USING STATISTICAL DISTRIBUTIONS TO GENERATE RANDOM TEST DATA

Jamie Zimmerman

Thesis Defense

Sometime Spring, 2018

**Abstract**

Many open-source software programs lack rigorous, wide-reaching testing. This is primarily because the testing process is deeply dependent on human invention and manual writing. Therefore, writing system tests is often avoided because of its financial and time expense on the software development lifecycle. However, the data for some system tests can be generated with special statistical distributions, which can exhibit certain trends or patterns and therefore tease out specific behaviors in the program that warrant being checked. This project aims to automate the creation of those data points to streamline the process of the writing large tests and test suites.

**Introduction**

Software testing is a crucial part of systems that run our lives. Industries that rely on predicting consumer habit trends or dispatching taxi cars are deeply affected when their software fails to provide accurate results. The gravity of this trust is even greater in safety-critical programs controlling gas-leak shut off valves or anesthetic delivery machines. Consequently, before these pieces of software are deployed, they must be checked and tested as rigorously as necessary, but no more. The software must eventually reach production (used in the hands of the customer or used in industrial practice “for real”), and cannot spend too much of its lifetime under test. Therefore, cost and convenience of testing is an important consideration in determining the rigor of testing.

A key component of getting correct results is knowing what that correctness looks like. Developers and testers must be careful not to confuse code that compiles and runs without errors with code that garners accurate results. To find accurate results, testers need an oracle, which is a way of determining how close the actual result of a program is to the expected result.

Unsurprisingly, attaining and checking this oracle is difficult. Given a particular input to a program, a tester must know what the expected result even is. Then, she must know how to read or understand the actual result and be able to compare it to the expected. Consequently, many companies often build the role of tester into the role of developer, since the developer has the most intimate knowledge of the design and implementation of the software, and therefore has a better sense of what the output of the program garners.

The breadth and completeness of testing is directly proportional to how much damage could occur if the software malfunctioned. However, there are some basic oracles and trends that can be tested that can offer high confidence that the program functions as anticipated. For example, it might be difficult to know what a piece of software should do, but significantly easier to know what the software should *not* do. Online banking software should not deposits millions of dollars into the account of a robber.

It is significantly easier to describe the trend of a certain test than it is to create the hundreds (if not more) data points that fit that description.

Moreover, unit testing is necessary but not sufficient to ensure the correct behavior of a system. As the scope of testing increases (from unit testing to integration testing to system testing), the confidence in knowing the *expected* result might decrease. This makes an oracle hard to write, since the tester does not know what she should check or to what she should compare the result.

Therefore, the goal of this paper is twofold. The first goal is to provide a framework that automates the generation of oracle properties, or provides some support for the tester to test concrete values. The follow up to this problem is to automate the injection of test data and oracle data into test code to eliminate the step of humans writing test code manually. The reality of the software development environment means that solutions must be adopted to ensure accurate software while still maintaining a practical timeline and budget.

**Literature Survey**

The essence of the oracle problem is best captured in the definition of non-testable programs, provided by Weyuker:

“A program should be considered non-testable [if] (1) there does not exist an oracle; (2) it is theoretically possible, but practically too difficult to determine the correct output”1

There are a variety of reasons why the oracle does not exist, or can be exercised in a practical capacity. Weyuker describes that some software systems are like magic calculators – they are meant to inform us of the answer. She additionally mentions that some programs produce output that is impossible to read, either because of its volume or complexity. Here lies the core of software testing challenges. It is impossible to solve for every oracle or be able to describe every detail of a complex oracle – every system is entirely different and no generic template could create a standardized oracle. However, it is possible to avoid oracles altogether, generate them partially, or give hints about what details are most important to check.

One method of avoiding oracles altogether is metamorphic testing, which exploits the relationships of different executions of different input data2, 3, 4. For example, upon white box inspection of a system, testers can see that outputs should relate directly to their inputs. They may not know much else about the system, or what its output means, but they know what differences they should see upon two different executions5. Upon a certain execution they get f(5) = 20, and since they know the input-output relates directly, then they should predict that f(6) > f(5). Though exceptional in avoiding oracles, ascertaining metamorphic relationships is as challenging as the oracle problem.

Other researchers have proposed methods that do not attempt to provide or calculate an oracle, but rather aid the tester, informed of the structure of the system, exactly what details she should watch and constrain about the oracle. This takes the form of determining, through a series of tests, which variables in the system are most effective in revealing faults or bad behavior, and then providing them to the tester to define the “correct” value. One research group used mutant generation to find which variables killed the most mutants and, therefore, found the most faults6. An alternative solution identified chains of dependent variables using probabilistic substitution graphs to find the most important variables7 (still unknown to the author how). These works make the best advances in the reducing the human effort needed to define and fulfill a complicated oracle.

