Budget-aware Random Testing with T3: Benchmarking at the SBST2016 Testing Tool Contest

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

ABSTRACT

Random testing has the advantage that it is usually fast. An interesting use case is to use it for bulk smoke testing, e.g. to smoke test a whole project. However, on a large project, even with random testing it may still take hours to complete. To optimize this, we have adapted an automated random testing tool called T3 so that it becomes aware of the time budget we set for a given target class. Test suites are now generated incrementally, and their refinements are adaptively scheduled towards maximizing the coverage, given the remaining time. This paper presents an evaluation of the performance of this adaptation, using the benchmark provided by the SBST 2016 Java Unit Testing Tool Contest.

Keywords

automated testing java; benchmark testing tools; unit testing; budgeted testing

1. INTRODUCTION

T3 [6, 7] is a light weight automated unit testing tool to test Java classes —it is the successor of T2 [9]. Table 1 shows T3's general profile. T3 is mostly random-based, so it is pretty fast. In an ideal case, thousands of test sequences can be generated in just few seconds. An interesting use case is to use it for bulk smoke testing, e.g. to smoke test a whole project. However, since T3 generates test sequences on the fly (that is, to know if a test sequence is valid, it executes it in-memory), the computation generated by the class under test (CUT) can greatly influence the performance. For example, with respect to the SBST2015 Benchmark [11], consisting of 63 real world classes, T3's average time to generate test suites is 48.5 seconds per CUT. Although this is the fastest among the participating tools in SBST2015, it still means that T3 needs almost a full hour to complete the whole set of benchmark. It only delivers in average 41% branch coverage, which is 0.91 of the coverage delivered by manual tests. In comparison, Evosuite [3], which is based on an evolutionary algorithm, performs (in terms of branch coverage) 1.08 times beter than manual tests, but it uses 7 times more time to generate the test suites.

Prerequisites	
Static or dynamic	dynamic + light static analysis
Software Type	Java.
Lifecycle phase	Mainly for unit testing.
Environment	Java
Knowledge & experience required	Java programming.
Input and Output of the tool: java classes (in), binary test suites (out).	
Operation	
Interaction	Used as a command line tool, or as a library, or interactively with Groovy shell.
Source of information	User manual [1]
Maturity	Released.
Technology behind the tool	Java 8, Groovy
Obtaining the tool and information: freely, GPL3 license, no dedicated customer support.	
Empirical evidence about the tool : [6, 7]	

Table 1: T3 profile.

We have since then improved T3 by including static analysis to gain knowledge on the literals that occur in a CUT, and by feeding back coverage information to direct T3's generators a la GRT [4]. So, T3 is now aware when a method in the CUT is not covered yet, and can keep on targeting it until the coverage is deemed enough. Since such a simplistic strategy can easily lead to starvation, we then implement a budgeting algorithm. Given a time budget T to test the CUT, the algorithm controls T3 to generate test suites incrementally. As it does so, it keeps track of the remaining budget. The suites are refined in rounds, which are dynamically scheduled. At the beginning, the refinement order is roughly random. After some time, the algorithm will switch to a more strategical scheduling, by preferring goals it considers easier, and in this way tries to maximize the utilisation of the remaining budget. This paper presents an evaluation for these improvements, using the benchmark provided by the SBST 2016 Java Unit Testing Tool Contest [10].

2. TEST SUITES GENERATION

We modify T3's algorithm [7] for generating test suites; it now works as follows. Suppose C_0 is the CUT, and time budget B_0 is given. If C_0 has static inner classes, T3 will also target each inner class, and B_0 is spread over all targets, proportional to their 'complexity' —the number of constructors and methods is used as an indicator of this. Since a target

SBST16, May 16-17, 2016, Austin, TX, USA

ACM ISBN 978-1-4503-4166-0/16/05.

