Information Sciences 334–335 (2016) 1–16

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Multi-objective optimisation for regression testing

Wei Zheng [a,*], Robert M. Hierons [b], Miqing Li [b], XiaoHui Liu [b], Veronica Vinciotti [b]

[a] School of Software and Microelectronics, Northwestern Polytechnical University, Yantan 127# the west road of you yixi'an, Shanxi 710072, China

[b] School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, Middlesex UB7 7NU, United Kingdom

ARTICLE INFO

Article history:
Received 14 March 2015
Revised 15 September 2015
Accepted 19 November 2015
Available online 2 December 2015

Keywords:
Software engineering
Regression testing
Test suite minimisation
Multi-objective search

ABSTRACT

Regression testing is the process of retesting a system after it or its environment has changed. Many techniques aim to find the cheapest subset of the regression test suite that achieves full coverage. More recently, it has been observed that the tester might want to have a range of solutions providing different trade-offs between cost and one or more forms of coverage, this being a multi-objective optimisation problem. This paper further develops the multi-objective agenda by adapting a decomposition-based multi-objective evolutionary algorithm (MOEA/D). Experiments evaluated four approaches: a classic greedy algorithm; non-dominated sorting genetic algorithm II (NSGA-II); MOEA/D with a fixed value for a parameter c; and MOEA/D in which tuning was used to choose the value of c. These used six programs from the SIR repository and one larger program, VoidAuth. In all of the experiments MOEA/D with tuning was the most effective technique. The relative performance of the other techniques varied, although MOEA/D with fixed c outperformed NSGA-II on the larger programs (Space and VoidAuth).

1. Introduction

Software testing is one of the main verification and validation methods used in software development. Regression testing occurs whenever the *system under test (SUT)* or its environment changes, with the regression testing process involving the SUT being tested with the current test suite T or some subset of this (possibly with additional test cases for new code). Although there are many tools that reduce the cost of regression testing by re-applying the test suite T, regression testing can be a time consuming and expensive process. This is the case if the test suite is large, there is some manual element in testing, or test execution takes up significant resources. It has been found that there are situations in which regression testing takes weeks to run [7]. In addition, many systems are now developed in a continuous manner, with builds being frequently tested, and in such situations the time taken by regression testing can be a significant issue. There has thus been considerable interest in methods that reduce the cost of regression testing (see, for example, [1–3,7,10,11,13,17,29,32,33,37,41]). A good overview of this work can be found in a recent survey [42].

A natural approach, to reducing the cost of regression testing, is to select a subset T' of the test suite T, with methods that do this being called *test suite minimisation* methods [42]. The aim of such methods is to choose an effective but small subset and many approaches have been devised [1–3,10,11,13,17,41]. In the absence of fault information, it is normal to use some form of coverage as a proxy for effectiveness. For example, we might want a small subset T' where the percentage of statements

* Corresponding author. Tel.: +86 13991976695.
E-mail address: zhengweizr@gmail.com, wzheng@nwpu.edu.cn (W. Zheng).

http://dx.doi.org/10.1016/j.ins.2015.11.027
0020-0255/

executed in testing (statement coverage) is high. The problem is then usually seen as either maximising coverage for a given test budget or minimising cost for a given coverage, with initial work on this topic devising greedy algorithms (see, for example, [2,3,11]).

The previously mentioned approaches fix one property (cost or coverage) and then aim to optimise the other property. It has been observed that instead we might treat this problem as a multi-objective optimisation problem: we attempt to optimise two or more properties (such as cost and coverage) at the same time [10]. In multi-objective optimisation it is normal to place a partial order on solutions by saying that solution *xPareto dominates* solution *y* if *x* is superior to *y* for at least one objective and is at least as good as *y* for all objectives. For a given problem, the *Pareto set* is the set of solutions that are not Pareto dominated. This Pareto set provides a range of trade-offs between the properties optimised: the tester can then choose from these solutions. While typically it is not feasible to produce the Pareto set for a problem, optimisation algorithms might return an approximation to the Pareto set. Yoo and Harman implemented a multi-objective approach, applying a greedy algorithm and two versions of *non-dominated sorting genetic algorithm II (NSGA-II)* [41]. The approaches were evaluated on Space and four programs from the Siemens suite. Interestingly, while the versions of NSGA-II outperformed the greedy algorithm on the programs from the Siemens suite, the greedy approach was superior for Space [41]. Harman has also identified many different factors that might be considered in the test suite minimisation problem [10].

