Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP)

DOI:
10.1145/3282517.3282529

Document Version
Accepted author manuscript

Published in:
ACM SigSoft Software Engineering Notes

Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP)

Lucas Cordeiro
University of Manchester
Manchester, UK
lucas.cordeiro@manchester.ac.uk

Daniel Kroening
University of Oxford
Oxford, UK
kroening@cs.ox.ac.uk

Peter Schrammel
University of Sussex
Brighton, UK
p.schrammel@sussex.ac.uk

ABSTRACT

Empirical evaluation of verification tools by benchmarking is a common method in software verification research. The Competition on Software Verification (SV-COMP) aims at standardization and reproducibility of benchmarking within the software verification community on an annual basis, through comparative evaluation of fully automatic software verifiers for C programs. Building upon this success, here we describe our proposal to re-use the ecosystem developed around SV-COMP for benchmarking Java verification tools. We provide a detailed description of the rules for benchmark verification tasks, the integration of new tools into SV-COMP's benchmarking framework and also give experimental results of a benchmarking run on three state-of-the-art Java verification tools, JPF-SE, JayHorn and JBMC.

1. INTRODUCTION

The complexity of software in smartphones and enterprise applications has dramatically increased over the last years. In particular, mobile applications based on the Android OS have gained popularity in the consumer electronics industry, reaching nearly 87% market share [1]; the Android OS essentially consists of a large set of libraries (approximately 13 million lines of code), containing both Java code and native code, which thus require methods to verify its security properties (e.g., sensitive data leakage) [2]. Similarly, Java remains popular in business applications (server-side), mainly owing to the existence of several robust frameworks (e.g., Spring [3]); therefore, verification of Java enterprise applications is also of particular interest.

Technology companies such as Facebook and Amazon increasingly invest effort and time to develop efficient and effective verification methods as testing alternatives [4, 5], to check correctness of some aspects of their systems with the goal to improve robustness and security [6]. Although there are several software verification for Java programs (e.g., Bandera [7], JPF-SE [8], JayHorn [9] and JBMC [10]), they are typically very difficult to compare in practice, mainly due to the lack of (1) a common set of benchmarks and (2) methods to standardize and reproduce the empirical evaluations.

The Software Verification Competition. SV-COMP is one of the main initiatives targeted at the evaluation of new software verification methods, technologies, and tools [11]. It has been running since 2012 as part of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Its main focus has been on evaluating different verification (and testing) tools for C programs. Currently, there are some powerful Java verifiers available, but there is no standard procedure to compare them fairly. One of the main difficulties to conduct such a comparison is the lack of a standard set of Java benchmarks and respective benchmarking infrastructure to obtain reliable, reproducible and accurate results.

The main contribution of this paper is to define a standard Java benchmark set and respective benchmarking infrastructure, which can drive the verification community to effectively evaluate state-of-the-art software verification tools for Java programs with the goal to achieve comparability and reproducibility. In particular, we collect and standardize a reasonable set of Java benchmarks from different sources [9, 10, 12, 13] and re-use existing benchmarking infrastructure [14], so that we allow the community to get beyond research prototypes to usable tools [15]. This can lead to further progress in the area since verification of Java programs, which is of particular interest due to its large deployment in industrial systems.

Java verification tools. Here, we consider the following tools:

JBMC [10][1] is based on the C Bounded Model Checker (CBMC) [16] to verify Java bytecode. JBMC consists of a frontend for parsing Java bytecode and a Java operational model (JOM), which is an exact but verification-friendly model of the standard Java libraries. A distinct feature of JBMC is the use of Bounded Model Checking (BMC) [17] in combination with Boolean Satisfiability and Satisfiability Modulo Theories (SMT) [18] and full symbolic state-space exploration, which allows JBMC to perform a bit-accurate verification of Java programs.