DOI: http://dx.doi.org/10.1145/2897010.2897019

may use less than its allocated budget, after the completion of each target we recalculate the distribution of the budget over the remaining targets. Suppose a class C is now an elementary target for T3, with budget B. T3 does the following:

1. A set P_C of 'prefixes' is first generated. These are sequences of calls to C's constructors and mutators intended to create various instances of C.

2. T3 determines its *testing goals* and put them in a *worklist*; by default these are all C's protected and public constructors and methods.

3. As long as the worklist is not empty, and the budget B is not exhausted, goals are removed from the worklist, in some order, and processed.

   Suppose m is a goal selected for processing. For simplicity, assume m is a non-static method. (a) If this is the first time m is processed, T3 generates a test suite T_m, consisting of test sequences; each has the form of $\sigma{+}{+}o.m(..){+}{+}\tau$ where $\sigma{\in}P_C$ and is used to create an instance o of C. Then, m is invoked on o. The suffix τ is intended to check the state of o after m, by calling non-mutators methods of C on o. All parameters needed along the sequence are generated randomly. The size of T_m is set to be linear to $|P_C|$.

   (b) If this is not the first time m is processed, so it already has a T_m, a new suite T'_m is generated. For each sequence in T'_m, we move it to T_m if it improves the code coverage of current T_m. This process is called *refinement*.

   In both cases, (a) and (b), if the delivered code coverage is below a certain limit, m is put back in the worklist for further refinement. However, there is an upper bound R_{max} in how many times m can be put back in the worklist. We will later discuss the importance of this decision.

Generated test suites are saved (in files), and can replayed and queried later to analyze failures found. The test suites can also be used for regression test; in this case T3 will generate and inject oracles into the suites.

The order with which goals are taken our from the worklist is at first random. After some portion of B is consumed, currently $0.5B$, "easier" goals are favored. A method m is considered easier than m' if it has less number of parameters; if this number is the same, then if m has currently less code coverage than m'.

The number of times a goal has been refined is called the *generation* number of the goal. To prevent starvation, we impose that no goal of generation $k{+}1$ can be refined, until all goals of generation k that are currently awaiting in the worklist have been processed.

When all goals of generation k have been processed, but there are still goals left in the worklist (which must be of higher generation), we also grow P_C. We first generate K new prefixes. However, only prefixes that improve the code coverage of P_C or produce instances of C that are different than those generated by P_C (with respect to some concept of structural equivalence) will be added to P_C.

In comparison, the original algorithm [7] creates each T_m in one go, rather than staging it in multiple rounds of refinement. Moreover, goals are processed in a fixed order. The original algorithm also defines a goal to be a pair of methods, thus enforcing pair-wise testing. This is in principle stronger, but is also much more expensive. Here, we do not do that, hoping that the combination of each goal m with prefixes and suffixes will cover most, if not all, pairs.

3. EXPERIMENT SETUP AND RESULTS

The SBST 2016 Java Unit Testing Tool Contest provides a benchmark of 80 real world classes. A testing tool is expected to generate a regression test suite to test each of these classes. The classes are assumed to be correct. The strength of the generated suites are measured in terms of code coverage and fault finding capability. The latter is measured via mutation test. Errors are injected as mutations, and the ability of the test suites to detect these errors is measured (mutation coverage). All errors, except one, are artificial.

The benchmark is run in several rounds of varying time budget, namely 60, 120, 240, and 480 seconds per target class. Notice that the last case can potentially takes 10 hours to run.

To speed up its execution, T3 does not save test sequences in JUnit classes. Instead, it saves test suites as serialized objects (binary), so that they can be readily loaded into JVM and run. However, the benchmark requires generated test cases to be presented as test methods in JUnit classes. For each testing goal m (see Section 2), we create thus a proxy JUnit class that simply loads and runs its binary test suite T_m. To be able to detect mutations, each generated test sequence is extended with oracles. Since as mentioned above the target class is assumed to be correct, these oracles simply assert some selected values along the sequence. No specification mining e.g. a la [2] is applied to save budget. For mutation test, the benchmarking tool will remove flaky tests. Suppose a method $k()$ in a test sequence σ returns an integer value v. To avoid generating flaky tests, T3 does not literally assert v as the expected return value of $k()$ in σ. Instead, it only asserts the magnitude of x (defined as $\lceil log(x/100) \rceil$). Still, the benchmarking tool removes flaky tests at the JUnit test method level, whereas T3's units of tests are test sequences. This may result in an entire test suite T_m to be rejected if there is a single test sequence in T_m that turns out to be flaky. To mitigate this, T_m is invoked three times in three test methods, each activates only a certain type of oracles.