Lastly, some researchers attempt to define an oracle and provide suggested values for what the most important oracle variables should be. One group proposed that Artificial Intelligence implementations could implicitly learn, after multiple executions and a preexisting suite of known solutions, the behavior of a system and accurately predict what output future test cases *should* produce8. Additionally, Natural Language Processing might be useful for inferring expected behavior from requirements documents and code comments. An approach defined by a group in Europe generated oracles for exceptional cases and improved existing test code fault-finding9. A similar group proposed a method of finding redundancy patterns in code and cross-checking the execution of those methods10. Uncovering differences in redundant methods may help to inform expected behavior.

Weyuker herself even proposed an early method of creating oracles, which was based on previous or even alternative version of a system as a side-by-side comparison machine1. However, she states that comparing the two outputs is just as tedious as checking an output against an oracle, and moreover, this method only confirms correct behavior if the two programs agree. If the two outputs do not match, the tester does not have any indication which piece of software is the correct one.

**Proposed Argument**

If a tester knows useful information about the input, and knows a reasonable level of detail about the implementation of the software program, then the tester can make reasonable, informed predictions about what they should see in the execution of the software – the output.

Given this hypothesis statement, I have devised a tool that accumulates the descriptions of what a tester wants in their tests, and uses the python languages’ random statistical functions to generate data points.

Why this works – discuss law of large numbers.

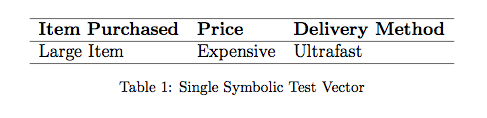
**Background**

How the tool works, design decisions

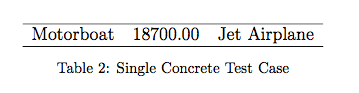
1. Pairwise Testing

Using a pairwise testing tool developed by the author’s advisor10, Michal Young, we can create symbolic test vectors that describe what each test case should generally look like, using English adjectives to describe what the input should be. Pairwise testing rests on the hypothesis that a majority of faults in a program are found with specific combinations of parameters. Creating the minimum number of test cases that satisfactorily describe all combinations of possible parameters creates the maximum code coverage. The success of finding all faults increase with greater numbers of combinations of parameters; that is to say that all combinations of four parameters will find more faults than all combinations of three. However, pairwise testing finds all combinations of two parameters and is often sufficient for maximum code coverage while still reducing the time, and therefore cost, of creating or running all test cases.

While translating a symbolic test vector to concrete data may present a challenge, it is extremely valuable in reducing the uncertainty of knowing what input test data looks like. Say, for example, we get a test vector that looks like



We know that in some test case we have we want to sending a large item by ultrafast means. In our injection of concrete data (described in the next section) into this test vector, we might end up with



Because this test case is relatively simple with only three parameters, we can see that we are trying to test our ecommerce website with the case of a customer trying to have an unshippable motorboat mailed to them through ridiculous means. However, imagine a much longer and complicated test vector – it is helpful to know that it was created from the test vector of Large Item - Expensive - Ultrafast. The symbolic test vector is perfectly descriptive in what the input looks like.

2. Context-Free Grammars

The aforementioned skipped step uses context-free grammars implemented in the Python templating tool Mako to generate concrete test code. This tool, GenSequence, has been developed by the author’s advisor11. Say a test case has the following form:

*Line --> Item + Price + DeliveryMethod*

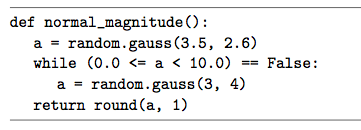
Where Item might have a production rule

*Item --> “Ship” | “Bike” | “Loaf”*

The grammar tool used is constructed such that a non-terminal can be the result of a function, which can be randomly generated. For example,

*Price --> lambda x: random.choice([i for i in range(x)])*

We can define a non-terminal symbol to end with the return result of a very sophisticated function - one perhaps that returns a hypergeometric distribution of data points. For example, take an earthquake modeling program, which takes a set of data points for magnitudes, and it can have all sort of descriptions attached to it: hypergeometric, normal, pre-clustered. All of these are included somewhere in the test vector, but there is not an easy way to turn the string “normal distribution” into



The series of testing tools relies on asking the tester what it means for magnitudes to be normally distributed (the above code example does not even reliably create a set of data points that are normally distributed). Moreover, generating good test data for actual magnitudes of an earthquake must be based on the knowledge that a magnitude cannot be negative or greater than 10.0. In addition, for every type a parameter could be, there is potentially a need for a different function that caters specifically to that type. A “pre-clustered” set of magnitudes will certainly need a different function.

