Competition on Software Verification*
(SV-COMP)

Dirk Beyer

University of Passau, Germany

Abstract. This report describes the definitions, rules, setup, procedure, and results of the 1st International Competition on Software Verification. The verification community has performed competitions in various areas in the past, and SV-COMP'12 is the first competition of verification tools that take software programs as input and run a fully automatic verification of a given safety property. This year's competition is organized as a satellite event of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS).

1 Introduction

The area of verification, in particular model checking, has grown to an own major research area within computer science, which is witnessed and acknowledged by a recent ACM Turing Award in the area and the growth of conferences in the field of verification to some of the top computer-science conferences with high impact on the research community. Model checking started to get adopted in software industry (e.g., Microsoft, NASA, NEC) about ten years ago, and major tool-development projects in software model checking began around that time (Blast at UC Berkeley, Slam at MSR, Magic at CMU).

Several new and powerful software-verification tools became available, but they have not been compared systematically in the past. The reason for this is that no widely distributed benchmark suite was available and most concepts were only validated in research prototypes. This can be changed by a competition. Comparison, and thus competition, is a driving force for the invention of new methods, technologies, and tools. This article describes the competition of software-verification tools, which this year is organized as a satellite event of the conference TACAS. SV-COMP'12 is the first competition that compares verification tools for software source code.

Only few research projects aim at producing stable tools that can be used by people outside the respective development groups, and the development of such tools is not continuous. PhD students and post-docs do not adequately benefit from tool development because theoretical papers are still considered more relevant than papers that present technical contributions, like tool papers. Through its visibility, the Competition on Software Verification changes this, by showing

* `http://sv-comp.sosy-lab.org`

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 504–524, 2012.

off the latest implementation of the research results in our community, and giving credits and benefits to researchers and students who spend considerable amounts of time implementing verification algorithms in practical software packages (winning the verification competition in a category serves as acknowledgment). More discussion on problems and barriers in developing tools for software verification can be found in a position paper by Alglave et al. [1].

A competition event fosters the transfer of theoretical and conceptual advancements in software verification into practical tools. The main goals of the Competition on Software Verification are the following:

- Establish a set of benchmarks for software verification in the community, i.e., create and maintain a set of programs together with explicit properties to check, and make those publicly available for researchers to be used in performance comparisons when evaluating a new technique.
- Provide an overview of the state-of-the-art in software verification for the community, i.e., compare, independently from particular paper projects and specific techniques, different verification tools in terms of precision and performance.
- Increase the visibility and credits that tool developers receive, i.e., provide a forum for presentation of tools and discussion of the latest technologies, and give students the opportunity to publish about the implementation work that they have done.

Related Events. Competitions are widely acknowledged as a means to improve the available tools, the visibility of their strengths, and to establish a publicly available set of benchmark problems. In the formal-methods community (loosely interpreted), there are competitions on, e.g., SAT [1], SMT [2], Planning [3], QBF [4], HWMC [5], and Theorem Proving [6]. These events seem to have a positive impact on the development speed and the quality of the participating software tools; theoretical results are transferred to practical tools almost instantly.

2 Procedure and Schedule

The competition compared state-of-the-art software verifiers with respect to effectiveness and efficiency. The overall process was composed of several phases, as described in the following.

Announcement and Benchmark Submission. The competition was publicly announced on July 19, 2011 at the conference event CAV. During the preparation phase, calls for contributions were made in various mailings, the web page was set

[1] http://www.satcompetition.org
[2] http://www.smtcomp.org
[3] http://ipc.icaps-conference.org
[4] http://www.qbflib.org/competition.html
[5] http://fmv.jku.at/hwmcc11
[6] http://www.cs.miami.edu/~tptp/CASC

up, and benchmark verification tasks were collected and classified into competition categories. Since this was the first competition, all contributed benchmarks were initially accepted, and we only disqualified benchmark programs (after discussion) if they violated the requirements below.

Training Phase. The set of all benchmark verification tasks was finalized and made publicly available on September 14, 2011. During the training phase, the teams of the competition candidates were able to download the benchmarks in order to train their tools on the given verification tasks. At the end of this phase, the competition contributions (consisting of the software together with a three-page description of the competition candidate) were submitted. Also during the training phase, some benchmark programs were corrected (without changing the verification outcome), and some verification tasks were disqualified (by the rules below and after community discussion) and removed from the benchmark set.

Benchmark Evaluation Phase. The submission of competition contributions ended on October 14, 2011; all competition candidates were downloaded and installed on a competition machine, and the verification tools were applied to the sets of benchmark verification tasks. All submitted artifacts of the competition contribution (tool description and software archive files) were stamped with SHA hash values. The hash values were sent to all members of the program committee (= jury) of the competition, in order to eliminate the possibility of undue advantages of any tool.

Also in this phase, all descriptions of competition candidates (the three-page summary papers) were reviewed, each by several members of the program committee, in order to ensure the quality standards of the TACAS proceedings.

Approval of Verification Results. After the results were obtained on a competition machine [7] (the number of solved instances and the run time were measured), each participating team received the (preliminary) results that were obtained using their submitted competition candidate. This step gave the jury the opportunity to discuss some unexpected results with the corresponding authors of the competing tools. This approval phase was completed by December 9, 2011. By this time, a list of all participating teams was publicly announced.

Notification. On December 16, the notification of acceptance of the competition contribution, together with the reviews, were sent to all authors. All teams were informed of the results of all competition candidates, and tables with rankings were made available to all teams.

