Università degli Studi di Camerino

Scuola di Scienze e Tecnologie

Corso di Laurea in Informatica (Classe L-31)

ReSuMo: A regression mutation testing tool for Solidity Smart Contracts

Laureandi

Caripoti Eugenio
Latini Ludovico
Longhi Sara

Relatore

Andrea Morichetta

Matricole

Caripoti Eugenio 106091
Latini Ludovico 106239
Longhi Sara 105829

Correlatore

Morena Barboni

A.A. 2020/2021

Indice

1. Introduction

In the last period, we often hear about new technologies related to blockchain and cryptocurrency. Most people believe that blockchains is another way to make transaction using the cryptocurrency and investing money on them, trying to speculate on their value and sell on the best moment. Others think that crypto and in particular *"bitcoin"* is used only to pay for illegal items bought on the dark web, using the feature of the blockchain to make untraceable transactions. But it is more than that, for example Ethereum allow user to enjoy a lot of different services like NFT, Metaverse and Multiverse. Thanks to the innovative smart contracts, developers became a part of this system and have the opportunity to create decentralized programs also known as DApps. These apps and other services permit the birth of the Web3.0, an evolution of the web towards a decentralized paradigm. An example of a blockchain service could be the supply chain, a process that allows to bring to the market a product or service, transferring it from the supplier to the customer. It is a complex process that involved a lot of professional figures. But not only services are born with the blockchain, another important aspect of the Web 3.0 are the applications of various kind that run it, for example, apps referred to payment, trading and investment like: Tornado cash, Augur or PoolTogether, or crowdfunding system for blockchain like Gitcoin Grants and also a digital collectibles world like in OpenSea.

However, because of the youth of these DApp, they have some limits and vulnerabilities, so it might be possible for a developer to releases a smart contract containing bugs. Fixing a deployed DApp is an expensive process due to the fact that it requires a great waste of resources and a payment of a fee, which does not make the process sustainable for numerous changes that a new DApp could require. A good approach for ensuring the correct functioning of the DApps is testing the code before the deployment. Furthermore, for a deeper analysis could be necessary carrying out a mutation testing process, which is more expensive but it surely a more complex and complete than every other testing approach. In order to solve these problems, the community developed a lot of frameworks. This work is focused on change one of those frameworks, in particular, we created ReSuMo which is a fork of SuMo[1] project. Our program tries to improve the performance, the accuracy, and the usability of SuMo. We sought to use it with some DApps, and we notice that its performance decrease as the complexity of the project increases, in addiction to that we observe that the result of the mutation testing process: *"mutation score"* was very rough. In order to solve these problems, we implemented a Trivial Compiler Equivalence algorithm directly in SuMo code. Another advance that we did in this project is the addition of a Regression testing framework to decrease the execution time.

Lastly, we found some difficulties using SuMo to testing projects, in fact SuMo re-

[1]SuMo is a project of mutation testing for Smart Contracts

quires a lot of parameters to set that need to be set manually because SuMo is a CLI framework. This policy of use could discourage a novice user, in fact we designed and implemented a user-friendly GUI.

This thesis born with the aim of present our improvements in SuMo that not only consist in what we said above but include the development of some reports in order to describe with greater precision the data coming from the mutation and the regression testing session. In addiction to that, we also describe the validation process of our feature.

2. Background

2.1 Blockchain

After the "financial industry crash" in 2008 an anonymous person or a collective named **Satoshi Nakamoto** created a paper where a protocol was developed for a digital cash that used an underlying cryptocurrency called **Bitcoin** based in a system for the management of information called **blockchain**. A blockchain can be thought as a chain of computers that combine their computing power in order to form a network, owners of these computers are called *"miners"*. It is composed by blocks link one to each other, a block is a data structure that contains information permanently recorded, in most cases these concern transactions and assets. Each block has two **hash** fields, one is related by the previous block and the other one belongs to the current block. These hash has made with a cryptography algorithm, using the information stored in the other's field of the block to create it, like in Fig. 2.1 and 2.2. Blockchain presents a lot of advantages for users in multiple fields, in fact it allows exchanging money in complete safety, enters into unchangeable contracts between the parties and develop apps without points of failure. The safety behind a blockchain concerned the fact that it is not possible to modify the contents of a block throughout the blockchain, hypothetically if an attacker changes the content of a block, he needs to change all the block of the chain, the entire history of commerce on that blockchain; he needs also to change this information in all the peer involved in the mechanism. For these reasons, blockchain for definition can not be hacked or modified without the consent of the rest of the network.

In addiction, this protocol is decentralized that means no one has the possibility to control it because it has no owner, so it cannot be censored or modified for the purposes of an individual. The blockchain belongs to the community that democratically decides its fate and keeps it active. This network is also much easier to scale, as you can simply add more machines to the network to add more compute power. Like an onion network or an IRC network, in decentralized networks information is not passing through a single point, but it passes through a number of different points, this makes it much more difficult to track across a network, so they offer more privacy than the centralized one. This type of system has many benefits, in fact it exploits the passage of messages and distributed networking to make sure to store data across its entire network and avoid having a single point of failure.

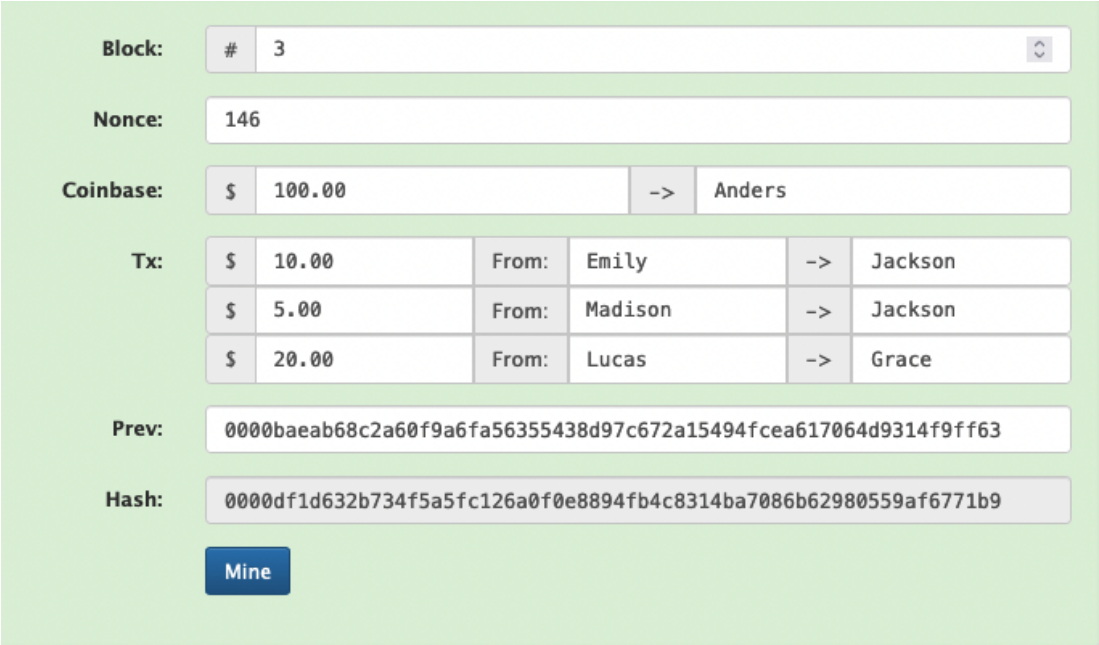


Figura 2.1: An example of a block with some transaction done

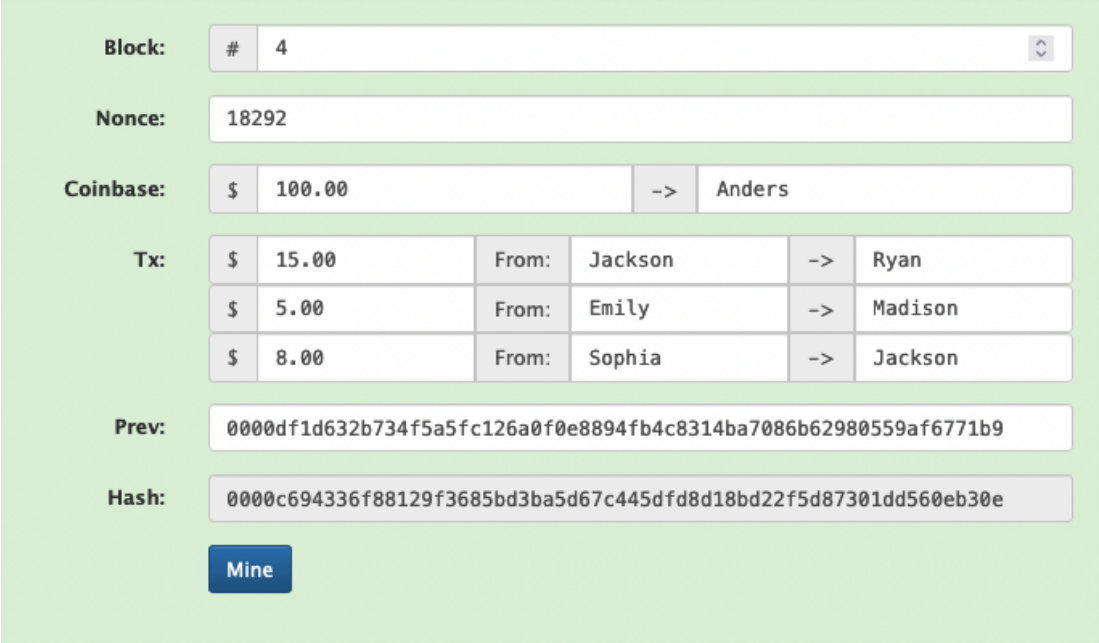


Figura 2.2: The consecutive block of Fig. 2.1

2.1.1 Ethereum Blockchain

Ethereum is an innovative blockchain described for the first time in a *white paper* [But13] from the programmer and the co-founder of the *"Bitcoin Magazine"* Vitalik Buterin in December 2013, but its development starts only in the early 2014. Ethereum, in addition to the exchange of ETH[1], permits others operations that are solely possible on this blockchain. For example, Ethereum allows the use of a special token

[1] Also known as ether, the Ethereum currency

which represents the deed of ownership and the certificate of authenticity stored in the blockchain called NFT, making it impossible to steal or copy the property of an object.

Ethereum's technology has also made possible the advent of Web 3.0. Web 3.0 represents the next phase of the evolution of the Word Wide Web. Web3 is built upon the core concepts of decentralization, openness, and greater user utility and allows people another type of web far from the concepts of using an app or a web app owned by a multinational which controls its behaviour. With this new web, data could be stored in multiple locations simultaneously and hence be decentralized. This would break down the massive databases currently held by internet giants like Meta and Google and would hand greater control to users. In addiction, web3-based applications are designed and programmed in an open-source perspective, it means that they are subject to a policy that sees, in the community, an opportunity to share the application code in order to make it freely available for possible modification and redistribution. Web 3.0 will also be trustless that means that the network allows participants to interact directly without going through a trusted intermediary and permissionless, that is, anyone can participate without authorization from a governing body. Actually, Web 3.0 is a utopian and auxiliary theory that could become real with the increase of the adoption of the system of the blockchain like system as the base of our interactions with internet. There are some business that are already developed their application on the Ethereum blockchain, some of the most famous application developed on this system are: Golem (a decentralized supercomputer that anyone could access), EtherTweet (an alternative to Twitter), Augur (a decentralized platform built on the Ethereum blockchain).