Java PathFinder (JPF)[2] is an explicit-state [8] and symbolic [12, 13] software model checker for Java bytecode. JPF is used to find and explain defects, collect runtime information as coverage metrics, deduce test vectors, and create corresponding test drivers for Java programs. JPF checks for property violations such as deadlocks or unhandled exceptions along all potential execution paths as well as user-specified assertions.

JayHorn[3] is a verifier for Java bytecode [9] that uses the Java optimization framework Soot [19] as a front-end and then produces a set of constrained Horn clauses to encode the verification condition (VC). JayHorn is able to check for user-specified assertions and is sound for Java programs that use a single thread, have no dynamic class loading and complex static initializers.

Overview. This paper proposes a Java Category for the Software Verification Competition (SV-COMP). First, we describe in detail the definition and set up of the category. Then, we report on the integration of the three tools mentioned above into the SV-COMP benchmarking infrastructure. Finally, we give the experimental results that we obtained by benchmarking the tools on the benchmark collection we propose.

[1]Available at `https://www.cprover.org/jbmc/`
[2]Available at `https://github.com/javapathfinder`
[3]Available at `https://github.com/jayhorn/jayhorn`

2. OUR PROPOSAL FOR A JAVA CATEGORY IN SV-COMP

We describe our proposal to host a Java category in SV-COMP. In particular, we define the structure and meaning of verification tasks, the properties to be verified, the execution environment and how verification results could be evaluated.

2.1 Definition of Verification Task

A verification task consists of a Java program and a specification. A verification run is a non-interactive execution of a competition candidate, i.e., a verifier, on a single verification task, in order to check whether the program satisfies its specification. According to the current SV-COMP rules,[4] the result of a verification run is a triple (*answer*, *witness*, *time*). *Answer* is one of the following outcomes given in Table 1.

Table 1: Definition of a Verification Result in SV-COMP [11].

TRUE	The specification is satisfied (i.e., there is no path that violates the specification).
FALSE	The specification is violated (i.e., there exists a path that violates the specification).
UNKNOWN	The tool cannot decide the problem or terminates by a tool crash, time-out, or out-of-memory (i.e., the competition candidate does not succeed in computing an answer TRUE or FALSE).

Time is the consumed CPU time until the verifier terminates. It includes the consumed CPU time of all processes that the verifier starts. If *time* is equal to or larger than the time limit, then the verifier is terminated and the *answer* is set to "timeout" (and interpreted as UNKNOWN).

There is no witness checking (as previously described in [20]) in the Java category at the moment, which represents an important feature to reproduce verification results given by verifiers. Additionally, the Java programs are partitioned into categories, which are defined in category-set files. At the moment there is only one category in Java, *ReachSafety*, which is concerned with specifications that consider an assert(*condition*) statement in the verification task whose non-violation must be proven or refuted.

2.2 Benchmark Verification Tasks

All Java verification tasks used in the competition must be part of the SV-COMP benchmark collection[5] prior to the benchmark contribution deadline (typically in September). The competition candidates can "train", i.e., run and tune, their verifiers on the verification tasks until the tool submission deadline (typically in November). Misclassified benchmarks can be corrected during this training period. Contentious benchmarks are excluded from the competition once they are identified. SV-COMP does not use verification tasks without training; this is particularly important for Java since it has many features of which verifiers only support a subset; participants should know before the competition which features they are expected to support.

Benchmark structure. Verification tasks are grouped in directories, e.g., jbmc-regression. Within these directories, each verification task is in its own directory, e.g., StringValueOf01_true-assert. The suffix of this directory contains the expected verification outcome, i.e., _true-assert means that there is no execution of the program that violates any of the assertions in that program. A verification task has the directory structure[6] shown in Table 2. Note that lib/ contains library dependencies of the given benchmark in the same style of Apache Ant [21], while target/ contains temporary files built from the benchmark sources, which will used by the verification tools.

