Abstract. In the context of deductive proof, formal specification (and thus proofs) may only cover a small part of the program behaviours. We discuss the relevance of applying a mutation analysis, proposed to evaluate the quality of a test set, to evaluate if a specification is able to detect predefined types of faults.

1 Introduction

A primary purpose of testing is to detect software failures. It implies running the software item in predetermined conditions (input selection), analyzing the obtained results, and identifying errors [16]. Testing can never completely establish the correctness of a program. For this reason, several methods (among which mutation analysis) have been put forward to increase confidence with respect to the test set provided.

Mutation analysis was introduced by DeMillo in 1978 [6]. Its main purpose is to evaluate the quality/adequacy of a test set. The basic idea is to insert changes into the program being tested, and to check if the tests are able to detect the difference between the original program and its variations. It is mainly used for unit testing evaluation. Since 1978, mutation analysis has been widespread, improved and evaluated. It has been adapted for several programming languages [12]. Andrews et al. have demonstrated that this technique can provide a good indication of the fault detection ability of a test suite [3].

Testing is often opposed to verification techniques. In the following, we focus on theorem proving, which aims at assessing that an artefact (code or model) conforms to a (set of) property thanks to a sequence of deductions [19][17]. It is considered to be a reliable validation technique since it is based on well-founded mathematical deductions. However, when verifying an artefact with respect to a description (specifications), it is not necessary to describe all the facets of the artefact’s behaviours. For instance, let us consider a program to sort a list. It can be specified that the resulting list should be sorted and should contain all the initial elements. But, one can forget to state that the size of the list should be unchanged. A code that adds new elements or removes some repeated values
can be proved to be correct with respect to the description although it is not the expected behaviour.

In the following, we provide an example of how mutation analysis can help increase confidence with respect to the specification. Section 2 gives details about mutation analysis in the context of testing. Section 3 shows how mutation analysis can be used for proof. Section 4 considers some perspectives.

2 Mutation Analysis for Testing

Mutation analysis consists in introducing a small syntactic change in the source code of a program in order to produce a mutant (for instance, replacing one operator by another or altering the value of a constant, etc.). Then the mutant behaviour is compared to the original program. If a difference can be observed, then the mutant is marked as killed. If the mutant has exactly the same observable behaviour as the original program, it is equivalent.

The original aim of the mutation analysis is the evaluation of a test set. To do that, one has to produce all mutants corresponding to a predefined fault model. If the test set can kill all non-equivalent mutants, the test set is declared mutation-adequate. This means that the tests are able to discriminate the behaviours of all the mutants from the original program. Mutation analysis does not confirm the correctness of the program under test. It only focuses on the adequacy of the test data: the original program may be faulty and a mutant correct. If the tests are able to differentiate their behaviours, the tester may be able to detect the fault in the original program and correct it.

The adequacy of the test set is evaluated thanks to the mutation score (also called adequacy score). The mutation score is the percentage of non-equivalent mutants killed. For a program $P$, let $M_T$ be the total number of mutants produced with respect to a particular fault model $F$. Let $M_E$ and $M_K$ be the number of equivalent and killed mutants. The mutation score of the test set $T$ with respect to the fault model $F$ is defined as: $MS(P, T, F) = \frac{M_K}{M_T - M_E}$. A test set is mutation-adequate if the mutation score is equal to 1.

Example: The max Program

Let us consider the following simple example of a program that computes the maximum of two values $x$ and $y$ (max).

```c
int max(int x, int y) {if (x > y) return x; else return y;}
```

If one applies a fault model such as the one defined in [1] on the max program, one would obtain mutants in which $>$ is replaced by $<$ or $\geq$; each instance of $x$ is replaced by $y$, $x-1$ or $x+1$; similarly each instance of $y$ is replaced by $x$, $y-1$ or $y+1$. For the sake of brevity, only three mutants are shown below.

---

1 Mutation testing aims at producing tests until the maximal mutation score is obtained.