ISSN 2186-7437

NII Shonan Meeting Report

No. 2014-13

Computational Intelligence for Software Engineering

Hong Mei (Shanghai Jiao Tong University, China)
Leandro Minku (The University of Birmingham, UK)
Frank Neumann (The University of Adelaide, Australia)
Xin Yao (The University of Birmingham, UK)

October 20–23, 2014

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Computational Intelligence for Software Engineering

Organizers:
Hong Mei (Shanghai Jiao Tong University, China)
Leandro Minku (The University of Birmingham, UK)
Frank Neumann (The University of Adelaide, Australia)
Xin Yao (The University of Birmingham, UK)

October 20–23, 2014

Computational intelligence (CI) techniques have provided many inspirations for improving software engineering, both in terms of the engineering process as well as the software product. The application of CI techniques in software engineering is a well established research area that has been around for decades. There have been dedicated conferences, workshops, and journal special issues on applications of CI techniques to software engineering.

In recent years, there has been a renewed interest in this area, driven by the need to cope with increased software size and complexity and the advances in CI. Search-based software engineering [1] provided some examples of how difficult software engineering problems can be solved more effectively using search and optimisation algorithms.

It is interesting to note that search-based software engineering does not provide merely novel search and optimisation algorithms, such as evolutionary algorithms, to solve existing software engineering problems. It helps to promote rethinking and reformulation of classical software engineering problems in different ways. For example, explicit reformulation of some hard software engineering problems as true multi-objective problems, instead of using the traditional weighted sum approach, has led to both better solutions to the problems as well as richer information that can be provided to software engineers [2,3]. Such information about trade-off among different objectives, i.e., competing criteria, can be very hard to obtained using classical approaches.

However, most work in search-based software engineering has been focused on increasing the efficiency of solving a software engineering problem, e.g., testing, requirement prioritisation, project scheduling/planning, etc. Much fewer work has been reported in the literature about CI techniques used in constructing and synthesizing actual software. Automatic programming has always been a dream for some people, but somehow not as popular as some other research topics.

The advances in evolutionary computation, especially in genetic programming [4], has re-ignited people's interest in automatic programming. For example, after the idea of automatic bug fixing was first proposed and demonstrated [5], industrial scale software has been tested using this approach and bugs fixed

[6]. The continuous need to test and improve a software system can be modelled as a competitive co-evolutionary process [7], where the programs try to improve and gain a higher fitness by passing all the testing cases while all the testing cases will evolve to be more challenging to the programs. The fitness of a testing case is determined by its ability to fail programs. Such competitive co-evolution can create "arms race" between programs and testing cases, which help to improve the programs automatically. In fact, competitive co-evolution has been used in other engineering design domains with success.

There are many other examples of highly innovative ideas from CI for tackling hard software engineering problems. The primary aim of this Shonan meeting has been to provide an interdisciplinary forum for researchers from the areas of computational intelligence and software engineering. There were roughly 50% participants from each of the two areas.

The meeting has included presentations by the participants as well as discussion groups for hot topics and future work. The discussion groups played a central role and reflected on the current state of the art in the different areas of software engineering and computational intelligence, foster interdisciplinary work, and establish new research directions. The outcomes of these discussion groups as well as the abstracts of the presentations are given in this Shonan meeting report such that it is available to all researchers interested in computational intelligence and software engineering.

References

1. M. Harman and B. F. Jones, Search-based software engineering. Information and Software Technology, 43:833–839, 2001.

2. K. Praditwong, M. Harman and X. Yao, Software Module Clustering as a Multi-Objective Search Problem. IEEE Transactions on Software Engineering, 37(2):264-282, March/April 2011.

3. Z. Wang, K. Tang and X. Yao, Multi-objective Approaches to Optimal Testing Resource Allocation in Modular Software Systems. IEEE Transactions on Reliability, 59(3):563-575, September 2010.

4. N. L. Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In J. J. Grefenstette, editor, Proc. of ICGA'85, pp.183-187, 1985.

5. A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC2008), (Piscataway, NJ), pp. 162–168, IEEE Press, 2008.

6. W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. Proc. of the 2009 International Conference on Software Engineering (ICSE), pp. 364-374, 2009.

7. A. Arcuri and X. Yao, "Coevolving programs and unit tests from their specification," Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'2007), (New York, NY), pp.397–400, ACM Press, 2007.