[7] One complete competition run of all candidates on all verification tasks required a total of 163 hours of non-stop machine time; several such competition runs were necessary.

3 Definitions and Rules

This section presents the definitions and rules that regulated the execution of the competition and how the results were evaluated towards a ranking.

Definition of Verification Task. A verification task consists of a C program and a safety property. For simplicity, the safety properties to be verified are reduced to reachability problems and encoded in the program source code (using the error label 'ERROR'). In other words, the competition candidate is asked, given a C program and the error label 'ERROR', whether there is a concrete execution path through the program such that the error label can be reached. A verification run is a non-interactive execution of a competition candidate on a single verification task. The result of a verification run is either

SAFE: there is no path that reaches the error location,
UNSAFE + Path: there exists a path that reaches the error location, or
UNKNOWN: the competition candidate does not succeed in computing an answer 'SAFE' or 'UNSAFE'.

There is no particular fixed format for the error path. The error path has to be written to a file or on stdout in a reasonable format to make it possible to manually check validity.

Benchmark Verification Tasks. All verification tasks were provided by the specified date on the competition web site[8]. Most programs were provided in CIL (C Intermediate Language). The programs were assumed to be written in GNU C (most of them adhere to ANSI C).

Potential competition participants were invited to submit benchmark verification tasks until the specified date. Programs had to fulfill two requirements to be eligible for the competition: (1) the program has to be written in GNU C or ANSI C, and can be successfully CIL-pre-processed[9] with the parameters `--dosimplify --printCilAsIs --save-temps --domakeCFG --no-convert-field-offsets --no-convert-direct-calls`, and (2) the property is instrumented into the program and is violated if the label 'ERROR' is reached.

As a further convention, a verification tool can assume that a function call `__VERIFIER_assume(expression)` has the following meaning: If `expression` is evaluated to '0', then the function loops forever, otherwise the function returns (no side effects). The verification tool can assume the following implementation:

```
void __VERIFIER_assume(int expression) {
  if (!expression) { LOOP: goto LOOP; }
  return;
}
```

[8] `http://sv-comp.sosy-lab.org`
[9] We used CIL version 1.3.7, from `http://cil.sourceforge.net`, with extensions.

Similarly, the following functions can be assumed to return an arbitrary value of the indicated type: `__VERIFIER_nondet_X()` (and `nondet_X()`, deprecated) with X being one of `int`, `float`, `char`, `short`, or `pointer` (no side effects, `pointer` refers to `void *`). The verification tool can assume that the functions are implemented according to the following template:

```
X __VERIFIER_nondet_X() {
  X val;
  return val;
}
```

Setup. The verification runs of the competition were (natively) executed on a dedicated unloaded compute server with a 3.4 GHz 64-bit Quad Core CPU (Intel i7-2600K) and a GNU/Linux operating system (x86_64-linux). The machine had 16 GB of RAM, of which exactly 15 GB were made available to the competition candidate. Every verification run had a run-time limit of 15 min. The run time was measured in seconds of CPU time.

The verification runs were started by a batch script that collects statistics and interprets the result of every competition candidate on every verification task as one of the following categories of verification results: SAFE (verifier states that the property holds), UNSAFE (verifier states that the property does not hold, an error path is reported), UNKNOWN (result does not fall into the other two categories: verification result not known, resources exhausted, verifier crashed).

Qualification. A verification tool was qualified to participate as competition candidate if the tool was publicly available (for the GNU/Linux platform, more specifically, it had to run on an x86_64 machine) and succeeded in more than 50 % of all training verification tasks to parse the input and start the verification process (a tool crash during the verification phase does not disqualify). A person (participant) was qualified as competition contributor for a competition candidate if the person was a contributing designer/developer of the submitted competition candidate (witnessed by occurrence of the person's name on the tool's project web page, a tool paper, or in the revision logs). A contribution paper was qualified if the quality of the description of the competition candidate sufficed to run the tool in the competition and was appropriate as competition-candidate representation for the TACAS proceedings.

A verification tool could participate several times as an independent competition candidate, if a significant difference of the conceptual or technological basis of the implementation is justified in the accompanying description paper. This applies to different versions as well as different configurations, in order to avoid forcing developers to create a new tool name for every new concept. Competition candidates were allowed to opt-out from certain categories.

Evaluation by Scores and Run Time. The scores were assigned according to the scoring schema in Table 1. Every verification task comes with an expected result, which was provided by the contributor of the verification task. The

Table 1. Scoring schema

Reported result	Points	Description
UNKNOWN	0	Failure to compute verification result, out of resources, program crash.
UNSAFE correct	+1	The error in the program was found and an error path was reported.
UNSAFE incorrect	−2	An error is reported for a program that fulfills the property (false alarm, imprecise analysis).
SAFE correct	+2	The program was analyzed to be free of errors.
SAFE incorrect	−4	The program had an error but the competition candidate did not find it (missed bug, unsound analysis).

interpretation of 'UNSAFE' is that a verification tool is supposed to find a path to the error label. The interpretation of 'SAFE' is that no executable path to the error label exists in the program, assuming the C semantics [2] and a standard POSIX run-time environment. The results of type 'SAFE' yield higher absolute score values compared to type 'UNSAFE', because it is expected to be heuristically easier to detect errors than it is to prove correctness. The absolute score values for incorrect results are higher compared to correct results, because a single correct answer should not be able to compensate for a wrong answer. This scoring schema ensures a disadvantage for (hypothetical) competition candidates that always return the same result or random results.

