Reducing Qualitative Human Oracle Costs associated with Automatically Generated Test Data

Phil McMinn
The University of Sheffield
Dept. of Computer Science
Regent Court, 211 Portobello
Sheffield, S1 4DP, UK

Mark Stevenson
The University of Sheffield
Dept. of Computer Science
Regent Court, 211 Portobello
Sheffield, S1 4DP, UK

Mark Harman
CREST
King's College London
Strand, London
WC2R 2LS, UK

ABSTRACT

Due to the frequent non-existence of an automated oracle, test cases are often evaluated manually in practice. However, this fact is rarely taken into account by automatic test data generators, which seek to maximise a program's structural coverage only. The test data produced tends to be of a poor fit with the program's operational profile. As a result, each test case takes longer for a human to check, because the scenarios that arbitrary-looking data represent require time and effort to understand. This short paper proposes methods to extracting knowledge from programmers, source code and documentation and its incorporation into the automatic test data generation process so as to inject the realism required to produce test cases that are quick and easy for a human to comprehend and check. The aim is to reduce the so-called qualitative *human oracle costs* associated with automatic test data generation. The potential benefits of such an approach are demonstrated with a simple case study.

1. INTRODUCTION

The automatic generation of software test cases has been a burgeoning research area of late. Several techniques for achieving structural coverage have been proposed; including symbolic execution [9], concolic execution [4, 17], and search-based testing [14]. While each technique has demonstrated moderate levels of success in generating test data [12], for example for obtaining branch coverage, the actual test data produced and their corresponding program outputs must usually be evaluated by a human tester. This is because an automated oracle is frequently not in existence. The effort expended by a human in this task is referred to as the *human oracle cost*. Surprisingly, there has been little work devoted to developing automatic test data generators so that human oracle costs are reduced.

While techniques have been proposed to minimize the sizes of generated test cases [13] and test suites [6], essentially tackling *quantitative* aspects of human oracle cost, there has been no work addressing the *qualitative* costs associated with generated test data. That is, the issue of how easily the generated test data can be comprehended and the scenario comprising a test case understood so that the corresponding program output can be evaluated accordingly.

The test data produced by automatic test data generators are not necessarily 'realistic' in the sense that they tend not to closely match the operational input profile of the program under test. Examples include strings that appear to be random sequences of characters rather than easily discernible pieces of data, such as a person's name, a country or a URL; or hard to interpret calendar dates that are several millennia in the past or future. Such test data require effort to understand, making the job of manually checking test cases harder, more laborious and more time consuming.

This short paper is the first work to propose methods to tackle the issue of qualitative human oracle cost, with the aim of producing test data that provide a better 'fit' with the operational input profile of a program; an issue not accounted for by the current state of the art. Clearly, the evaluation of the techniques proposed techniques requires human judgement, and so further work is required with real testers in order to assess them. Nevertheless, the paper shows, by means of a case study and an initial experiment, that the incorporation of even a little knowledge about the type of expected inputs into a search-based test data generator can produce dramatically simplified and more recognisable test data.

The contribution of this short paper is the proposal of techniques that address the qualitative human oracle cost problem in the following ways:

1. They aim to ascertain details about the operational input profile of a program through provided test cases and by reverse engineering knowledge from a program

2. They aim to incorporate this knowledge into search-based optimisation methods to encourage the generation of more 'realistic' test data, guiding the generation of test cases through seeding the initial stage of the search and through biasing search operators.

3. The paper also proposes techniques to identify opportunities for *re-using* input variable values for similar units, since partial or even full re-use of test input vectors across different units will avoiding the need for testers to spend as much time acquainting themselves with completely new test cases.

4. The techniques proposed may impact the fault-finding capability of a test suite, suggesting that test cases that of an ill-fit with the expected input profile should not be ignored entirely. In this scenario, multi-objective test data search approaches are proposed to balance possible competing constraints.

These ideas are presented in Sections 3-6. Prior to this, a motivating case study is introduced with the results of an initial experiment with it in Section 2. Finally, Section 7 concludes with closing remarks and further comments regarding further work.