Participants

- Bradley Alexander, The University of Adelaide, Australia, https://cs.adelaide.edu.au/~brad/
- Carl K. Chang, Iowa State University, USA, http://www.cs.iastate.edu/~chang/
- Shing-Chi Cheung, The Hong Kong University of Science and Technology, Hong Kong, http://www.cse.ust.hk/~scc/
- Jens Krinke, University College London, UK, http://www0.cs.ucl.ac.uk/staff/J.Krinke/
- William Langdon, University College London, UK, http://www0.cs.ucl.ac.uk/staff/w.langdon/
- Zheng Li, Beijing University of Chemical Technology, China, http://cist.buct.edu.cn/zheng/
- Hong Mei, Shanghai Jiao Tong University, China, http://en.sjtu.edu.cn/about-sjtu/administration/mei-hong
- Leandro Minku, The University of Birmingham, UK, http://www.cs.bham.ac.uk/~minkull/
- Frank Neumann, The University of Adelaide, Australia, http://cs.adelaide.edu.au/~frank/
- Jerry Swan, University of Stirling, UK, http://www.cs.stir.ac.uk/~jsw/
- Ke Tang, University of Science and Technology of China, China, http://staff.ustc.edu.cn/~ketang/
- Markus Wagner, The University of Adelaide, Australia, http://cs.adelaide.edu.au/~markus/
- Hironori Washizaki, Waseda University, Japan, http://www.washi.cs.waseda.ac.jp/eng/AssociateProfessor.html
- Andreas Zeller, Saarland University, Germany, http://www.st.cs.uni-saarland.de/
- Jian Zhang, Chinese Academy of Sciences, China, http://www.ios.ac.cn/~zj/
- Lu Zhang, Peking University, China, http://sei.pku.edu.cn/~zhanglu/
- Jianjun Zhao, Shanghai Jiao Tong University, China, http://cse.sjtu.edu.cn/~zhao/

Program

Sunday (19 October)

19:00-21:00: Welcome Reception

Monday (20 October)

09:00-09:15: Introduction

09:15-09:45: Bill Langdon: Genetic Improvement

09:45-10:15: Jens Krinke: Computational Stupid: Language-Independent Program Slicing

10:15-10:45: Time for discussion

10:45-11:15: Break

11:15-12:00: Discussion and identification of important future research topics. Form of smaller groups based on research interest.

12:00-13:30: Lunch

13:30-14:00: Group Photo

14:00-14:30: Jerry Swan: Automatic Improvement Programming

14:30-15:00: Markus Wagner: Maximising Axiomatization Coverage and Minimizing Regression Testing Time

15:00-15:30: Discussions

15:30-16:00: Break

16:00-16:30: Lu Zhang: The Structure Problem in Search-Based Software Engineering

16:30-18:00: Small group discussions based on research interests

Tuesday (21 October)

09:00-09:30: Andreas Zeller: Search-Based System Testing

09:30-10:00: Leandro Minku: Software Effort Estimation Models Past, Present and Future

10:00-10:30: Time for discussion

10:30-11:00: Break

11:00-12:00: Discussion on funding opportunities with focus on international collaborations

12:00-13:30: Lunch

13:30-14:00: Hironori Washizaki: Predicting Release Time Based on Generalized Software Reliability Model

14:00-14:30: Brad Alexander: Searching Software Design Spaces Using Genetic Programming

14:30-15:00: Time for discussion

15:00-15:30: Break

15:30-16:00: Jianjun Zhao: Debugging with Online Slicing and Dryrun

16:00-16:30: Jian Zhang: Backtracking search techniques in software testing and analysis

16:30-18:00: Small group discussions based on research interests

Wednesday (22 October)

09:00-09:30: Hong Mei: Can Big Data bring us new opportunities for software automation?

09:30-10:00: Zheng Li: Efficient Search based Regression Test case prioritization

10:00-10:30: Time for discussion

10:30-11:00: Break

11:00-12:00: Discussion or time to summarize small group discussions

12:00-13:30: Lunch

13:30-21:30: Excursion and Banquet

Thursday (23 October)

09:00-09:30: Jerry Swan: An API for modular metaheuristics

09:30-10:30: Time for discussion

10:30-11:00: Break

11:00-12:00: Wrap-up and final discussions

Overview of Talks

Searching Software Design Spaces Using Genetic Programming

Bradley Alexander, The University of Adelaide, Australia

his talk summarises a small number of projects that I have worked on that exploit heuristic search to produce human competitive software artefacts. These projects range from the automatic construction of rules defining the core of an optimising compiler; the discovery of non-trivial recurrences from partial call trees; the evolution of navigation functions for robotic control; and the search for patches for software repair.