Table 2: Directory structure of a verification task.

src	
src/main/java/Main.java	The Main class.
src/main/java/	This directory contains the remaining Java source tree.
lib/	This directory contains the library dependencies (without java.* and javax.*).
target/	After building, this directory contains .jar assembling .class files built from src and from within .jar files in lib.

The programs are assumed to be written in Java 1.8. The .java source files are in the source tree in the src sub-directory of the benchmark directory. The program may call the Java standard library (java.*, javax.*). Calls to other libraries are allowed under the condition that the library is provided as a .jar file in the lib sub-directory of the benchmark directory. The benchmark must have a Main class with a public static void main(String[]) method in the root package. The Main.java file must have a copyright header indicating the source of the benchmark as well as name, version and links to the source code of all used libraries that are included in lib.

Potential competition participants are invited to submit verification tasks until the benchmark contribution deadline by submitting a Pull Request to the benchmark collection repository.[7] Verification tasks must comply with the aforementioned format. New proposed categories will be included if at least three different tools or teams participate in the category (i.e., not the same tool twice with a different configuration). In the following, we list a few conventions that are used in the Java verification tasks.

Assertions. For checking (un)reachability, we use the assert keyword provided in the Java language. It is assumed that the AssertionError thrown on violation of the assertion always leads to abortion of the program, i.e., it is not caught in the program.

Nondeterminism. The arguments to public static void main(String[] args) are assumed to be nondeterministic under the following constraints: assume(args != null && for all i. 0<=i<args.length => args[i] != null).

We do not specify custom methods for introducing nondeterminism as done in the C categories of SV-COMP (cf. __VERIFIER_nondet) [11]. Instead, we use the java.util.Random class; the methods in that class are expected to return a nondeterministic value instead of a random value, but satisfying the same constraints on their value range.

Moreover, we do not specify a custom assume method. It is recommended to use return or System.exit() to achieve the desired behavior as they do not impact the termination behavior of a

[4] A detailed description of the current rules can be found in `https://sv-comp.sosy-lab.org/2018/rules.php`

[5] `https://github.com/sosy-lab/sv-benchmarks`

[6] This structure is inspired by the Maven standard directory layout (cf. `https://maven.apache.org`).