Yoo and Harman [41] appear to be the first researchers to apply a multi-objective optimisation approach to the test suite minimisation problem. This was an important step that led to valuable results and insights, but the work (naturally) had a number of limitations. One limitation was that while the multi-objective evolutionary algorithm used (NSGA-II) is known to perform well on many problems, there is the potential to apply more specialised approaches. In the evaluation, the main approach used to compare algorithms operated as follows: take the union of the non-dominated solutions produced by the algorithms, form a new set *P* of non-dominated solutions from these, and for an algorithm *A* determine how many of its solutions are not Pareto dominated by solutions in *P*. The authors also compared the sizes of the sets of non-dominated solutions returned by the algorithms, arguing that an algorithm that returns many solutions provides the tester with more options. While this approach is sensible, the evolutionary optimisation community has developed other approaches. Finally, the authors considered two and three objective problems.

In this paper we aim to significantly develop the multi-objective agenda pioneered by Yoo and Harman in three major ways, with the following contributions. Firstly a 'modern' evolutionary algorithm *decomposition-based multi-objective evolutionary algorithm (MOEA/D)* [43], not previously used, has been utilised to overcome some of the limitations as stated above. Secondly, the *Hypervolume (HV)* [49] of a solution has been used in the evaluation as it has a number of important benefits and is widely used in comparing the performance of multi-objective and many-objective algorithms. Thirdly, we have extended a multi-objective solution as proposed in Yoo and Harman's paper to a many-objective one as this would enable one to consider the testing of other important types of software. It transpired that it was necessary to adapt MOEA/D and in doing so we introduced a parameter *c* used in the normalisation. Some initial runs were performed and in these a value of $c = 0.3$ was found to be reasonably effective. This led to two versions of MOEA/D being used in the experiments: one in which we used this fixed value of *c* and one in which the value of *c* was chosen for each experiment ('variable *c*'). We also explored problems with two to four objectives; Yoo and Harman considered two and three objective problems. One might expect the nature of the optimisation problem to change as the number of objectives increases since, for example, fewer pairs of solutions will be related (under the relation that one dominates the other). In addition, our seven experimental subjects included an additional real-world program; one that implements a web-service.

Yoo and Harman considered two and three objective problems, but we have chosen several combinations of objectives in this paper. The most basic included two objectives only (cost and statement coverage) and might be seen as corresponding to the development of 'normal' software. The three objective case added in branch coverage. Branch coverage is required in a number of areas including automotive software (see, for example, [39]). For four objectives we used cost, statement coverage, branch coverage, and modified condition/decision coverage (required for safety critical components in avionics software [34]) so this might be seen as corresponding to the testing of critical software. In this way we have moved from a multi-objective optimisation problem to a many-objective one, allowing the testing of additional types of software.

In this paper, we compare four optimisation algorithms: a greedy algorithm, NSGA-II, and MOEA/D (with a fixed value of *c* or variable *c*). We used five smaller experimental subjects plus Space from the software-artifact infrastructure repository (SIR) and one larger real world system, VoidAuth, that implements the main processes of the Universal Payment Gateway (UPG) service. In all cases the experiments found MOEA/D with variable *c* to be the most effective technique, with the differences being statistically significant. The relative performance of the other methods varied, although the greedy algorithm was worst for the smaller SIR programs. Similar to Yoo and Harman, we found that the greedy algorithm outperformed NSGA-II on Space but, interestingly, both variants of MOEA/D performed better than the greedy algorithm. For VoidAuth the greedy algorithm outperformed NSGA-II for the two and three objective problems and outperformed MOEA/D with fixed *c* for the three objective problem. The relative performance of NSGA-II and MOEA/D with fixed *c* varied for the SIR programs, with neither being consistently better. However, in all cases MOEA/D with fixed *c* outperformed NSGA-II for VoidAuth. We also considered a specific scenario, where the tester decides to use a test suite that provides full coverage for every metric in the optimisation process. In every experiment MOEA/D with variable *c* returned the smallest test suite.

The paper is structured as follows. Section 2 provides an overview of previous work on test suite minimisation for software regression testing and Section 3 reviews approaches to evolutionary multi-objective optimisation. Section 4 explains how we