In addiction, one of the main features that distinguishes Ethereum from others blockchain is that it makes it possible to run *"smart contracts"*. The term "Smart Contract" means a digital agreement between two or more parties that is made reliable by the blockchain. In fact, the two parts are forced to respect the terms, and they cannot break the agreement, because of the automatic execution of the smart contract acting as an intermediate. A typical example for explaining Smart Contracts are the crowdfunding campaign made by a lot of websites. In this context, the two figures, who create the campaign and who donate money, need to trust the intermediary that has the task to give the money back if the project don't take place or give them to the founder to finance the project. A smart contract act as intermediate and make sure that the two parts don't need to believe each others or rely on a third party figure because the blockchain ensures the fact that the initial agreements will be respected, enforcing the terms defined within the code of the smart contract. They can be safely used to perform numerous operations: election systems, domain name registration, financial markets, crowdfunding platforms, intellectual property, etc. They are written in Solidity, a Turing-complete coding language similar to JavaScript, and it could be used for more than one activity, in fact Smart Contracts are involved in the developing of decentralized application that take the name of *"DApp"*.

Decentralized applications run in Ethereum blockchain through a series of rules that must be respected, in fact they need Ether founds to work so they have to be linked to a special account. There are two types of accounts: **Externally Owned Account** (EOA) and **Contract Account** (CA). they differ for some particular aspects. The **EOA** account is a type of account that Ethereum users have to create in order to execute transactions, it has a private key, that users can use for controlling founds and transactions, and a public key that becomes a unique reference to that account for the other users. This type of account can send transactions to both EOA and CA accounts.

On the other hand, **CA** accounts are multi-restricted type of accounts, CA can

be only used for operations on its storage and fire calls to other contracts, but only in response to a transaction. With a CA, you cannot access to an Ethereum smart contract until it is deployed on the blockchain. Finally, it has an address that permit other accounts to interact with it.

An important characteristic of Ethereum accounts is the fact that they are linked to an **Account State** that contains these data[2] (Information as in figure 2.3):

1. **Balance**: With balance is intended the total number of Ether owned from an account.
2. **StorageRoot**: A particular attribute that contains the Merkle-Patricia trie root[3], and linked only to CA accounts
3. **CodeHash**:A hash referred to the code of an account on the Ethereum Virtual Machine(EVM)
4. **Nonce**: Count the transaction made by an EOA or the number of generated contracts for a CA

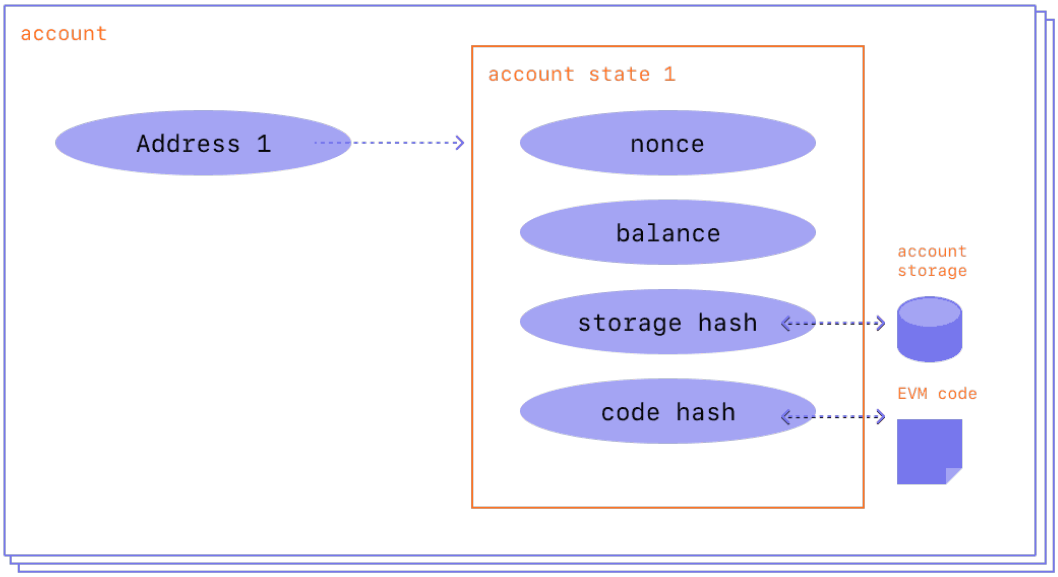


Figura 2.3: An example of an "Account State" data

[2]https://ethereum.org/en/developers/docs/accounts/
[3]https://medium.com/@chiqing/merkle-patricia-trie-explained-ae3ac6a7e123

As we have seen, users have to use accounts in order to make transactions, these transactions are saved in a particular data structure named **World State trie** that combined with the old information formed a state. Ethereum uses blocks like an image of the state of the blockchain and every time an account perform an operation in the blockchain is created a new block so a new snapshot of the state of the blockchain. The last block of the chain represents the current world state, in fact, it is possible to know information about the balances referred to all the Ethereum accounts after recent operations have been executed, by looking this block. A visual example of the state concept can be seen in figure 2.4.

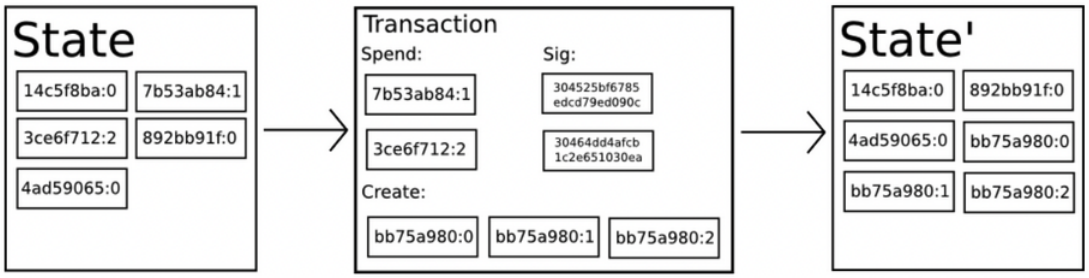


Figura 2.4: An example of the World State trie

The execution of a transaction alters the state of the blockchain, this process is irreversible and unchangeable.

In Ethereum, three types of transactions are permitted:

1. **Regular Transaction**: A ETH transfer between two wallet
2. **Contract Deployment Transaction**: The creation of a Contract Account on Ethereum
3. **Execution of a contract**: The execution of an operation written in a Smart Contract

As seen before, smart contracts can be executed in the Ethereum blockchain, to make it possible Ethereum has created a stack-based environment for running the code that takes the name of **Ethereum Virtual Machine** or **EVM**. EVM allows running Turing complete languages and participating each Ethereum node in the verification protocol. But as a result of complex operations written in smart contracts, the validation process could become computationally expensive. In addiction, programmers could create a non-terminating program executions that can potentially freeze the participating nodes. In order to solve these problems, Ethereum creates a **gas mechanism** with two specific purpose: Who commissions a transaction must pay for its execution, for its storage and its computation in order to limit the use of the computing power and compensate miners for their work.

All the mechanism of a smart contract on the Ethereum blockchain is managed through the parameters of *gas price* and *gas limit*. In particular, gas price represents the price of a single unite of gas and it is governed by the market that, based on the availability of the miners, decides the cost. The gas limit represents the amount of gas that can be use for executing the transaction.

The Ethereum virtual machine allows the employ of smart contracts, in particular they are written in high-level programming language such as Solidity, but they need

to be translated to a low-level language in order to be compiled from machines. When a smart contract starts its compiling process, it is translated in **bytecode** and the bytecode can be translated in a series of operation codes named **opcodes**. These opcodes are the part of a smart contract that is associated with a cost expressed in terms of gas, and are therefore the parts that increase the price of the compilation. Smart contracts and blockchain environment present several problems related to the technology itself and its youth:

- **Smart Contracts bug fix**: Smart Contracts once they are deployed on the blockchain cannot be easily fixed. Correcting a bug is a very expensive process, that requires the deployment of a new correct smart contract, paying the fee and consuming gas. Smart contract does not follow the typical software life process, the deployed version should be the final one.
- **Smart Contracts bug consequences**: Smart contracts, being type of contracts that involve a cryptocurrency blockchain, they include money in their transactions and a bug may cause a loss of assets.
- **Smart Contracts programming language**: The two main programming language for writing Smart Contracts are Solidity and Viper. These programming language are young and in continuously evolving, and for these reasons they open the possibility to a series of issue and vulnerability that are still being discovered.

In this context, writing good code and secure its correct functioning is one of the possible operation for preventing future problems.For these reasons, testing Smart Contracts is not only a well practice, but is a necessary practice in order to be sure to writing safe code.

Smart contract as said before is a type of digital contract that needs to be tested a lot because it and the Ethereum blockchain system present a series of not negligible problems. In fact, they usually control balances and for a code bug is possible to lost money, in addiction, transaction are irreversible, so if there are no code bugs it is however possible that the programmer is wrong to make a transaction and it is not possible to recover assets lost during the smart contract execution. Furthermore, correcting a bug on a deployed smart contract or DApp is a But one of the most dangerous problems is the youth of Solidity, in fact a young and continuously evolving programming language such as Solidity open the possibility to a series of issue and vulnerability that are still being discovered. So testing solidity smart contract in not only a well practice but is a necessary practice in order to be sure to writing safe code.

2.2 Mutation testing

One important part of a software development is the testing. Testing activity allows the developer to verify that the code is working, and it can do it via different approach:

- System test
- Integration test
- Unit test

If good tests assure our code quality, how can we check if our tests are good enough? To answer this question, several methods to assure the test suites quality have arisen. There are a lot of approach to do this type of testing like: the code coverage tools which measure how much code is executed during testing, or the property based testing which uses several input on a single test case and checks the program behaviour.

The framework on which we have worked adopt the technique of **Mutation Testing.** It is a white-box technique for evaluating and improving the quality of a test suite. Mutation testing is based on errors' insertion in the code with the purpose to simulate a programming mistake, creating a faulty copy of the original file called **mutant**, in Listing 2.1 there is an example of mutant. Later, running the test suite on it allows developers to understand if the error has been detected or not. In fact, the test failure means that the mutation inserted in the code has been recognized, and the test suite is qualitatively good. Mutation testing need a set of rules called **operators** that specify how the source code must be changed to create mutants.

```
1 //Original
2 function sum ( uint a , uint b ) public returns ( uint ) {
3 return(a+b) ;
4 }
5
6 //Mutant
7 function sum ( uint a , uint b ) public returns ( uint ) {
8 return(a-b) ;
9 }
```

Codice 2.1: Mutant in Solidity

In the example the mutation simply changes the sign of the return operation, in order to simulate an error, there are also more complex operators that modify for example the visibility and the type of variable, others swap the returns values and in generals they simulate every type of programming error that could occur during the development. The main purpose of the mutation testing is to try to create a wrong piece of code that a test might recognize. Studying the behaviour of a test suite, developers can understand the accuracy of their code. If a test run on a mutant and fails, it means that the test detect the changes and correctly returns an error, in this case the mutant will be reported as **killed**. Otherwise, if the test suite does not detect the change made by the mutation and the test gives a positive response, the mutant will be reported as **live**. It can happen that, after a mutation applies the changes, the created mutant cannot be compiled and consequently cannot be tested because of the syntactic errors in the code. These mutants are called **stillborn**, we have an example of stillborn in Listing (2.2). They are very common and represent a waste of time and resources that

slow down the mutation testing process.

```
1 //Original
2 function sum ( uint a , uint b ) public returns ( uint ) {
3 return(a+b) ;
4 }
5
6 //Mutant
7 function sum ( uint a , uint b ) public returns ( uint ) {
8 return(a--b) ;
9 }
```

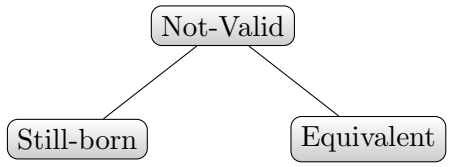
Codice 2.2: Stillborn mutant in Solidity

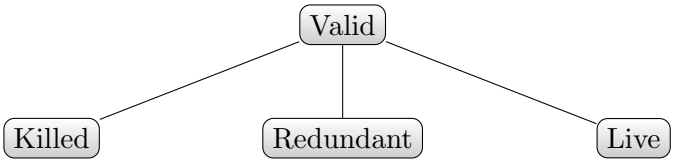
However, is not correct saying that all the lives mutants represent the possible enhancement of a test suite because of the presence of the **redundant** and the **equivalents** mutants. Equivalents are mutants that have the same behaviour of the original program as is shown in Listing (2.3), whereas redundants behaves in the same way of other mutant. Both of them are useless for the results, and detecting them before the testing phase is one of the better way to make the mutation testing process faster, less expensive and more reliable results.

```
//Original
function sum ( uint a , uint b ) public returns ( uint ) {
 return(a + b) ;
 }
//Equivalent Mutant
function sum ( uint a , uint b ) public returns ( uint ) {
 return(a -(- b)) ;
}
```

Codice 2.3: Equivalent mutant in Solidity

We can distinguish mutants in two categories: **Valid** are mutants that produce a relevant result, different from the one produced by the original code, while **Not-Valid** are mutants that cannot be tested or if tested produce a result equivalent to the code without mutations applied. The first cat:





The measure of adequacy of the test suite is made by running it on each mutant created, taking trace of the results previously obtained, and at the end of the process

calculate a rate of the detected mutations. A typical approach to estimate the quality of the test is to calculate the percentage of the killed mutants upon the totality of the valid in order to make a numerical estimate of the quality of the test suite. With this purpose, the mutation testing process eject a specific value that is called **Mutation Score** (Equation 2.1).

$$mutationscore = \frac{valid - live}{valid} \times 100 \tag{2.1}$$

The mutation score has the advantage to have at first glance feedback from the testing process. The higher is the mutation score, the more test suites can be considered reliable.

