Multi-objective Regression Test Suite Minimisation for Mockito

Andrew J. Turner[1], David R. White[2], and John H. Drake[3]

[1] University of York, Department of Electronics, York, YO10 5DD, UK
`andrew.turner@york.ac.uk`

[2] CREST, Department of Computer Science, University College London, Gower Street, London, WC1E 6BT, UK
`david.r.white@ucl.ac.uk`

[3] The OR Group, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London, E1 4NS, UK
`j.drake@qmul.ac.uk`

Abstract. Regression testing is applied after modifications are performed to large software systems in order to verify that the changes made do not unintentionally disrupt other existing components. When employing regression testing it is often desirable to reduce the number of test cases executed in order to achieve a certain objective; a process known as as test suite minimisation. We use multi-objective optimisation to analyse the trade-off between code coverage and execution time for the test suite of Mockito, a popular framework used to create mock objects for unit tests in Java. We show that a large reduction can be made in terms of execution time at the expense of only a small reduction in code coverage and discuss how the described methods can be easily applied to many projects that utilise regression testing.

1 Introduction

Search-based software engineering (SBSE) refers to methodologies that apply computational search techniques to software engineering problems [6]. Software testing is an extremely popular area for SBSE research, with a recent study suggesting that over half of the SBSE literature is concerned with software testing [7]. A key area of software testing that may be improved by search-based methods is regression testing.

Regression testing is performed when a system is updated from one version to the next, to check whether any of the new added features interfere with previous, existing features. Regression testing is also continuously performed during test driven development, where the aim is to produce a system that meets a specification embodied by a suite of tests. A *retest-all* approach executes an entire test suite on the system, however this can be extremely costly with respect to computational or time limitations. The fields of regression test suite minimisation, selection and prioritisation seek to optimise the amount of effort needed to perform regression testing on a particular system. For a detailed survey of regression testing minimisation, selection and prioritisation, we direct the interested reader to Yoo and Harman [9].

Evolutionary Algorithms (EAs) are population-based metaheuristics inspired by the process of Darwinian evolution [1]. Given an initial population of candidate solutions, at each generation, strong solutions are chosen as parents to generate new solutions by recombination (crossover) and mutation. The newly generated solutions then compete with the original population for a place in the next generation. In the case of multi-objective optimisation, where multiple objectives are optimised simultaneously, a multi-objective EA (MOEA) attempts to find a set of *Pareto*-optimal solutions for which no improvement can be made in a single objective without having a detrimental effect on at least one of the other objectives [2]. This set of solutions, known as the Pareto front, gives a representation of the trade-off that exists between two conflicting objectives.

The nature of regression testing lends itself to formulation as a multi-objective problem, where the goal is to maximise the extent to which the test suite covers the target software whilst minimising the cost of executing that test suite. Indeed, many studies exist in the literature considering multi-objective regression testing [4, 5, 8], however these methods are still vastly outnumbered by studies considering single-objective variants.

Mockito (mockito.org) is a widely used Java-based framework for creating mock objects in automated unit testing. A mock object is used to mimic the behaviour of a real object when it is not possible to use the real object in a unit test, such as mocking the interface to a database. Thus, we are applying test suite optimisation to a set of tests written by programmers who are expert in testing; this offers a unique opportunity to examine the redundancy (with respect to code coverage) of a high-quality test suite. In this paper we apply the well-known multi-objective Non-dominated Sorting Genetic Algorithm II (NSGA-II) [3], to the test suite of the core components of a recent *beta* version of Mockito 2.0 (2.0.44). The objective of the optimisation algorithm is to minimise the running time of a selected subset of the test suite and maximise the proportion of the code-base covered by the selected tests in terms of branch coverage.

2 Methodology

The Mockito project uses the Gradle build system (gradle.org) to manage project structure, compilation and testing. The Gradle build system provides plugin support such that additional functionality can be executed as one or more 'tasks'. The standard plugin for Java (the language predominately used by Mockito) provides a *test* task. This task can be used to automatically run all unit tests associated with a project and produces information including pass/fail rates and time taken to execute each individual test. The specific tests to be executed may be manipulated by changing the Gradle build file (gradle.build). A further Gradle plugin provides support for using JaCoCo (eclemma.org/jacoco) that produces test coverage information of the executed unit tests. The JaCoCo plugin provides a range of common code coverage metrics including branch coverage and line coverage: we choose branch coverage in this work, but JaCoCo and similar plugins provide an easy route to apply other metrics. Both the Java and JaCoCo plugins provide high-level summaries as web-based report or comma separated value files, which can be easily parsed by external programs.

By specifying which specific tests are to be run via the *gradle.build* file, the Java and JaCoCo plugins can be used to measure the wall-clock time required to execute a subset of all available Mockitos unit tests and report code-coverage.

In this work a bit-string of length n, where n is the total number of available tests, is used to encode subsets of tests; where a '1' at position x in the bit-string represents than the x^{th} test, out of all available tests, should be used and a '0' represents that it should be left out.