A common thread through all this research has been aggressive reduction in the search space through the target representations used and exploiting various mechanisms to keep the search focused on feasible solutions where possible. I conclude the talk with proposals to combine recently produced frameworks to further focus search in automated software repair.

Computational Stupid: Language-Independent Program Slicing

Jens Krinke, University College London, UK

Current slicing techniques cannot handle systems written in multiple programming languages because it requires knowledge on the semantic of all used languages and their interaction. Observation-Based Slicing (ORBS) is a language-independent slicing technique capable of slicing multi-language systems, including systems which contain (third party) binary components. A potential slice obtained through repeated statement deletion is validated by observing the behaviour of the program: if the slice and original program behave the same under the slicing criterion, the deletion is accepted. The resulting slice is similar to a dynamic slice. We evaluate five variants of ORBS on ten programs of different sizes and languages showing that it is less expensive than similar existing techniques. We also evaluate it on bash and four other systems to demonstrate feasible large-scale operation in which a parallelised ORBS needs up to 82% less time when using four threads. The results show that an ORBS slicer is simple to construct, effective at slicing, and able to handle systems written in multiple languages without specialist analysis tools.

Genetic Improvement

William Langdon, University College London, UK

Genetic programming can optimise software. There may be many ways to trade off resource consumption (such as time, memory, battery life) vs. functionality. Human programmers cannot try them all. Also the best multi-objective Pareto point may change with time, with hardware infrastructure, with network characteristics and especially for individual user behaviour. It may be GP can automatically suggest different trade offs for each new market. Recent results

include substantial speed up by evolving a new version of a program customised for a special case, medical image registration on GPUs and even growing small grafts containing new code (grow and graft GP, GGGP) which can be inserted into much bigger human written code.

Efficient Search based Regression Test case prioritization

Zheng Li, Beijing University of Chemical Technology, China

Test case prioritization (TCP) aims to improve the effectiveness of regression testing by ordering the test cases so that the most beneficial are executed first. TCP can be formulated into a search problem and then varied search techniques are applied in TCP. In this presentation, we discuss multiple objective search algorithms used in TCP, including NSGA-II, PSO, ACO and a coevolutionary algorithm. We illustrate the efficient problem when applied these multiple objective algorithms in TCP, and present a GPU-based parallel fitness evaluation and three parallel crossover computation schemes based on ordinal and sequential representations, which can highly speed up the computation of the evolutionary algorithms for TCP.

Can Big Data bring us new opportunities for software automation?

Hong Mei, Shanghai Jiao Tong University, China

Software automation (SA) is the process of automatically generating artifacts, especially programs, in the whole life cycle with a computer based on an informal (or formal) specification. In the histroy, there were a lot of researchers who paid efforts in this field and there were some exciting results in some areas of academic research, such as the research on automated programming. But SA is not successfully adopted by practical software development. With the coming of Big Data era, massive software engineering data, including various large repositories of source code, are available in many software companies and over the Internet. Can the Big Data in software engineering bring new opportunities to Software Automation? Can we develop a data-driven approach (the past research on software automation mainly focused on rule-based approach) to automatically synthesizing real large programs based on source code with rich documents and other related data in these repositories by using machine learning and statistical data analysis techniques? In this presentation, I will give a brief historical overview of Software Automation and some thoughts on Big Data based approach to software engineering, especially the software automation.

Software Effort Estimation Models Past, Present and Future

Leandro Minku, The University of Birmingham, UK

As the effort (e.g., person-hours, person-months) required to develop software projects is frequently the most important factor for calculating the cost of software projects, the task of estimating effort is of strategic importance for

software companies. Due to the difficulty of this task, researchers have been investigating the use of machine learning to create software effort estimation models that can be used as decision-support tools for software engineers. In this talk, I will briefly go through some key points in terms of the past, present and future of the field of machine learning for creating software effort estimation models. I will go from (a) conclusion instability to (b) ensembles and locality to (c) the importance of performing temporal and cross-company learning, generating insights, and obtaining a better understanding of when, why and how our models work (or dont work). Even though this talk is in the context of software effort estimation models, several of these ideas are also applicable to other types of software prediction models, such as defect predictors.