The participating competition candidates are ranked according to the sum of points. Competition candidates with the same sum of points are sub-ranked according to success run time. The success run time for a competition candidate is the total CPU time over all verification tasks for which the competition candidate reported a correct verification result.

The participants had the opportunity to check the verification results against their own expected results and discuss inconsistencies with the competition chair (cf. Sect. 2). A candidate that opted out from a category or obtained a negative total score in a category, was assigned zero points in that category as total score.

To ensure that no undue advantage occurs from knowing the benchmark programs beforehand, we obfuscated all benchmark programs (by renaming all variable and function names, as well as the file name) and ran the competition candidates on the obfuscated versions of the benchmark programs. All verification results obtained using obfuscated versions matched the verification results of the corresponding original program.

Publication and Presentation of the Competition Candidates. A description of every qualified competition candidate (contribution paper) was published in the LNCS proceedings of TACAS 2012. In addition, every qualified competition candidate was granted a demonstration slot in the TACAS program to present the competition candidate to the TACAS audience.

Competition Jury. The program committee that oversees the process of the competition consists of one member of each participating team. The tasks of this committee are to review the competition contribution papers and help the organizer to resolve any disputes that might occur. Deviation from the competition rules need to be approved by the committee. The 2012 competition jury consists of the following members:

Dirk Beyer, University of Passau, Germany (Chair)
Bernd Fischer, University of Southampton, UK
Vadim Mutilin, Russian Academy of Sciences, Russia
Andrey Rybalchenko, TU Munich, Germany
Carsten Sinz, Karlsruhe Institute of Technology, Germany
Michael Tautschnig, University of Oxford, UK
Helmut Veith, TU Vienna, Austria
Tomas Vojnar, Brno University of Technology, Czech Republic
Georg Weissenbacher, Princeton University, USA
Philipp Wendler, University of Passau, Germany
Daniel Wonisch, University of Paderborn, Germany

The term of the jury is one year, and the next jury consists of the chair and one member of each participating team of the next competition.

4 Benchmark Verification Tasks

All verification tasks are available for browsing and download via the public SVN repository for the Competition on Software Verification [10]. The competition was organized in several categories of benchmark verification tasks, which are explained in the following.

The benchmark verification tasks were contributed by several research and development groups. After the submission deadline for benchmarks, a group of people (organizer and participants) were working on improving the quality of the verification tasks. This means that after the benchmark sets were made public, some programs were removed (not qualified, no property encoded, unknown architecture), and some programs were technically improved (CIL simplifications, compiler warnings, memory model). These changes have improved the overall quality of the final set of verification tasks for the competition, and have not changed the intended verification result; all changes are tracked in the public repository.

The expected verification result is encoded in the file name of each verification task: the sub-strings 'BUG' and 'unsafe' indicate that the program violates the property, i.e., the error label is reachable.

[10] `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp12`

Control Flow and Integer Variables. The first set of verification tasks consists of the programs in the set ControlFlowInteger:

```
ntdrivers-simplified/*_BUG.cil.c
ntdrivers-simplified/*[!G].cil.c
ntdrivers/*.BUG.i.cil.c
ntdrivers/*[!G].i.cil.c
ssh-simplified/*_BUG.cil.c
ssh-simplified/*[!G].cil.c
ssh/*.BUG.i.cil.c
ssh/*[!G].i.cil.c
locks/*.BUG.c
locks/*[!G].c
```

The programs and properties in this category use problems that relate mostly to control-flow structure and integer variables. There is no particular focus on pointers, data structures, and concurrency. The verification tasks were taken from the source-code repositories of the tools Blast [6] and CPAchecker [9].

The directories 'ntdrivers*' contain 19 verification tasks that were derived from (parts of) device drivers of the Windows NT kernel. The directories 'ssh*' contain 61 verification tasks `s3_clnt*` and `s3_srvr*`, which represent the subroutine for the connection handshake protocol (a state machine) of the SSH client and server. The different versions represent various protocol-specific safety properties (one program for each property). The directories with the suffix 'simplified' contain versions of the drivers and SSH programs that were manually pre-processed in order to remove heap access. The verification tasks with the suffix 'BUG' have artificial bugs injected, which cause the assertions to fail. The 13 verification tasks in directory 'locks' were taken from the CPAchecker project, where they served the purpose of demonstrating the advantage of adjustable-block encoding [5, 10].

Linux Device Drivers 32-bit. This category consists of problems that require the analysis of pointer aliases and function pointers (32-bit machine model):

```
ldv-regression/*-unsafe*.cil.c
ldv-regression/*-safe*.cil.c
ddv-machzwd/*_BUG.cil.c
ddv-machzwd/*[!G].cil.c
```

The 46 verification tasks in directory 'ldv-regression' were contributed by the Linux Driver Verification (LDV) project [11]. The verification tasks are used in the LDV project as regression tests for Blast and CPAchecker. The benchmark set consists of small programs that check for features rather than imposing a high verification load; some of these tests are inspired by the problem patterns that were seen in real device-driver code.

The 13 verification tasks in the directory 'ddv-machzwd' were generated using DDVerify [30]. The main file `ddv_machzwd_all` contains several assertions. Then,

[11] `http://linuxtesting.org/project/ldv`

there is one separate file for each assertion in file `ddv_machzwd_all`; the file names of these separate files have a suffix that indicates the name of the function in which the assertion occurs.