Advantages of MT[4]

1. A good approach to get the source program coverage. are irreversible
2. Ability to fully test the program.
3. A good sensitivity to errors
4. Detects redundancies and ambiguities in source code, capability to detect all faults in the program.
5. Increases system reliability and stability.

Disadvantages of MT

1. Extremely expensive as well as time-consuming due to large number of mutant programs that have to be generated.
2. An automation tool is a must for mutation testing due to its high complications.
3. Each mutation has the same number of test cases as of the original program; thus, numerous mutant programs have to be tested with the actual test suite.
4. It involves source code changes, so cannot be applied to Black box testing.

2.2.1 Trivial Compiler Equivalence

Trivial Compiler Equivalence is a very common technique used during the mutation testing process that exploits optimizations by existing compilers to detect equivalences in the set of mutants. TCE has the purpose to identify and exclude from the testing process **equivalent** and **redundant** mutants because if they were tested they would produce the same results as previous mutants or as the original contract. In addiction, they could slow down the mutation testing process and creating duplicated results in the final mutation score.

Two programs, after the compilation phase, could have the same binary code, comparing these binary files, TCE can detect mutants that are **equivalent** to the original program or mutants that are identical to others mutants that are **redundant** mutants. Formally, we can define the TCE function as follows[DS19]:

[4]https://www.tutorialspoint.com/mutation-testing-in-software-testing-mutant-score-and-analysis-example

- **o** is the original program
- **M** are the set of mutants generated by o
- **a** and b are two programs
- **Ω** is a binary relation, such a Ω b if a and b have identical binary code
- **TCE-equivalents mutants**: TCE_EQ is a set of equivalents mutants (Equation 2.2):

$$TCE\_EQ = \{a \in M | o\ \Omega\ a\} \tag{2.2}$$

- **TCE-redundant mutants**: TCE_RE is a set of redundant mutant (Equation 2.3):

$$TCE\_RE = \{a \in M | \exists b \in M : a\ \Omega\ b\} \tag{2.3}$$

  There are symmetrical (Equation 2.4) and transitive (Equation 2.5) properties associated:

$$\forall a, b \in TCE\_RE, a\ \Omega\ b \Rightarrow b\ \Omega\ a \tag{2.4}$$

$$\forall a, b, c \in TCE\_RE, a\ \Omega\ b \wedge a\ \Omega\ c \Rightarrow b\ \Omega\ c \tag{2.5}$$

Studies have confirmed the fact that mutation testing techniques are an effective way to measure the adequacy of test cases, in fact studies in [Pap+15], TCE could detect more than 7% and 21% of mutants respectively between redundant and equivalent. However, a major drawback of mutation testing is high cost involved in creating mutants and executing test cases against that mutant program.

2.3 Regression Testing

A software product must be tested during the development phases and every change made, in order to certify it is correct functioning, needs to be tested too. But after the first testing session, not all parts of a software need to be tested. In fact, the only part that could generate an error is the new one, so being able to know what part of a software is changed and needs to be tested becomes very important in order to reduce the time taken for the testing session. Regression testing is a technique used in order to verify that a code change in the software does not impact the existing functionality of the product. It can identify a **software regression** that is a type of software bug that consist in a feature that has worked before that stops working now. There are three important type of changes that could produce this type of bug:

- A software change
- An environment change
- A library linked to the software change

Regression testing is a technique that has the purpose of being sure of the effectiveness of a feature after a code change[Won+97]. During the development of the Smart Contract the programmer could introduce numerous changes inside its code, in fact, programmer with the purpose to change, update or improve its functionality could introduce a bug in its DApp that takes the name of: **Software Regression**. Software regression could cause some problems to the end-user and it could be fixed only by

regression tests. Developers have the capability to identify software regression via regression testing, so they re-run function tests to discover some revision control errors. But the peculiarity behind regression testing is the fact that coders have the possibility to test only the changed part of their code in order to don't test all the code and save time. There are several ways to perform regression testing:

- **Regression test selection**: This Regression testing technique perform testing in only a selected part of code, and it could be useful only if the cost of selecting the part of the test suite is less than the retest all.
- **Test case prioritization**: This technique consist of prioritize tests and run before tests which have higher priority. There are some parameters that can be use in order to prioritize test suites:
  - **General prioritization**
  - **Version-specific prioritization**
- **Hybrid**: An hybrid technique that includes parts of both the other technique: Regression test selection and Test case prioritization.

So regression testing is a very important resource, but every programmer should consider a series of advantages and disadvantages that it comprehends:

Regression Testing Advantages

- Provides that any code change does not involve negatively other functionality
- Ensures that already fixed bugs do not come back
- Serves as a risk mitigation strategy during testing
- It is a very simple technique to understand and apply

Regression Testing Disadvantages

- It could be very time-consuming without automatization
- Must be applied for every small code change
- Requires creating complex test cases
- Things stand at present if a programmer changes a large part of code, running regression testing is equal to run all tests in all the code.

2.4 SuMo - Solidity Mutator

SuMo[5] is a mutation testing tool for the development of Solidity smart contracts in order to verify test suites quality, it is written in JavaScript and available by CLI commands. As already said, deploying a Smart contract on the Ethereum blockchain implies some difficulties due to the peculiarity of the environment. Some main problems are: the new languages uses to write this contract and the lack of best practice, the immutability of Smart Contract and the irreparable consequences due to the irreversibility of transactions. In SuMo are implemented 44 operators, 25 of them are Solidity specific operators (Table 2.1) and 19 are generals (Table 2.2) [**13-29**; BMP21]

[5]SuMo - Solidity Mutator created by the PhD Morena Barboni

Operator	***Extended Name***	***Aspect***
AVR	Address Value Replacement	Address
CCD	Contract Constructor Deletion	Construction
DLR	Data Location keyword Replacement	Data Location
DOD	Delete Operator Deletion Delete	Keyword
EED	Event Emission Deletion Event	Emission
FVR	Function Visibility Replacement	Visibility
GVR	Global Variable Replacement	Block and Transaction Properties
MCR	Mathematical and Cryptographic function Replacement	Global functions
MOC	Modifiers Order Change	Function behaviour modifiers
MOD	Modifier Deletion	Function behaviour modifiers
MOI	Modifier Insertion	Function behaviour modifiers
MOR	Modifier Replacement	Function behaviour modifiers.
OMD	Overridden Modifier Deletion	Function behaviour modifiers
PKD	Payable Keyword Deletion	Function state modifiers
RSD	Return Statement Deletion	Function state modifiers
RVS	Return Values Swap	Function state modifiers
SCEC	Switch Call Expression Casting	Address
SFD	Self-destruct Function Deletion	Global functions
SFI	Self-destruct Function Insertion	Global functions
SFR	SafeMath Function Replacement	Libraries
TOR	Transaction Origin Replacement	Block and Transaction Properties
VUR	Variable Unit Replacement	Units
VVR	Variable Visibility Replacement	Visibility

Tabella 2.1: All Solidity specific operators applied in SuMo

Operator	**Extended Name**	**Aspect**
ACM	Argument Change of overloaded Method call	Overloading
AOR	Assignment Operator Replacement	Expression
BLR	Boolean Literal Replacement	Literal
BOR	Binary Operator Replacement	Expression
CBD	Catch Block Deletion	Control
CSC	Conditional Statement Change	Control
ECS	Explicit Conversion to Smaller type	Type
ER	Enum Replacement	Type
HLR	Hexadecimal Literal Replacement	Literal
ICM	Increments Mirror	Expression
ILR	Integer Literal Replacement	Literal
LSC	Loop Statement Change	Control
OLFD	Overloaded Function Deletion	Overloading
ORFD	Overridden Function Deletion	Overriding
SKD	Super Keyword Deletion	Overriding
SKI	Super Keyword Insertion	Overriding
SLR	String Literal Replacement	Literal
UORD	Unary Operator Replacement and Deletion	Expression

Tabella 2.2: All General operators applied in SuMo

SuMo also implement an extended version of some Specific operators (Table 2.1) called **Non-Optimized**. In fact, AOR, BOR, ER, GVR, SFR, RVS and VUR could include a more complex version of the operators that generate more mutants guaranteeing a more complete mutation testing process. However, using this version of the operator will lead to a more waste of resources and time-consuming process increasing the possibility of generating many redundant mutants, this version is useful in full-scale mutation analysis. The normal version of operators, the **Optimized** one, contains simplified rules and is suggested in incremental mutation testing process. Anyhow, the choice of using one of these version can be done by the user. When the mutations are generated, these are saved in a directory using a unique **hash** as a name for each mutant. This hash is made relying on the mutation itself, so there cannot be identical hash. Then, SuMo takes a mutation at a time and modifies the original contract by the appliance of the mutation and the test are performed on it. In Listing 2.4 the '—' raw contains the part of code that will be replaced by the raw starting with '+++'. When the test ends, the mutation is deleted from the contract file by the usage of a specific

function in order to recreate the original contract without the mutation.

```
119 | uint256 moduleCount = 0;
120 | address currentModule = modules[start];
121 | ---  while (currentModule != address(0x0) && currentModule !=
   SENTINEL_MODULES && moduleCount < pageSize) {
    | +++  while (currentModule != address(0x0) && currentModule !=
   SENTINEL_MODULES && moduleCount <= pageSize) {
122 | array[moduleCount] = currentModule;
123 | currentModule = modules[currentModule];
```

Codice 2.4: Example of a mutant file

SuMo provides different functions that allow to perform more than just mutation testing in Smart Contracts. In fact, with SuMo it is possible to generate a list of mutants and save them in a directory or simply mutate contracts in order to know how many mutations a DApp generate. Obviously, the main purpose of SuMo remains performing mutation testing, but it allows users to do that with a lot of specifiable parameters. It is possible, on SuMo, to specify more than just one operator taken from the list above, it is also possible to decide in what contracts perform the mutation testing in order to select only the useful contracts for the user. The entire process is customizable too, in fact, this program permits to enable or disable process parameters like optimization or the use of **Ganache**[6]. Finally, it also gives the possibility to use a custom test script during the mutation testing. SuMo is also a versatile program because it considers several variables that could change from project to project. SuMo uses package manager for the JavaScript programming language as Npm and Yarn, and two development frameworks for Ethereum, Truffle and HardHat.

