JTExpert at the Third Unit Testing Tool Competition

Abdelilah Sakti, Gilles Pesant, Yann-Gaël Guéhéneuc
Department of Computer and Software Engineering
École Polytechnique de Montréal
Montréal, Quebéc, Canada
Email: {abdelilah.sakti, gilles.pesant, yann-gael.gueheneuc}@polymtl.ca

***Abstract*—JTExpert is a software testing tool that automatically generates a whole test suite to satisfy the branch-coverage criterion on a given Java source code. It takes as inputs a Java source code and its dependencies and automatically produces a test-case suite in JUnit format. In this paper, we summarize our results for the Unit Testing Tool Competition held at the third SBST Contest, where JTExpert receives 159.16 points and was ranked sixth of seven participating tools. We discuss the internal and external reasons that were behind the relatively poor score and ranking.**

***Keywords*-— test-case generation; classes testing; unit testing; random testing; static analysis.**

I. Introduction

This paper describes and discusses the results obtained by applying the test-case generation tool JTExpert [4] on the benchmarks used to compare participating tools in the unit-testing competition held as apart of the International Workshop on Search Based Software Testing (SBST) held in Florence, Italy, from 18 to 19 May 2015. More details on the competition and the benchmarks can be found in [3]. In this competition, JTExpert receives 159.16 points and was ranked sixth.

II. JTExpert

JTExpert is a software testing tool that has been developed to automatically generate a whole test suite that satisfies the branch coverage criterion on a given Java source code [4]. Table I summarizes the main features of JTExpert. JTExpert automatically generates a JUnit [2] test suite for the Class Under Test (CUT). It can be used through a command line interface. It takes as inputs a Java file (.java) and its dependencies and automatically produces a test-case suite in JUnit format. JTExpert is available as an executable Jar file. It is based on four main components: a source code analyzer, a test case candidates builder, an instances generator, and a random search strategy.

Table I
Features of the tool JTExpert

Prerequisites	
Static or dynamic	Dynamic testing
Software Type	Java source code (.java)
Lifecycle phase	Unit testing for Java projects
Environment	Java
Knowledge required	JUnit [2]
Experience required	unit-testing knowledge
Input and Output of the Tool	
Input	A Java source code and its dependencies
Output	A test-case suite in JUnit 4 format
Operation	
Interaction	Through the command line
User guidance	
Source of information	https://sites.google.com/site/saktiabdel/JTExpert
Maturity	Still under development
Technology behind the tool	Random testing guided by static analyses
Obtaining the tool and information	
License	
Cost	Free
Support	None
Empirical evidence about the Tool	
Completeness	
Effectiveness	see [4]
Efficiency	see [4]
Defect types	
Scalability	see [4]
Comprehensibility	
Learnability	
Subjective satisfaction	
Other	

A. Source Code Analyzer

JTExpert uses the Source Code Analyzer (SCA) to determine the set of methods that are likely to change the state of a given data member and the set of methods that may reach a given branch. The SCA analyzes the source code to collect constants and path information to reach a given branch. SCA provides JTExpert others components with relevant information to guide them throughout the process of test-case generation.

B. Test Case Candidates Builder

JTExpert uses the Test Case Candidates Builder (TDCB) to explore only relevant sequences of method calls. Using the collected information by SCA, the test-case generation problem is represented by a vector composed of relevant means-of-instantiation of the CUT, methods that are likely to change the object state by changing a data member, and methods that may reach the branch target. Thus, JT-Expert represents a test-case candidate by: (1) a means-of-instantiation of the class under test (i.e., a constructor, a method factory, a data field, or an external method from the CUT); (2) a sequence of method calls whose length (i.e., number of method calls) is bounded by the number of declared data members in the CUT, each method in a sequence being called in the hope to put a given data member in a relevant state; (3) a method call that is likely to reach the test target; (4) a means-of-instantiation for each needed argument.

The TDCB is a key novelty of JTExpert compared to other tools because it allows JTExpert to avoid exploring useless sequences and thus to generate tests faster without compromising the coverage.

C. *Instances Generator*

JTExpert uses a customized instance generator that is based on a seeding strategy and a dynamic strategy to diversify generated instances of classes. The seeding strategy gets collected constants for each primitive data type or string, then seeds them while generating data. It defines a seeding probability of each data type according to the number of collected constants. Also, it seeds the null value with a constant probability while generating instances of classes. The diversification strategy aims to generate different types of needed instances by using different means-of-instantiation (e.g., constructors, factory methods, subclasses).

The instance generator allows JTExpert to improve the exploration of the search space, reaching more branches, and thus increasing code coverage in less time.