2. BACKGROUND AND CASE STUDY

The human oracle cost associated with automatically generated test data can be demonstrated with the C function shown in Figure 1. The function returns the number of days between two dates, each represented by three integers corresponding to the month, day of the month, and the year.

In a simple initial experiment, the alternating variable method (due to Korel [11], implemented as detailed by Harman and McMinn [8]) was used to generate a test suite for branch coverage of the program. The alternating variable method is a meta-heuristic local search method that modifies a single input vector in accordance with a fitness function until the current branch of interest is executed (or resources are exhausted). The initial input vector is usually chosen at uniform random from the program's input domain, thus, the value of each input variable for the program is drawn from the range of the C `int` type, *i.e.* $-32,768$ to $32,767$.

While successfully covering all branches, the test data generated correspond to rather obscure dates, as can be seen in Table 1. In the first column of test data, a different random seed is used to generate the starting point (the initial input vector) for the test data search for each branch, producing dates such as '-5455/23195/-30879' (the program sanitizes month and day inputs that are out of range). Using the same random seed for each search leads to the same starting point being selected for each branch, leading to a higher degree of similarity amongst generated inputs (second column of test data). While this similarity arguably contributes to a reduced oracle cost, the data still require additional effort to comprehend when contrasted with the third column of test data. Here, test data are generated using a human-supplied test case (1/1/2010, 1/1/2010) as the starting point of the search. The impact of introducing this small piece of domain knowledge has a dramatic effect. The test data found to cover each branch is based around the supplied dates, and as such represent more eaily-recognizable test cases.

The question posed by this paper is, in what ways can input domain knowledge about a program be retrieved and used to guide the test data generation process? Test data generated in this fashion may have a reduced fault-finding capability. If so, how might these two potentially competing constraints be managed in the test data generation process?

3. GATHERING KNOWLEDGE ABOUT EXPECTED INPUT PROFILES

The case study demonstrates that information regarding the program's expected input profile is required to produce more 'realistic' test data that is easier for a human oracle to evaluate. A programmer or tester may in the position to provide a limited amount of knowledge in the form of a few test cases. This would not need to be a lengthly or onerous process, since a programmer is likely to have run their program at least once with a 'sanity check' to ensure it is working more or less as intended. A tester may even desire to bias the generation of test cases so as to incorporate their own knowledge into the process, including their own favourite 'corner' cases, and so on. Having being constructed by hand, such checks are likely to represent useful, realistic scenarios that can be used as the starting point from which to base the generation of further test cases. Yoo and Harman [18] have already demonstrated the possibility of generating test data from existing test data, but in the context of automating regression testing.

The program under test may itself be used to build up knowledge about its likely input profile. Many programs contain input sanitisation routines or defensive programming constructs, and these often contain explicit checks for set membership, boundaries, special cases, exclusions and other properties of inputs; all of which may be extracted from the program through static or dynamic program analysis. One such sanitization routine is present in the program of Figure 1. When a month integer or day of the month integer is out of range, the program corrects the input. Where the sanitization code is itself not being tested, such routines could be used to 'correct' automatically generated inputs, presenting more recognisable information to the human oracle. For example, the second and third columns of Table 1 show automatically generated test data containing values considered as 'out of range' by the program, which may be corrected to recognisably in-range values for the purposes of covering certain branches in the program. The sanitization performed in one function may be further used to sanitize inputs of another. For example, day and month inputs are sanitized in `days_between` and then used later in the body of the function when calling `month_days` and `is_leap_year`.

The names given to input variable identifiers give rise to further clues with respect to a program's input profile, particularly when identifier names contain a string that indicates a pre-defined class of values, for example '`dayOfTheWeek`', '`country`' or '`url`'. The common practice of `CamelCasing` and `under_scoring` variable names makes splitting identifiers into words and removing stop words such as 'the' and 'of' is a trivial task [1]. The extracted words and their variable type information could then be analysed in conjunction large-scale natural language lexicons such as WordNet [3] and type information to suggest appropriate input values. In general, however, domain-specific lexicons are likely to be required to cope with specific types of program.

4. DOMAIN-AWARE TEST DATA GENERATION

Assuming the existence of domain knowledge; in the form of example test cases, sets of appropriate values or constraints; several possibilities are available for using the information in a search-based test data generator to aid the production of more 'realistic' test data.