Automatic Improvement Programming

Jerry Swan, University of Stirling, UK

A recent methodology for automated improvement of (traditionally human-designed) programs is Genetic Improvement Programming (GIP), which has typically used methods inspired by Darwinian evolution to generate programs (variously in source or binary form) that meet some user-specified quality measure. This talk will describe two recent research directions in Automatic Improvement Programming an extension of GIP which features:

- An online adaptive approach, as exemplified by the Gen-O-Fix monitor framework for self-improving software systems.
- The incorporation of formal techniques from category theory, yielding a significant new direction, viz. semantics-preserving transformations that come with a guarantee of asymptotic improvement.

An API for modular metaheuristics

Jerry Swan, University of Stirling, UK

The modularisation and self-assembly of metaheuristics can be considered as a special case of software component assembly. It is our contention that this should be a significant future concern for SBSE. There are a number of reasons for this:

- Tackling larger and more difficult problems in SBSE will require more sophisticated metaheuristics than the random search and hillclimbing approaches that predominated when the field was in its infancy.
- Selecting/devising/tuning metaheuristics is an onerous and time-consuming activity. A modular decomposition addresses this by providing a basis for automated assembly. In particular, modularity allows metaheuristic design to be easily scaled (e.g. populations of metaheuristics, evaluatable in parallel).
- Dynamic problems motivate adaptive solutions, with the attendant desire for metaheuristics to be generated online.

The presentation discusses how to build such a modularized framework.

Maximising Axiomatization Coverage and Minimizing Regression Testing Time

Markus Wagner, The University of Adelaide, Australia

The correctness of program verification systems is of great importance, as they are used to formally prove that safety- and security-critical programs follow their specification. One of the contributing factors to the correctness of the whole verification system is the correctness of the background axiomatization, which captures the semantics of the target program language. We present a framework for the maximization of the proportion of the axiomatization that is used (covered) during testing of the verification tool. The diverse set of test cases found not only increases the trust in the verification system, but it can also be used to reduce the time needed for regression testing.

Predicting Release Time Based on Generalized Software Reliability Model

Hironori Washizaki, Waseda University, Japan

Development environments have changed drastically, development periods are shorter than ever and the number of team members has increased. These changes have led to difficulties in controlling the development activities and predicting when the development will end. We propose a generalized software reliability model (GSRM) based on a stochastic process, and simulate developments that include uncertainties and dynamics. We also compare our simulation results to those of other software reliability models. Using the values of uncertainties and dynamics obtained from GSRM, we can evaluate the developments in a quantitative manner. Additionally, we use equations to define the uncertainty regarding the time required to complete a development, and predict whether or not a development will be completed on time.

Search-Based System Testing

Andreas Zeller, Saarland University, Germany

Writing tests is hard, maintaining them is even worse. Where symbolic reasoning is impossible because of scale or complexity, generating test cases is a scalable, fully automatic technique to detect errors. Search-based techniques use genetic algorithms to systematically evolve a population of inputs towards a coverage goal. They form a middle ground between the scalability and applicability of random testing and the corner-case reasoning of symbolic techniques. When applied to generate system inputs, one gets powerful test generators that can be easily adapted towards arbitrary testing goals simply by changing the fitness function.

Backtracking search techniques in software testing and analysis

Jian Zhang, Chinese Academy of Sciences, China

Like many search methods used in SBSE, backtracking search techniques can also play a significant role in software engineering, especially in software testing and static analysis of source code. We describe our previous works on automatic test data generation for unit testing, automatic test data generation for combinatorial testing, as well as static analysis of C programs. All of them rely heavily on backtracking search. It is interesting to compare such search techniques with other search methods used in SBSE. We also present our recent work on optimization with respect to complex constraints.

The Structure Problem in Search-Based Software Engineering

Lu Zhang, Peking University, China

In many areas of software engineering, it is typical that a candidate solution has a structure, often a complex structure. In my talk, I argue that such a structure may pose an obstacle to search-based software engineering. When we search for a good solution via a meta-heuristic algorithm, such a structure may make it difficult to generate and/or synthesize new candidate solutions from existing solutions. When we search for a good solution via an existing mathematical optimization framework (such as ILP and SAT), such a structure may incur an inflation of the size of the formulated ILP or SAT problem. In my opinion, this difficulty may lie in that existing search frameworks do not allow incomplete solutions during the search. This indicates that we may need to invent new search frameworks for search-based software engineering.

