T3 @SBST2018 Benchmark, and How Much We Can Get from Asemantical Testing

I.S.W.B. Prasetya
Utrecht University
s.w.b.prasetya@uu.nl

ABSTRACT

This paper discusses the performance of the automated testing tool for Java called T3 and compares it with few other tools and human written tests in a benchmark set by the Java Unit Testing Tool Contest 2018. Since all the compared tools rely on randomization when generating their test data, albeit to different degrees and with different heuristics, this paper also tries to give some insight on just how far we can go without having to reconstruct the precise semantic of the programs under test in order to test them.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Empirical software validation*;

KEYWORDS

automated testing, automated unit testing Java, random testing, fuzzing, unit testing

ACM Reference Format:
I.S.W.B. Prasetya. 2018. T3 @SBST2018 Benchmark, and How Much We Can Get from Asemantical Testing. In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3194718.3194727

1 INTRODUCTION

T3 [8] is a tool to automatically test Java classes. This paper discusses its performance with respect to a benchmark taken from the Java Unit Testing Tool Contest 2018, and how it compares to other testing tools and manually written test cases. The Contest is organized annually by the international workshop on Search-Based Software Testing (SBST) since 2013 to measure testing tools' capability to *automatically* test real life Java classes. In 2018 there are three other participating tools: Randoop [6], JTeXpert [9], and Evosuite [2]. Although all these tools have their own algorithm and heuristics for generating tests, they do share one thing in common: they are *asemantical*. That is, they abstract away from the precise semantics of the class under test (CUT) as opposed to tools like CUTE [10] and Pex [11]. Being able to reconstruct the semantic, e.g. by employing Dijkstra predicate transformer [1, 3], is indeed very powerful, as it would allow, at least in theory, the testing tool to calculate which methods to call and which inputs to give them in order to cover any program location in the CUT. T3 does not do this out of a pragmatic consideration: such an approach requires complicated tooling and it requires the source code of CUT's whole dependency to be analyzed, which we do not always have. In this paper we will also try to give some insight on how good asemantical approaches can be when dealing with real life Java classes.

SBST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194727

2 T3 AND THE OTHER TOOLS

Given a Java class to test (class under test or CUT), T3 will test it by randomly generating test sequences, where each sequence consists of calls to the CUT's constructors and methods. When a call to a method (or constructor) needs parameters, parameters of primitive types will be randomly generated, whereas a parameter of an object type T will be generated by locating a constructor or a factory method that can produce an instance of T. This in turn may recursively trigger more method or constructor calls to be generated. Randoop, JTeXpert, and even Evosuite work in principle in the same way, though each does have its own heuristics in how the sequences are being generated:

(1) Randoop [6] avoids generating duplicate sequences. Duplication is checked lexically and by comparing objects generated through the sequences. By default the latter comparison uses the method `equals()`. T3 also drops duplicates, but it only checks for lexical duplicates.
(2) To test a non-static method m of the CUT, JTeXpert [9] generates, essentially, test sequences consisting of prefixes followed by a call to m. The prefixes have the important role to create CUT's instances with enough variety to subsequently test m. To this end, different fields need to be varied. JTeXpert uses static analysis to infer which methods modify which fields, so that it does not waste effort on just randomly trying different methods to build the prefixes.
(3) Of all the tools, Evosuite [2] employs the most sophisticated heuristic, namely a genetic algorithm to iteratively improve the overall coverage of the test suite (the set of test sequences) it generates. Then, after every predefined number of iterations it also employs local search to do a more controlled (but more expensive) improvement.
(4) For the Contest we do not use the standard T3 [8] as described in its manual, but a variation called T3G2 described in [7]. It has two notable features that were not present in the standard T3: (1) rather than generating sequences targeting the whole CUT it considers each method in the CUT as a "*goal*" of its own. It still generates test sequences to test each goal, but when given a time budget it will try to divide

the budget over the available goals so that it at least tries to allocate effort to each. (2) A test sequence is dropped if it does not improve the currently achieved coverage (but it does not know upfront if this sequence would turn out to be non-contributing, so the effort to build its candidacy still has to be expended). T3G2 also employs prefix-build up similar to JTeXpert, but its heuristic is more simplistic, relying only on name patterns to identify the getter methods, and then use them randomly to build the prefixes. It builds a pool of common prefixes which are reused to target different goals. To get diversity, prefixes are checked if they build CUT instances of different structures. The pool is also gradually increased in size whenever effort to raise the coverage of some goal stagnates.