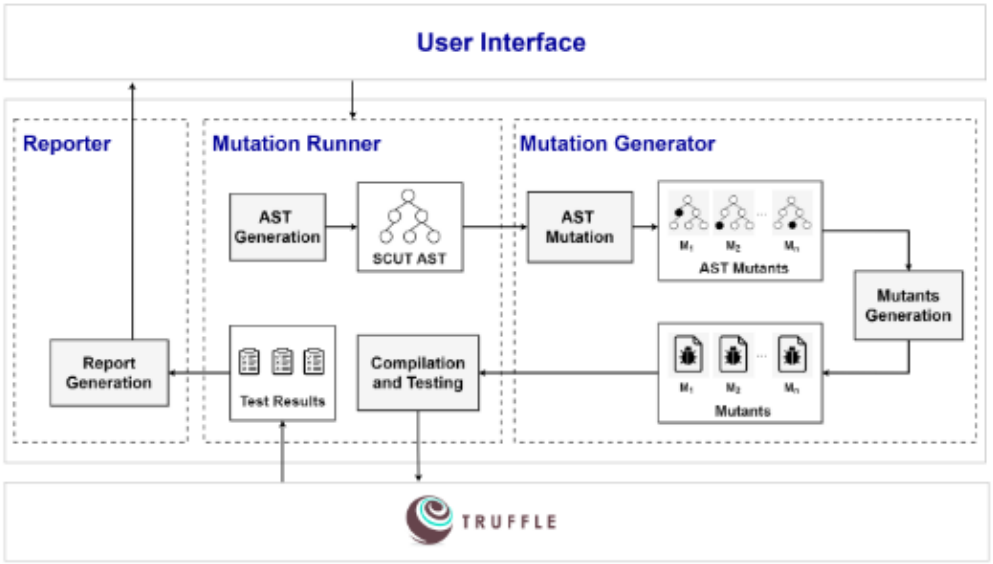


Figura 2.5: The mutation testing process in SuMo

There is the mutation testing. When all parameters are set, the user can start the mutation testing process via a CLI command. The first version of SuMo, that we

[6]Ganache is a personal Ethereum blockchain which you can use to run tests, execute commands, and inspect state while controlling how the chain operates.

studied in order to implements some features, used a Solidity parser for creating the AST or Abstract Syntax Tree of each contract that the user wanted to test. Then SuMo, in accord with the operator chosen by the user in the setting up phases, started to visit each AST and generate the mutations. Later, SuMo did not implement a TCE algorithm module, so the user was forced to pass the resulting set of contracts to an external Java module in order to delete redundant and equivalent mutant. After delete the not useful mutants and after re-upload the resulting set of contracts, SuMo started its testing process with the use of a development framework for Ethereum (for example Truffle) and a personal Ethereum blockchain like Ganache. Information about the mutation testing process was written in a report produced by SuMo. All this SuMo mutation testing process is depicted in Fig. 2.5.

2.4.1 Problems related to mutation testing and SuMo

The main target of our work was to increase the **efficiency** and the **usability** of SuMo. Efficiency issues are referred to the mutation testing process that might take a long time. In facts, projects of average dimension could take more than 6-7 hours to be completely tested by SuMo, depending on the machine used. The efficiency issue, in particular, includes the *"cost of mutation testing for evolving systems"*[Ojd+21], in fact, software development is subjected to a cyclical evolution that involve code modifications and could involve problems in previously working functionality. Mutate and test contracts that are not modified can be very time-consuming, so it can be considered a strong efficiency problem. To avoid this problem, we integrate **ReSuMe**[7] in SuMo, a regression testing framework written in JavaScript.

The "*Impact of Equivalent and Redundant mutants*" is another big issue that concerns the efficiency problems related to the mutation testing approach, in particular the fact that producing a lot of mutants increase the possibility to have some equivalent or redundant mutants. This problem if is ignored could lead to time costs and generating untrue reports, decreasing the effectiveness of the mutation testing. Using **TCE** (Trivial Compiler Equivalence) this problem can be avoided by eliminating redundant and equivalent mutants generated by the process before the start of testing session.

SuMo presents not only efficiency problems, in particular it is a CLI software that requires a lot of settings which leads to a usability problem. Being easy and intuitive to use is one of the most important software prerogatives, and being able to set the settings graphically can significantly reduce the pre-testing time and the learning curve of SuMo and decreasing the possibility to commit an error. For this reason, we have created a GUI version of SuMo named **SuMo-GUI**.

Another usability problem of SuMo is the fact that it doesn't produce a single complete report of the mutation testing session that could be very useful to the user. For this reason, we created an all-inclusive **reports** in **CSV** and in **XLSX** formats that contain information about all mutation testing session.

[7] https://github.com/FrancescoCasoni/ReSuMe

3. ReSuMo implementation

In this chapter, we discuss the description of our improvements implemented on SuMo in order to create a new version of the Solidity Mutator named ReSuMo. We work on SuMo with the aim to improve performance, accuracy, and usability. Our advancements are done by applying the TCE algorithm, allowing not to use the external Java module for removing redundant and equivalent mutants and improving usage time and accuracy. As already said in section 2.4, another modification done in SuMo is the implementation of ReSuMe in order to permit a policy of regression testing to SuMo. Finally, we also implemented a GUI and a reporting system to increase the usability. These upgrades are better described in the page below.

3.1 Trivial Compiler Equivalence

The implementation of TCE algorithm in SuMo helps us to solve the redundant and equivalent mutants' problem. This algorithm is usually implemented in the mutation testing process, and we adopt and integrate this algorithm for its capacity to increase the testing performance, as we said in paragraph 2.2.1. Our implementation of TCE was made in the main function of ReSuMo (Test()), and we have provided the possibility to activate the Algorithm in the config file. In addiction, ReSuMo performs some changes to the configuration file of the Solidity compiler named *"truffle-config"* or *"hardhat.config"*. In fact, ReSuMo add to this file some lines in order to permit the use of the Trivial Compiler Equivalence algorithm thanks to the module *"Instrumenter"*, in particular these lines allow deleting the "metadata" from the bytecode produces after the compiling phase.

In the workflow implemented, TCE is involved when all mutants are being generated and the testing process starts on them. In particular, the Compiler Equivalence algorithm works thanks to the solidity compiler that ejects a JSON file shown in Listing 3.3 in a specific folder, this file contains a univocal bytecode that is the result of the compilation of the contract (mutated or not). Bytecodes are compared one to each other and through this comparison it is possible to find out equalities with the purpose to discover redundant and equivalent mutants. The TCE mechanism can be disabled by the user, modifying the configuration file before starting the testing process.

As show in the sequence diagram in Fig. 3.1, when the user starts the testing process, Mutation Runner compiles each original file of the project under test and saves the bytecodes thus generated in a collection for a later usage (Listing 3.4). Then the mutants' generation phase starts, and to each contract the Mutation Generator applies the set of all operators, generating the mutants, the fields associated and save this information in a file. For each mutant created (Fig. 3.2), ReSuMo tries to compile them one at a time, if it fails, the mutant is marked as **stillborn** and is restored, and then Mutation Runner restarts the process for the next mutant. Instead, if the

mutant is compiled, the TCE (Listing 3.2) algorithm starts the comparison. Firstly, the comparison is made between the bytecode of the original contract and the mutant one, in order to find an **equivalent** mutant. Then, if it is not equivalent, the comparison is made between the mutant bytecode and each other's bytecode of others tested mutants bytecodes to detect **redundant** mutants. Finally, if no equivalences has been found the mutant is submitted to the test suite that would produce a live or killed mutant, furthermore the considered bytecode is saved in a collection to carry out the finding of the next redundant mutant. When the test phase ends, the mutated contract is restored to the original one using the *restore()* function, and the process is repeated for each other contracts of the project under test. Part of the implementation can be read in code (Listing 3.4).

```
1  function test() {
2    prepare(() =>
3        //  ... set up var to use during the test process
4        spawnCompile();
5        const originalBytecodeMap = new Map();
6        //  ... do regression testing
7            originalBytecodeMap.set(parse(file).name, saveBytecodeSync(parse(
       file).name + ".json"));}
8        instrumenter.instrumentConfig();
9        reporter.setupMutationsReport();
10       //Generate mutations
11       const mutations = generateAllMutations(files);
12       var startTime = Date.now();
13       //Compile and test each mutant
14       for (const file of originalBytecodeMap.keys()) {
15         runTest(mutations, originalBytecodeMap, file);
16       }
17       //  ... save mutation testing session information
18     })
19   );
20 }
```

Codice 3.1: Test() function used to mutate and test all contracts

```
1  function tce(mutation, map, file, originalBytecodeMap) {
2    mutation.bytecode = saveBytecodeSync(file + ".json");
3    if (originalBytecodeMap.get(file) === mutation.bytecode) {
4      mutation.status = "equivalent";
5    } else if (map.size !== 0) {
6      for (const key of map.keys()) {
7        if (map.get(key) === mutation.bytecode) {
8          mutation.status = "redundant";
9          break;
10       }
11     }
12     if (mutation.status !== "redundant") {
13       map.set(mutation.hash(), mutation.bytecode);
14     }
15   } else {
16     map.set(mutation.hash(), mutation.bytecode);
17   }
18 }
```

