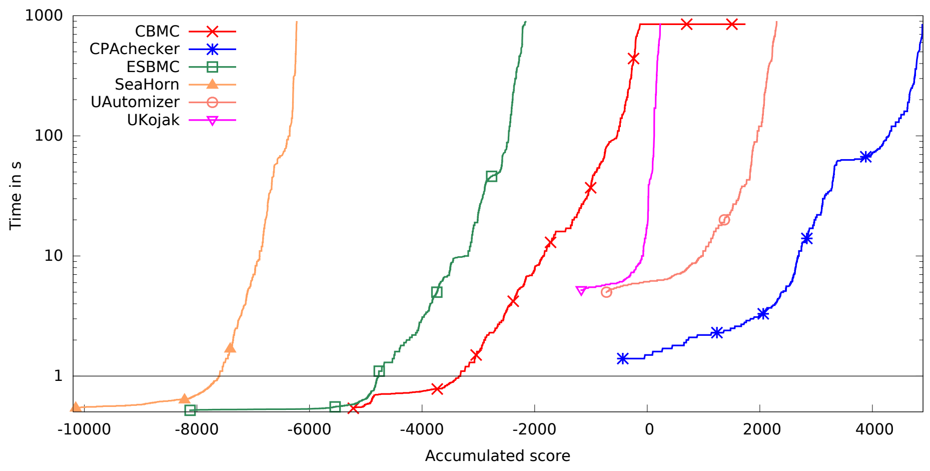

**Fig. 1.** Quantile functions for category *Overall.* We plot all data points $(x, y)$ such that the maximum run time of the $n$ fastest correct verification runs is $y$ and $x$ is the accumulated score of all incorrect results and those $n$ correct results. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s.

*Amount of Correct Verification Work (Length of Graph).* The length of the graph indicates the amount of correct results: for example, Cbmc and Esbmc both produce a large amount of correct results.

*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, UltiAutomizer, and UltiKojak). (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 right-most graphs and those closest to the $x$-axis. (4) Also the shape of the graph can give interesting insights: From Cbmc's horizontal line just below the time limit at 850 s, we can see that this bounded model checker returns a result just before the time limit is reached. The quantile plot for CPAchecker shows an interesting bend at 60 s of run time, where the verifier suddenly switches gears: it gives up with one strategy (without abstraction) performs an internal restart and proceeds using another strategy (with abstraction and CEGAR-based refinement).

**Robustness, Soundness, and Completeness.** The best tools of each category show that state-of-the-art verification technology is quite advanced already: Table 7 (last two columns) reports a low number of wrong verification results, with a few exceptions. Cbmc and Esbmc are the two verifiers that produce the most wrong safety proofs (missed bugs): both of them are bounded model checkers. In three categories, the top-three verifiers did not report any wrong proof.

**Verifiable Witnesses.** One of the objectives of program verification is to provide a witness for the verification result. This was an open problem of verification technology: there was no commonly supported witness format yet, and the verifiers were not producing accurate witnesses that could be automatically assessed for validity. SV-COMP 2015 changed this (restricted to error witnesses for now): all verifiers that participated in categories that required witness validation supported the common exchange format for error witnesses, and produced error paths in that format. We used a witness checker to validate the obtained error paths.

## 5 Conclusion

The 4th edition of the Competition on Software Verification was successful in several respects: (1) We introduced *verifiable witnesses* (the verifiers produced error paths in a common exchange format, which made it possible to validate a given error path by using a separate witness checker). (2) We had a record number of *22 participating verification tools* from 13 countries. (3) The repository of verification tasks was extended by two new categories: *Arrays and Floats.* (4) The properties to be verified were extended by a liveness property: *Termination.* (5) The total number of verification tasks in the competition run was doubled (compared to SV-COMP'14) to a total of *5 803 verification tasks.* Besides the above-mentioned success measures, SV-COMP serves as a yearly overview of the state of the art in software verification, and witnesses an enormous pace of development of new theory, data structures and algorithms, and tool implementations that analyze real C code. As in previous years, 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.