All tools except Randoop use some form of instrumentation to be aware of the current coverage of the generated suite (and can act on this information when the current coverage is not enough). All tools except Randoop scan the CUT's source code to collect used constants and use this information when generating test data. Evosuite also collects constants dynamically (at the runtime) e.g. from expressions of the form $a == b$ when a is a method's parameter.

3 THE BENCHMARK

The Contest of 2018 offers a set of 59 real life classes from various open sources projects (7) as targets [5]. For each CUT, each participating tool is asked to generate a test suite within a given time limit, e.g. 120 seconds. Importantly, the CUTs are *not known* until the Contest ran, so tools' authors could not customize their tools towards these CUTs. Some (but not all) of these CUTs come with accompanying test suites (from their developers), which provides the opportunity to compare the tools' performance against developers written test suites.

A tool's overall performance is expressed as a single performance score that aggregates various indicators e.g. the branch and mutation coverage delivered by the generated test suites and the time the tool takes to generate them. The precise scoring formula can be found in [5]. In this paper we will focus on the tools' performance in terms of the delivered *branch coverage*, with time limit 120 seconds. It is important to note that for the purpose of the Contest all the CUTs *are assumed to be correct*. No specifications describing the intent of the CUTS are given. In this setup, the tools' objective is *not* to find bugs, but to simply deliver as much coverage as possible.

The Contest scoring tool also measures mutation coverage. The tools need to extend each test sequence they generate with an artificial oracle that essentially remembers the states of all objects created by the sequence when it is constructed. A mutation is 'killed' (detected) when executing the sequence on the mutated program produces at least one different state. Note that despite the oracle, to kill a mutation the tool still needs to generate the right execution that would behave differently under the mutation. The oracle is 'artifical' because we cannot really use it for real testing, because then we cannot assume that the CUT is correct (and if so, there is no point to test it). In fact, the more important part of this setup of mutation testing is the tools' ability to generate the right executions, and not the oracles that they use. Unfortunately the configuration used by T3 in the Contest injected a weaker oracle and consequently the measured mutation coverage does not reflect T3's 'intrinsic' mutation coverage if a proper oracle was injected. For this reason we will not discuss this metric further.

Code	Name	#l	#m	#c
DUBBO-10	com.alibaba.dubbo.common.bytecode.Wrapper	173	21	98
DUBBO-2	...common.utils.ReflectUtils	472	45	418
DUBBO-3	...common.utils.StringUtils	178	27	164
DUBBO-4	...common.utils.ClassHelper	60	11	24
DUBBO-5	...common.io.UnsafeByteArrayOutputStream	33	12	18
DUBBO-6	...common.utils.CompatibleTypeUtils	92	2	106
DUBBO-7	...common.beanutil.JavaBeanDescriptor	65	28	48
DUBBO-9	...common.io.Bytes	295	55	130
FASTJSON-1	com.alibaba.fastjson.parser.JSONLexerBase	3091	99	2197
FASTJSON-10	...fastjson.serializer.StringCodec	45	7	20
FASTJSON-2	...fastjson.util.TypeUtils	1372	70	1142
FASTJSON-3	...fastjson.parser.DefaultJSONParser	943	64	564
FASTJSON-4	...fastjson.JSONArray	149	68	30
FASTJSON-5	...fastjson.util.JavaBeanInfo	519	11	452
FASTJSON-7	...fastjson.util.IOUtils	388	20	262
FASTJSON-8	...fastjson.parser.JSONReaderScanner	157	23	96
FASTJSON-9	...fastjson.util.ASMUtils	96	10	64
JSOUP-1	org.jsoup.parser.TokenQueue	141	32	130
JSOUP-2	org.jsoup.select.QueryParser	233	22	142
JSOUP-3	org.jsoup.helper.DataUtil	103	15	90
JSOUP-4	org.jsoup.parser.Parser	41	19	10
JSOUP-5	org.jsoup.parser.Tokeniser	153	27	90
OKIO-1	okio.Buffer	820	126	514
OKIO-10	okio.Timeout	49	12	30
OKIO-2	okio.ByteString	192	58	142
OKIO-3	okio.SegmentedByteString	121	32	60
OKIO-4	okio.RealBufferedSource	201	47	162
OKIO-5	okio.RealBufferedSink	104	29	66
OKIO-6	okio.Okio	35	18	28
OKIO-7	okio.Segment	55	9	24
OKIO-8	okio.AsyncTimeout	68	14	44
OKIO-9	okio.Utf8	30	3	26
ZXING-10	com.google.zxing.qrcode.encoder.Encoder	276	26	155
ZXING-3	...zxing.common.StringUtils	88	3	118
ZXING-4	...zxing.client.result.ResultParser	107	18	82
ZXING-5	...zxing.qrcode.encoder.MatrixUtil	155	21	80
ZXING-7	...zxing.pdf417.decoder.ec.ModulusPoly	109	13	76
ZXING-9	...zxing.oned.CodaBarWriter	63	3	49