There are four tools participating in the Contest: T3, Randoop [5], Evosuite [3], and JTexpert [12], which allows us to compare the results. Randoop is purely random-based; Evosuite uses an evolutionary algorithm; JTexpert uses a search heuristic. The Contest also defines a scoring function, which produces a single value that summarizes various other statistics. This function is used to determine the contest winner, see [10].

The tools are not allowed to be custom configured for the target classes. For this reason, the classes were kept secret until the start of the contest. Unfortunately, this has the consequence that the tools crashed on some but different targets. In the case of T3, a bug in the new algorithm caused it to fail on targets which are abstract classes or interfaces. Table 2 shows the results on the target classes (22) on which no tool failed, and on which the benchmarking tool itself had no issue. The classes are sorted in increasing number of conditional nodes they have. We will refer to this subset as *Set-22*. Due to limited space, we do not include the results

of JTexpert; see [10].

On this Set-22, T3 performs consistently better than the pure random-based tool Randoop. There are cases where T3 performs very well, for example GrayPaintScale for which T3 delivers 100% condition coverage in 14 seconds (Evosuite delivers 100% as well, but uses 59 seconds). On ArrayUtils, which contains over 800 conditions, T3 delivers 97% condition coverage in 60 seconds (Evosuite gives 58% in 460 seconds). There are also some cases that T3 performs poorly, e.g. SimplexTableau (T3 4.4%, Evosuite 96%) and PeepholeFoldConstants. The last case is disputable, and should perhaps be disregarded: all tools appear to perform badly on this case, whereas T3's internal coverage measurement indicates high class coverage (>90%).

The rows "overall average" in Table 2 show the achieved coverage over the whole set[1]. Over the whole Set-22, T3 can thus deliver 59% condition coverage and 74% mutation kills in 1062 seconds (17.7 minutes), which is improved to 63% condition coverage by using 26.3 minutes. If we disregard the controversial PeepholeFoldConstants, T3 gives 69% condition coverage in 16.6 minutes, and 74% in 25 minutes. In comparison, Evosuite delivers 76% condition coverage, using 4.5 hours. We can also see that whereas other tools are eager to use up the given time budget, T3 tends to be conservative. E.g. on the 480s budget, it only uses on average 126 seconds per CUT. This behavior is controlled by the variable R_{max} that specifies the maximum number of times a test suite can be refined (see Section 2). R_{max} was set to 8. Setting it to a higher value will cause T3 to behave more eagerly. Whether we should do so is open for discussion. On the 60s budget, T3 should have indeed behaved more eagerly, since the productivity was still high. However, when the productivity has dropped too low, insisting on further generation would be wasteful The rows "productivity" show *additional % coverage gained per additional minute spent*. For example on the 120s and 240s budgets we see that on the latter Randoop's conditional coverage productivity dropped from 0.14 to 0.06. This means that from this point on, it needs to invest 16 minutes to get further 1% improvement. Actually, it will be more expensive since the more coverage we get, the harder it is to improve it. So, the productivity will drop further as we proceed. However, Randoop does not implement budget management beyond a simple time out, so it simply uses all the budget given (even exceeding it). On the 240s budget it already spent in total 5493 seconds (1.5 hour). When given twice the budget, it went on spending 3 hours, despite that the productivity drops even further to 0.03. In comparison, on the 240s budget T3's conditional coverage productivity drops from 0.51 to 0.15. It spent in total 34 minutes, which is already much shorter than the other two tools, and delivering beter coverage. When the budget is doubled to 480s, it restrains from simply doubling the effort, and stops after 46 minutes. Afterwards, this is justified, because by then its productivity has dropped further to 0.06.