[7] `https://github.com/sosy-lab/sv-benchmarks`

Tool	JayHorn 0.5.1				JBMC 5.9				JPF rev 32			
Limits	timelimit: 900 s, memlimit: 15000 MB, CPU core limit: 8											
Host	localhost											
OS	Linux 4.4.0-131-generic x86_64											
System	CPU: Intel Core i7-6700HQ CPU @ 2.60GHz with 8 cores, frequency: 3500 MHz, Turbo Boost enabled; RAM: 16014020 kB											
Date of execution	2018-07-29 02:53:52 BST				2018-07-28 18:40:30 BST				2018-07-29 16:02:12 BST			
Run set	jayhorn.sv-comp18				jbmc.sv-comp18				jpf.sv-comp18			
../git-sv-benchmarks/java/	status	cputime	walltime	memUsage	status	cputime	walltime	memUsage	status	cputime	walltime	memUsage
jbmc-regression/ArithmeticException1_false-assert	true	2.895s	1.657s	195948544	false(reach)	0.260s	0.273s	43819008	false(reach)	1.011s	0.595s	67473408
jbmc-regression/ArithmeticException5_true-assert	true	2.308s	1.345s	152911872	true	0.346s	0.365s	43417600	false(reach)	0.998s	0.612s	66256896
jbmc-regression/ArithmeticException6_false-assert	true	2.345s	1.346s	150769664	false(reach)	0.304s	0.328s	43786240	true	0.938s	0.584s	64716800
jbmc-regression/ArrayIndexOutOfBoundsException1_false-assert	false(reach)	4.108s	2.300s	197890048	false(reach)	0.297s	0.308s	44023808	true	1.011s	0.601s	66256896
jbmc-regression/ArrayIndexOutOfBoundsException2_false-assert	false(reach)	4.161s	2.296s	193990656	false(reach)	0.301s	0.316s	43749376	true	0.979s	0.590s	63488000
jbmc-regression/ArrayIndexOutOfBoundsException3_false-assert	false(reach)	5.137s	2.827s	230653952	false(reach)	0.311s	0.324s	43880448	true	0.918s	0.577s	64274432
jbmc-regression/BufferedReaderReadLine_false-assert	false(reach)	7.010s	3.658s	260804608	false(reach)	0.439s	0.447s	44457984	true	1.006s	0.616s	67321856
jbmc-regression/CharSequenceBug_false-assert	false(reach)	4.433s	2.416s	192462848	false(reach)	0.387s	0.402s	44269568	true	0.959s	0.597s	63365120
jbmc-regression/CharSequenceToString_true-assert	false(reach)	4.515s	2.493s	193818624	true	0.712s	0.731s	44658688	true	0.970s	0.580s	63336448
jbmc-regression/ClassCastException1_false-assert	false(reach)	3.961s	2.215s	194396160	false(reach)	0.328s	0.335s	44134400	false(reach)	1.000s	0.604s	64888832
jbmc-regression/ClassCastException2_true-assert	true	4.357s	2.356s	189190144	true	0.442s	0.456s	44097536	true	0.950s	0.581s	64630784
jbmc-regression/ClassCastException3_false-assert	false(reach)	3.969s	2.186s	192733184	false(reach)	0.297s	0.316s	43929600	false(reach)	0.950s	0.584s	64626688
jbmc-regression/Class_method1_true-assert	true	3.129s	1.754s	184451072	true	0.467s	0.486s	43941888	true	0.920s	0.568s	63279104
jbmc-regression/Inheritance1_true-assert	true	5.023s	2.652s	217104384	true	0.459s	0.471s	44077056	true	0.950s	0.578s	64221184
jbmc-regression/NegativeArraySizeException1_false-assert	true	2.399s	1.394s	153817088	false(reach)	0.291s	0.303s	43745280	false(reach)	0.961s	0.578s	64442368
jbmc-regression/NegativeArraySizeException2_false-assert	true	2.205s	1.305s	153120768	false(reach)	0.285s	0.312s	43921408	false(reach)	0.985s	0.605s	63938560
jbmc-regression/NullPointerException1_true-assert	unknown	1.303s	0.761s	116211712	true	0.431s	0.441s	43900928	false(reach)	0.952s	0.595s	63512576
jbmc-regression/NullPointerException2_false-assert	unknown	1.273s	0.736s	115384320	false(reach)	0.286s	0.294s	43929600	false(reach)	0.949s	0.575s	64266240
jbmc-regression/NullPointerException3_false-assert	unknown	1.325s	0.752s	115838976	false(reach)	0.266s	0.275s	43737088	false(reach)	0.973s	0.574s	63655936
jbmc-regression/NullPointerException4_false-assert	unknown	1.218s	0.700s	114372608	false(reach)	0.282s	0.296s	43929600	false(reach)	1.002s	0.613s	64077824
jbmc-regression/RegexMatches01_true-assert	false(reach)	5.222s	2.814s	195457024	false(reach)	38.274s	38.288s	237780992	true	1.001s	0.620s	63860736
jbmc-regression/RegexMatches02_false-assert	false(reach)	6.126s	3.235s	218169344	false(reach)	38.763s	38.780s	243040256	true	0.974s	0.603s	63315968
jbmc-regression/RegexSubstitution01_true-assert	false(reach)	10.211s	5.299s	412655616	true	4.537s	4.554s	117723136	true	1.038s	0.645s	65220608
jbmc-regression/RegexSubstitution02_false-assert	false(reach)	9.948s	5.200s	384372736	true	4.592s	4.612s	89309184	true	0.978s	0.598s	64528384
jbmc-regression/RegexSubstitution03_true-assert	false(reach)	7.912s	4.162s	297971712	true	1.741s	1.765s	49369088	true	1.068s	0.638s	65187840
jbmc-regression/StaticCharMethods01_true-assert	false(reach)	4.988s	2.604s	233947136	true	0.944s	0.966s	46419968	true	0.966s	0.579s	62652416
jbmc-regression/StaticCharMethods02_false-assert	false(reach)	5.032s	2.683s	210714624	false(reach)	1.094s	1.104s	53043200	true	0.960s	0.585s	63422464
jbmc-regression/StaticCharMethods03_false-assert	false(reach)	4.816s	2.622s	198123520	false(reach)	1.084s	1.096s	51294208	true	1.022s	0.632s	64135168
jbmc-regression/StaticCharMethods04_false-assert	false(reach)	4.843s	2.636s	200192000	false(reach)	1.046s	1.055s	51044352	true	0.906s	0.551s	63311872
jbmc-regression/StaticCharMethods05_false-assert	false(reach)	7.822s	4.077s	278511616	false(reach)	1.824s	1.833s	59617280	unknown	0.986s	0.598s	60248064
jbmc-regression/StaticCharMethods06_true-assert	false(reach)	6.272s	3.271s	237772800	true	2.687s	2.697s	53399552	true	1.008s	0.624s	64339968
jbmc-regression/StringBuilderAppend01_true-assert	false(reach)	15.775s	8.096s	521629696	unknown	874.936s	875.035s	63283200	true	1.008s	0.610s	65589248
jbmc-regression/StringBuilderAppend02_false-assert	false(reach)	16.517s	8.540s	539189248	unknown	874.962s	875.030s	129449984	true	1.007s	0.616s	63279104