Linux Device Drivers 64-bit. This category consists of problems that require the analysis of pointer aliases and function pointers (64-bit machine model):

```
ldv-drivers/*-unsafe*.cil.c
ldv-drivers/*-safe*.cil.c
```

The verification tasks in this category were contributed by the LDV project. The directory contains 41 recent (Sept. 2011) driver-verification tasks that were taken directly from the x86_64 Linux kernel. Among them are 16 programs with bugs, which are accompanied by sample error traces. Some of these are confirmed bugs that were reported by the LDV project to the kernel developers.

Heap Manipulation. The problems in this category require the analysis of data structures on the heap and consist of the programs in the set HeapManipulation:

```
heap-manipulation/*BUG.cil.c
heap-manipulation/*[!G].cil.c
list-properties/*.cil.c
```

The eight verification tasks in directory 'heap-manipulation' were provided by the Predator project [12]. The program `bubble_sort_linux` is a bubble-sort implementation that operates on Linux lists. Verification tasks with the suffix 'BUG' have an artificial bug injected. The program `dll_of_dll` operates on a NULL-terminated doubly-linked list of doubly-linked lists. The program creates a doubly-linked list of doubly-linked lists, and then destroys the data structure in several phases. The program `merge_sort` is an implementation of the merge-sort algorithm that operates on two-level singly-linked lists. The program `sll_to_dll_rev` converts a singly-linked list to a doubly-linked list, then reverses the list, and coverts it back to a singly-linked list.

The six verification tasks in the directory 'list-properties' are taken from a supplementary web page of the Blast 3.0 project [7]. This set contains several C programs that manipulate list data structures containing integers as data elements. The programs `simple` and `simple_built_from_end` both create a list that represents a sequence of integers that matches 1*0 (regular expression), i.e., an arbitrary number of list elements that are initialized with the data value 1 with the last element initialized with 0. Then, the programs traverse the list to check that every element is set to 1 and the last to 0. The difference between the two programs is the order in which the list elements are created. The program `list` creates a sequence that matches 1*2*3. The program `list_flag` creates a sequence that matches c*3, where c is a constant determined by a flag. Then, the program traverses the list to check that the integers occur in the correct order.

[12] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator`

The program `alternating` is similar to `list` except that the list begins with alternating 1s and 2s, and ends with a value 3, i.e., it creates a sequence that matches (12)*3. The program `splice` first builds the same list as `alternating`. Then, the list is split into two different lists: the first list contains the nodes at odd positions and the second list contains nodes at even positions of the original list, without the last value 3. Each new list is then traversed to check that all its elements have the same data value.

SystemC. This category contains SystemC-related problems:

```
systemc/*BUG.cil.c
systemc/*[!G].cil.c
```

This set of 62 verification tasks was provided by the SyCMC project [14]. The programs were transformed to sequential C programs by incorporating the scheduler into the C code. More details can be found in the research article that defines the benchmark [14].

Concurrency. Some concurrency problems are contained in this set:

```
pthread/*BUG.cil.c
pthread/*[!G].cil.c
```

This benchmark set of eight verification tasks was contributed by the ESBMC project [13]. The program `fib_bench` starts two threads, which are together computing a Fibonacci number, and then compares if the results of the two threads are smaller than an upper bound. The program `fib_bench_BUG` is a version which checks a wrong bound and thus is expected to yield an error. The programs `fib_bench_longer` and `fib_bench_longer_BUG` are using the same algorithm but a larger number of iterations.

The programs `queue_ok` and `queue_BUG` operate on a queue data structure, where the former is expected to work correctly and the second to reach the error label. Two threads are started, one trying to write to the queue and one trying to read from the queue, after acquiring a mutex lock, respectively, while the programs check for some properties to hold.

The program `reorder_5_BUG` lets a set of threads write values to two variables and another set of threads verify that the two values are either both untouched or both changed to the new values. Due to certain interleavings in the execution, a violation of the property is possible. The program `twostage_3_BUG` creates two sets of threads. One set of threads is writing a value to one global variable and an increased value to a second variable. The other set of threads verifies the success of the first set of threads. Again, due to certain interleavings, the property might be violated in some executions.

Overall. The category 'Overall' consists of the union of all above-mentioned sets of verification tasks.

[13] `http://esbmc.org`

5 Participating Teams

In the following, we briefly introduce the competition candidates, listed in alphabetical order. Table 2 gives an overview of the participating candidates. The top-three placements achieved in the competition for each category are given in the paragraph for the corresponding tool. The detailed summary of the results is presented in Sect. 6.

Table 3 provides an overview of the technologies and concepts used by the various competition candidates. The technique of counterexample-guided abstraction refinement (CEGAR) [15] is used by the majority of tools. Other techniques that are offered by the competition candidates are predicate abstraction [3, 19], bounded model checking [12], shape analysis [23], construction of an abstract reachability tree (ART) as proof of correctness [6], lazy abstraction [21], and Craig interpolation for discovering new predicates to refine a predicate analysis [17, 25]. Only three tools provide verification of concurrent programs.

Blast 2.7 [26], submitted by *Pavel Shved, Vadim Mutilin, and Mikhail Mandrykin* (Institute for System Programming of the Russian Academy of Sciences, Russia), has achieved the following placements:

- *Winner* in DeviceDrivers64
- *Bronze* in DeviceDrivers

Blast 2.7[14] is a model checker that is based on predicate abstraction, with a focus on verifying control-flow intensive programs such as device drivers and system programs. It is based on the CEGAR algorithm [15] and uses Craig interpolation [17] on infeasible error paths to discover new predicates for increasing the precision of the predicate abstraction. The tool was originally developed at the University of California at Berkeley and at EPFL Lausanne [6], but later significantly improved by the Linux Driver Verification group at the Institute for System Programming of the Russian Academy of Sciences in Moscow. The tool uses Cvc 3 [28] as SMT solver, CSIsat [11] as interpolation procedure, and is implemented in OCaml.