Debugging with Online Slicing and Dryrun

Jianjun Zhao, Shanghai Jiao Tong University, China

Efficient tools are indispensable in the battle against software bugs during both development and maintenance. In this talk, we will introduce two techniques that target different phases of an interactive and iterative debugging session. To help fault diagnosis, we split the costly computation of backward slicing into online and offline, and employ incremental updates after program edits. The result is a vast reduction of slicing cost. The possibility of running slicing in situ and with instant response time gives rise to the possibility of editing-time validation, which we call dryrun. The idea is that a pair of slices, one forward from root cause and one backward from the bug site, defines the scope to validate a fix. This localization makes it possible to invoke symbolic execution and constraint solving

Discussion on Computational Intelligence to Improve the Software Development Process (summary by Leandro Minku)

Software development involves many dynamic events and uncertainty. For example, the requirements of the software may change during the development, the effort required to develop the software may differ from the originally estimated effort, the team developing the software project may change, etc. Such dynamism and uncertainty is inherent in the software development. Therefore, modern software development processes such as Agile have been embracing such dynamism and uncertainty by dealing with it, rather than proposing measures to try to make the software development process free of dynamism and uncertainty.

We have identified three different opportunities for using computational intelligence to deal with dynamic events and uncertainty during the software development process, so that the software development process can be improved. The first opportunity relates to the estimation of effort required to develop a software project. Machine learning approaches such as k-nearest neighbours, regression trees and ensembles of learning machines have been used for software effort estimation at early stages of software development. Early estimates can help to determine the cost and schedule of the software project. However, such estimates are usually rough and the real effort required to develop the software project may vary due to uncertainty. More accurate estimates could be obtained over time by using additional information acquired during the development process. These updated estimates could then be used to help software managers in making decisions to deliver the software on time, for example. Machine learning approaches may be able to update software effort estimations during the development of the project by using additional input attributes with previously unknown / too uncertain values, or by incorporating in the learning process the software effort estimation errors obtained in previous phases of the project.

The second opportunity relates to the prediction of how the number of defects will change during the course of the project. The number of defects existing in a software changes with time during development. In order to best determine when to release a software, it is useful to know whether the number of defects is unlikely to considerably drop any further. Current techniques for describing changes in the number of defects require manual provision of a distribution to describe such changes. Software data are then used to determine the parameters of this distribution. However, manually determining which is the best distribution to be used is a difficult task. Machine learning approaches could be used to automatically identify which distribution best describes the changes in defects for a given project, based on past projects.

The third opportunity relates to predicting what changes may happen during the development of a software project, so that action can be taken to minimise the risks of such changes. Even though we have not discussed during the meeting what types of change could be predicted, we regarded the possibility of investigating new machine learning problems related to predicting changes during the software development process as worthy of mention.

Besides the opportunities to deal with dynamic events and uncertainty, we have also identified the following opportunity to use complex machine learning algorithms such as graph classification algorithms in order to perform predictive

tasks involving data that are not easily vectorised, i.e., that cannot be easily converted into vectors of input and output attributes to be learnt. An example of problem that may benefit from that is modelling software architectures so as to predict their efficiency and development cost.

Discussions on Optimisation for Software Engineering (summary by Frank Neumann)

Optimisation methods have been widely used in the areas of operations research and engineering. Different opportunities arise in the area of software engineering to apply optimization. For example, one might to minimize the energy consumption of a program (app) running on a mobile device or optimize the schedule for the development process for a new software product. We discuss these two domains in the following.

Optimizing data structures and the use of sub-algorithms in software

Software uses different data structures and/or subalgorithms that impact their performance. The effectiveness of a datastructure depends on parameters during the execution of the program which is often not known in advance. Hence, it might be desirable to develop software in a general framework and leave the choice of data structures and algorithms (for example sorting algorithms) open and decide on them at at later stage. The choice can then be optimized for the device and its parameters that the software is running on, e.g. an android phone or a graphic cards.

A first prototype could be an android application that switches data structures and array sizes and optimize for energy consumption. There would be a need for good benchmark to test configurations for energy consumption,. Benchmarking of configurations might be difficult as energy consumption depends on a lot of other external factors such as filling of caches, etc.

Software project scheduling

Software project scheduling is the allocation of employees to tasks. Each task requires a set of skills and an amount of effort. Each employee has a salary. The goal is to optimize this allocation. Hereby, constraints like overwork have to be incorporated. An important question is what are the features that make software project scheduling different from classical scheduling like timetabling. Such features might be the incompatability of resources like software engineers or version control. The task would be to datamine such features, determine their impact and make use of them for the software schedule. Aggregating features might be harder in software engineering than in other areas. An important question is how to get data for feature selection and feature aggregation.

Discussions on Computational Intelligence and Testing (summary by William Langdon)

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.

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].