D. *Random Search Strategy*

JTExpert uses a random search that targets all uncovered branch at the same time: it does not focus on only one branch, instead it generates a test-case candidate uniformly at random for every uncovered branches. This strategy allows JTExpert to reach a good branch coverage quickly because it does not waste efforts on unreachable branches and it benefits from the significant number of branches that may be covered accidentally.

III. Benchmark results

The detailed results of JTExpert on all benchmarks are presented in Table II. On average, JTExpert achieved 46.38% instructions coverage, 41.48% branch coverage, and 29.51% mutation coverage. Overall, JTExpert produced 14 test cases for each class and took an average of 94.32 seconds to generate a suite of test cases for a given class.

A. *Class Loading*

The results are in line with our expectations except for the 0% score on the seven classes from the library guava [1]. While, such a score is expected for a hard class to instantiate (which guava is not), what we did not expect is the use of the guava library as benchmark to compare JTExpert to other tools. The problem is that JTExpert uses this library. JTExpert produced 0% coverage on the seven classes from this library (4,279 statements, 354 branches), not because these classes are hard to test, but because JTExpert loads the library guava itself before starting the process of test-case generation. Therefore, technically, the Java Virtual Machine prevents JTExpert from reloading an instrumented version of a class-under-test from guava or any other class that is already loaded. This is similar to trying to generate a test-case suite for the class `java.lang.String` using a Java-based tool. This is not possible without a special classes loader and put JTExpert at a disadvantage.

B. *Low Branch Coverage*

Besides the guava library, JTExpert produced low branch coverage on some classes, especially those form the Java Wikipedia Library (JWPL). Some classes from the JWPL require an instance of the class `Wikipedia` that needs an instance of the class `DatabaseConfiguration`. To instantiate the class `DatabaseConfiguration`, JTExpert must generate five strings representing a host name, a database name, a user name, a password, and a language. All these information must reflect an existing Wikipedia environment. To generate a string, JTExpert uses a random string generator that seeds extracted constants from the source code. Because JTExpert could not find any required strings in the source code and because randomly generating such strings is almost impossible, JTExpert failed to instantiate the CUT and generate their test cases. The low coverage score represents only the exceptions raised in constructors using `null` instances for the class `Wikipedia`.

JTExpert produced 0% on the class `AbstractLoader` from the library Checkstyle. This revealed a bug in JT-Expert. We accidentally deactivated a part of the code that generates stubs for abstract classes. Consequently, JT-Expert failed to instantiate this class and generate test cases. For the same reason, JTExpert achieved low coverage on the classes `ScopeUtils`, `AnnotationUtility`, and `AutomaticBean` (respectively 43%, 21%, and 11% branch coverage).

Finally, JTExpert produced 0% on the class `ExceptionDiagnosis` from the library Twitter4j but in our local environment JTExpert covers this class, so we could not determine the actual reason behind this low coverage.

After this analysis, we can confirm that JTExpert' Instance Generator may fail to instantiate classes or reach branches that require meaningful parameters, e.g.,

Table II
DETAILED RESULTS OF JTExpert ON THE SBST Contest BENCHMARKS.