CPAchecker 1.0.10-abe [24], submitted by *Stefan Löwe and Philipp Wendler* (University of Passau, Germany), has achieved the following placements:

- *Winner* in ControlFlowInteger
- *Silver* in Overall
- *Bronze* in SystemC
- *Bronze* in HeapManipulation

CPAchecker 1.0.10-abe is based on a predicate analysis, with an application focus similar to that of Blast. CPAchecker [9][15] is a flexible verification framework that implements the formalism of configurable program

[14] `http://mtc.epfl.ch/software-tools/blast`
[15] `http://cpachecker.sosy-lab.org`

Table 2. Competition candidates with their system-description references and representing jury members

Competition candidate	Ref.	Representing jury memb.	Affiliation
Blast 2.7	[26]	Vadim Mutilin	Moscow, Russia
CPAchecker 1.0.10-abe	[24]	Philipp Wendler	Passau, Germany
CPAchecker 1.0.10-memo	[31]	Daniel Wonisch	Paderborn, Germany
Esbmc 1.17	[16]	Bernd Fischer	Southampton, UK
FShell 1.3	[22]	Helmut Veith	Vienna, Austria
Llbmc 0.9	[27]	Carsten Sinz	Karlsruhe, Germany
Predator 2011-10-11	[18]	Tomas Vojnar	Brno, Czech Republic
QArmc-Hsf(c)	[20]	Andrey Rybalchenko	Munich, Germany
SATabs 3.0	[4]	Michael Tautschnig	Oxford, UK
Wolverine 0.5c	[29]	Georg Weissenbacher	Princeton, USA

Table 3. Technologies and features that the competition candidates offer

Competition candidate	CEGAR	Predicate Abstraction	Bounded Model Checking	Shape Analysis	ART-based Analysis	Lazy Abstraction	Interpolation	Concurrency Support
Blast	✓	✓			✓	✓	✓	
CPA-abe	✓	✓			✓	✓	✓	
CPA-memo	✓	✓			✓	✓	✓	
Esbmc			✓					✓
FShell			✓					
Llbmc			✓					
Predator				✓				
QArmc-Hsf(C)	✓	✓			✓		✓	✓
SATabs	✓	✓						✓
Wolverine	✓				✓	✓	✓	

analysis (CPA) [8]. The competition candidate CPAchecker 1.0.10-abe uses the concept of adjustable-block encoding [10], which is implemented as a CPA in the framework. The algorithm uses an interpolation-based refinement of the predicate precision and explores the abstract state space by building an abstract reachability graph. The framework currently uses MathSAT [13] as SMT solver and interpolation procedure, and is implemented in Java.

CPAchecker 1.0.10-memo [31], submitted by *Daniel Wonisch* (University of Paderborn, Germany), has achieved the following placements:

- *Winner* in Overall
- *Silver* in ControlFlowInteger
- *Silver* in DeviceDrivers64
- *Bronze* in HeapManipulation

CPAchecker 1.0.10-memo is based the verification framework CPAchecker, configured for large-block encoding [5] and boolean predicate abstraction. The novel feature of the competition candidate CPAchecker 1.0.10-memo is the integration of the concept of block-abstraction memoization as a CPA. Intermediate analysis results of large blocks are cached in order to avoid repeated verification of similar program traces. This concept yields a significant improvement over the standard configuration of CPAchecker in the category 'DeviceDrivers64', as shown in Table 4.

Esbmc 1.17 [16], submitted by *Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer* (University of Southampton, UK), has achieved the following placements:

- *Winner* in SystemC
- *Winner* in Concurrency
- *Bronze* in Overall

Esbmc 1.17[16] is a bounded model checker that is based on the concept of generating verification conditions for the program, which are then passed to an SMT solver for checking if a feasible error path exists. The focus of Esbmc is to provide a context-bounded verification of multi-threaded C programs, in addition to sequential C programs. The tool uses components of the CProver framework [17], the external solvers Z3 [18] and Boolector [19], and is implemented in C++.

FShell 1.3 [22], submitted by *Andreas Holzer, Daniel Kröning, Christian Schallhart, Michael Tautschnig, and Helmut Veith* (TU Vienna, Austria), is a test-generation tool for C programs, which is based on bit-precise bounded model checking for identifying program paths that fulfill a given test-coverage

[16] http://esbmc.org
[17] http://www.cprover.org
[18] http://research.microsoft.com/projects/z3
[19] http://fmv.jku.at/boolector

criterion, for which test vectors can be derived using satisfying assignments. The tool FShell[20] is based on the CProver framework, uses the SAT solver MiniSAT [21], and is implemented in C++.

Llbmc 0.9 [27], submitted by *Carsten Sinz, Stephan Falke, and Florian Merz* (Karlsruhe Institute of Technology, Germany), has achieved the following placements:

- *Winner* in DeviceDrivers
- *Silver* in HeapManipulation

Llbmc 0.9[22] is a bounded model checker that operates on LLVM's intermediate representation, with a focus on providing a bit-precise analysis of C code, in particular for detecting violations of safe memory usage. The tool is based on the Llvm compiler infrastructure, and passes the verification conditions to the SMT solver Stp [23], which supports bit-vectors and arrays. Llbmc is implemented in C++.

