Immune and Evolutionary Approaches to Software Mutation Testing

Pete May[1], Jon Timmis[2], and Keith Mander[1]

[1] Computing Laboratory, University of Kent, Canterbury, Kent, UK
petesmay@gmail.com, k.c.mander@kent.ac.uk
[2] Departments of Computer Science and Electronics, University of York, York, UK
jtimmis@cs.york.ac.uk

Abstract. We present an Immune Inspired Algorithm, based on CLONALG, for software test data evolution. Generated tests are evaluated using the mutation testing adequacy criteria, and used to direct the search for new tests. The effectiveness of this algorithm is compared against an elitist Genetic Algorithm, with effectiveness measured by the number of mutant executions needed to achieve a specific mutation score. Results indicate that the Immune Inspired Approach is consistently more effective than the Genetic Algorithm, generating higher mutation scoring test sets in less computational expense.

1 Introduction

Software testing can be considered to have two aims [1]. The primary aim is to prevent *bugs* from being introduced into code - prevention being the best medicine. The second is to discover those un-prevented bugs, i.e. to indicate their *symptoms* and allow the *infection* to be *cured*.

Curing an infection is a two stage process of identifying and then correcting faults. These continue until all bugs in the code have been found, at which point a set of tests will have been generated that have reduced the failure rate of the program. Unfortunately, a tester does not know *a priori* whether faults are present in the software, posing an interesting dilemma: *how does a tester distinguish between a "poor" test that is incapable of displaying a fault's symptoms, and a "good" test when there are simply no faults to find?* Neither situation provides a useful metric. A heuristic to help aid this problem uses the notion of *test set adequacy* as a means of measuring how "good" a test set is at testing a program [2]. The key to this is that "goodness" is measured in relation to a predefined *adequacy criteria*, which is usually some indication of program coverage. For example, statement coverage requires that a test set executes every statement in a program at least once. If a test set is found inadequate relative to the criteria (e.g. not all statements are executed at least once), then further tests are required. The aim therefore, is to generate a set of tests that fully exercise the adequacy criteria.

Typical adequacy criteria such as statement coverage and decision testing (exercising all true and false paths through a program) rely on exercising a program with an increasing number of tests in order to improve the reliability of that program. They do not, however, focus on the cause of a program's failure, namely the faults. One criteria does. Known as *mutation testing*, this criteria generates versions of the program containing simple faults and then finds tests to indicate their symptoms. If an adequate test set can be found that reveals the symptoms in all the faulty program versions, then confidence that the program is correct increases. This criterion forms an adequacy measure for the *cure*.

In previous work, we outlined a vision for a software mutation system that exploits Immune-Inspired principles [3]. In this paper we present a limited set of our results from our investigations, detailed further in [4]. The remainder of this paper is organised as follows: Section 2 describes the mutation testing process. Next, in section 3, the notion of algorithm effectiveness with respect to evolving test data using mutation testing is introduced. Section 4 details the Immune and Genetic algorithms, which are compared in section 5.

2 Mutation Testing

Mutation testing is an iterative procedure to improve test data with respect to a program, as indicated in Figure 1. The initial parameters to the process are the *PUT* (Program Under Test), a set of mutation operators (Mutagens), and a test set population, *T*. Initially, by using an oracle, the PUT must be shown to produce the desired outputs when executed with the test set T. If not, then T has already demonstrated that the PUT contains a fault, which should be corrected before resuming the process.

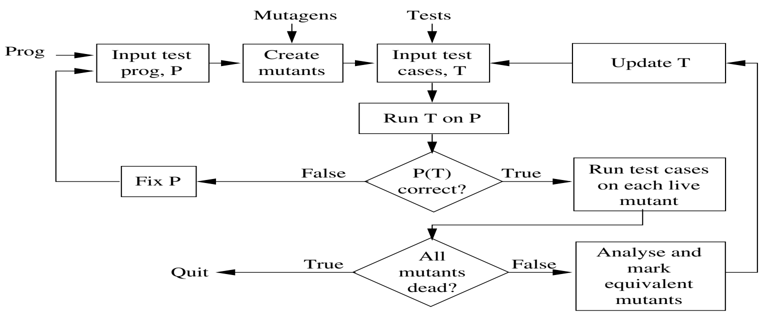

Fig. 1. The Mutation Testing process. *Diagram reproduced from [4], modified from [5].*

The next stage is to generate a set, *M*, of fault induced variants of the PUT that correct for simple faults that could have occurred. Each variant, or *mutant*, differs from the PUT by a small amount, such as a single lexeme, and is generated by a mutagen. These mutation operators alter the semantics of the PUT depending on the faults they classify. For example, the *relational operator* mutagen will generate a number of mutants where each one has an instance of