Figure 1: Per benchmark comparison table as produced by BenchExec [14]. The detailed description of the scores of each tool can be found in `https://pschrammel.bitbucket.io/schrammel-it/research/sv-comp-java-2018/`.

program. Using while(true); would make any program with assumptions classified non-terminating when a potential *Java Termination* category might be introduced in future.

Operating System Model. Any library methods that make system calls are not allowed in verification tasks. Exceptions with well-defined behaviors can be explicitly granted if they allow a wider range of benchmarks to be included in the collection. For instance, new java.util.Date() is expected to create a Date object with a nondeterministic timestamp.

2.3 Properties

In SV-COMP, the specification to be verified for a program path/dirname is given either in a file with the name path/dirname.prp or in a file Category.prp. The definitions in these .prp have been designed for extensibility in order to allow new properties for future categories to be specified. For instance, in the C categories, four different properties are in use.

For our Java category, the definition in ReachSafety.prp states CHECK(init(Main.main()), LTL(G assert)). Here, init(Main.main()) gives the initial states of the program by a call of function main (under the assumptions on the inputs of main stated in the previous section). LTL(f) specifies that formula f holds at every initial state of the program. In particular, the linear-time temporal logic (LTL) operator G f means that f globally holds. The proposition assert is true if all assert statements in the program hold.

2.4 Competition Environment and Requirements

In the SV-COMP environment, each software verifier is assumed to run on a machine with a GNU/Linux operating system (x86_64-linux, Ubuntu 16.04). SV-COMP also sets three resource limits to evaluate each software verifier, which are: (*i*) a memory limit of 15 GB of RAM, (*ii*) a runtime limit of 900 seconds of CPU time, and (*iii*) a limit to 8 processing units of a CPU. Note that if a software verifier does not consume CPU time, then it is killed after 900 seconds of wall clock time, and the resulting runtime is set to 900 seconds. For the Java category, JDK 1.8 is assumed to be installed on the competition machines. The modest resource requirements have been chosen in order to allow everybody to reproduce the competition results on a reasonably sized machine.

