The Use of Automatic Test Data Generation for Genetic Improvement in a Live System

Saemundur O. Haraldsson*,John R. Woodward* and Alexander I.E. Brownlee*
*Department of Computing Science and Mathematics
University of Stirling
Stirling, Scotland
Email: soh@cs.stir.ac.uk

***Abstract*—In this paper we present a bespoke live system in commercial use that has been implemented with self-improving properties. During business hours it provides overview and control for many specialists to simultaneously schedule and observe the rehabilitation process for multiple clients. However in the evening, after the last user logs out, it starts a self-analysis based on the day's recorded interactions and the self-improving process. It uses Search Based Software Testing (SBST) techniques to generate test data for Genetic Improvement (GI) to fix any bugs if exceptions have been recorded. The system has already been under testing for 4 months and demonstrates the effectiveness of simple test data generation and the power of GI for improving live code.**

***Keywords*-Search Based Software Engineering; Test data generation; Bug fixing; Real world application**

I. Introduction

Genetic Improvement (GI) is a growing area within Search Based Software Engineering (SBSE) [1] which uses computational search methods to improve existing software. When improving programs, whether functional or non-functional properties, it is necessary to ensure that the enhanced version of the program behaves correctly. Traditionally GI has used testing rather than other formal verification methods for that purpose [2]. Moreover test cases have been used to evaluate the improvements as well [3]–[5]. GI and SBST should therefore be used in conjunction with each other, specifically when the existing software has limited test data. It is then necessary to generate more test cases. Earliest papers of the SBST literature were mostly seeking to generate such test data automatically with search methods [6] which fits with GI's philosophy.

It is not uncommon to launch programs before they can be completely tested. Often it is because the number of conceivable use case scenarios is huge and therefore impossible to test in practice. Instead the application is put in use after a reasonable amount of testing and the developer collects data from the users, both recording performance and reliability. Then putting effort and resources into maintenance of the software, regularly providing updates and patches throughout the lifetime of the software.

In this paper we present a live system, Janus Manager (JM), that collects data only when user input produces errors and uses it to generate test data for fixing itself. This decreases significantly the cost of maintenance after the initial delivery of the product. It is a bespoke program for a vocational rehabilitation centre, developed and maintained by Janus Rehabilitation Centre in Reykjavik, Iceland.

The remainder of the paper is structured as follows. Section II lists some related work and inspirations. Section III details what the system does during business hours and how it keeps records for later generating test data. Section IV explains how the daily data is used to generate and utilise test data and Section V summarises the current data gathered since the launch of JM. Section VI gives an overview of what future directions we are currently contemplating.

II. Related work

The SBSE literature has expanded considerably [7] since Harman and Jones coined the term [1] and with it the SBST literature [6]. Moreover the research challenges for software engineering for self-adaptive systems are regularly being recognized [8], [9].

Much of the SBST research has been about the generation of test data such as Beyene et al. [10] where they generated string test data with the objective of maximising code coverage. The essential objective of the test data generation process is maximum code coverage, where every new instance of test data has never been seen before and therefore might be covering code that previous test cases have not. It is still a simple random sampling of test data and not as sophisticated as uniform sampling with Bolzman samplers [11] or like Feldt and Poulding do when searching for data with specific properties [12], [13]. The random search can also be replaced with an alternatives such as hill-climbing [14], an Evolutionary Algorithm [15] or less commonly used optimisation algorithms [16].

The majority of the generated test data in JM comes from emulating actual inputs from a graphical user interface (GUI). There are however examples of work that generate test data for GUI testing by representing input fields with symbolic alternatives [17]. That however demands that the developer knows in better detail about how the software will be used, which in our case is near impossible since every client's route through the rehabilitation is unique.

Our set up is in practice a slower working example of test data and program co-evolution for bug fixing [18], [19] with the addition that the usage evolves as well.

978-1-5386-2789-1/17 $31.00 © 2017 IEEE
DOI 10.1109/SBST.2017..10

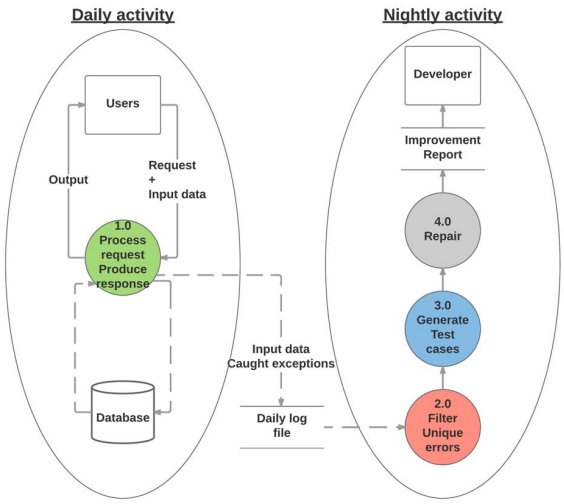

Fig. 1. JM functionality divided into daytime processes and night-time processes.