Predator 2011-10-11 [18], submitted by *Kamil Dudka, Petr Muller, Petr Peringer, and Tomas Vojnar* (Brno University of Technology, Czech Republic), has achieved the following placements:

- *Winner* in HeapManipulation
- *Silver* in DeviceDrivers

Predator 2011-10-11[24] is a program analyzer that is based on separation logic, with a focus on verifying C programs with dynamically linked list data structures. The separation-logic formulas that describe (infinite) sets of heaps are internally represented as heap graphs. The main objective of the Predator project is to support the verification of system code, which also uses low-level programming techniques like pointer arithmetics. The tool uses no external decision procedure, is designed as a plug-in for GCC, and is implemented in C++.

QArmc-Hsf(c) [20], submitted by *Sergey Grebenshchikov, Ashutosh Gupta, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko* (TU Munich, Germany), has achieved the following placements:

- *Bronze* in ControlFlowInteger

QArmc-Hsf(c)[25,26] is a model checker that is based on predicate abstraction with a special focus on liveness properties in addition to being able to check safety

[20] `http://code.forsyte.de/fshell`
[21] `http://minisat.se`
[22] `http://baldur.iti.uka.de/llbmc`
[23] `http://sites.google.com/site/stpfastprover`
[24] `http://www.fit.vutbr.cz/research/groups/verifit/tools/predator`
[25] This tool participated in the competition as QArmc and was renamed to Hsf(c) when the proceedings were due.
[26] `http://www7.in.tum.de/tools/hsf`

properties. The tool is based on the CEGAR algorithm, but instead of operating on transition systems, it operates directly on Horn-clause representations of the program and its proof rules. The tool is based on the ARMC infrastructure and the constraint solver CLP(Q). The frontend CIL is used as parser and for the transformation of C programs into the internal representation; the backend is implemented in Prolog and requires the SICSTUS compiler package.

SATABS 3.0 [4], submitted by *Alastair Donaldson, Alexander Kaiser, Daniel Kröning, Michael Tautschnig, and Thomas Wahl* (Oxford University, UK), has achieved the following placements:

- *Silver* in SystemC
- *Silver* in Concurrency
- *Bronze* in DeviceDrivers64

SATABS 3.0[27] is a model checker that is based on predicate abstraction with a focus on bit-precise analysis of program variables. The tool implements an explicit abstract-check-refine loop of the CEGAR algorithm for sequential and concurrent programs. In every iteration, an abstract (boolean) program is computed, then this abstract program is model-checked, then the error path is checked for feasibility, and predicates are discovered in order to compute a more precise abstract program in the next iteration. The tool uses components from the CPROVER framework, SMV or BOOM as model checkers, MINISAT as SAT solver, and is implemented in C++.

WOLVERINE 0.5C [29], submitted by *Georg Weissenbacher, Daniel Kröning, and Sharad Malik* (Princeton University, USA), is a model checker that is based on interpolation-based predicate analysis without computing predicate abstractions during single post-operations. Instead of discovering predicates and collecting them in a predicate precision, the interpolants from infeasible paths are directly used as part of the abstract states. WOLVERINE 0.5C[28] is based on an integrated interpolating decision procedure, uses components from the CPROVER framework, and is implemented in C++.

6 Results and Discussion

The results in this paper represent the state-of-the-art in software verification in terms of precision and performance, as available and participated, when the benchmark verification runs for the 1st Competition on Software Verification were performed. We sent all results to the participating competition teams for review; all results shown in this paper are approved by the competing teams. The total quantitative overview is provided in Table 4. The run time in the tables is given

[27] `http://www.cprover.org/satabs`
[28] `http://www.cprover.org/wolverine`

Table 4. Summary of all results. The tools are listed in alphabetical order. In every table cell for competition results, we list the points in the first row and the CPU time for successful runs in the second row (cf. Table 1 for the scoring schema). The top-three candidates have their score set in bold face and in larger font size. The entry '—' means that the competition candidate opted-out or obtained a total of less than 0 points in the category.

Competition candidate Representing jury member Affiliation	**ControlFlowInteger** 144 points max. 93 verification tasks	**DeviceDrivers** 103 points max. 59 verification tasks	**DeviceDrivers64** 66 points max. 41 verification tasks	**HeapManipulation** 24 points max. 14 verification tasks	**SystemC** 87 points max. 62 verification tasks	**Concurrency** 11 points max. 8 verification tasks	**Overall** 435 points max. 277 verification tasks
Blast							
Vadim Mutilin Moscow, Russia	71 9900 s	**72** 30 s	**55** 1400 s	—	33 4000 s	—	231 15000 s
CPA-abe							
Philipp Wendler Passau, Germany	**141** 1000 s	51 97 s	26 1900 s	**4** 16 s	**45** 1100 s	0 0 s	**267** 4100 s
CPA-memo							
Daniel Wonisch Paderborn, Germany	**140** 3200 s	51 93 s	**49** 500 s	**4** 16 s	36 450 s	0 0 s	**280** 4300 s
ESBMC							
Bernd Fischer Southampton, UK	102 4500 s	63 160 s	10 870 s	1 220 s	**67** 760 s	**6** 270 s	**249** 6800 s
FShell							
Helmut Veith Vienna, Austria	28 580 s	20 3.5 s	0 0 s	—	—	0 0 s	48 580 s
LLBMC							
Carsten Sinz Karlsruhe, Germany	100 2400 s	**80** 1.6 s	1 110 s	**17** 210 s	8 2.4 s	—	206 2700 s
Predator							
Tomas Vojnar Brno, Czech Republic	17 1100 s	**80** 1.9 s	0 0 s	**20** 1.0 s	21 630 s	0 0 s	138 1700 s
QArmc-Hsf(c)							
Andrey Rybalchenko Munich, Germany	**140** 4800 s	—	—	—	8 820 s	—	148 5600 s
SATabs							
Michael Tautschnig Oxford, UK	75 5400 s	71 140 s	**32** 3200 s	—	**57** 5000 s	**1** 1.4 s	236 14000 s
Wolverine							
Georg Weissenbacher Princeton, USA	39 580 s	68 65 s	16 1300 s	—	36 1900 s	—	159 3800 s