2.5 Evaluation by Scores and Runtime

SV-COMP has strict rules to evaluate the verification results provided by each software verifier. In particular, each verifier is heavily penalized if they produce an incorrect result for a specific verification task with the goal to favor correctness. The scores are assigned to each software verifier according to Table 3.[8]

The higher score and penalty for the TRUE case is justified because it is usually more difficult to prove a program correct than to find a bug [11].

3. INTEGRATION INTO THE COMPETITION INFRASTRUCTURE

BenchExec[9] [14] is the framework used in SV-COMP for reliable benchmarking and resource measurement. It can be easily installed to run experimental comparisons and to reproduce competition results. For the Java category, we extended the framework by introducing a new assert proposition for specifying properties.

[8] `https://sv-comp.sosy-lab.org/2018/rules.php`
[9] `https://github.com/sosy-lab/benchexec`

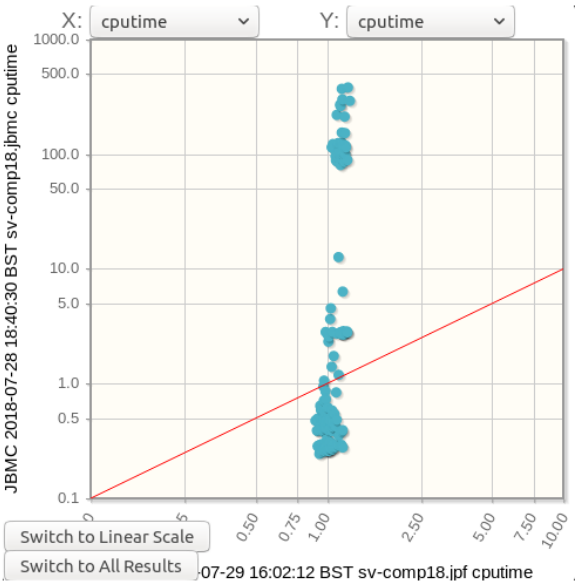

Figure 2: Example of a scatter plot comparing JPF and JBMC as produced by BenchExec [22].

Table 3: Evaluation by Scores and Runtime in SV-COMP [11].

Points	Answer	Description
0	UNKNOWN	Failure to compute verification result, out of resources, program crash.
+1	FALSE correct	The error in the program was found.
-16	FALSE incorrect	An error is reported for a program that fulfills the specification (false alarm, incomplete analysis).
+2	TRUE correct	The program was analyzed to be free of errors.
-32	TRUE incorrect	The program had an error but the competition candidate did not find it (missed bug, unsound analysis).

Integrating a new tool into the framework requires the addition of two files:

- The *tool-info module* is a Python module located in the benchexec/tools directory that implements the tool interface to connect a verifier to BenchExec. Essentially, it must provide functions for running the verifier with a given verification task and to translate the tool output into an *answer* TRUE, FALSE or UNKNOWN (see Section 2.1).
- The *benchmark definition*[10] is an XML file that specifies which categories can be run with a given verifier and which tool command line options to use.

Here, we have implemented and added these files for the three tools under consideration (i.e., JPF, JBMC and JayHorn). After installing a verifier (e.g., JPF) in the base directory of BenchExec, it can be run with the command bin/benchexec jpf.xml. There are various options to run subsets of the verification tasks and overriding time and memory limits, for instance.

4. JAVA BENCHMARK COLLECTION

Previously, there existed 64 "minepump" benchmarks in the SV-COMP repository from earlier attempts to run a Java category; these benchmarks were already classified as "safe" and "unsafe" by the community. Beyond these few files, there was no standard benchmark suite for Java verification available in the community.[11] Therefore, we took the entire JBMC regression test suite ("jbmc-regression"), consisting of 177 benchmarks (including known bugs and hard benchmarks that JBMC cannot yet handle); these benchmarks test common Java features (e.g., polymorphism, exceptions, arrays, and strings) and they were classified by the JBMC developers. We also used 23 benchmarks ("jayhorn-recursive") taken from the JayHorn repository [9]. These are mainly C benchmarks from the recursive category that have been translated into Java by keeping the original classification from SV-COMP. Additionally, we have extracted 104 benchmarks from the JPF regression test suite [12] ("jpf-regression"); for these particular benchmarks, we have manually inspected and classified them as "safe" and "unsafe". Table 4 summarizes the characteristics of the benchmark sets.[12]