III. JM Daily activity

JM is a software system that is developed by JR as a tool in their vocational rehabilitation service. The motivation for its development is to provide the best possible service to their clients by giving the specialists a user friendly management tool. Moreover it is a tool for the directors to be able to continuously improve the rehabilitation process with statistical analysis of client data and performance of methods and approaches. It has to manage multiple connections between users, specialists and clients.

A. Usage

The left side of Figure 1 displays the daily routine of JM and Figure 2 is a simplified map of currently possible usage and features. The users are all employees of JR, over 40 in total, including both, specialists and administrators. They interact with JM by either requesting or providing data which is then processed and saved. The requests are for an example internal communications between the interdisciplinary team of specialists about clients, a journal record from a meeting or an update to some information regarding the client. The system can also produce reports and bills in pdf format or rich text files.

The clients have access to specialised and standardised questionnaires that measure various aspects of the clients welfare and progress. The specialists then use those questionnaires to plan a treatment or therapy.

While all of this is happening, every time an input data causes an exception to be thrown JM logs the trace, input data and the type of exception in a daily log file shown in the middle of Figure 1.

B. Structure

JR provides individualised vocational rehabilitation and as such users of JM regularly encounter unique use cases.

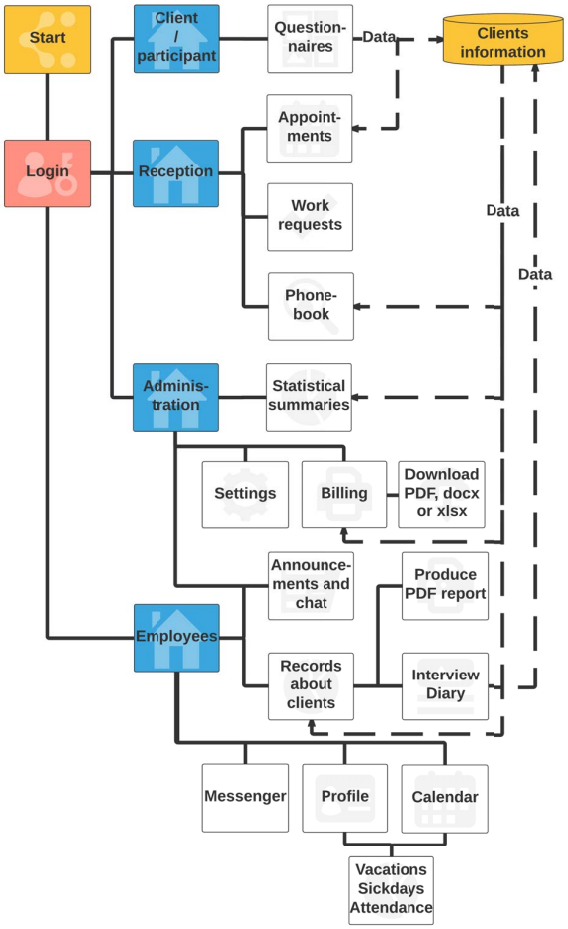

Fig. 2. A simplified map of JM current features.

Therefore JM is in active development while being in use. Features are continuously added based on user experience, feedback and convenience. Currently the system is over 25K lines of Python (300 classes and more than 600 functions).

JM runs as a web service on an Apache server running on a 64 bit Ubuntu server with 48GiB RAM and two 6 core Intel processors. The GUI is a web page that JM serves up from pre-defined templates.

IV. JM Nightly activity

After the last user logs off in the evening the nightly routine in Figure 1 initiates. The process runs until the next morning or until all bugs are fixed. During the night JM analyses the logs, generates new test data and uses GI to fix bugs that have been encountered during the day.