Figure 1: The class under tests; #l, #m, #c are their number of lines, methods, and conditions.

The Contest measures the tools under several time limits per CUT, namely 10, 60, 120, and 240 seconds. Here we will focus only on the 120 seconds experiments, mainly because we consider this to be reasonable for a tool to be able to generate tests for a class.

We also want to focus on comparing the tools' performance to manually written test suites. So, of the original set of 59 CUTs we drop 20 CUTs which either do not have the accompanying test suites or whose test suites fail to deploy during the Contest (e.g. due to missing dependency). We drop one more CUT on which all the tools failed to deploy (another missing dependency issue). This leaves us with 38 CUTs listed in Figure 1.

4 RESULTS

Figure 2 shows T3's performance in terms of delivered branch coverage in the scale of 0 - 100% compared to human written test suites. For ease of viewing, the CUTs are sorted in the increasing order of T3's coverage on them. On 16 out of 38 CUTS T3 delivers at least 60% coverage. On 10 CUTs both could *not* reach 60%, but on the other CUTs (28) T3 and human complement each other quite well: on 13 of them (28) T3 outperforms human by at least a 10% margin (and delivers at least 60% coverage), and on 11 CUTs human outperforms T3 by at least a 10% margin (and on the remaining 4 CUTs both are quite close to each other).

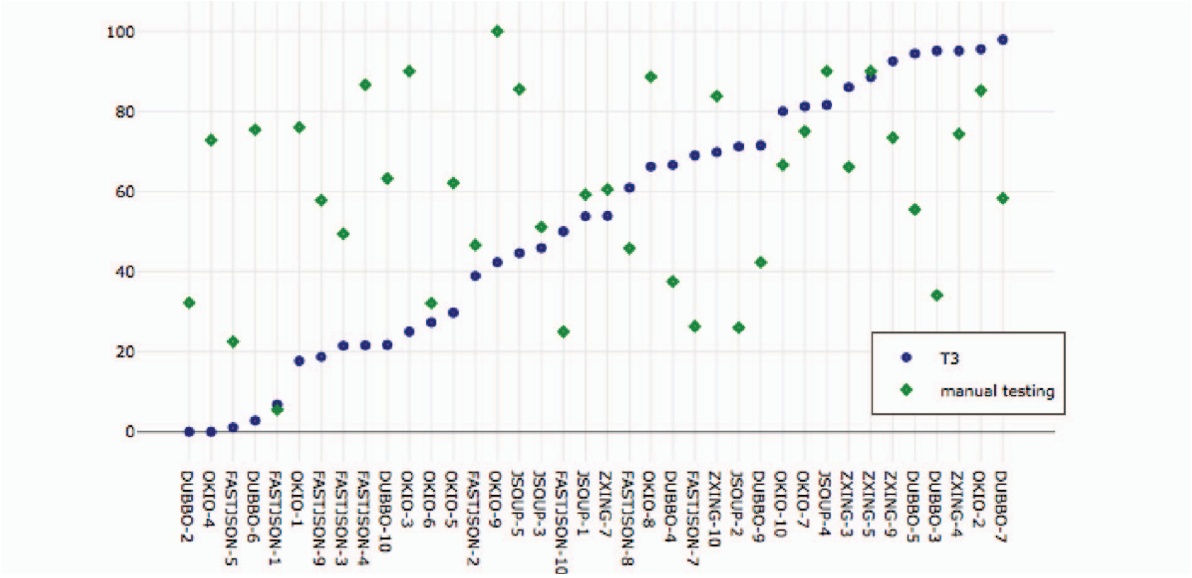

Figure 2: T3's delivered branch coverage, compared with manually written test cases.