Table 4: Characteristics of the Java Benchmark Sets.

benchmark set	total	safe	unsafe	avg. LOC
jbmc-regression	177	89	88	25
jpf-regression	104	52	52	52
jayhorn-recursive	23	14	9	35
minepump	64	8	56	62
total	368	163	205	40

These benchmarks are a good start to launch the proposed Java category, but are not yet fully representative for the breadth of challenges that we face in verifying Java programs. We will rely on the community to contribute and continuously enrich the collection of Java benchmarks in the future.

5. BENCHMARKING RESULTS

The results of running a verifier using BenchExec as explained in Section 3 are collected in a timestamped format in the results directory with the BenchExec base directory. This contains a .zip file with the log files and a .xml.bz2 file with the results in a structured format. One or more of the latter files (potentially of different tools) can be passed to bin/table-generator in order to generate an HTML report that compares the benchmarking runs.

A part of this report is shown in Figure 1.[13]. This comparison used JBMC v5.9-3c2e55e, JayHorn v0.5.1, and JPF rev 32. The HTML report allows to filter rows and columns and display the most important comparison charts, such as scatter plots — Figure 2 compares the CPU time of JPF and JBMC, for example — and quantile ("cactus") plots as depicted in Figure 3. The latter plot shows the cumulative time (y-axis) required for a verifier to solve its n fastest benchmarks (x-axis). This allows us to compare the scaling behavior of the tools, i.e., the longer a graph extends to the right the more verification tasks were solved by the tool, the closer to the bottom the faster it is.

6. CONCLUSIONS

We described our proposal to run the first Java category in SV-COMP, given that it is currently focused on evaluating C software verifiers only. In particular, we defined the structure and meaning of verification tasks, the properties to be verified, the execution environment and how verifiers are integrated into the benchmarking framework and how verification results, produced by each Java verifier are evaluated. SV-COMP is one of the most

[10] https://github.com/sosy-lab/sv-comp

[11] There is a community effort in collecting Java benchmarks in http://sir.unl.edu, but they are not currently classified.

[12] The benchmarks are currently under review: https://github.com/sosy-lab/sv-benchmarks/pull/565.

[13] The full results are available at https://pschrammel.bitbucket.io/schrammel-it/research/sv-comp-java-2018/