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/`

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.

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

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

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.

Future

As mentioned in Section , 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 and).

As mentioned in Section , 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).

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].

Discussion on Automatic Fault Localisation and Repair **(summary by Bradley Alexander)**

Locating and fixing faults is not easy – even when have tests that reveal the presence of bugs. Even when the source of a problem is known it can take a long time to effect a repair and then the fixes are often incorrect. These problems have motivated a growing body of research into automated fault repair. Frameworks such as GenProg[22] have demonstrated that CI can be usefully applied to debugging moderately-sized applications. However, there is still room for improvement in terms of scale and generality. The following lists some research opportunities (in no particular order) for future research that may lead to significant improvements in the effectiveness of fault localisation and repair.

Creating Better-Informed Evaluative Functions

Problem Current frameworks for automated bugfixing have very basic evaluative functions that count the number of tests passed and failed for each candidate patch. Such course-grained evaluative functions lead to a very challenging and uninformative search space.

Proposed Approach Measure the dynamic impact of applying the patch by measuring the differential impact of that change on passing and failing cases. The idea is that a good patch would have maximum impact on failing test cases and minimum impact on passing cases. Impact could be measured in a number of ways such as changes of control flow and changes of input and output values of individual functions.

Tools/Approaches to help measure impact include Pin, Valgrind, Byte-Code instrumentation and ASN.

Faster Evaluative Functions

Problem Evaluating a patch requires running the test suite over the entire program. This can slow down the rate of search.

Proposed Approach Instead of running the whole program for each set of changes - run just the function the changes are occurring in. This has the potential to be very much faster than running the whole program. The search order might need to be changed to batch proposed changes to functions. The system will need to handle changes to global state and intercept systems calls (sandboxing).

As a first step we can use existing work *Carving System Tests into Unit Tests* by Elbaum et al.[23].

Dynamic Test Selection

Problem Some tests do not need to be run with some changes. For example, if a test does not cover the line of code where the change resides it will not have an impact. Likewise changes in some locations, even if executed have limited impact on the outcomes of some tests.

Proposed Approach Use code coverage tools to establish which tests are most likely to be useful. Also dynamically track the sensitivity of tests to changes in some lines of code. Tests with low sensitivity to a location in the past can be avoided.

Automatically Engineer the Inputs that Cause Failure

Problem When a system crashes we often have just the crash stack. We rarely have the inputs that caused the failure. We would like to find the inputs automatically.

Proposed Approach Use search to try to derive inputs that progressively replicate the crash stack. Perhaps borrow some ideas from ReCore by Rossler et. al.[25].

Searching for Errors of Omission in Similar Code

A lot of bugs are errors of omission. It might be useful to focus some search specifically on patterns that might be missing by detecting code that is similar other code but is missing some parts.

Work has been done to enhance cp-miner to detect omitted code in clones. This work might be generalised to performing such detection in code that is only structurally similar rather than cloned.

Automatic Mining of Existing Patches for Software Repair

Recent work has shown that mining existing patches and applying them to other parts of the code can be very productive. Kim et al.[26] showed that applying manually mined patches can give better performance on benchmarks than GenProg. Andersen et al.[24] showed that automatically mined patches can be effectively applied. Work in this area would explore techniques for improving patch mining and exploring the possibility and benefits of making the patches more generic or abstract.

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.

[22] Le Goues, C., ThanhVu Nguyen ; Forrest, S. ; Weimer, W.: GenProg: A Generic Method for Automatic Software Repair IEEE Trans on Software Engineering, (Volume:38 , Issue: 1), 2012, pp 54-72.

[23] Elbaum, Sebastian; Chin, Hui Nee; Dwyer, Matthew B.; Dokulil, Jonathan: Carving differential unit test cases from system test cases Proceedings of the 14th ACM SIGSOFT, 2006, pp 253–264.

[24] Andersen, J; Anh Cuong Nguyen ; Lo, D. ; Lawall, J.L. ; Siau-Cheng Khoo: Semantic Patch Inference Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on. IEEE, 2012, pp. 382-385

[25] Rler, J.; Zeller, A.; Fraser, G.; Zamfir, C.; Candea, G.: Reconstructing Core Dumps ICST '13: Proceedings of the Sixth IEEE International Conference on Software Testing, Verification and Validation, 2013, pp. 114-123

[26] Kim, D.; Nam, J.; Song, J.; Kim, S.: Automatic patch generation learned from human-written patches Proceedings of the 2013 International Conference on Software Engineering, pp. 802-811