Figure 3 shows T3's performance compared to other tools. It turns out that Evosuite outperforms other tools on 17 CUTs, whereas T3 outperforms the others in 6 CUTs. However, in many of these cases at least two of the tools also have quite similar performance, differing with less than 10% margin. If we only look on CUTs where T3 delivers at least 60% coverage, only on two CUTs it outperforms other tools by a margin of at least 10%. On the other hand, on CUTs where Evosuite delivers at least 60%, on five CUTs it outperforms other tools by a margin of at least 10%. The table below gives the summary:

	$\delta \geq 0$	$\delta \geq 10$	$cov \geq 60\% \wedge \delta \geq 10$
T3 outperforms others	6	2	2
Randoop outperforms others	1	1	1
JTeXpert outperforms others	9	5	1
Evosuite outperforms others	17	7	5

The second column shows on how many CUTs a tool T outperforms the others, the third column shows on how many CUTs T outperforms others by a margin of at least 10%, the last column is as the second column, but we only count it when T delivers at least 60% coverage.

Each of the compared tools indeed has its own unique heuristic. The table above shows that each has at least one case where it clearly excels. Since all their heuristics are asemantical, we may wonder how this family of approaches performs compared to manual testing. This is shown in Figure 4. For each CUT we select the tool that delivers the best coverage on the CUT. We take the coverage number of this tool as representing the performance of the asemantical family and show it on the graph, along with the coverage of human written test suite for that CUT. For ease of viewing, the CUTs in Figure 4 are sorted in the increasing order of the tools' 'collective' coverage on them. There are 25 CUTs where the best tools deliver at least 60% coverage. In fact, on 19 CUTs the tools deliver at least 80%. They do not outperform the human in all CUTs. The two sides do complement each other: out of the set where the tools deliver at least 60%, on 18 CUTs they outperform human by a margin of at least 10 %, and on the other hand, there are 6 CUTs where human delivers at least 60% and outperforms tools by a margin of at least 10%.

5 CONCLUSION AND FUTURE WORK

There are cases where T3 and likewise all four tools deliver at least 60% branch coverage as well as significantly outperform manual testing. But they do not outperform human in all cases, and perhaps it is also not reasonable to expect them to. Considering the used classes are real life classes, and the tools have been used without CUT-specific customization, it would be well justified to conclude that asemantical tools, especially when used together, would do well to complement manual testing. While Evosuite's heuristic outperforms other tools in more cases, every tool does have cases where they outperform the others, suggesting that it is hard for a single algorithm to be good in all cases. This seems to be particularly true for asemantical approaches: their success would be quite influenced by how well their specific heuristics match with the CUT's semantic. Without semantical information, instrumentation becomes important since it provides the only means for the tool to know what happens inside a method when it is called. Randoop's lower performance can perhaps be blamed to its lack of instrumentation (it is the only tool that does not have it). If we follow this conjecture, it would be interesting to try to employ deeper instrumentation e.g. a la [4] that would allow arbitrary monitoring code to be inserted in any part of a target method, for example to keep track which fields of the CUT that part modifies.

We did not actually measure the coverage obtained if we would use *all* test suites generated by the tools (instead, we only take the best test suites), since this information was not measured by the Contest, nor the coverage we would get if we combine tools' test suites with human test suites. This would give better insight on