Codice 3.2: tce() JavaScript implementation

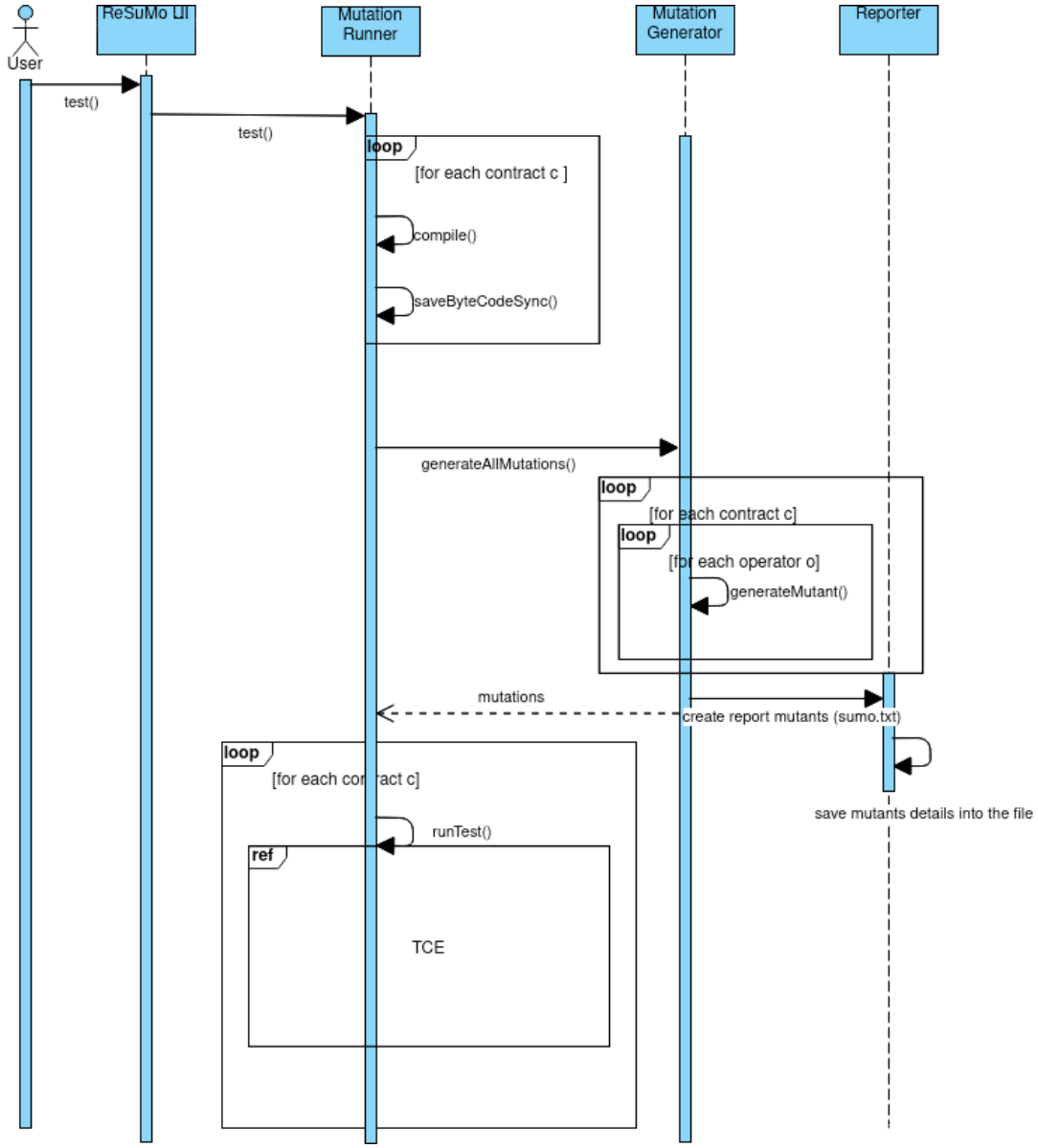


Figura 3.1: Introduction to TCE Sequence Diagram

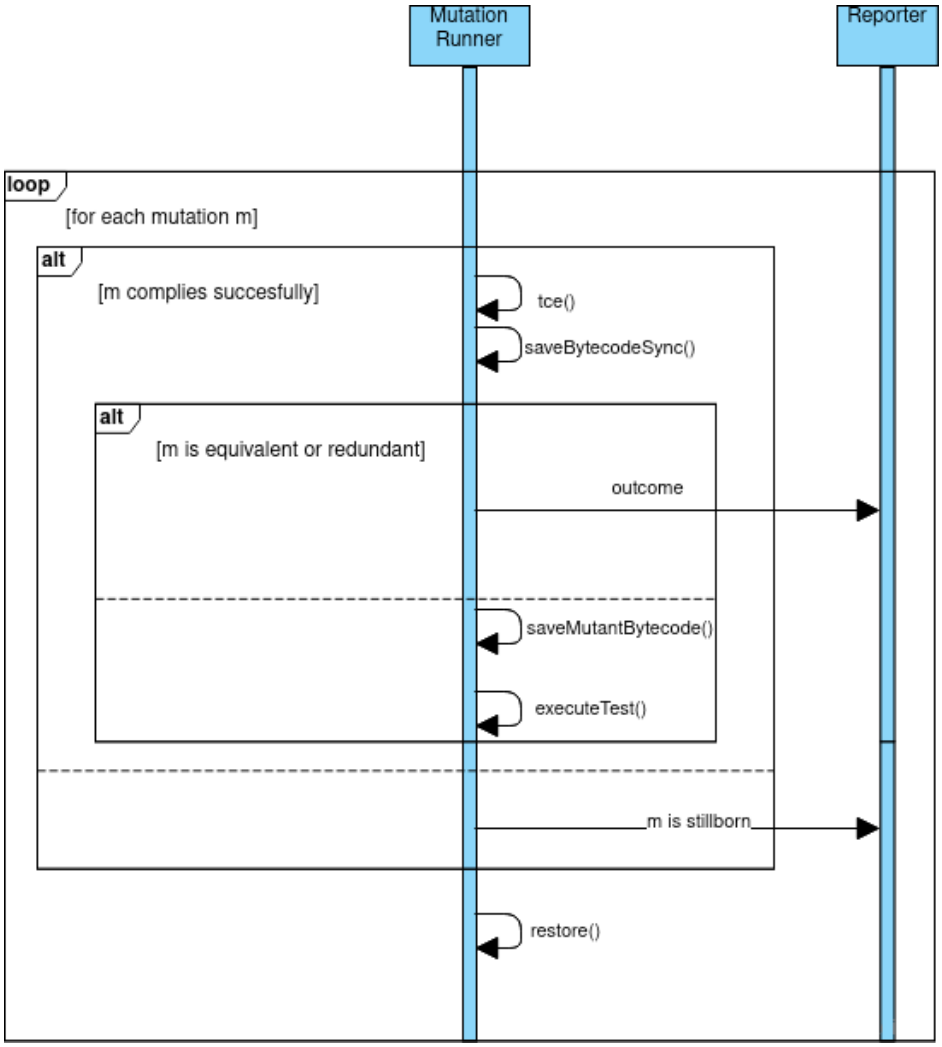


Figura 3.2: TCE Sequence Diagram

```
1  {
2    "contractName": "AscendingOrderedStack",
3    "abi": [],
4    "metadata": "{ [...] }",
5    "bytecode": "0x61073401a60731461001957fe5b306000526073815382
     81f3fe730000000000000000000000000000000000000030146080604
     052600436106100565760003560e01c8063189bb5e31461005b5780633c6
     42a7f146100875780633d7ce16d146100b157806364d [...] 4300060c
     000a",
6    "deployedBytecode": "0x730000 [...] 0060c000a",
7
8    "immutableReferences": {},
9
10   [...]
```

Codice 3.3: An extract of the JSON file product by the compiler

3.2 ReSuMe - Regression Testing tool for SuMo

3.2.1 A Regression Testing tool

ReSuMo, as already said, in order to decrease the testing-time of SuMo, integrates a Regression Testing framework named **ReSuMe**. This framework was already developed like an external module, but we have integrated it into SuMo inserting it into the main function of testing process. ReSuMe born with the aim to perform selective Regression Testing at file level, in fact, it is able to discover if a test or a contract is changed during the development. To carry out its work, ReSuMe saves the entire code of contracts and tests thanks to its hashing algorithm that permits to reduce the entire code in a single string name: **checksum**. Thanks to the checksum, it is able to establish which contract and which test must be used for the mutation testing. There are 5 policies that regulate what needs to be used, and they are:

- A new contract or test
- A changed contract or test
- A contract (changed or not) dependent on another changed contract.
- A test (changed or not) dependent on another changed test file.
- A test (changed or not) used for testing a changed contract, because every changed contract must be tested with its test.

ReSuMe also allows saving contracts and tests used during the mutation testing process, this feature is created with the aim of restoring files that, for any reasons, could corrupt themselves. This possibility works thanks to the fact that the Regression Testing framework saves all the files in directories that take the names of **baseline**.

3.2.2 ReSuMe module and its implementation in SuMo

The implementation of ReSuMe (Fig. 3.3) in SuMo has been made to automate the exclusion from testing contracts and tests that are not changed or are not involved in the change, speeding up the whole process. To be changed is not the only condition for which code should be subjected to ReSuMo testing process, in fact if a module is imported into another, or its functions are used in a part of code which directly has changed, even if the module or the function did not get any changes it has to be tested too. When the testing process starts, ReSuMo runs ReSuMe functions and takes the information about which contracts and which tests need to be executed during the mutation process, saving in a baseline all the contracts and tests. When tests are run for the first time in a project, ReSuMe saves their contract and their checksum, mutates and tests all contracts. Instead, when the mutation process is executed in an already tested project, ReSuMe checks which contracts are changed and which are not by means of the checksum field. Contracts that do not need to be tested are ignored, and tests that do not need to be used are deleted from their directory for a Truffle limitation that makes running only a part of the tests impossible, as we can see in the first loop of the Fig. 3.3. The process continues by applying mutations, and testing all remaining contracts. When the mutation process is completed, ReSuMo restores the deleted tests from their baseline. ReSuMe produces a report (see Fig. 3.4) that lets the user know how many contracts and tests are present in the project, it also

presents a list of files which refer to the contracts and tests that are changed since the last test run. Finally, it shows the list of files that will be used in the next mutation testing activity. We also updated the integration of results in ReSuMo main function. Before our work ReSuMe printed results of regression testing in a CLI matrix but we updated the idea behind saving information by creating the functionality that allows the program itself to process data and save them in a CSV files as shown in Fig. 3.3. ReSuMo Project