Table 5. Overview of the top-five candidates for each category. The run time is given in seconds of CPU usage for the verification tasks that were successfully solved. The column 'False Alarms' indicates the number of verification tasks for which the tool reported an error but the program was safe (false positive), and column 'Missed Bugs' indicates the number of verification tasks for which the tool claims that the program is safe although it contains a bug (false negative).

Rank	Candidate	Score	Run Time	Solved Tasks	False Alarms	Missed Bugs
ControlFlowInteger						
1	**CPAchecker-abe**	**141**	1000	91		
2	CPAchecker-memo	140	3200	91		
3	QArmc-Hsf(c)	140	4800	91		
4	Esbmc 1.17	102	4500	70		4
5	Llbmc 0.9	100	2400	79	5	3
DeviceDrivers						
1	**Llbmc 0.9**	**80**	1.6	46		
2	Predator	80	1.9	46		
3	Blast 2.7	72	30	51	6	1
4	SATabs 3.0	71	140	43		1
5	Wolverine 0.5c	68	65	48	2	3
DeviceDrivers64						
1	**Blast 2.7**	**55**	1400	33		
2	CPAchecker-memo	49	500	33	2	
3	SATabs 3.0	32	3200	17		
4	CPAchecker-abe	26	1900	23	2	
5	Wolverine 0.5c	16	1300	12		
HeapManipulation						
1	**Predator**	**20**	1.0	12		
2	Llbmc 0.9	17	210	10		
3	CPAchecker-abe	4	16	9	5	
3	CPAchecker-memo	4	16	9	5	
5	Esbmc 1.17	1	220	6	3	1
SystemC						
1	**Esbmc 1.17**	**67**	760	58		4
2	SATabs 3.0	57	5000	40		
3	CPAchecker-abe	45	1100	34		
4	CPAchecker-memo	36	450	30		
5	Wolverine 0.5c	36	1900	25		
Concurrency						
1	**Esbmc 1.17**	**6**	270	7		1
2	SATabs 3.0	1	1.4	1		
Overall						
1	**CPAchecker-memo**	**280**	4300	209	20	
2	CPAchecker-abe	267	4100	203	20	
3	Esbmc 1.17	249	6800	191	9	11
4	SATabs 3.0	238	15000	149		1
5	Blast 2.7	231	15000	158	6	1

in seconds of CPU time. All measurement values are rounded to two significant digits. The points are calculated according to the scoring schema in Table 1. Some more details on the top-five tools for each category are given in Table 5.

The main result of this competition is that there is currently no single technique that is absolutely superior in comparison with the other tools. The competition candidates have scored differently in the various categories, with no single candidate being the absolute winner.

Towards Robustness. There is one single competition candidate that achieved positive scores in all categories: ESBMC 1.17. The following candidates participated in all categories, with a non-negative score in all categories: CPACHECKER 1.0.10-ABE, CPACHECKER 1.0.10-MEMO, ESBMC 1.17, and PREDATOR 2011-10-11.

Towards Soundness. There are four competition candidates that never reported the answer 'SAFE' for a benchmark program that actually contains a bug (missed a bug): CPACHECKER 1.0.10-ABE, CPACHECKER 1.0.10-MEMO, PREDATOR 2011-10-11, and QARMC-HSF(C).

Towards Completeness. There are three competition candidates that never reported a bug for a safe program (false alarm): FSHELL 1.3, QARMC-HSF(C), and SATABS 3.0.

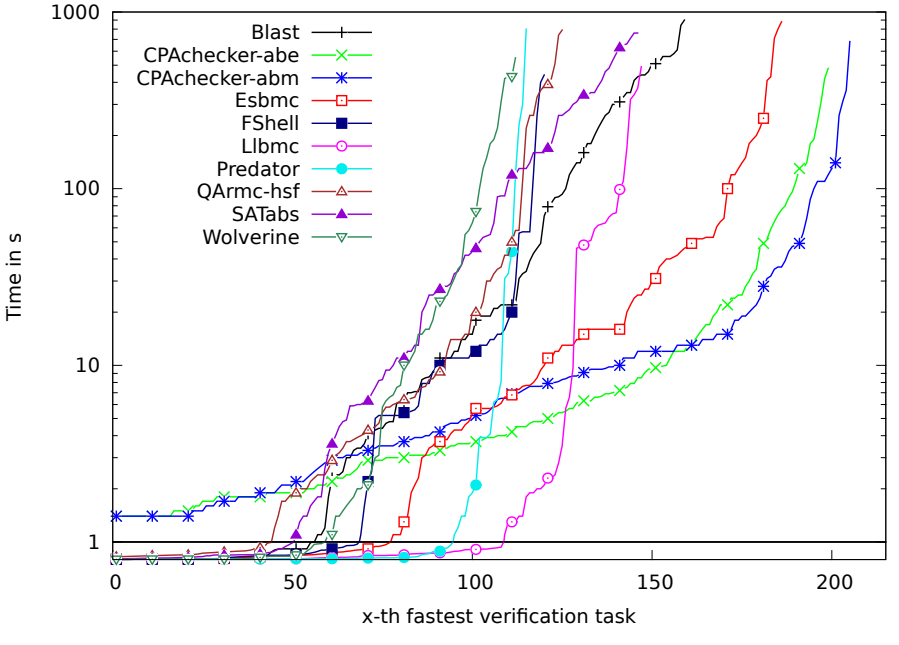

Fig. 1. Quantile functions: For each competition candidate, we plot all pairs (x, y) such that the maximum run time of the x fastest results is y. A logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s. The graphs are decorated with symbols at every tenth data point in order to make the graphs distinguishable on gray-scale prints.