The simplest approach to incorporating knowledge into a search-based optimisation algorithm is to simply 'seed' the initial stage of the search with inputs that are already known to represent appropriate values, as with the experiment performed in the case study. The presence of several example inputs would help seed the initial stage of a global search,

```
        int days_between(int start_month, int start_day, int start_year,
                         int end_month, int end_day, int end_year)
        {
          int days = 0;

          // sanitize month inputs
(1)       if (start_month < 1) start_month = 1;
(2)       if (end_month < 1) end_month = 1;
(3)       if (start_month > 12) start_month = 12;
(4)       if (end_month > 12) end_month = 12;

          // sanitize day inputs
(5)       if (start_day < 1) start_day = 1;
(6)       if (end_day < 1) end_day = 1;
(7)       if (start_day > month_days(start_month, start_year))
            start_day = month_days(start_month, start_year);
(8)       if (end_day > month_days(end_month, end_year))
            end_day = month_days(end_month, end_year);

          // swap dates if start date before end date
(9)       if ((end_year < start_year) ||
                  (end_year == start_year && end_month < start_month) ||
                  (end_year == start_year && end_month == start_month &&
                                end_day < start_day)) {
             int t = end_month; end_month = start_month; start_month = t;
             t = end_day; end_day = start_day; start_day = t;
             t = end_year; end_year = start_year; start_year = t;
          }

          // calculate days
(10)      if (start_month == end_month && start_year == end_year) {
            days = end_day - start_day;
          } else {
            days += month_days(start_month, start_year) - start_day;
            days += end_day;
(11)        if (start_year == end_year) {
              int month = start_month + 1;
(12)          while (month < end_month) {
                days += month_days(month, start_year); month ++;
              }
            } else {
              int year; int month = start_month + 1;
(13)          while (month <= 12) {
                days += month_days(month, start_year); month ++;
              }
              month = 1;
(14)          while (month < end_month) {
                days += month_days(month, end_year); month ++;
              }
              year = start_year + 1;
(15)          while (year < end_year) {
                days += 365;
(16)            if (is_leap_year(year)) days ++;
                year ++;
          } } }
          return days;
        }
```

Figure 1: Case study code: a simple C function that takes two dates and returns the number of days between them. Branch IDs appear in brackets next to left of the decision statement concerned. Due to space constraints, the referenced functions `month_days` and `is_leap_year` are not listed but return the number of dates in a month and whether a year is a leap year respectively