Class Name	**AVG # Test Cases**	**Time**		**% Coverage**		
		Generation	**Execution**	**Instruction**	**Branch**	**Mutation**
com.google.gdata.data.AttributeHelper	40.50	3.41	0.15	57.97	70.73	40.87
com.google.gdata.data.DateTime	25.17	1.26	0.10	77.06	65.00	68.77
com.google.gdata.data.Kind	11.00	1.14	0.04	43.67	30.68	23.19
com.google.gdata.data.Link	30.83	3.06	0.10	70.85	55.63	12.78
com.google.gdata.data.OtherContent	13.17	2.13	0.05	40.06	35.98	10.83
com.google.gdata.data.OutOfLineContent	18.33	2.07	0.05	60.39	41.07	14.58
com.google.gdata.data.Source	27.67	2.46	0.10	26.79	8.33	6.52
net.sf.javaml.core.AbstractInstance	13.33	1.56	0.04	72.58	60.71	26.56
net.sf.javaml.core.Complex	7.83	0.17	0.03	100.00	0.00	5.91
net.sf.javaml.core.DefaultDataset	8.83	2.77	0.03	30.77	24.58	13.98
net.sf.javaml.core.DenseInstance	20.67	2.34	0.08	91.39	83.85	54.03
net.sf.javaml.core.Fold	11.17	3.05	0.04	37.41	28.33	17.71
net.sf.javaml.core.SparseInstance	19.33	1.43	0.07	98.14	84.90	50.00
net.sf.javaml.tools.data.ARFFHandler	2.00	0.51	0.01	38.98	25.00	17.59
twitter4j.ExceptionDiagnosis	1.00	0.37	0.01	0.00	0.00	0.00
twitter4j.GeoQuery	15.67	0.17	0.06	99.96	86.11	49.74
twitter4j.OEmbedRequest	19.17	0.66	0.07	95.74	83.64	46.17
twitter4j.Paging	20.00	0.62	0.07	94.48	94.91	44.89
twitter4j.TwitterBaseImpl	21.67	3.36	0.28	45.70	49.51	28.41
twitter4j.TwitterException	12.50	1.39	0.05	65.11	41.50	36.95
twitter4j.TwitterImpl	114.17	4.09	1.30	11.01	47.16	4.81
com.puppycrawl.tools.checkstyle.api.AbstractLoader	1.00	0.88	0.01	0.00	0.00	0.00
com.puppycrawl.tools.checkstyle.api.AnnotationUtility	8.17	0.99	0.03	50.34	43.33	36.36
com.puppycrawl.tools.checkstyle.api.AutomaticBean	3.33	1.53	0.02	44.40	21.43	11.63
com.puppycrawl.tools.checkstyle.api.FileContents	21.50	1.90	0.10	88.16	75.64	40.33
com.puppycrawl.tools.checkstyle.api.FileText	10.50	1.55	0.05	71.20	69.23	63.83
com.puppycrawl.tools.checkstyle.api.ScopeUtils	7.50	1.01	0.03	25.63	11.00	13.93
com.puppycrawl.tools.checkstyle.api.Utils	16.00	1.21	0.08	79.62	82.05	42.86
com.google.common.base.CharMatcher	1.00	0.54	0.01	0.00	0.00	0.00
com.google.common.base.Joiner	1.00	0.95	0.01	0.00	0.00	0.00
com.google.common.base.Objects	1.00	0.37	0.01	0.00	0.00	0.00
com.google.common.base.Predicates	1.00	0.89	0.01	0.00	0.00	0.00
com.google.common.base.SmallCharMatcher	1.00	0.69	0.01	0.00	0.00	0.00
com.google.common.base.Splitter	1.00	0.38	0.01	0.00	0.00	0.00
com.google.common.base.Suppliers	1.00	0.38	0.01	0.00	0.00	0.00
org.hibernate.search.SearchException	5.00	0.20	0.02	66.67	0.00	0.00
org.hibernate.search.Version	2.33	0.16	0.02	98.33	0.00	0.00
org.hibernate.search.backend.BackendFactory	9.33	0.90	0.05	84.21	81.25	70.00
org.hibernate.search.backend.FlushLuceneWork	4.67	0.38	0.02	99.15	91.67	70.83
org.hibernate.search.backend.OptimizeLuceneWork	4.50	0.38	0.02	100.00	100.00	75.00
org.hibernate.search.util.logging.impl.LoggerFactory	3.00	0.53	0.02	56.25	0.00	50.00
org.hibernate.search.util.logging.impl.LoggerHelper	3.00	0.18	0.02	100.00	0.00	22.22
de.tudarmstadt.ukp.wikipedia.api.CategoryDescendantsIterator	1.17	3.25	0.01	0.00	0.00	0.00
de.tudarmstadt.ukp.wikipedia.api.CycleHandler	2.00	2.53	0.01	5.57	0.00	0.00
de.tudarmstadt.ukp.wikipedia.api.Page	2.17	2.76	0.02	0.25	0.67	2.67
de.tudarmstadt.ukp.wikipedia.api.PageIterator	2.33	2.66	0.01	9.12	0.00	5.21
de.tudarmstadt.ukp.wikipedia.api.PageQueryIterable	1.83	2.84	0.01	8.40	3.88	0.00
de.tudarmstadt.ukp.wikipedia.api.Title	7.33	0.59	0.03	91.98	75.00	90.48
de.tudarmstadt.ukp.wikipedia.api.WikipediaInfo	2.00	2.87	0.01	1.92	2.00	0.00
org.asynchttpclient.AsyncHttpClient	23.67	3.58	0.16	66.86	45.14	53.85
org.asynchttpclient.AsyncHttpClientConfig	37.17	1.21	0.17	85.87	47.78	56.21
org.asynchttpclient.FluentCaseInsensitiveStringsMap	33.67	3.99	0.12	77.90	74.37	71.04
org.asynchttpclient.FluentStringsMap	30.33	3.42	0.11	77.41	73.45	69.19
org.asynchttpclient.Realm	38.00	0.77	0.13	89.15	51.69	35.07
org.asynchttpclient.RequestBuilderBase	24.83	3.85	0.16	72.50	53.55	53.99
org.asynchttpclient.SimpleAsyncHttpClient	36.33	2.71	0.38	84.96	57.65	47.39
org.scribe.model.OAuthConfig	7.83	0.10	0.03	91.14	100.00	66.67
org.scribe.model.OAuthRequest	4.83	0.51	0.02	100.00	79.17	80.00
org.scribe.model.ParameterList	10.67	0.43	0.04	100.00	95.37	80.43
org.scribe.model.Request	18.83	0.80	0.07	57.54	45.45	37.98
org.scribe.model.Response	2.00	0.42	0.01	1.77	0.00	0.00
org.scribe.model.Token	10.83	0.14	0.04	95.80	77.08	59.09
org.scribe.model.Verifier	2.00	0.07	0.01	100.00	0.00	41.67
Average per class	**13.60**	**89.04(s)**	**4.57(s)**	**54.60%**	**39.29%**	**30.51%**

`Source$SourceHandler` requires two Strings containing a `namespace` and a `localName`.