**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. D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.
2. D. Beyer. Second competition on software verification. In *Proc. TACAS*, LNCS 7795, pages 594–609. Springer, 2013.
3. D. Beyer. Status report on software verification. In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.
4. D. Beyer, S. Löwe, and P. Wendler. Benchmarking and resource measurement. Unpublished manuscript, 2015.
5. D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS 7976, pages 1–17. Springer, 2013.

6. F. Cassez, T. Matsuoka, E. Pierzchalski, and N. Smyth. Perentie: Modular trace refinement and selective value tracking (competition contribution). In *Proc. TACAS*. Springer, 2015.
7. Y.-F. Chen, C. Hsieh, M.-H. Tsai, B.-Y. Wang, and F. Wang. CPArec: Verifying recursive programs via source-to-source program transformation (competition contribution). In *Proc. TACAS*. Springer, 2015.
8. M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic. In *Proc. TACAS*. Springer, 2015.
9. P. G. de Aledo and P. Sanchez. Framework for embedded system verification (competition contribution). In *Proc. TACAS*. Springer, 2015.
10. A. Gurfinkel, T. Kahsai, and J. A. Navas. SeaHorn: A framework for verifying C programs (competition contribution). In *Proc. TACAS*. Springer, 2015.
11. A. Haran, M. Carter, M. Emmi, A. Lal, S. Qadeer, and Z. Rakamarić. SMACK+Corral: A modular verifier (competition contribution). In *Proc. TACAS*. Springer, 2015.
12. M. Heizmann, D. Dietsch, J. Leike, B. Musa, and A. Podelski. Ultimate Automizer with array interpolation. In *Proc. TACAS*. Springer, 2015.
13. L. Holik, M. Hruska, O. Lengal, A. Rogalewicz, J. Simacek, and T. Vojnar. Forester: Shape analysis using tree automata. In *Proc. TACAS*. Springer, 2015.
14. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A lazy sequentialization tool for C (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 398–401. Springer, 2014.
15. D. Kröning and M. Tautschnig. CBMC: C bounded model checker (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 389–391. Springer, 2014.
16. T. C. Le, S. Qin, and W. Chin. Termination and non-termination specification inference. In *Proc. PLDI, to appear*. ACM, 2015.
17. J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer. ESBMC 1.22 (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 405–407. Springer, 2014.
18. P. Muller, P. Peringer, and T. Vojnar. Predator hunting party (competition contribution). In *Proc. TACAS*. Springer, 2015.
19. T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Unbounded Lazy-CSeq: A lazy sequentialization tool for C programs with unbounded context switches (competition contribution). In *Proc. TACAS*. Springer, 2015.
20. A. Nutz, D. Dietsch, M. M. Mohamed, and A. Podelski. Ultimate Kojak with memory safety checks (competition contribution). In *Proc. TACAS*. Springer, 2015.
21. H. Rocha, R. S. Barreto, L. Cordeiro, and A. D. Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Proc. IFM*, LNCS 7321, pages 128–142. Springer, 2012.
22. P. Shved, M. Mandrykin, and V. Mutilin. Predicate analysis with Blast 2.7 (competition contribution). In *Proc. TACAS*. Springer, 2012.
23. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination and memory safety of C programs (competition contribution). In *Proc. TACAS*. Springer, 2015.
24. E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. MU-CSeq 0.3: Sequentialization by read-implicit and coarse-grained memory unwindings (competition contribution). In *Proc. TACAS*. Springer, 2015.
25. C. Urban. FuncTion: An abstract domain functor for termination (competition contribution). In *Proc. TACAS*. Springer, 2015.
26. W. Wang and C. Barrett. Cascade (competition contribution). In *Proc. TACAS*. Springer, 2015.