Branch	Different random seed used to generate the starting point for the test data search for each branch		Same random seed used to generate the starting point for the test data search for each branch		Supplied test case (1/1/2010, 1/1/2010) used as the starting point	
1T	-4048/-10854/-29141	3308/-25426/-11998	-1247/-17004/9006	3305/6393/-10930	0/1/2010	1/1/2010
1F	4091/-31366/-23576	-9671/1283/-29866	15136/-17004/9006	3305/6393/-10930	1/1/2010	1/1/2010
2T	10430/3140/6733	-14884/-8416/-18743	15136/-17004/9006	-790/6393/-10930	1/1/2010	0/1/2010
2F	-31846/-3340/4891	7021/-24358/13435	15136/-17004/9006	3305/6393/-10930	1/1/2010	1/1/2010
3T	3063/31358/8201	9560/32094/-23160	15136/-17004/9006	3305/6393/-10930	16/1/2010	1/1/2010
3F	-2459/13917/984	6289/31510/-21766	-1247/-17004/9006	3305/6393/-10930	1/1/2010	1/1/2010
4T	9581/-9706/-310	12557/-4068/28941	15136/-17004/9006	3305/6393/-10930	1/1/2010	16/1/2010
4F	28420/31728/-12768	-1091/32060/-28710	15136/-17004/9006	-790/6393/-10930	1/1/2010	1/1/2010
5T	-32246/-6571/-25398	-12031/-25636/5098	15136/-17004/9006	3305/6393/-10930	1/0/2010	1/1/2010
5F	-6344/2436/16612	-28416/6792/30588	15136/15763/9006	3305/6393/-10930	1/1/2010	1/1/2010
6T	-11506/30843/-23045	-16620/-141/-3609	15136/-17004/9006	3305/-1798/-10930	1/1/2010	1/0/2010
6F	-25410/604/24374	-18405/1156/20386	15136/-17004/9006	3305/6393/-10930	1/1/2010	1/1/2010
7T	-23522/26246/8293	16112/18444/32681	13175/13978/22957	-12590/9615/12387	2/32/2010	1/1/2010
7F	-28918/-18094/4121	-5455/23195/-30879	15136/-17004/9006	3305/6393/-10930	1/1/2010	1/1/2010
8T	6986/24411/27186	-27238/10414/-26385	15136/-17004/9006	3305/6393/-10930	1/1/2010	2/32/2010
8F	17014/-25186/11482	-8643/-10992/9497	15136/-17004/9006	3305/-1798/-10930	1/1/2010	1/1/2010
9T	22499/16890/32767	-27155/28640/14378	15136/-17004/9006	3305/6393/-10930	2/1/2010	1/1/2010
9F	30047/-8218/-21009	25985/25832/-8283	15136/-17004/-23761	3305/6393/-10930	1/1/2010	1/1/2010
10T	-11755/-32245/-17139	-15931/-792/-17139	15136/-17004/-6866	3305/6393/-6866	1/1/2010	1/1/2010
10F	-27408/-22701/-8570	-5726/-22240/7212	15136/-17004/9006	3305/6393/-10930	2/1/2010	1/1/2010
11T	-28081/-27324/-1948	18799/15668/-1948	-28081/-27324/-6523	3305/6393/-6523	2/1/2010	1/1/2010
11F	24451/-28781/-16101	21202/-16138/26431	15136/-17004/9006	3305/6393/-10930	2/1/2009	1/1/2010
12T	21693/-14199/28802	-25586/-20457/28802	-28081/-27324/-6523	3305/6393/-6523	4/1/2010	1/1/2010
12F	-24836/-19878/20875	30494/9192/20875	-28081/-27324/-6523	3305/6393/-6523	2/1/2010	1/1/2010
13T	-4260/11218/18103	-12655/11307/-4945	-28081/-27324/8201	-28073/-1070/-2428	2/1/2009	1/1/2010
13F	-12813/6164/8579	9148/-18272/-29747	15136/-17004/9006	3305/6393/-10930	2/1/2009	1/1/2010
14T	13175/-15807/22957	-12590/9615/-21770	15136/-17004/9006	3305/6393/-10930	2/1/2009	2/1/2010
14F	-18841/23081/1943	24695/8345/-7519	15136/-17004/9006	3305/6393/-10930	2/1/2009	1/1/2010
15T	21370/-23147/27879	-1540/-7439/30075	15136/-17004/9006	3305/6393/-10930	2/1/2007	1/1/2010
15F	-13081/3756/-14974	-29578/6978/28020	15136/-17004/9006	3305/6393/-10930	2/1/2009	1/1/2010
16T	18475/6506/-20585	7605/26536/15658	15136/-17004/9006	3305/6393/-10930	2/1/2007	1/1/2010
16F	-17048/-32204/-13540	26210/20124/-12635	15136/-17004/9006	3305/6393/-10930	2/1/2007	1/1/2010

Table 1: Automatically generated test data using search-based optimisation for each branch ('T' and 'F' refer to the respective true and false branches). The search-based method used is the alternating variable method. In the first column of test data, inputs are generated for each branch using a random starting point decided using a different random seed for each branch. When the same random seed is used to begin the test data search for each branch, similar test data emerge, as seen in the subsequent column. In the final column, a human supplied test case (1/1/2010, 1/1/2010) was used as the starting point for the search, resulting in the generation of more recognisable dates for each branch

such as a Genetic Algorithm, which relies on several starting points; as well as helping to increase the variance of inputs, which may impact the fault-finding capability of the generated test suite (see Section 6).