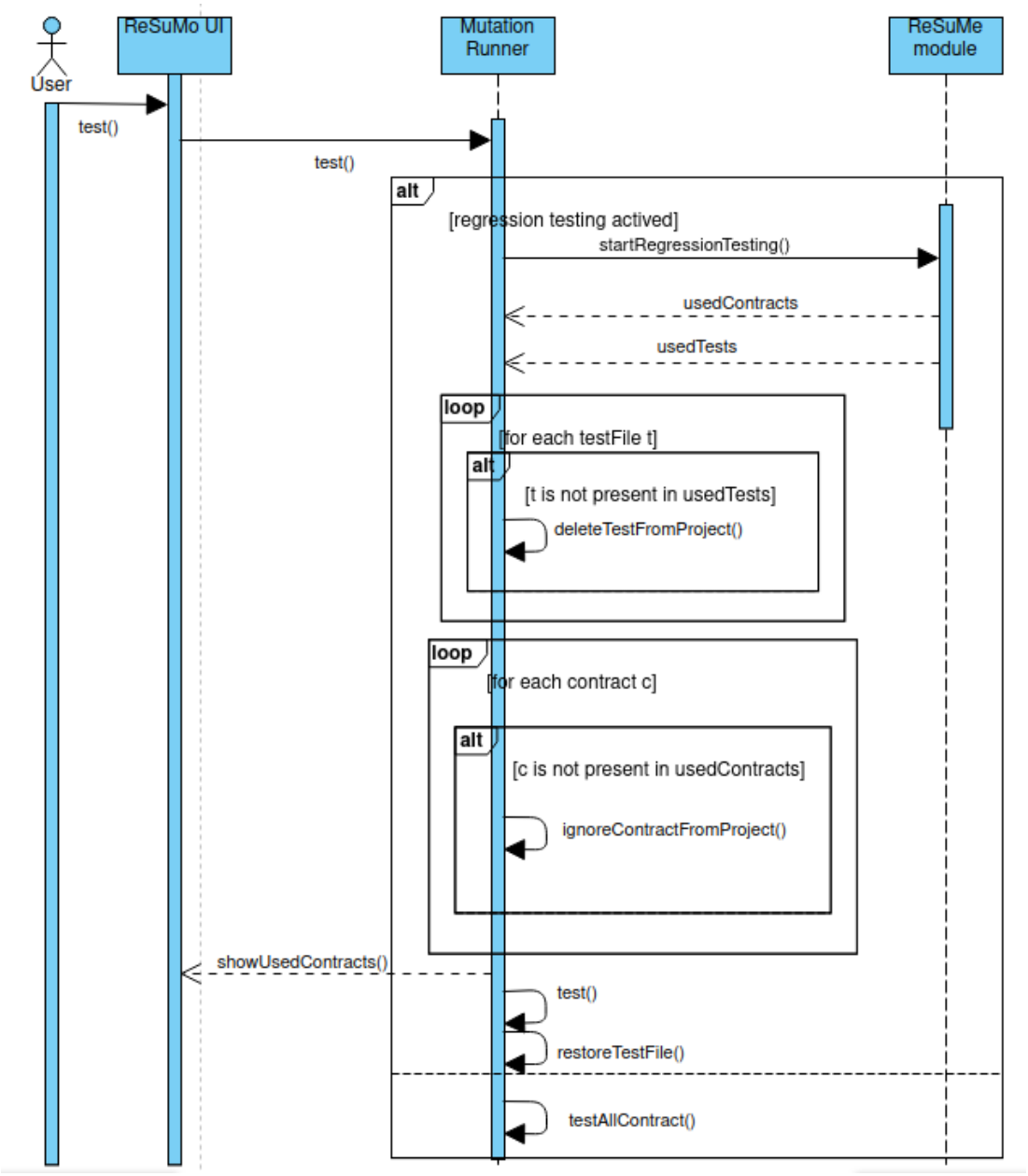


Figura 3.3: Sequence diagram representing ReSuMe interactions with ReSuMo

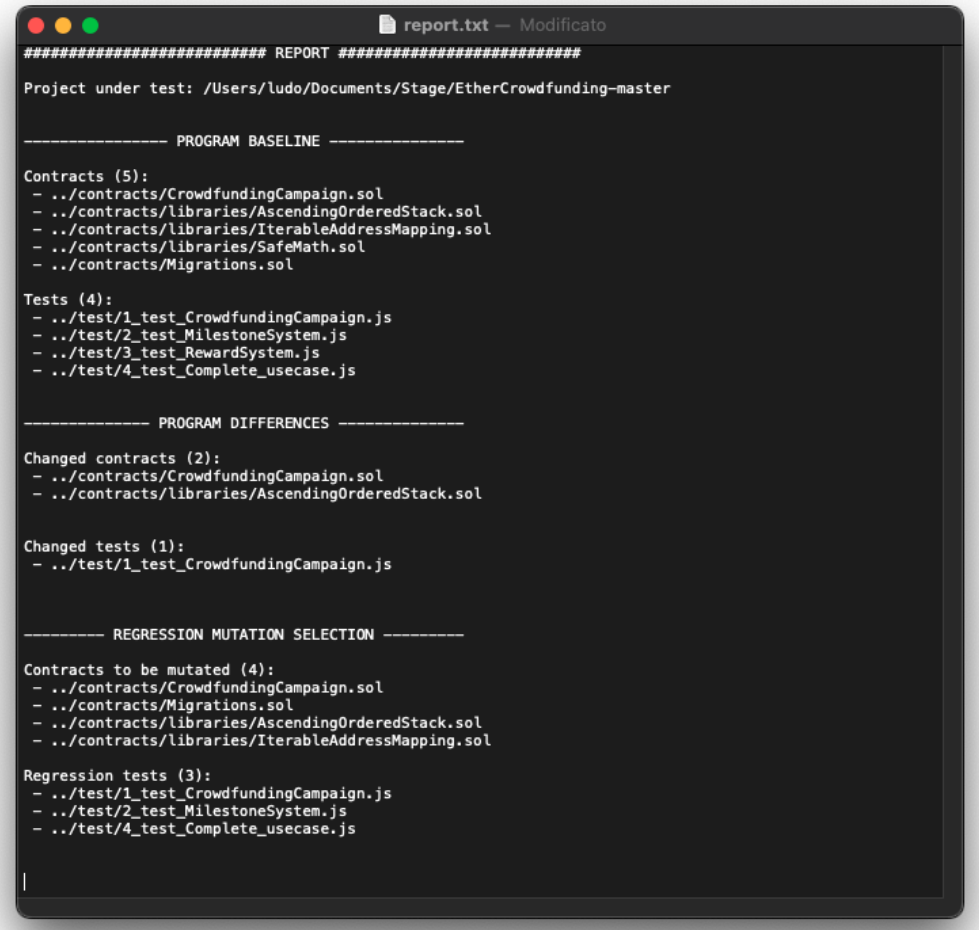


Figura 3.4: Resume report

3.3 Integration of results

3.3.1 Motivations behind the new report

SuMo produces a lot of reports to pick up information that derive from the mutation testing session, therefore at some point it became necessary to create a report containing the most relevant information about testing process and also the only ones useful to users. Thanks to these considerations, we created a CSV report, resulting from the selection of various reports produced by SuMo. This report contains information about mutants generated by SuMo and testing results, sorted by column. CSV report, furthermore, merges of new information and old information taken from a previous report about the last execution of mutation testing.

The entire work of integration of results has been made with the purpose of having a mutation score that reflects the quality of the all test suite on all contracts without the obligation to re-run it.

3.3.2 Report implementation

CSV report (in Fig. 3.5) was made to easily find a specific mutant by his hash and finding which test suites has been performed and their result, it also contains data about mutants, as the operator that generates it and which contract it refers to. There are three types of test result, if the test suite pass we write "**L**" means live, "**K**" means killed and "**?**" is used if a test has not been performed on a certain mutant. CSV report came from merging two other reports, one generated by Mochawesome a custom testing reporter used with JS framework mocha, that generates a JSON file for each tested mutant and writes on it any kind of information about the testing process. The other one, is created by SuMo when the testing process finishes, it contains each mutant and writing information about which test passed and which did not.

3.3.3 Merging of the result algorithm

The merge algorithm in Fig. 3.6 provides us to produce the clearest and the richest report, it is the result of many checks and it allows having information deriving from all the mutation testing executions. The algorithm that permit to marge information derived from old and new run can be summed up in 3 cases:

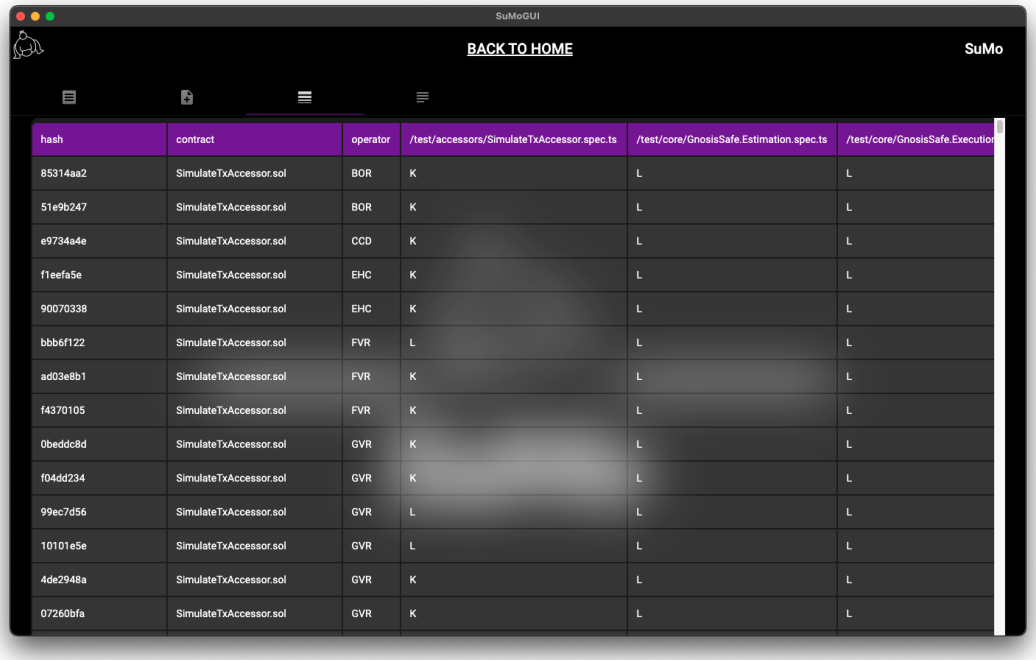
1. For all the hashes contained in the new run but not in the old one, ReSuMo add this hash in order to report if they are *"Live"* or *"Killed"* . The reason behind the fact that it could be new hashes is that, for a modification of the code, some mutations points raised up.

2. For all the hashes contained in the new and in the old run, ReSuMo update the information of these hashes with those arising from the new report.

3. For all the hashes present in the old run but not in the new one, ReSuMo performs a series of check:

    - If the hash is linked to a contract that is not presented in the contracts list of the new mutation testing run, ReSuMo adds this hash information because it is a mutation created by a contract that are not being mutated in the newest run for regression reasons.

- If the hash is linked to an operator that is not presented in the operators list of the new mutation testing run, ReSuMo adds this mutant because it is the result of an operator that the user does not choose for the new run.
- If the hash is linked to a contract and to an operator involved even in the new run, ReSuMo discards this mutant because it is probably created from a mutation that has been fixed before the new run.



hash	contract	operator	/test/accessors/SimulateTxAccessor.spec.ts	/test/core/GnosisSafe.Estimation.spec.ts	/test/core/GnosisSafe.Execution
85314aa2	SimulateTxAccessor.sol	BOR	K	L	L
51e9b247	SimulateTxAccessor.sol	BOR	K	L	L
e9734a4e	SimulateTxAccessor.sol	CCD	K	L	L
f1eefa5e	SimulateTxAccessor.sol	EHC	K	L	L
90070338	SimulateTxAccessor.sol	EHC	K	L	L
bbb6f122	SimulateTxAccessor.sol	FVR	L	L	L
ad03e8b1	SimulateTxAccessor.sol	FVR	K	L	L
f4370105	SimulateTxAccessor.sol	FVR	K	L	L
0beddc8d	SimulateTxAccessor.sol	GVR	K	L	L
f04dd234	SimulateTxAccessor.sol	GVR	K	L	L
99ec7d56	SimulateTxAccessor.sol	GVR	L	L	L
10101e5e	SimulateTxAccessor.sol	GVR	L	L	L
4de2948a	SimulateTxAccessor.sol	GVR	K	L	L
07260bfa	SimulateTxAccessor.sol	GVR	K	L	L

Figura 3.5: Example of a CSV report from the GUI

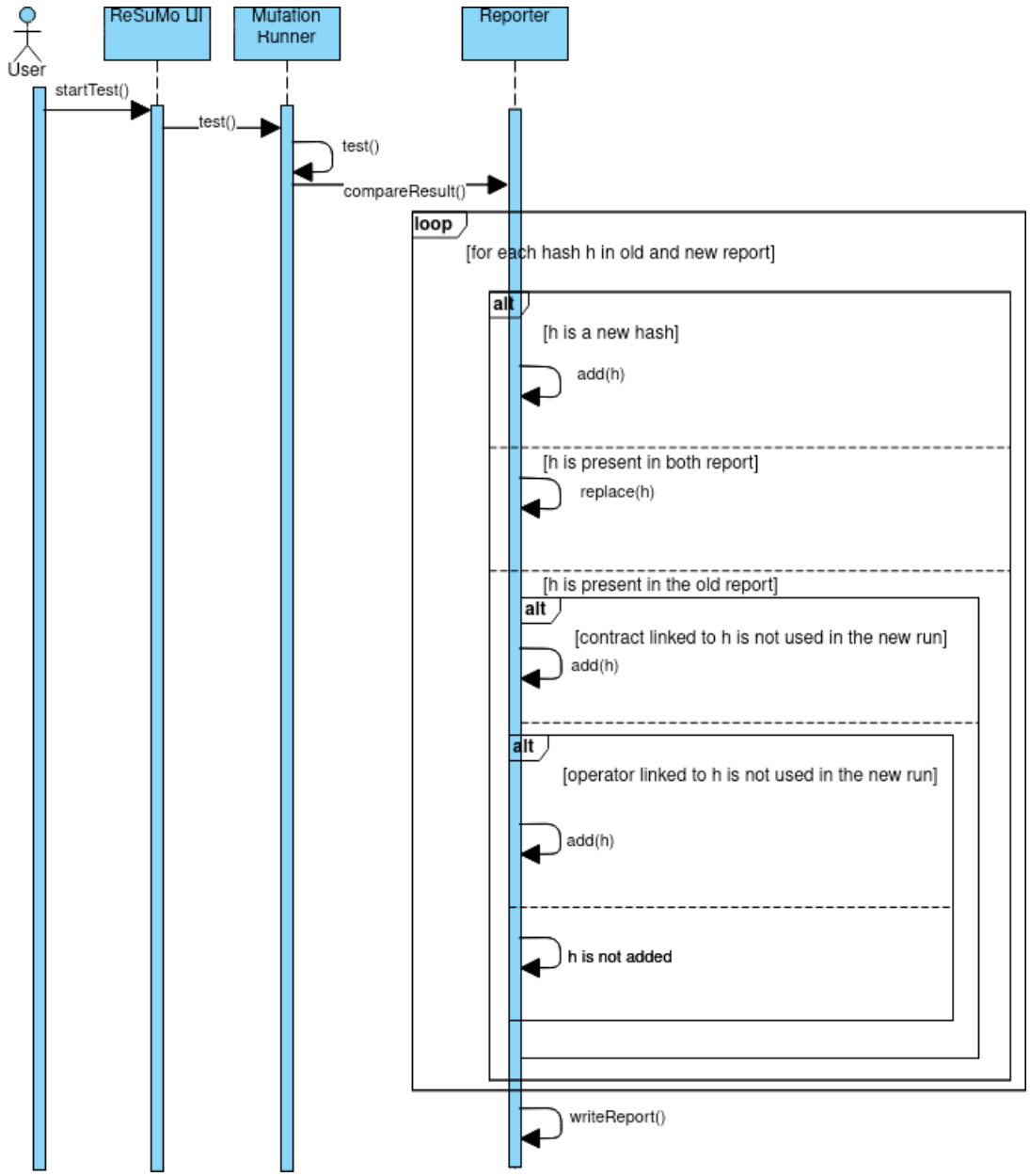


Figura 3.6: Sequence Diagram of the merging algorithm implemented in ReSuMo

3.4 Mutation Testing Report

3.4.1 Motivations behind the new report

SuMo stores information about the behaviour of operators with a specific module called *"reporter"*. Users might want to analyse operators based on how they act on their project. From this need arises XLSX report, which allows dividing the results by operators that produce mutants that in turn produce those results. In this report we can find one line for each operator and several columns that are referred to these values:

- Total number of mutations that the operator produce
- Number of redundant mutations that the operator produce
- Number of equivalent mutations that the operator produce
- Number of valid mutations that the operator produce
- Number of killed mutations that the operator produce
- Number of live mutations that the operator produce
- Number of timed-out mutations that the operator produce
- Number of still-born mutations that the operator produce
- Mutation score of the mutants produce by the same operator
- Time to run all test suites in the mutants produce by the same operator

This report shown in Fig. 3.7 becomes useful to the user when he wants to understand when and how an operator impacts his project.