Using this encoding, we employ a multi-objective optimisation algorithm to search over the space of possible subsets. We use the NSGA-II [3] implementation from the MOEA framework (http://moeaframework.org/). The objectives optimised by NSGA-II are the elapsed wall-clock time from running a subset of tests (provided by the Java plugin) and the branch coverage of those tests (provided by the JaCoCo plugin).

2.1 Experimental Set-up

All of the results presented in this paper are based on version 2.0.44-beta of the Mockito framework. The parameters of NSGA-II were left at the defaults specified in the MOEA framework version 2.9; which include a population size of 100. NSGA-II was left to run for 5 hours, which is an appropriate length of time to be incorporated into nightly builds i.e. ready to be used by the team of developers the following day. The experiments presented minimise the number of tests used by the core packages of the Mockito framework. The Mockito packages considered are: org.mockitousage.basicapi.*, org.mockitousage.bugs.*, org.mockitousage.misuse* and org.mockitousage.verification.*, which contain 420 individual tests in total.

3 Results and Discussion

The Pareto front of possible subsets of tests generated by applying NSGA-II to test suite minimisation of Mockito is given in Figure 1. Running all available tests, for the packages described previously, requires 5.93 wall-clock seconds[4] and results in 47.82% branch coverage; the run time and branch coverage in Figure 1 are in relation to these values. Although the time savings may appear small, even short pauses can have a significant impact in breaking a developer's "flow" during the repeated write-compile-test cycle.

As can be seen in Figure 1, NSGA-II was able to significantly reduce the wall-clock time whilst maintaining a high proportion of the original branch coverage. Additionally, the generated Pareto front provides a graceful degradation

[4] Running on an Intel ©Core™ i7-4600U CPU @ 2.10GHz x 2.

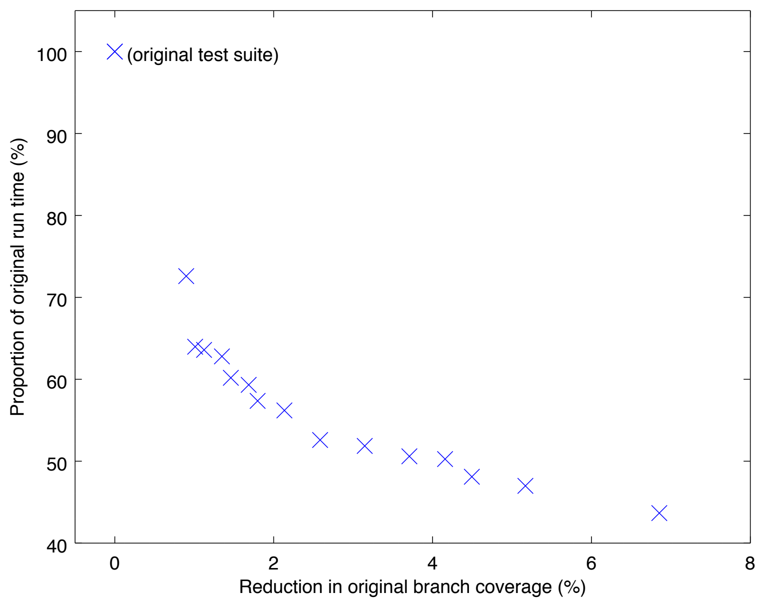
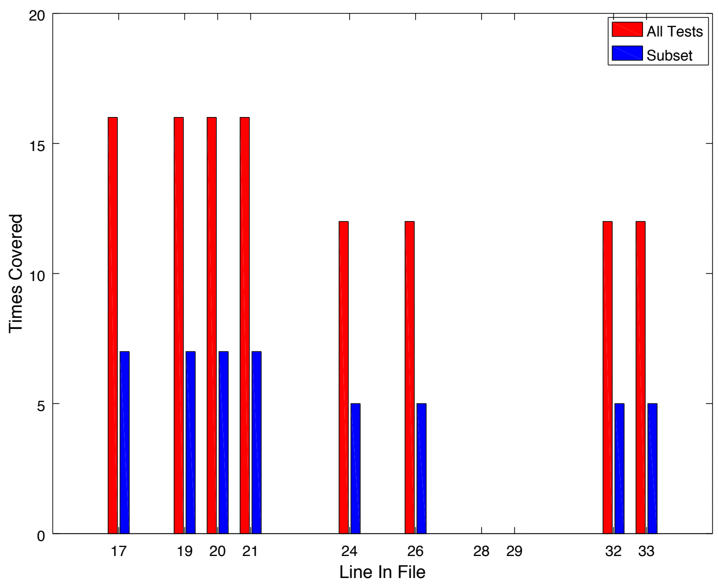


