UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note

RN/14/13

Computational Intelligence and Testing

3 November 2014

W. B. Langdon

Abstract

Discussion of the state of the art in and future research on Software Testing during NII Shonan Meeting Computational Intelligence for Software Engineering, Seminar 053, 20-23 October 2014. The discussions between software engineers and experts in artificial intelligence were mainly lead by Andreas Zeller and Jens Krinke.

1 Automated Test Case Generation

We did not spend much time on automated test case generation as it was regarded as pretty much a solved problem. However typically even the best test suites only cover about 80% of the software being tested. The remaining 20% may be either to hard to reach or even it may be it can never be executed. Unreachable code may have been written as a defensive measure to catch error conditions which should never happen.

Although test cases can be generated to cover a large fraction of human written code, it remains an open question how to check the program has indeed done the right thing for each test. This is known as the test "Oracle Problem" [1].

2 Mutation Testing

Mutation testing is a well established technique for assessing the effectiveness of testing processes by artificially inserting faults into the software being tested (known as mutants) and seeing how many of these artificial bugs are found by the test suite [2, 3]. Notice mutation testing does not require a test oracle [4]. That is, test cases do not need to include the desired answer(s). Instead it is sufficient to see if the mutated code yields the same answer(s) as the original un-mutated code. If a mutation causes a different answer, it is regarded as having failed that test. It is said to have been killed by the test. The more mutants a test suite kills, the better the test suite is.

Although the discussion was about software testing, mutation can be widely applied. E.g. the effectiveness of proof reading a book can be estimated by randomly inserting errors into the text and then seeing what fraction of seeded errors are reported. Since mutation testing is about the effectiveness of the test process, it is also sometimes called "mutation analysis".

Even for modestly sized code, a large number of mutants are created. Compiling and testing each of these is expensive. Computational costs can sometimes be reduced by compiling source code containing all mutations and then enabling them one at a time at run time [5].

Typically only a single change is made to the source code. E.g. replacing $<$ by $\leq$. These are known as first order mutants. It is becoming more common to consider making multiple simultaneous changes (known as higher order mutants [6]). Potentially higher order mutants can reduce computational overhead as a higher order mutant *may* be caught more easily by testing [7].

There are two major reason why mutation testing has not been widely adopted: 1) equivalent mutants [8] and 2) expense. In general it is impossible to tell if two programs will always generate the same answers (given the same inputs) as each other. That is, again in general, we do not know if a test suite has failed to kill a mutant because of a weakness in the test suite of because the mutant genuinely has made no difference to the program. (It is an "equivalent mutant".) In practise mutation testing creates large numbers of mutants which are not killed and thus creates a manual problem to decide if more and better tests are needed or if the undead mutant is truly an equivalent mutant (and thus cannot be killed).

As the mutated code may not be well behaved it is common to run the mutated program in some form of *sand box*. Potentially this might be provided at low overhead by a virtual machine. Alternatives include enabling array bounds checking[1] and intercepting system calls [3]. Since system calls, including I/O, can be slow, a sand box which replaces them with dummies can, in some circumstances, actually speed up testing.

It remains a long term goal to create a virtuous closed co-evolutionary loop in which mutation testing finds untested parts of programs and then automatic testing creates additional test cases which cover it [9].

Yue Jia maintains a repository of papers on mutation testing [10]. It can be found at: `http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository/`

[1] William Bader has a patch for GCC which provides bounds checking for C.

3 Unit Testing

As generally units are only a small fraction of the total system, unit testing is easier and faster than system testing. With fewer paths through the code, it may be possible (at the level of individual function level) to test them all.

Although there are commercial tools to assist unit testing, these are not widely used. This may in part be because they rely on pre- and post- conditions, which are often not available. (Notice that contract programming, such as in the Eiffel programming language, requires the programmer to give pre- and post-conditions.) That is, it may be possible to generate test cases to cover all the lines of code in a function (i.e. a unit) but when the tests are run, without more information (such as pre- and post- conditions), we do not know if the function has calculated the right answer or not. Also there may be a large number of cases where the function creates an exception or a segmentation error because the unit test creates conditions the code was not designed to handle. For example, instead of passing the address of a data structure, the test case calls the function with a null pointer. If there are a large number of these problems the user may lose faith in the tool.

There may be scope for automatically learning pre- and post- conditions, perhaps inconjunction with discovering invariants, e.g. with Daikon [11], or other machine learning techniques. During normal operation, Daikon might learn that a pointer to a data structure is never null.

4 Interplay Between System Testing and Unit Testing

A future testing tool might support both system and unit level testing and the interaction between them. For example a machine learning approach might be invoked when running system tests to discover details of how functions are called. These might then be used to control the ranges of inputs automatically generated during unit testing. E.g. if during system testing, a function is always passed the address of a constant string, this might be treated as a fact when generating test cases for when it is tested in isolation. Thus reducing the load on the user caused by unit tests failing because they called the function with an illegal address or the address of non-string data.

Additionally the invariants might be used to automatically add assertions to the code. For example, `assert(p!=null);`

Testing may proceed by alternating between system and unit level testing. At the unit level, the `assert()` documents an assumed invariant but during system tests it should not be triggered. Thus during system testing it becomes an automated test oracle, since a test that fails it indicates a problem.