3.4.2 Report implementation

Our implementation, of XLSX report, consists in taking information about each mutant during the mutation testing. We store data concerned to the hash of the mutant, its status that could be: redundant, equivalent, timed-out, killed, live or still-born and the time to run tests on this mutant that the program takes. Other fields mentioned above and not derivable from mutation testing are created by a mathematical formula, in particular:

$$valid = killed + live \tag{3.1}$$

We implemented the mechanism for taking information in the main SuMo module shown in Listing 3.4 that run mutation testing too (*"command"*).

During the mutation testing session, when mutation phase is already done, we start the time counter and run *"runTest"* method shown in Listing 3.5. This method allow us to store data about the state of the mutant. When SuMo has stored all the information that it needs, we stop the time counter and we process it in order to be able to know how long did the testing process take. Next we start reporter methods in order to save all the data that we need to create: **sumo**, **CSV** and **XLSX** reports, and then we start reporter methods, created with the purpose of storing information for XLSX report.

In the *"runTest"* method we store all data about mutants, in particular we take information about the compile process and we store a status that depends on how response to the compilation, the TCE algorithm and the testing process. We also register another time counter related to the time that a single mutant take to run tests.

Information are store and calculate by *reporter* module that save data of mutants during the test process, in particular it is able to memorize: the contract from which the mutant is generated, the operator that generate the mutant and the state of the mutant. *"Reporter"* module uses this information in order to create a XLSX report.

The module *Reporter* receives a list of mutants coming from the mutation session, it creates columns of the new report that could be:

- Redundant
- Valid
- Killed
- Live
- Timedout
- Stillborn
- Mutation Score
- Occurred time

Then it begins to fill the rows of the columns with the information taken, and reworked into maps. In particular, all operators active during the mutation testing session are placed first, then the other columns are filled with the length of status of mutants created by the operator and finally the *Reporter* write the time that all mutants generated by a certain operator takes to be tested.

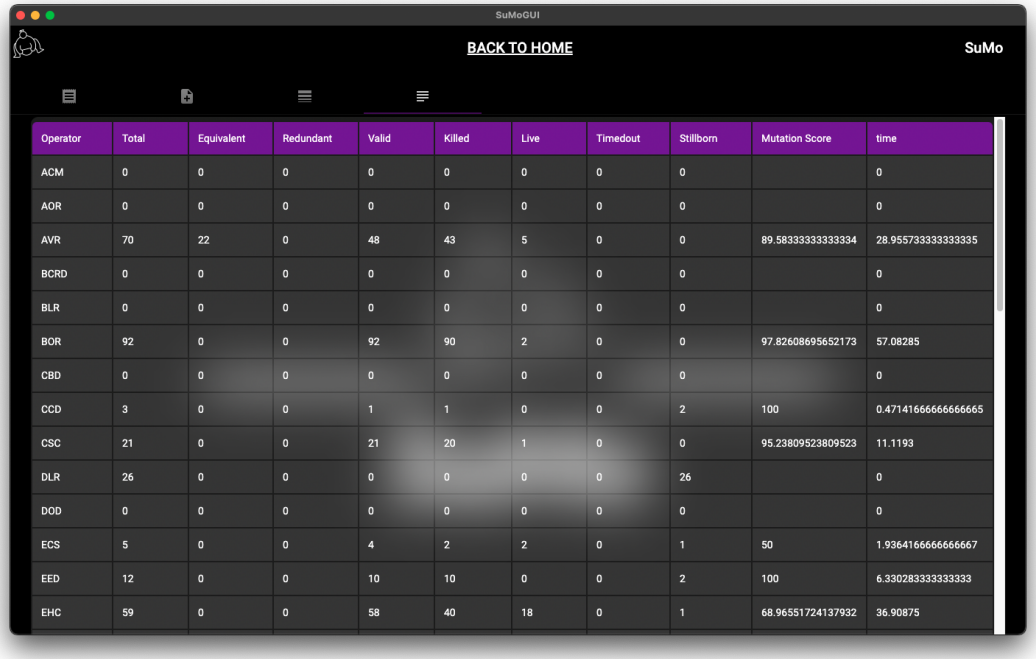
```
1  function test() {
2    prepare(() =>
3
4      glob(contractsDir + contractsGlob, (err, files) => {
5
6        ...
7
8          instrumenter.instrumentConfig();
9          reporter.setupMutationsReport();
10         //Generate mutations
11         const mutations = generateAllMutations(files);
12         var startTime = Date.now(); // Start time counter for testing
       process
13
14
15         //Compile and test each mutant
16         for (const file of originalBytecodeMap.keys()) {
17           runTest(mutations, originalBytecodeMap, file);
18         }
19
20         var testTime = ((Date.now() - startTime) / 60000).toFixed(2);  //
       End time counter for testing process
21
```

```
        instrumenter.restoreConfig();

        // reporter functicodingon
        reporter.saveMochawesomeReportInfo();
        reporter.testSummary();
        reporter.printTestReport(testTime);
        reporter.restore();

        //exReporter function that permits to create xlsx report
        exReporter.saveData();
        exReporter.restore();
        ...
      })
    )}
```

Codice 3.4: test() functions of ReSuMo

```
function runTest(mutations, originalBytecodeMap, file) {
  const bytecodeMutantsMap = new Map();
  for (const mutation of mutations) {
    if ((mutation.file.substring(mutation.file.lastIndexOf("/") + 1)) === (
    file + ".sol")) {
     var startTime=Date.now()
      ganacheChild = spawnGanache();
      mutation.apply();
      reporter.beginCompile(mutation);
      const isCompiled = spawnCompile();

      ....

        if (mutation.status !== "redundant" && mutation.status !== "
    equivalent") {
          reporter.beginTest(mutation);

         ....

        }
        if (mutation.status !== "redundant" && mutation.status !== "
    equivalent" && mutation.status !== "timedout") {
          reporter.extractMochawesomeReportInfo(mutation);
        }
      } else {
        mutation.status = "stillborn";
      }
      reporter.mutantStatus(mutation);
      exReporter.mutantStatus(mutation);

        ....

      mutation.time=Date.now()-startTime;
    }
  }
....
}
```

Codice 3.5: runtest() function of ReSuMo



Operator	Total	Equivalent	Redundant	Valid	Killed	Live	Timedout	Stillborn	Mutation Score	time
ACM	0	0	0	0	0	0	0	0		0
AOR	0	0	0	0	0	0	0	0		0
AVR	70	22	0	48	43	5	0	0	89.58333333333334	28.955733333333335
BCRD	0	0	0	0	0	0	0	0		0
BLR	0	0	0	0	0	0	0	0		0
BOR	92	0	0	92	90	2	0	0	97.82608695652173	57.08285
CBD	0	0	0	0	0	0	0	0		0
CCD	3	0	0	1	1	0	0	2	100	0.47141666666666665
CSC	21	0	0	21	20	1	0	0	95.23809523809523	11.1193
DLR	26	0	0	0	0	0	0	26		0
DOD	0	0	0	0	0	0	0	0		0
ECS	5	0	0	4	2	2	0	1	50	1.9364166666666667
EED	12	0	0	10	10	0	0	2	100	6.330283333333333
EHC	59	0	0	58	40	18	0	1	68.96551724137932	36.90875

Figura 3.7: Example of XLSX report.

3.5 Test Suite Report

3.5.1 Introducing to the testSuiteReport

Another report can be generated from the user with the aim to explore the results that each test suite and each test case produce at the end of a run. This report is named *"testData.xlsx"* shown in Fig. 3.8 and it could be created by command *"npm run sumo generateExel"*. It is built from the information taken regarding the tests, and it is created by a specific function on the module *"reporter"*.

When the testing process on a single mutant ends, a JSON file containing all the information about the test is written. They are sorted by every single test file, test suite and test case for every mutant and they are useful to the user in order to perform data analysis.

The fields that the report use are:

- Testing file name
- Test name
- Test suite name
- Test case name

Reporting this information is useful to understand precisely to which test the information refers to. The results that are written under the columns of the various test cases referred to mutants. Report uses three more fields in order to identify these mutants:

- Hash
- Operator
- Contract

As in the Fig. 3.8 all the mutants can be recognized by its unique hash, it is associated with the operator that generated the mutant and the contract where it comes from. In addition to that, we can also find coloured cells with inside the results of the test case referred to a certain mutant. Cells could be:

- K if the result is killed. That means the test case is failed.
- L if the result is live. That means the test case is passed.
- - if the result is null. That means the test case is null.

	A	B	C	D	E	F	G	
1				/test/accessors/SimulateTxAccessor.spec.ts				/test/core/(
2				SimulateTxAccessor				GnosisSafe
3				estimate				requiredTx(
4	Hash	Operator	Contract	should enforce delegatecall	simulate call	simulate delegatecall	simulate revert	should reve
5	0017b5d0	RSD	CompatibilityFallbackHandler.sol	L	L	L	L	L
6	0085b1ea	BOR	DefaultCallbackHandler.sol	L	L	L	L	L
7	00a38686	FVR	HandlerContext.sol	L	L	L	L	L
8	00c040f8	FVR	GnosisSafeProxyFactory.sol	K	K	K	K	K
9	022107c7	FVR	GnosisSafeProxyFactory.sol	L	L	L	L	L
10	037f6079	ILR	OwnerManager.sol	L	L	L	L	L
11	0436bfb0	FVR	GnosisSafeProxyFactory.sol	L	L	L	L	L
12	04963722	VVR	ModuleManager.sol	L	L	L	L	L
13	04b4de04	CSC	OwnerManager.sol	L	L	L	L	L
14	05910bac	BOR	OwnerManager.sol	L	L	L	L	L
15	06492178	MOI	ModuleManager.sol	L	L	L	L	L
16	0666769f	ILR	OwnerManager.sol	L	K	L	L	L
17	07260bfa	GVR	SimulateTxAccessor.sol	L	K	K	K	L
18	07c79c40	ILR	OwnerManager.sol	L	L	L	L	L
19	07d55065	EHC	OwnerManager.sol	L	L	L	L	L
20	08705d37	SLR	OwnerManager.sol	L	L	L	L	L
21	0993a7a8	AVR	OwnerManager.sol	L	L	L	L	L
22	09f6eece	GVR	ModuleManager.sol	L	L	L	L	L
23	0a5271b8	VVR	OwnerManager.sol	L	L	L	L	L
24	0a5e18d2	EHC	OwnerManager.sol	L	L	L	L	L
25	0a978247	CSC	Executor.sol	L	L	K	L	L
26	0ab78802	BOR	OwnerManager.sol	K	K	K	K	K
27	0ad67ee6	SLR	CompatibilityFallbackHandler.sol	L	K	K	K	L
28	0ba4c63b	MOD	FallbackManager.sol	L	L	L	L	L
29	0ba5ed5c	AVR	OwnerManager.sol	L	L	L	L	L
30	0bc02b5b	ILR	ModuleManager.sol	L	L	L	L	L
31	0beddc8d	GVR	SimulateTxAccessor.sol	L	K	K	K	L
32	0c1478d4	EHC	ModuleManager.sol	L	L	L	L	L
33	0d1127f1	EHC	ModuleManager.sol	L	L	L	L	L
34	0d5b74dd	ILR	OwnerManager.sol	K	K	K	K	K
35	0d722fee	AVR	OwnerManager.sol	L	L	L	L	L

Figura 3.8: This is an example of the *"testData.xlsx"*.