Another approach to incorporating information into the search may be to simply influence the search towards existing points in the input domain, yet still with the target of covering a branch, by biasing search operators such as crossover and mutation in a Genetic Algorithm. The biased operators would essentially favour searching around values identified to be part of the program's expected input profile. Estimation of Distribution Algorithms [16] may represent an appropriate meta-heuristic approach in this regard, as these algorithms are based around the sampling of a probability distribution, which could be generated from knowledge gathered about the program's operational profile. Such a distribution may be a useful way to solve the problem in the presence of only incomplete or partial information.

5. TEST DATA RE-USE

Another means of reducing human oracle costs associated with test data comprehension is to *re-use* input values across related program units, as the tester may already be familiar with these inputs from testing other parts of the system. For example, month and year inputs from the `days_between` function in Figure 1 could be usefully re-used for the `days_in_month` or `is_leap_year` functions that it calls. In the case of object-oriented systems, lengthy sequences of pre-scripted method calls may be required to achieve a necessary test objective pre-condition. This pre-condition may correspond to a common state of an object or sub-system, for which the same sequence may be re-used.

The main obstacle to automating test data re-use is identifying related units and harvesting existing test suites for which inputs and test sequences can be re-applied. Call graph analysis is an obvious way of finding links from one unit to another. Algorithms such as Google's PageRank [15] may then help in analyzing links to decide which is the most 'important' unit for which test data should be initially generated and then re-used in another.

Techniques from Information Retrieval [1] have been shown to be useful for software quality analysis by comparing comments and identifier names in and across program units. Such methods could also be applied to this problem. For example, they could be applied to compare units using the names of identifiers and units as well as documentation in the form of comments in program code. String comparison techniques such as Levenstein distance [5] to include information about similar identifier names in the search. Comparison of program structures is another means of identifying similar units, and therefore clone detection [10] and methods developed to detect plagiarism within answers submitted for programming exercises [2] may also be useful.

6. FAULT-FINDING CAPABILITY OF THE GENERATED TEST SUITES

There is a danger that biasing test data generation toward specific values in a program's input domain will reduce the fault-finding capability of the test suites produced, and that more diverse or randomly generated test cases are not without purpose. Empirical studies involving Mutation Analysis need to be performed to confirm whether these effects occur in practice. If they do, it would be possible to mitigate against the problem using multi-objective search-based processes to find the optimal trade-off between reduced human oracle costs and increased fault-finding ability. Multi-objective search has already been applied in search-based test data generation, for example, to find test suites that cover as many branches as possible while also attempting to maximise memory usage for memory leak detection [7].

7. CONCLUSIONS AND FUTURE WORK

This short paper has raised the issue of human oracle costs associated with automatically generated test data. A simple case study has shown how even very simple techniques may help alleviate the problem. The paper also presented several other techniques that may be further applied to allow automated test data generation to take account of human oracle cost and minimize it, while also maintaining the fault-finding capability of test suites. Further work is required to implement and evaluate these ideas. Since test data comprehension is a human task, qualitative evaluation is required with real human testers.

8. REFERENCES

[1] D. Binkley, H. Feild, D. Lawrie, and M. Pighn. Increasing diversity: Natural language measures for software fault prediction. *Journal of Systems and Software*, 2009.

[2] P. Clough. Plagiarism in natural and programming languages: an overview of current tools and technologies. Tech. rep. CS-00-05, Dept. Comp. Sci, University of Sheffield, 2000.

[3] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database and some of its Applications*. MIT Press, 1998.

[4] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005.

[5] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[6] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. *3rd Search-Based Software Testing workshop (SBST 2010)*, 2010. IEEE digital library.

[7] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. *GECCO 2007*, pp. 1098–1105. ACM.

[8] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, to appear.

[9] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[10] K. Kontogiannis, R. Demori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1), 1996.

[11] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[12] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? *TAIC PART 2009*, pp. 95–104. IEEE.

[13] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. *ASE 2007*, pp. 417–420. ACM.

[14] P. McMinn. Search-based software test data generation: A survey. *Soft. Testing, Ver. and Reliab.*, 14(2):105–156, 2004.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[16] M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Apps*, 21(1):5–20, 2004.

[17] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *ESEC/FSE 2005*, pp. 263–272. ACM.

[18] S. Yoo and M. Harman. Test data augmentation: generating new test data from existing test data. Tech. rep. TR-08-04, King's College London, 2008.