Fig. 1: Pareto front of regression test suite solutions found using NSGA-II

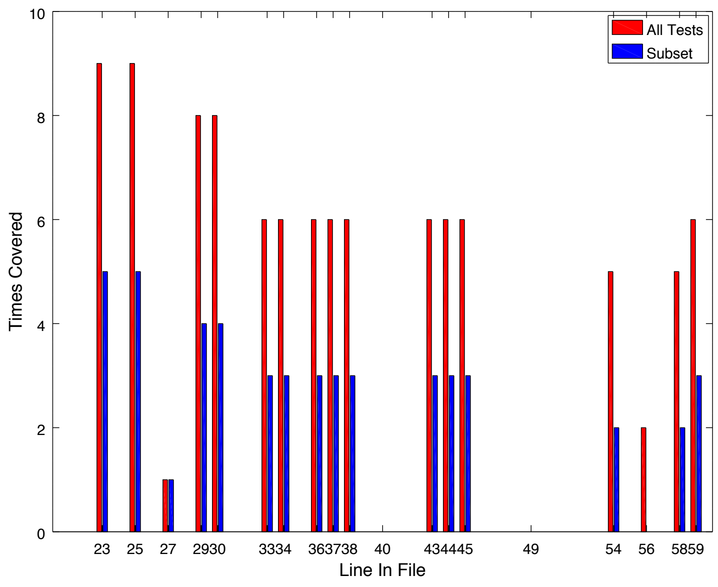
in branch coverage as the wall-clock time is reduced. This demonstrates how the presented method could easily be incorporated into a decision support process; where a human in-the-loop decides what level of branch coverage degradation is acceptable given the computational speed-up in test cycle time. This would be particularly useful when applying minimisation to larger test suites.

In order to demonstrate how this significant reduction in wall-clock time is afforded, with minimal reduction in branch coverage, a detailed view of the effect of removing redundant tests is presented. Figure 2 shows the number of times each line of two Mockito source files are executed at least once by each test in the original test suite. These results are contrasted with employing a subset of tests found using NSGA-II. Lines not covered by any of the tests can also be identified in the given plots e.g. line 28 in Figure 2a. The space between the bars represent non-executable lines i.e. comments and bracket placement.

As can be seen in Figure 2, when using all available tests each line of a given source file is often evaluated by a large number of individual tests. This indicates that there is a high level of redundancy in the test suite. However, when applying a subsets of tests found using NSGA-II, the number of tests which execute the same lines is reduced. This is why very comparable levels of test coverage are maintained whilst utilising a reduced test suite; redundant tests have been removed. The compromise of this method is that a small number of lines which were previously covered may no longer be tested; such as line 56 in Figure 2b.



(a) *AtLeastXNumberOfInvocationsChecker.java*



(b) *AtMost .java*

Fig. 2: Number of times each executable line of two Mockito source files are evaluated at least once by each individual employed test. Coverage from using all available tests and coverage from using a subsets of tests found using NSGA-II is shown.

4 Conclusions and Future Work

This paper applied test suite minimisation to a Java project through integration with the Gradle build system; no specific information regarding the Mockito was required. Therefore it is possible to apply the same approach to any project that uses Gradle, or even develop a plugin that automates the process. This is one instance of a wider trend in software development: the move towards automated build and deployment systems employing standardised interfaces represents an opportunity for the application and dissemination of SBSE.

Possible future work is to implement such a general plugin. The impact of test minimisation on the fault-finding ability of the test suite is of general interest and, by integrating with standard build tools as presented here, a larger-scale study could investigate this relationship.

In the described system all tests associated with each new candidate test suite were re-run in order to record overall converge and testing time; resulting in 5 hours of training time. However, this can be dramatically reduced in future work by caching the coverage and elapsed time of each individual test. This means that no code has to be re-evaluated when assessing new candidate test suites, only a series of computationally cheap look-ups.

Multi-objective optimisation was successful in minimising the Mockito framework's regression suite. The wall-clock time required by the test suite was reduced by ~50% whilst maintaining ~96% of the original code coverage. Therefore using such methods can save significant developer time during the write-compile-test cycle with limited effect on the amount of code covered by the test suite.

References

1. Back, T., Fogel, D.B., Michalewicz, Z.: Handbook of evolutionary computation. IOP Publishing Ltd. (1997)
2. Deb, K.: Multi-objective optimization using evolutionary algorithms, vol. 16. John Wiley & Sons (2001)
3. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evolutionary Computation 6(2), 182–197 (2002)
4. Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K.: Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In: Proc. International Symposium on Software Testing and Analysis (2015)
5. Gu, Q., Tang, B., Chen, D.: Optimal regression testing based on selective coverage of test requirements. In: Proc. Parallel and Distributed Processing with Applications (2010)
6. Harman, M., Burke, E., Clark, J.A., Yao, X.: Dynamic adaptive search based software engineering. In: Proc. Empirical Software Engineering and Measurement (2012)
7. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys 45(1), 11 (2012)
8. Yoo, S., Harman, M.: Pareto efficient multi-objective test case selection. In: Proc. International Symposium on Software Testing and Analysis (2007)
9. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)