About Solved Instances and Run Time. Figure 1 illustrates the competition results using the quantile functions over all benchmark verification tasks. The function graph for a competition candidate yields the maximum run time y for the x fastest computed correct results. On the left, the plot shows that two candidates need a few seconds of run time even for the simplest benchmark programs; this seems due to the setup time for the Java virtual machine that these two candidates are using. The right-most data point of each graph yields the number of successfully solved verification tasks by the corresponding competition candidate. The area below a graph (its integral) is the accumulated run time for all successfully solved verification tasks.

7 Summary and Future Plans

The competition on software verification was well received in the research community, and the participants were enthusiastic about the event. The participation of ten teams in the first competition, which exceeded the expectation, witnesses the need for such an event. The organizer and the jury were making 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 conclusion is that the event shall be held annually from now on. One important objective for the next competition is to significantly extend the benchmark set, especially in the categories 'Heap-Manipulation' and 'Concurrency'. Since software verification becomes more and more relevant in practice, we are convinced that the pool of available benchmarks will considerably grow in the next few years. We also hope that the number of participants even increases in the next years, and that a wider range of verification technologies will be covered.

Acknowledgments. We thank the TACAS steering committee and the program chairs for hosting the Competition on Software Verification as satellite event of the conference TACAS, and for the encouragement and support during the design of the event. Most importantly, we thank the participating teams for contributing their tools and system descriptions. In particular, we want to thank (among others) Pavel Shved, Kamil Dudka, Georg Weissenbacher, Corneliu Popeea, Bernd Fischer, and Carsten Sinz for their help in preparing the benchmark verification tasks for the competition (contributing verification tasks, sending patches and comments). The biggest thanks goes to Karlheinz Friedberger, who programmed the benchmark processing script and helped with configuring the tools and infrastructure that we used for the competition.

References

1. Alglave, J., Donaldson, A.F., Kröning, D., Tautschnig, M.: Making Software Verification Tools Really Work. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 28–42. Springer, Heidelberg (2011)

2. American National Standards Institute. ANSI/ISO/IEC 9899-1999: Programming Languages — C. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA (1999)
3. Ball, T., Rajamani, S.K.: The Slam Project: Debugging System Software via Static Analysis. In: Proc. POPL, pp. 1–3. ACM (2002)
4. Basler, G., Donaldson, A., Kaiser, A., Kröning, D., Tautschnig, M., Wahl, T.: SATAbs: A Bit-Precise Verifier for C Programs. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 551–554. Springer, Heidelberg (2012)
5. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software Model Checking via Large-Block Encoding. In: Proc. FMCAD, pp. 25–32. IEEE (2009)
6. 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)
7. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy Shape Analysis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)
8. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program Analysis with Dynamic Precision Adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008)
9. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
10. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010)
11. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
12. 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)
13. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
14. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: A Software Model Checking Approach. In: Proc. FMCAD, pp. 51–59. FMCAD Inc. (2010)
15. 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)
16. Cordeiro, L., Morse, J., Nicole, D., Fischer, B.: Context-Bounded Model Checking with Esbmc. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 534–537. Springer, Heidelberg (2012)
17. Craig, W.: Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. J. Symb. Log. 22(3), 250–268 (1957)
18. Dudka, K., Müller, P., Peringer, P., Vojnar, T.: Predator: A Verification Tool for Programs with Dynamic Linked Data Structures. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 544–547. Springer, Heidelberg (2012)
19. 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)
20. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier Based on Horn Clauses. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 548–550. Springer, Heidelberg (2012)

21. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. POPL, pp. 58–70. ACM (2002)
22. Holzer, A., Kröning, D., Schallhart, C., Tautschnig, M., Veith, H.: Proving Reachability Using FShell. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 537–540. Springer, Heidelberg (2012)
23. 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)
24. Löwe, S., Wendler, P.: CPAchecker with Adjustable Predicate Analysis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 527–529. Springer, Heidelberg (2012)
25. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
26. Shved, P., Mandrykin, M., Mutilin, V.: Predicate Analysis with Blast 2.7. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 524–526. Springer, Heidelberg (2012)
27. Sinz, C., Merz, F., Falke, S.: Llbmc: A Bounded Model Checker for Llvms Intermediate Representation. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 541–543. Springer, Heidelberg (2012)
28. Stump, A., Barrett, C.W., Dill, D.L.: CVC: A Cooperating Validity Checker. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 500–504. Springer, Heidelberg (2002)
29. Weissenbacher, G., Kröning, D., Malik, S.: Wolverine: Battling Bugs with Interpolants. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 555–557. Springer, Heidelberg (2012)
30. Witkowski, T., Blanc, N., Kröning, D., Weissenbacher, G.: Model Checking Concurrent Linux Device Drivers. In: Proc. ASE, pp. 501–504. ACM (2007)
31. Wonisch, D.: Block Abstraction Memoization for CPAchecker. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 531–533. Springer, Heidelberg (2012)