4. CONCLUSION AND FUTURE WORK

Enhanced with light weight static analysis and directed random testing, T3 can deliver decent coverage on real life target classes. This can be further improved by spending effort to customize T3 to its given targets —a convenient way for the user to specify custom value generators is provided[8]. T3 is also budget aware. It splits the given target class into multiple test goals, and tries to incrementally and strategically spread the given time budget over these goals. On bigger budget this results in much less wasteful behavior than the other tools. On smaller budget, it however stops too early, while it is still productive. Improving the budgeting algorithm is future work. The performance can also be improved. Using up 30 - 45 minutes for smoke testing a project of 22 classes is still unpractical. In theory it is possible to handle the test goals in parallel if we have multi cores. This may greatly reduce the time consumption. This too is future work.

[1]In particular, it does *not* refer to the average over the coverage percentages of the CUTs; this would ignore that some CUTs are larger than the others.

5. REFERENCES

[1] T3 site. https://git.science.uu.nl/prase101/t3.

[2] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

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

[4] L. Ma, C. Artho, C. Zhang, H. Sato, M. Hagiya, Y. Tanabe, and M. Yamamoto. GRT at the SBST 2015 tool competition. In *8th Int. Workshop on Search-Based Software Testing*, pages 48–51, 2015.

[5] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conf. on Object-oriented programming systems and applications (OOPSLA)*, pages 815–816. ACM, 2007.

[6] I. S. W. B. Prasetya. T3, a combinator-based random testing tool for Java: Benchmarking. *Int. Workshop Future Internet Testing*, 2014.

[7] I. S. W. B. Prasetya. T3: Benchmarking at third unit testing tool contest. In *IEEE/ACM 8th Int. Workshop on Search-Based Software Testing (SBST)*. IEEE, 2015.

[8] I. S. W. B. Prasetya. T3i: A tool for generating and querying test suites for java. In *10th Joint Meeting of ESECï££ and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2015.

[9] 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.

[10] U. Rueda, R. Just, J. Galeotti, and T. Vos. Unit testing tool competition – round fout. In *ACM/IEEE 9th International Workshop on Search-Based Software Testing (SBST)*. ACM/IEEE, 2015.

[11] U. Rueda, T. Vos, and I. S. W. B. Prasetya. Unit testing tool competition – round three. In *IEEE/ACM 8th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2015.

[12] 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.