C. Low Mutation Coverage

The mutation coverage is low compared to the code coverage because JTExpert has been developed to generate test data that has a high level of branch coverage and the version used for the Contest is the first attempt to generate test cases with oracles. JTExpert component that automatically generates the oracle is not yet completely developed. Its current version is based only on primitive types and methods returning values: during the test-data generation process if the method under test returns a primitive value then this value is used in an assertion statement.

To better understand where we should direct our effort to enhance JTExpert' Oracle Generator Component (OGC), we carefully analyzed the classes where we observed a large difference between branch coverage and mutation coverage.

In the classes `Kind`, `OtherContent`, and `OutOfLineContent` from the library Gdata, we observed a significant difference between branch coverage (55.63%, 35.98%, 41.07%) and mutation coverage (12.78%, 10.83%, 14.58%). In these classes, we found that almost all methods return either an object or void that the OGC can not handle. Therefore the OGC could not generate enough assertions to kill mutants in such classes. Also, we observed the same behavior on the classes `AbstractInstance`, `Complex`, and `DenseInstance` from the Javaml library.

After this analysis, we can confirm that the current version of the OGC could not generate enough assertions to kill mutants and get a mutation coverage aligned with the branch coverage.

IV. Discussions

The poor score is not only the result of JTExpert but also of our ignorance of the Contest rules. To win the competition, in addition to a good tool, we needed information about the way a score is computed. Unfortunately, we did not get any information about the way the score is calculated until the end of the Contest. Consequently, we wrongly focused both on the running time and the branch coverage while the only important criterion is the mutation coverage. One hundred percent of mutation coverage is equivalent to four hours execution time, no matter how many mutants are injected in the CUT [3]. Also we wrongly focused on large classes but in the contest they have exactly the same weight, i.e., a class containing one instruction and one mutant injected has exactly the same weight as a class containing one million instructions and one million mutants injected which may favor "lazy" tools: let us suppose that we have two CUTs $c1$ and $c2$ and two competing tools $t1$ and $t2$; $c1$ contains two branches, two statements, and two mutants injected; $c2$ contains 200 branches, 200 statements, 200 mutants injected; if $t1$ reached 100% coverage on $c1$ after 1 hour and 0% on $c2$ then its score is 6 [3]; if $t2$ reached 50% coverage on $c2$ after 1 minute and 0% on $c1$ then its score is 3.48 [3]; Therefore, the tool $t1$ that covered only two instructions, two branches, and killed 2 mutants after one hour receives a better score than the tool $t2$ that covered 100 instructions, 100 branches, and killed 100 mutants in one minute. Therefore, the score does not reflect neither the standard mutation coverage nor the standard code coverage. It favors tools that choose a class and kill high percentage of injected mutants no matter how many mutants or if the class is challenging or not. This is one reason why manual testing was ranked first. It seems difficult to draw any conclusion based on such a score. The way score is computed should be deeper discussed to make future editions of the competition more interesting.

V. Conclusion

The way the score is computed, the use of the Guava library as benchmark, and the reasons explained at Subsections III-B and III-B are the main internal and external factors that negatively affected our score and ranking.

In the software testing area, developing an ideal tool for test-case generation is a dream that many researchers share. The SBST Contest 2015 is a stepping stone towards achieving our dream. Despite low results, the competition provides a unique opportunity to compare JTExpert to other tools and to identify its lucks. Also, the competition allowed determining new research directions to enhance JTExpert in particular and software testing in general.

References

[1] Guava: The guava project contains several of google's core libraries. `http://code.google.com/p/guava-libraries/` (2013), [Online; accessed JUN-2013]

[2] JUnit: Junit is a simple framework to write repeatable tests. `http://www.junit.org` (2013), [Online; accessed JUN-2013]

[3] Molina, U.R., Vos, T., Prasetya, I.: Unit testing tool competition : Round three. In: Software Engineering, Search Based Software Testing Workshops (ICSEW), 2015 IEEE 37th International Conference on (MAY 2015)

[4] Sakti, A., Pesant, G., Guéhéneuc, Y.G.: Instance generator and problem representation to improve object oriented code coverage. IEEE Transactions on Software Engineering pp. 1–1 (To appear, 2015)