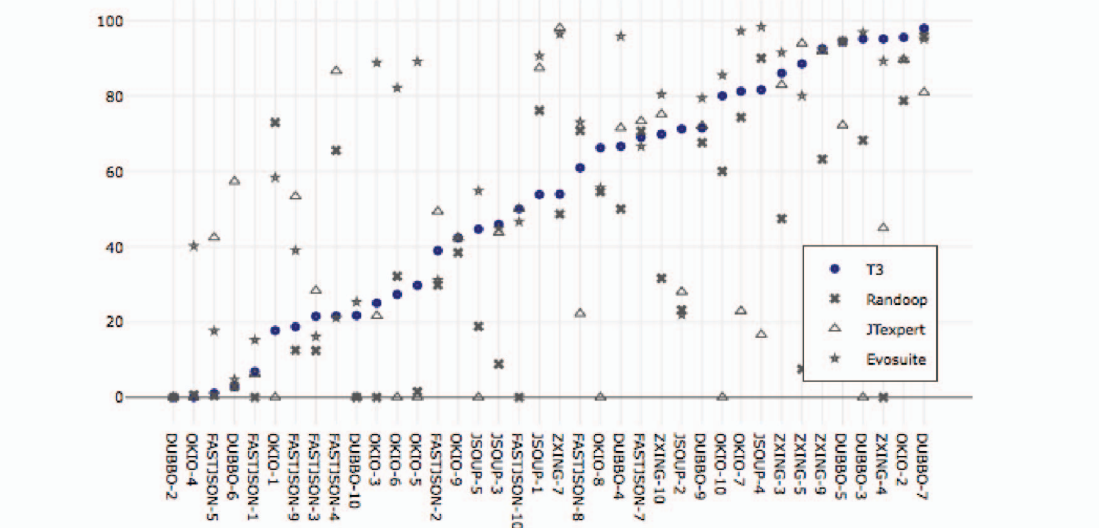

Figure 3: T3's delivered branch coverage, compared with Randoop, JTeXpert, and Evosuite.

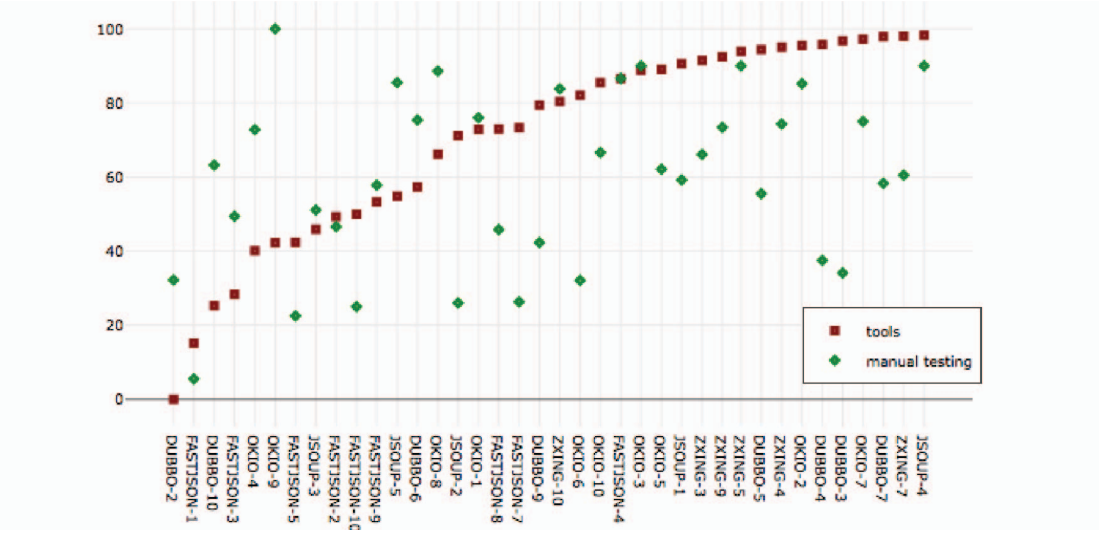

Figure 4: Best tools branch coverage, compared with manually written test cases.

the tools' combined performance, and how complementary they are with the human. We would recommend this as the future work. Additionally, comparison with semantic-aware tools like CUTE [10] would give valuable insight.

REFERENCES

[1] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[2] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419, 2011.

[3] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking java programs via guarded commands. In *ECOOP Workshops*, pages 110–111, 1999.

[4] A. Middelkoop, A. Elyasov, and I. Prasetya. Functional instrumentation of action-script programs with ASIL. In *Proc. Symposium on Implementation and Application of Functional Language (IFL)*. Springer, 2011.

[5] U. R. Molina, F. Kifetew, and A. Panichella. Java unit testing tool competition - sixth round. In *11th Int. Workshop on Search-Based Software Testing*. ACM, 2018.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th Int. Conf. on Software Engineering*, pages 75–84. IEEE, 2007.

[7] I. S. W. B. Prasetya. Budget-aware random testing with T3: benchmarking at the SBST2016 testing tool contest. In *9th Int. Workshop on Search-Based Software Testing*, pages 29–32. ACM, 2016.

[8] I. S. W. B. Prasetya, T. E. J. Vos, and A. Baars. Trace-based reflexive testing of OO programs with T2. *1st Int. Conf. on Software Testing, Verification, and Validation (ICST)*, 2008.

[9] A. Sakti, G. Pesant, and Y. Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2015.

[10] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[11] N. Tillmann and J. de Halleux. Pex–white box test generation for .NET. In *Tests and Proofs*, pages 134–153. Springer, 2008.