					$B = 60s$			$B = 120s$			$B = 240s$			$B = 480s$		
CUT	L	C	M	tool	$\%C$	$\%M$	T_G	$\%C$	$\%M$	T_G	$\%C$	$\%M$	T_G	$\%C$	$\%M$	T_G
GrayPaintScale	25	10	37	ran	80.0	75.7	64	80.0	75.7	124	80.0	75.7	244	80.0	75.7	484
				t3	100.0	91.4	14	100.0	92.8	14	100.0	91.0	14	100.0	91.4	15
				evo	100.0	91.9	59	100.0	65.3	100	100.0	73.4	183	100.0	82.9	323
ShapeList	34	16	28	ran	43.8	17.9	64	43.8	17.9	124	43.8	17.9	246	43.8	17.9	490
				t3	50.0	25.0	24	50.0	25.0	25	50.0	25.0	23	50.0	25.0	24
				evo	37.5	15.5	60	49.0	23.8	114	49.0	19.0	187	50.0	20.2	327
NumericEntityUnescaper	25	16	80	ran	6.2	10.2	64	6.2	10.2	126	6.2	10.4	249	6.2	10.2	495
				t3	18.8	15.8	14	18.8	15.4	14	18.8	13.5	15	14.6	15.4	16
				evo	6.2	8.3	59	27.1	17.7	95	51.0	24.2	165	79.2	49.8	305
ProperFractionFormat	63	19	46	ran	36.8	4.3	63	36.8	4.3	124	36.8	4.3	245	57.9	30.4	487
				t3	57.9	37.0	61	57.9	37.0	77	57.9	37.0	76	57.9	37.0	71
				evo	46.5	25.0	60	70.2	32.2	102	76.3	40.9	171	70.2	35.1	308
HypergeometricDistribution	66	26	175	ran	84.6	76.3	68	100.0	97.2	132	100.0	98.5	257	100.0	98.7	664
				t3	100.0	79.5	37	100.0	79.2	58	100.0	81.5	57	100.0	81.3	45
				evo	96.8	39.3	66	96.2	84.8	118	98.7	89.5	226	99.4	90.8	375
MultidimensionalCounter	70	32	138	ran	15.6	10.1	66	15.6	10.1	126	15.6	10.1	247	15.6	10.1	488
				t3	97.4	77.2	28	95.3	76.9	51	99.5	81.0	42	97.9	80.4	40
				evo	94.3	48.8	64	95.8	56.2	121	98.4	74.3	233	98.4	74.4	429
TimePeriodValues	149	54	254	ran	74.1	53.1	65	85.2	61.0	127	87.0	64.2	250	88.9	66.9	493
				t3	94.4	59.9	56	94.4	60.5	58	94.4	60.9	52	94.4	59.9	63
				evo	38.3	21.7	62	89.2	63.9	121	95.4	67.3	232	98.5	61.2	376
MutableDateTime	247	63	90	ran	67.5	14.3	65	70.9	30.9	126	75.1	32.8	249	79.6	55.4	494
				t3	51.9	19.4	65	68.3	38.5	125	78.3	65.4	226	87.6	90.6	376
				evo	79.6	35.4	64	84.4	47.6	123	90.5	55.9	236	96.8	57.6	449
Period	288	64	467	ran	52.1	2.7	66	56.8	3.2	127	57.8	3.2	251	58.9	3.2	493
				t3	88.3	63.5	69	79.9	55.0	129	92.4	67.2	248	96.9	78.8	489
				evo	81.0	25.1	69	84.4	45.1	125	90.6	73.4	241	93.2	82.2	465
LocaleUtils	76	64	177	ran	59.4	20.9	63	59.4	20.9	123	59.4	20.9	244	59.4	20.9	485
				t3	70.3	41.9	16	71.1	42.6	16	70.3	41.6	17	70.8	42.5	16
				evo	75.0	40.5	61	70.8	39.2	112	66.4	37.7	179	70.1	38.4	319
DateTimeFormatter	208	88	176	ran	67.6	20.1	65	75.4	20.8	127	78.4	22.3	249	79.7	25.6	494
				t3	19.3	10.9	68	21.6	11.7	117	24.8	13.1	149	25.0	13.0	149
				evo	84.1	39.6	65	80.3	46.1	125	84.1	54.6	237	87.5	59.3	436
Fraction	149	90	360	ran	48.9	33.3	78	73.3	53.9	141	71.1	54.2	282	78.9	62.3	527
				t3	83.3	61.9	33	83.5	62.2	35	83.5	62.2	35	83.3	62.0	40
				evo	94.1	58.7	64	95.9	69.1	122	97.4	70.8	226	98.5	70.0	348
SimplexTableau	131	90	392	ran	4.4	1.0	66	4.4	1.0	125	4.4	1.0	248	4.4	1.0	669
				t3	4.4	1.0	63	4.4	1.0	74	4.4	1.0	78	4.4	1.0	81