There was a brief discussion of bug masking, where two or more faults (bugs) are present but the code appears to be working. This can create the paradox that removing one bug causes the (apparently) working code to fail until the other bug is also fixed.

5 Future

As mentioned in Section 2, the integration of mutation testing and automatic test case generation remains a long term goal, however progressing it will probably also require solving some of the problems with each (mentioned in Sections 1 and 2).

As mentioned in Section 4, there seems to be great scope for testing strategies which mix system and unit testing and some form of invariant learning or even automated solutions to the Oracle Problem (Section 1).

Although there are many tools for detecting duplicated code [12], in practise cloned code is not removed or consolidated by refactoring. This is particularly true in financial applications where code consolidation is regarded as increasing the risks (or expense) of a single point of failure whereas multiple copies of equivalent code are each regarded as posing less risk to the user (i.e. the bank).

Mostly the discussion was aimed at testing to discover bugs and assumed the use of automated test scripts rather than manual or interactive testing, even for GUI and web based applications. We briefly touched on test case prioritisation during regression testing [13]. Other topics include fuzz testing and stress or performance testing. There is some evidence that the fraction of bugs removed from new code follows a logistic curve with time (see talk "Predicting Release Time Based on Generalized Software Reliability Model" by Hironori Washizaki at NII Shonan Meeting Seminar 053). A future topic might be to predict the number [14] (and especially the severity) of remaining bugs [15]. A common business decision appears to ship when 95% of bugs have been found and fixed. Research could continue to investigate other trade-offs between release date and software quality.

There has been some recent work looking at creating or enhancing existing software by transplanting code [16, 17]. Such plastic surgery approaches may take their feed stock from wide spread open source repositories (e.g. SourceForge and GitHub) or internal software might be used to donate code. Since transplanting code seems feasible, perhaps future work should investigate the scope for computation intelligence techniques for *transplanting test suites*, *software requirements* or even *user expectations*. There may also be scope for using datamining and bigdata approaches (e.g. as used by Google Translate) to find matches between informal requirements and (fragments) of code implementations. While Google Translate is based on learning from United Nations documents which have already been translated into multiple languages [18], there is lots of open source code which might be used to find fraglets of code [19] which might be assembled into complete programs, perhaps using an enhance form of genetic programming [20].

References

[1] Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, UK (2013) To appear in IEEE Transactions on Software Engineering.

[2] DeMillo, R.A., Offutt, A.J.: Constraint–based automatic test data generation. IEEE Transactions on Software Engineering **17**(9) (1991) 900–910

[3] Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software **83**(12) (2010) 2416–2430

[4] Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: A comprehensive survey of trends in oracles for software testing. Technical Report Research Memoranda CS-13-01, Department of Computer Science, University of Sheffield (2013)

[5] Langdon, W.B., Harman, M., Jia, Y.: Multi objective mutation testing with genetic programming. In Bottaci, L., Kapfhammer, G., Walkinshaw, N., eds.: TAIC-PART, Windsor, UK, IEEE (2009) 21–29

[6] Harman, M., Jia, Y., Langdon, W.B.: Strong higher order mutation-based test data generation. In Zeller, A., ed.: 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2011), Szeged, Hungary, ACM (2011) 212–222

[7] Harman, M., Jia, Y., Reales Mateo, P., Polo, M.: Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In: ASE. (2014)

[8] Harman, M., Yao, X., Jia, Y.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: 36^{th} International Conference on Software Engineering (ICSE 2014), Hyderabad, India (2014)

[9] Harman, M., Jia, Y., Langdon, W.B.: A manifesto for higher order mutation testing. In du Bousquet, L., Bradbury, J., Fraser, G., eds.: Mutation 2010, Paris, IEEE Computer Society (2010) 80–89 Keynote.

[10] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5) (2011) 649 – 678

[11] Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2) (2001) 1–25

[12] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transaction on Software Engineering **33**(9) (2007) 577–591

[13] Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: A survey. Journal of Software Testing, Verification & Reliability **22**(2) (2012) 67–120

[14] Mockus, A., Weiss, D.M., Zhang, P.: Understanding and predicting effort in software projects. In: Proceedings of the 25th International Conference on Software Engineering. ICSE '03, Portland, Oregon, USA, IEEE (2003) 274–284

[15] Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S.: A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering **38**(6) (2012) 1276–1304

[16] Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., Garcia-Sanchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K., eds.: 17th European Conference on Genetic Programming. Volume 8599 of LNCS., Granada, Spain, Springer (2014) 137–149

[17] Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In Orso, A., Storey, M.A., Cheung, S.C., eds.: 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014), Hong Kong (2014)

[18] Tanner, A.: Google seeks world of instant translations. Reuters (2007)

[19] Yamamoto, L., Tschudin, C.F.: Experiments on the automatic evolution of protocols using genetic programming. In Stavrakakis, I., Smirnov, M., eds.: Autonomic Communication, Second International IFIP Workshop, WAC 2005, Revised Selected Papers. Volume 3854 of Lecture Notes in Computer Science., Athens, Greece, Springer (2005) 13–28

[20] Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008) (With contributions by J. R. Koza).

[21] Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. (IEEE Transactions on Evolutionary Computation) Accepted.