A. Log analysis

Going through the daily logs involves filtering the exceptions to obtain a set of unique errors in terms of input, type and location. The input is defined as the argument list at every function call on the trace route from the users' request to the location of the exception. The type of the exception can be

Procedure 1 Test data search

1: $\Theta \leftarrow [\theta]$ {Start with the original input}
2: $n \leftarrow 0$
3: $\Theta^{new} \leftarrow [\theta]$
4: **while** $(n < 1000)$ AND $(|\Theta^{new}| != 0)$ **do**
5: extend Θ with Θ^{latest}
6: $\Theta^{latest} \leftarrow \Theta^{new}$
7: $\Theta^{new} \leftarrow [\ ]$
8: **for** $i = 1$ until $i == 100$ **do**
9: $\theta^r \leftarrow$ random choice Θ^{latest}
10: $\theta^{mutated} \leftarrow$ mutate θ^r
11: **if** $\theta^{mutated} \rightarrow$ causes exception **then**
12: append $\theta^{mutated}$ to Θ^{new}
13: **end if**
14: $n+ = 1$
15: **end for**
16: **end while**

any subclass of *Exception* in Python, both built in and locally defined.

The errors are sorted in decreasing order of importance, giving higher significance to errors that occurred more often, arbitrarily choosing between draws. This measure of importance assumes that these are use case scenarios that happen often and are experienced by multiple users and not a single user who repeatedly submits the same request.

B. Generate test data

The test data generation is done with a simple random search of the neighbourhood of the users' input data. The input is represented by a Python *dictionary* object, where elements are key, value pairs and the values can be of any type or class. However, most values are strings, dates, times, integers or floating point numbers. The objective of the search is to find as many versions of the input data as possible that trigger the same exception. Procedure 1 details the search for new test data,

Starting with the original input θ we make 100 instances of $\theta^{mutated}$ where a single value has been randomly changed. For each instance the value to be mutated is randomly selected while all other values are kept fixed. Every $\theta^{mutated}$ that causes the same exception as the original is kept in Θ, essentially given fitness 1, others are discarded. This is then repeated by randomly sampling from the latest batch of $\theta^{mutated}$, Θ^{latest} (see line 9) until either no new instances are kept or the maximum of 1000 instances have been evaluated (line 4).

The mutation mechanism in line 10 first chooses randomly between key, value pairs in θ^r only considering pairs where values are of type string, date, time, integer or float. Then depending on the type, the possible mutations are the following

String mutations randomly add strings from a predefined dictionary with white space and special characters, keeping the original as a sub-string.

Date mutations can change the format (e.g. 2017-01-27 becomes 27-01-17), the separator or randomly pick a date within a year from the original

Time mutations can change the format (e.g. 7:00 PM becomes 19:00), the separator or randomly pick a time within 24 hours from the original

Int. mutations add or subtract 1, 2 or 3 from the original.

Float mutations change the original with a random sample from the standardised normal distribution $N(0, 1)$

All of the instances in Θ along with the original θ are then the inputs of the new unit tests. The assertion for each of them will check that the response is of the specific exception type and the tests will fail if the input triggers that exception. The new unit tests are then added to the existing test suite, automatically expanding the library of test cases.

The problem with this approach is that it does not check whether new test cases are complementary or not, i.e. if the two or more test cases are validating the same part of the code.

C. Genetic Improvement

The GI part of the nightly process relies on the new test cases in conjunction with a previously available test suite. The assumption is that given the test suites the program is functioning correctly if it passes all test cases and so is awarded highest fitness. Otherwise fitness is proportional to the number of test cases the program passes of the whole suite.

The process is inspired by Langdon's et al. work [3] by evolving edit lists that operate on the source code. The edit lists define the operations replace, delete and copy for code snippets, lines and statements.

The evolution is population based with 50 edit lists in each generation. Each generation is evaluated in parallel to minimise GI's execution time and to utilise the full power of the server. Edit lists are selected in proportion to their fitness. Only half of the population gets selected and they undergo mutation to start the next generation, crossover is not used in the current implementation. The other half of the subsequent generation are randomly generated new edit lists.

The GI only stops if it has found a program variant that passes all tests or just before the users are expected to arrive to work. It then produces an html report detailing the night's process for the developers. The report lists all exceptions encountered, new test cases and possible fixes, recommending the fittest. If more than a single fix is found, then the report recommends the shortest in terms of number of edits. However it is always the developers choice to implement the changes as they are suggested, build on them or discard them.

V. Summary

Development on JM started in March 2016 and quite early on it was launched for general use in JR. Since late September, early October 2016 the self-healing processes have been running as a permanent service in JM. During that time 22 unique exceptions have been reported and always a single error at a time. Table I lists exception types that have been

TABLE I
SUMMARY OF ENCOUNTERED EXCEPTION TYPES, THE NUMBER OF OCCURRENCES AND HOW MANY TEST CASES THEY PRODUCED

Exception type	Number of occurrences	Mean total number of input test data produced	Mean number of complementary test cases
IndexError	4	15.25	1.25
TypeError	6	11.17	1.33
UnicodeDecodeError	3	44.33	1.67
ValueError	9	16.33	1.11

encountered, the number of times for each type, how many test cases it produced and how many of them were complementary.