				evo	2.2	0.5	60	74.8	41.6	119	86.9	52.8	228	96.1	57.1	401
Partial	249	106	336	ran	34.9	11.2	65	37.3	11.0	127	48.7	14.3	249	57.1	24.1	493
				t3	93.9	78.4	54	98.6	81.1	91	98.6	80.3	100	98.3	79.0	122
				evo	55.5	29.6	63	61.0	38.7	122	87.1	67.9	234	93.6	73.9	436
Rotation	327	122	2706	ran	29.5	5.8	66	29.5	5.9	124	31.1	7.8	245	32.7	9.6	488
				t3	48.2	60.2	57	47.7	57.2	67	48.6	61.3	63	48.0	63.6	62
				evo	45.5	52.8	65	59.0	58.8	122	88.7	79.5	238	91.5	82.8	415
DefaultIntervalCategoryDataset	186	128	273	ran	53.0	51.1	65	54.7	53.6	127	58.6	54.7	251	60.5	56.3	495
				t3	32.4	22.7	59	32.0	22.3	59	34.1	24.8	66	35.2	24.3	64
				evo	74.6	47.3	63	78.1	48.6	119	80.1	52.7	231	85.0	55.4	412
DateUtils	208	175	406	ran	67.4	33.8	63	69.1	33.9	124	70.3	33.7	245	71.4	36.2	487
				t3	90.3	53.4	34	90.0	55.5	35	90.2	53.2	32	90.3	55.3	34
				evo	56.6	28.1	62	81.3	35.7	119	87.9	51.9	221	93.6	55.3	367
PeriodFormatterBuilder	665	458	1139	ran	56.6	27.9	65	59.7	27.8	126	63.7	29.7	248	68.7	36.0	492
				t3	21.1	13.9	69	41.3	21.5	129	43.5	23.5	249	46.4	24.0	489
				evo	29.0	11.9	66	31.0	16.8	124	44.6	25.1	238	54.5	30.9	463
StrBuilder	656	464	1584	ran	75.1	44.8	64	80.9	56.8	124	83.8	65.3	246	86.9	71.2	488
				t3	91.7	79.0	59	97.5	86.4	119	97.6	86.1	140	97.6	86.0	146
				evo	67.2	23.8	71	72.9	34.0	127	77.0	44.9	242	81.3	48.6	480
DateTimeFormatterBuilder	1034	595	1743	ran	51.1	20.4	65	51.7	20.7	126	57.9	28.3	248	62.4	25.9	493
				t3	63.2	28.9	61	70.8	36.1	119	74.5	37.9	203	75.4	38.4	262
				evo	25.3	9.3	70	25.5	11.2	129	52.6	27.7	244	60.9	39.4	469
PeepholeFoldConstants	765	624	880	ran	0.5	0.1	66	1.1	0.1	128	1.1	0.1	253	1.1	0.1	497
				t3	3.7	1.2	64	3.6	1.2	74	3.1	0.9	72	3.2	1.1	74
				evo	0.2	1.9	64	0.2	2.0	119	5.0	2.5	239	6.5	3.2	445
ArrayUtils	957	806	2015	ran	91.2	73.1	64	94.9	78.7	125	96.2	81.2	247	97.4	82.8	489
				t3	97.0	96.9	60	98.1	96.3	93	97.9	97.3	95	98.1	96.6	100
				evo	61.5	24.4	75	66.5	35.0	124	72.2	56.6	238	75.1	58.2	460
overall coverage				ran	54.0	64.1		57.2	67.2		59.7	68.8		62.3	70.5	
				t3	59.2	74.4		63.6	76.9		64.8	77.9		65.5	78.0	
				evo	44.1	63.1		50.2	69.5		60.6	80.0		65.5	83.4	
total generation time				ran			1439			2785			5493			11181
				t3			1062			1579			2052			2780
				evo			1410			2601			4870			8805
productivity ($cov/minute$)				ran				0.14	0.14		0.06	0.04		0.03	0.02	
				t3				0.51	0.29		0.15	0.13		0.06	0.01	
				evo				0.31	0.32		0.28	0.28		0.07	0.05	

Table 2: ***Results, on a subset of CUTs where no tool crashes.*** *L and C are the number of lines and conditional nodes in the CUT. M is the number of injected mutantions in the CUT. B is the allocated time budget (per CUT). $\%C$ and $\%M$ are the delivered condition and mutation coverage (strong kills). T_G is time used for generating test suites (in seconds).*