3. Law of Large Numbers

4. Special Options

**Methods**

How I will measure the success of my program, intended experiment

**Results**

**Concluding Thoughts**

**References**

[1] Weyuker, Elaine J. “On Testing Non-Testable Programs”. *The Computer Journal* Volume 25, Issue 4 (1 November 1982): 465-470, accessed October 22, 2017 https://doi.org/10.1093/comjnl/25.4.465

[2] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. “Metamorphic model-based testing of autonomous systems”. *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 35-41, (2017).

[3] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. “A template-based approach to describing metamorphic relations”. *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 3-9, (2017).

[4] Tsong Yueh Chen. “Metamorphic testing: a simple method for alleviating the test oracle problem”. *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, 53-54 (2015).

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, “The Oracle Problem in Software Testing: A Survey”. *IEEE Transactions on Software Engineering* Volume 41, No. 5 (May 2015): 507-525.

[6] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. “Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing”. *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, 870-880, (2012).

[7] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. “Supporting oracle construction via static analysis”. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, 178-189, (2016).

[8] William B. Langdon, Shin Yoo, and Mark Harman. “Inferring automatic test oracles”. *Proceedings of the 10th International Workshop on Search-Based Software Testing* (SBST '17). IEEE Press, Piscataway, NJ, 5-6 (2017).

[9] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. “Automatic generation of oracles for exceptional behaviors”. *Proceedings of the 25th International Symposium on Software Testing and Analysis* (ISSTA 2016). ACM, New York, NY, USA, 213-224 (2016).

[10] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. “Cross-checking oracles from intrinsic software redundancy”. *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). ACM, New York, NY, 931-942 (2014).

[11] https://github.com/TestCreator/GenPairs

[12] https://github.com/TestCreator/GenSequence

**Annotated Bibliography**

Barr, E.T, and M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507-525, May 1 2015. DOI: 10.1109/TSE.2014.2372785 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6963470&isnumber=7106034

This paper describes different testing techniques and methods of parsing an oracle.

Carzaniga, Antonio, and Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. 2014. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). ACM, New York, NY, USA, 931-942. DOI: https://doi.org/10.1145/2568225.2568287

Crosscheck results by replacing methods with their redundant friends, which automatically injects an oracle.

Chen, Junjie, and Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, USA, 178-189. DOI: https://doi.org/10.1145/2970276.2970366

Static collection of fault-finding variables is a variant of the method that Staats, Gay, and Heimdahl proposed.

Chen, Tsong Yuen. 2015. Metamorphic testing: a simple method for alleviating the test oracle problem. In *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, USA, 53-54.

A discussion on metamorphic testing practices and applications.

Goffi, Alberto, and Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (ISSTA 2016). ACM, New York, NY, USA, 213-224. DOI: https://doi.org/10.1145/2931037.2931061

Manually written test cases might inadvertently miss boundary cases, but automatically generated cases do not have an oracle or know what it is testing. This group proposes a technique to process the comments on code with a natural language processing kit to generate oracles for exceptional edge cases.

Jahangirova, Gunel. 2017. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017). ACM, New York, NY, USA, 444-447. DOI: https://doi.org/10.1145/3092703.3098235

A discussion on the benefits of oracle location: inside or outside the source code.

Langdon, William B., and Shin Yoo, and Mark Harman. 2017. Inferring automatic test oracles. In *Proceedings of the 10th International Workshop on Search-Based Software Testing* (SBST '17). IEEE Press, Piscataway, NJ, USA, 5-6. DOI: https://doi.org/10.1109/SBST.2017..1

Machine learning and artificial intelligence can generate partially correct oracles.

Lindvall, Mikael, and Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic model-based testing of autonomous systems. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 35-41. DOI: https://doi.org/10.1109/MET.2017..6

Segura, Sergio, and Amador Durán, Javier Troya, and Antonio Ruiz Cortés. 2017. A template-based approach to describing metamorphic relations. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 3-9. DOI: https://doi.org/10.1109/MET.2017..3

Since metamorphism is thriving, this group attempts to standardize the practice with a template so that practitioners who begin using the technique have a baseline approach that makes the transition easier.

Staats, Matt, and Gregory Gay, and Mats P. E. Heimdahl. 2012. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 870-880.

By seeding faults in mutant versions of a program, an existing test suite can find the variables most likely to cause faults in a program. By narrowing the size of the expected oracle values, the tester has significantly less work.

Weyuker, Elaine J. On Testing Non-Testable Programs, *The Computer Journal*, Volume 25, Issue 4, 1 November 1982, Pages 465–470, <https://doi.org/10.1093/comjnl/25.4.465>

This paper defines what it means to be non-testable, and clearly describes the oracle problem and why it is so difficult to provide every expected value in an oracle.