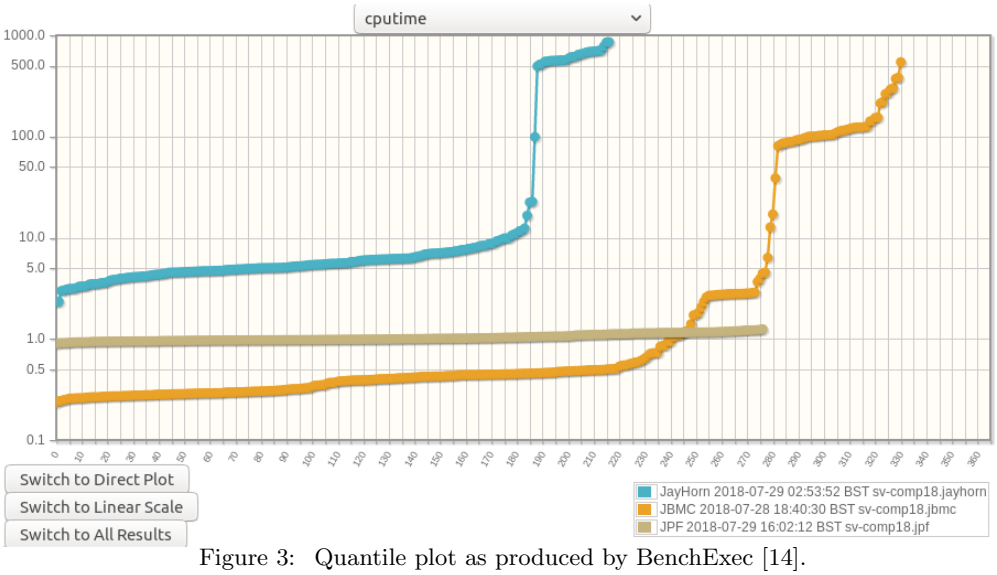

Figure 3: Quantile plot as produced by BenchExec [14].

successful software verification competitions, which is annually held by TACAS. Although the first edition of SV-COMP took place in 2012 and has been a successful so far, there has been no verification track to evaluate software verifiers targeted for Java programs. As a next step, Java verifiers need to be extended in order to produce witness files (for violation and correctness) that adhere to the witness exchange format defined by SV-COMP [20]. In this respect, witness checkers for Java verifiers also need to developed in order to check validity of the verification results provided by each verifier [23].

7. REFERENCES

[1] IDC: Smartphone OS Market Share, 2016 Q3. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp` Accessed 2018-07-29

[2] Bai, G., Ye, Q., Wu, Y., Botha, H., Sun, J., Liu, Y., Dong, J.S., Visser, W.: Towards model checking Android applications. IEEE Trans. Software Eng. **44**(6), 595–612 (2018)

[3] Gupta, K.: Why Is Spring More Popular than Other Java Frameworks? `https://www.freelancinggig.com/blog/2018/04/26/spring-popular-java-frameworks/` Accessed 2018-07-29

[4] O'Hearn, P.W.: Continuous reasoning: Scaling the impact of formal methods. In: LICS, pp. 13–25 (2018)

[5] Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R.: Model checking boot code from AWS data centers. In: CAV. LNCS, vol. 10982, pp. 467–486 (2018)

[6] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of Heartbleed. In: IMC, pp. 475–488 (2014)

[7] Iosif, R., Dwyer, M.B., Hatcliff, J.: Translating java for multiple model checkers: The bandera back-end. Formal Methods in System Design **26**(2), 137–180 (2005). doi:10.1007/s10703-005-1491-3

[8] Visser, W., Havelund, K., Brat, G.P., Park, S.: Model checking programs. In: ASE, pp. 3–12 (2000)

[9] Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A framework for verifying Java programs. In: CAV. LNCS, vol. 9779, pp. 352–358 (2016)

[10] Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: CAV. LNCS, vol. 10981, pp. 183–190 (2018)

[11] Beyer, D.: Software verification with validation of results (report on SV-COMP 2017). In: TACAS. LNCS, vol. 10206, pp. 331–349 (2017)

[12] Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to Java PathFinder. In: TACAS. LNCS, vol. 4424, pp. 134–138 (2007)

[13] Pasareanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: ASE, pp. 179–180 (2010)

[14] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. International Journal on Software Tools for Technology Transfer (to appear) (2017)

[15] Alglave, J., Donaldson, A.F., Kroening, D., Tautschnig, M.: Making software verification tools really work. In: ATVA. LNCS, vol. 6996, pp. 28–42 (2011)

[16] Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. LNCS, vol. 2988, pp. 168–176 (2004)

[17] Biere, A.: Bounded Model Checking. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 457–481 (2009)

[18] Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 825–885 (2009)

[19] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot – a Java bytecode optimization framework. In: CASCON, p. 13 (1999)

[20] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: FSE, pp. 326–337 (2016)

[21] Apache: The Apache Ant Project. `https://ant.apache.org/`. Accessed: 11-09-2018 (2018)

[22] Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: SPIN. LNCS, vol. 9232, pp. 160–178 (2015)

[23] Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses – execution-based validation of verification results. In: TAP. LNCS, vol. 10889, pp. 3–23 (2018)