Every single one of the exceptions revealed a bug in the program that was subsequently fixed by the GI process. In total 408 test cases have been produced, however a manual post-process revealed only 6% of those were testing unique parts of the code. The most obvious example is a *UnicodeDecodeError* caused by incorrect handling of a special character in a string input. The generative method for strings, described in Section IV, made multiple versions of a string containing the same special character as a sub-string and therefore all of them were invoking the same error.

VI. FUTURE WORK

The system introduced in this paper is fully implemented and live, however only 22 exceptions have been recorded during the first few months.The total number of new test cases is 408 of which 28 are unique in terms of code coverage, which is not enough to make statistical inference. Our next steps are to monitor the system while it is being developed further and gather data on the bugs that are caught and fixed. Ideally we want to be able to use the data to make predictions for expected inputs to the system and thus make it possible to generate test data that imitates unseen future inputs.

While a random search has been effective up until now, we would like to improve the process to find more unique test cases per exception encountered. That involves implementing a fitness function that is not binary and a better sampling method and adding constraints to the search to maximize code coverage while minimizing the number of test cases.

ACKNOWLEDGMENT

The work presented in this paper is part of the DAASE project which is funded by the EPSRC. The authors would like to thank JR for the collaboration and providing the platform for which made the development possible.

REFERENCES

[1] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, dec 2001.

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "{GenProg}: {A} Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[3] W. B. Langdon and M. Harman, "Optimising Existing Software with Genetic Programming," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 118–135, feb 2015.

[4] J. Petke, W. B. Langdon, and M. Harman, "Applying Genetic Improvement to MiniSAT," in *5th International Symposium, SSBSE 2013*, ser. Lecture Notes in Computer Science, St. Petersburg, Russia: Springer Berlin Heidelberg, aug 2013, pp. 257–262.

[5] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using Genetic Improvement \& Code Transplants to Specialise a {C++} Program to a Problem Class," in *17th European Conference on Genetic Programming, EuroGP 2014*, ser. Lecture Notes in Computer Science, Granada, Spain: Springer Berlin Heidelberg, 2014, pp. 137–149.

[6] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[7] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7007, pp. 1–59.

[8] B. H. C. Cheng et al., "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for SelfAdaptive Systems*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, vol. 5525, no. January, pp. 1–26.

[9] R. de Lemos et al., "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 7475, pp. 1–32.

[10] M. Beyene and J. H. Andrews, "Generating String Test Data for Code Coverage," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 270–279.

[11] P. Duchon and G. Louchard, "Boltzmann Samplers For The Random Generation Of Combinatorial Structures," *Combinatorics Probability and Computing*, vol. 13, no. 4-5, pp. 577–625, 2004.

[12] R. Feldt and S. Poulding, "Finding Test Data with Specific Properties via Metaheuristic Search," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 350–359.

[13] S. Poulding and R. Feldt, "Generating structured test data with specific properties using Nested Monte-Carlo Search," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. Vancouver: ACM, 2014, pp. 1279—-1286.

[14] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*. Austin, Texas: ACM Press, 2016, pp. 45–54.

[15] K. Lakhotia, M. Harman, and P. Mcminn, "A Multi-objective Approach to Search-based Test Data Generation," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. London, England: ACM, jul 2007, pp. 1098–1105.

[16] R. Feldt and S. Poulding, "Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary," *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pp. 1–7, 2015.

[17] K. Salvesen, J. P. Galeotti, F. Gross, G. Fraser, and A. Zeller, "Using Dynamic Symbolic Execution to Generate Inputs in Search-Based GUI Testing," *Proceedings - 8th International Workshop on Search-Based Software Testing, SBST 2015*, pp. 32–35, 2015.

[18] A. Arcuri, "On the Automation of Fixing Software Bugs," in *ICSE Companion '08 Companion of the 30th international conference on Software engineering*. Leipzig, Germany: ACM, 2008, pp. 1003–1006.

[19] A. Arcuri, D. R. White, J. Clark, and X. Yao, "Multi-Objective Improvement of Software using Co-evolution and Smart Seeding," in *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL'08)*, 2008, pp. 1–10.