8. Mutation Analysis

Test-suites Assessment Using Program Mutation

Andrea Polini

Fundamentals of Software Testing
MSc in Computer Science
University of Camerino

Mutation Analysis

- Mutation Analysis is an alternative techniques used to assess the quality of a test suite
- It can be generally considered a White box technique
- The tecnique is based on the concept o Program Mutation, i.e. the generation of a program from the original one with “planned differences”

What about its effectiveness?

Syntax and semantics of mutants

Let's consider the following function:

$$f(x,y) = \begin{cases} x + y & \text{if } x \leq y \\ x \times y & \text{otherwise} \end{cases}$$

A mutant is a syntactically different function that can be generated introducing a "small" modification in the definition

The idea of small has to do with the number of values for which the function changes its evaluation (semantics)

Strong and weak mutations

In order to distinguish a mutant from its parent we can compare their evaluation of specific values:

- strong mutations: the comparisons is performed only on the produced output
- weak mutations: the comparisons is performed on intermediate states while the evaluation is under execution

Why mutating a program?

- The competent programmer hypothesis
- The coupling effect

Testing assessment using mutation

Mutation analysis is structured over the following steps:

1. program execution over the test suite $\mathcal{T}$
2. mutant generation
3. mutant execution and classification

Mutant generation

- from the original program (parent) a set of similar programs are generated (mutants)
- each mutant differs from the parent for a single “slight” detail (first order mutants) obtained applying a substitution rule described in a mutation operator
- Mutation operators are generally language and context dependent
- Mutants are stored for being successively retrieved one by one

It is possible to generate high order mutants but some issues do not make this option generally practical

Mutant execution and assessment

- Each mutant previously generated is submitted to the test suite $\mathcal{T}$ till a test observes a divergence with respect to the parent program
- otherwise in case no test is able to spot a divergence the next mutant is considered
- According to the result of the previous assessment the mutant is marked as:
  - killed a test was able to observe a difference
  - alive no test was able to observe a difference

Disclaimer: some of the generated mutants could be directly rejected by the compilers

Equivalent mutants

It is possible the applying a modification the generated mutant is semantically equivalent to the parent program

Unfortunately the “equivalence problem” cannot be computationally solved in the general case.

Some heuristics can help the identification of equivalent mutants

- Trivial Compiler Equivalence (TCE)

In the general case the tester will have to understand if alive mutants are indeed equivalent mutants Let’s consider the following function:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ x^2 & \text{otherwise} \end{cases}$$

Mutation score

Let's consider $\mathcal{M}$ as the set of generated mutants, $\mathcal{E}$ the set of equivalent mutants, and $\mathcal{D}$ the set of detected mutatnts, the mutation score is given by the following formula:

$$MS(\mathcal{T}) = \frac{|\mathcal{D}|}{|\mathcal{M}| - |\mathcal{E}|}$$

Mutation operators

The definition of the mutation operators are really the central point of the approach! Generally they are the result of many empirical research activitiescarried on by different groups that given a language, and in some cases an application context, can define the most useful mutation operators

Finding the right balance is not easy....the more mutants the more time you need to perform the assessment

In any case having th epossibility to select a subset of mutation operators is generally possible

The case of Java

- Traditional operators
  - Aritmetic expressions
  - Bynary aritmetic expressions
  - Logical Connectors
  - Relational operators
  - Arithmetic of logical expressions (unary operators)
- Inheritance
  - Variables – removal of redefinitions in subclasses
  - Subclasses – add the definition of a variable in the superclass
  -
- Polymorphisms and dynamic binding
- Method overloading
- ...

The case of Solidity

The Journal of Systems & Software 193 (2022) 111445

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

ELSEVIER

SuMo: A mutation testing approach and tool for the Ethereum blockchain☆

Morena Barboni [a,b], Andrea Morichetta [b,*], Andrea Polini [b]

[a] IASI–CNR, Rome, Italy
[b] University of Camerino, Camerino, Italy

ARTICLE INFO

Article history:
Received 19 April 2021
Received in revised form 9 May 2022
Accepted 14 July 2022
Available online 21 July 2022

Keywords:
Test automation
Mutation testing
Smart contract
Blockchain
Solidity

ABSTRACT

Blockchain technologies have had a rather disruptive impact on many sectors of the contemporary society. The establishment of virtual currencies is probably the most representative case. Nonetheless, the inherent support to trustworthy electronic interactions has widened the possible adoption contexts. In the last years, the introduction of Smart Contracts has further increased the potential impact of such technologies. These self-enforcing programs have interesting peculiarities (e.g., code immutability) that require innovative testing strategies. This paper presents a mutation testing approach for assessing the quality of test suites accompanying Smart Contracts written in Solidity, the language used by the Ethereum Blockchain. Specifically, we propose a novel suite of mutation operators capable of simulating a wide variety of traditional programming errors and Solidity-specific faults. The operators come in two flavors: Optimized, for faster mutation testing campaigns, and Non-Optimized, for performing a more thorough adequacy assessment. We implemented our approach in a proof-of-concept work, SuMo (SOlidity MUtator), and we evaluated its effectiveness on a set of real-world Solidity projects. The experiments highlighted a recurrent low Mutation Score for the test suites shipped with the selected applications. Moreover, analyzing the surviving mutants of a selected project helped us to identify faulty test cases and Smart Contract code. These results suggest that SuMo can concretely improve the fault-detection capabilities of a test suite, and help to deliver more reliable Solidity code.

1. Introduction

Starting from their first appearance the adoption of blockchain technologies has drastically increased, and novel application scenarios are constantly investigated. This has greatly to do with the inherent trust that can be brought, in some contexts, by the adoption of these technologies. Moreover, in recent years many blockchain technologies have been augmented with support for smart contracts. These can be considered as software programs that can be deployed and executed over a blockchain and that, during their execution, will generate data (transactions) that will be stored within the blockchain itself. Such smart contracts permit the stakeholders that take part in the smart contract execution, to have enough guarantees so as to trust that the activities performed by the other interacting parties are in line with what is specified in the contract itself. Such mechanisms have been fruitfully adopted in the engineering of software systems (Porru et al., 2017), and in particular in the integration of inter-organizational and collaborative software systems (Corradini et al., 2020, 2022). Compared to traditional software systems, the development of smart contracts presents unique characteristics and challenges as a consequence of the underlying deployment and execution environment (Zheng et al., 2017; Destefanis et al., 2018). There is then a need for novel and appropriate testing tools that allow the developer community to write and deploy safer code. In particular, it is possible to identify several reasons why smart contracts generally ask for high-reliability guarantees, and a thorough testing process. The following is a, probably non-exhaustive, list of relevant characteristics (Barboni et al., 2021):

- **Smart Contracts manage valuable assets**: Smart contracts can control large amounts of cryptocurrency and other valuable assets. Deploying faulty code can result in the accidental loss of the assets held by the contract. The potential financial gain, and the anonymous nature of the blockchain further act as an incentive for attackers. Even a small loophole in the code can allow malicious users to drain large amounts of funds. A typical example is the famous DAO attack (Mehar et al., 2019), in which a reentrancy vulnerability

☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail addresses: morena.barboni@unicam.it (M. Barboni), andrea.morichetta@unicam.it (A. Morichetta), andrea.polini@unicam.it (A. Polini).

https://doi.org/10.1016/j.jss.2022.111445
0164-1212/© 2022 Elsevier Inc. All rights reserved.