3.6 ReSuMo GUI

3.6.1 Technologies in ReSuMo-GUI

An important implementation done in SuMo is the building of a Graphic User interface, in order to increase the Framework usability. In fact, SuMo requires several configurations and the user must set them in a configuration file written in JavaScript, this approach consent to have a grater possibility to customize the program, but also involve a lot of mistakes. In addiction, programmers that want to use SuMo must know the list of commands that the Solidity Mutator offers and to be faster in testing must also learn their syntax. For all these reasons and to complete SuMo giving the possibility to the user to read in the program the reports that the program produces, we implemented a single page application.

The GUI has been made using Angular, a framework to build WebApp using HTML, CSS and TypeScript. Angular framework could allow us to use ReSuMo as a light WebApp with an offline back-end and front-end. It communicates with ReSuMo by a service that is linked to an HTTP controller through GET and POST requests, which acts as a web-API. The implementation of the controller was required because Angular has no access to the file system for security purpose whereas it is used for web coding, so without it, we cannot call functions that modify the configurations files and launch the SuMo commands. The controller opens a server in localhost 8000, and when it receives a URL request from the service, it launches specific ReSuMo commands or saves the user settings. We also use the cross-platform framework "Electron", we created the ReSuMo app and in this way it can be used as a browser web or as an App. The use of Angular also permit us to implement a material design already drawn. ReSuMo thanks to this fact has taken a modern look absolutely comparable to the new modern applications.

In future, the new version of the Solidity Mutator will probably be distributed by Docker, a container of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another, and it probably is divided in a remote and centralized back-end and a distributed front-end.

Description of the ReSuMo-GUI

In the Fig. 3.9 is possible to see the first ReSuMo page, it focuses all ReSuMo commands in some buttons:

- **Test**: thanks to the test button, ReSuMo starts the mutation testing with the already configured setting. It sends a GET request via its specific service to the HTTP controller in order to communicate with the command module and active the test() function. Actually, ReSuMo permits to know when the process is finished via the terminal emulator, but in the future will show the whole process as the CLI version already does.
- **Preflight**: this button permits to generate all the mutants, without saving them but just showing them on the console and crating a summary, shown in Fig.3.12. Even in this case, when the button is press, ReSuMo send a GET request to the controller in order to activate the Preflight function of the command module.
- **mutate**: when the button *"mutate"* is pressed, ReSuMo communicates with the web-API to the command module in order to start the process and mutate all

contracts of the SUT project. The mutations are printed in the console to give, to the user, the opportunity to know how many and which mutations are present in its software. Moreover, the mutations are saved in a specific folder for being used in a second moment by other functions, creating files containing the mutations, like the one shown in listing 2.4.

- **Clean**: thanks to the clean button, ReSuMo deletes the *".sumo"* directory with the aim to clean all data stored in last mutation testing section. Once again, all the process is done thanks to the Web-API created by us in JavaScript.
- **Restore**: This button give to the user the possibility to restore contracts and test. It could be essential if, for any reasons, something goes wrong, the testing process stop and contracts remain with a mutation in their code.
- **Delete**: similar to the *clean*, delete button permits to clean the regression testing directory, and can be use when:
  - A new project is under testing.
  - Programmer wants to delete the past regression testing information.

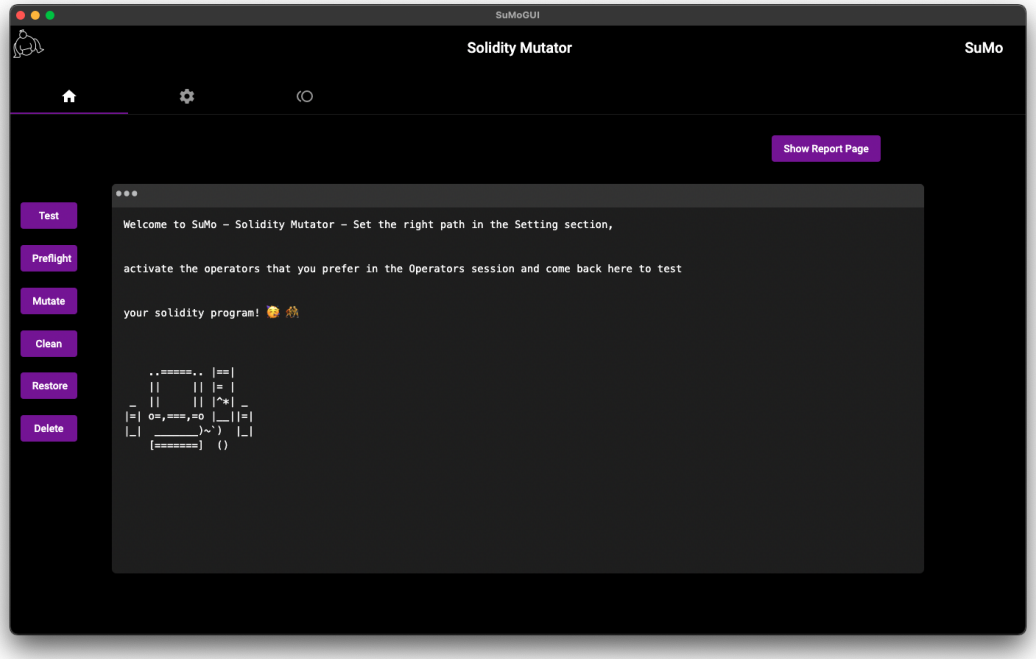


Figura 3.9: The first page of the GUI

Another important component included in the first GUI page is the setting component that has made possible to fix problems related to the configuration and previously described. This component can be seen in Fig. 3.10.
This component contains settings that can be found in the configuration file named *"config.js"*. Without our improvement, like said before, programmers had to change this file and risk making mistakes. Thanks to the GUI, this modus-operandi changes and configuring ReSuMo become much easier. In fact, ReSuMo need to be setted by this component, and send via Web-API all new parameters to the back-end that saves

them in the right way, when the saving process start ReSuMo also controls the TCE checkbox, in fact, if it is active, for security reasons, the Framework alerts the user to the facts. The parameters that the user need to set are:

- **SuMo Path**: This parameter is the local path where sumo is saved. In future, it might be omitted if ReSuMo will be hosted in a remote server.
- **Project Path**: This path permits to know where is saved the target project to test, in order to read its configuration files.
- **Contracts Path**: This path is referred to the main directory, where contracts of the target project are stored.
- **Test Path**: User must write there the path of the main test directory of the SUT project.
- **Path to save the compiled file**: In this field can be written the path where the programmer wants to save all the compiled contracts, this directory is going to contains the mutated and the not-mutated contracts.
- **Set TimeOut in second**: Thanks to ReSuMo is now possible to the user to set the time-out of the testing process, this field is an important introduction imported by ReSuMo, in fact, it consents to set a time after which a test of a mutant cannot continue in order to avoid the loop-test.
- **Select contracts to ignore**: Is a button that ReSuMo uses to show to the user the contracts present in the "Contracts Path" in order to give the possibility to choose what contracts to ignore and on which not applying the mutation testing. This new ReSuMo feature is very important because the contracts to ignore path is not handwritten any more by the programmer in the configuration file. Some contracts need to be ignored because they could appertain to the Solidity language libraries.
- **Ganache**: This checkbox gives to the user the possibility to start ganache during the mutation testing process.
- **TCE**: With this checkbox, it is possible to enable and disable the TCE tool during the testing process.
- **Personal test script**: Some programmers might need to launch a personal script for testing its contracts, with this checkbox active, ReSuMo runs the script specified in the package.json of the under test project.
- **Optimizer**: This checkbox permit to enable or disable the operators' optimization. To know how operators work and which ones are not optimized, you have to refer to the SuMo section in the background.
- **Regression Testing**: This checkbox can be use in order to enable and disable the regression testing during a testing process. But with this checkbox disable, information related to the old regression testing session and stored in the *".resume"* will be not deleted, for this operation user can use the delete button of the first component of the first page of ReSuMo-GUI.

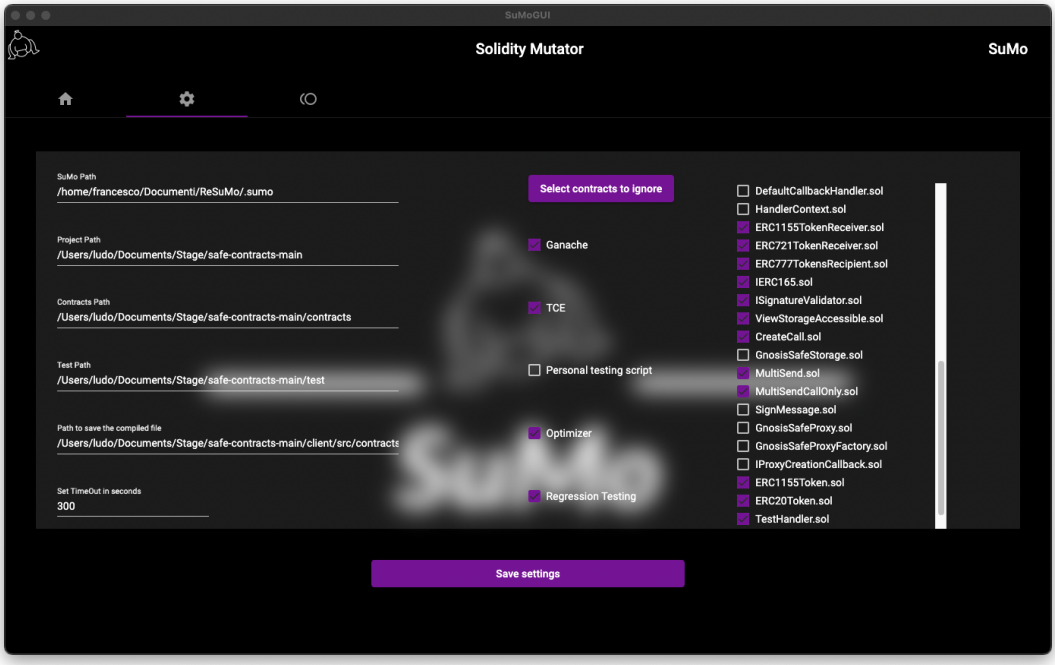


Figura 3.10: The second component of the GUI

The last component of the first GUI page shown in Fig. 3.11 is referred to the operators. This component permits the user to choose operators that ReSuMo is going to use during the mutation testing process. Operators are divided in two macro-categories:

- **General Operators**: Operators that are used generally in the mutation testing. They can be found in the table 2.2
- **Solidity Operator**: They are specific operators for Solidity smart contract. They are specified in table 2.1.

All these operators can be enabled and disabled by their specific Slide-toggle. User can also switch the entire group of operators by specific buttons imposed on them. All this operation has to be done before the mutation testing process starts, when programmer decides what operators active, it can save its preferences with the button *"Save operators"*, placed below the page. When the user presses the button, start the saving process and ReSuMo sends new preferences to the back-end by the HTTP controller created that accept POST and GET calls. Then, when the process is finished, ReSuMo confirms the saving to the user by the terminal.

The second page of the ReSuMo-GUI is an entire page dedicated to reports. Reports are another important implementation of ReSuMo that has the aim to improve the usability of SuMo. This page can be accessible by the button placed in the up-right corner of the first component of the first ReSuMo-GUI page. When the button is pressed, a new page is loaded and ReSuMo send an HTTP request to the controller that response with the content of reports, ReSuMo format the text and make it viewable, each one in its specific component. The new version of the Solidity Mutator contains 4 different sections to display four different reports that it produces during the mutation testing.

The reports that ReSuMo allows to see are:

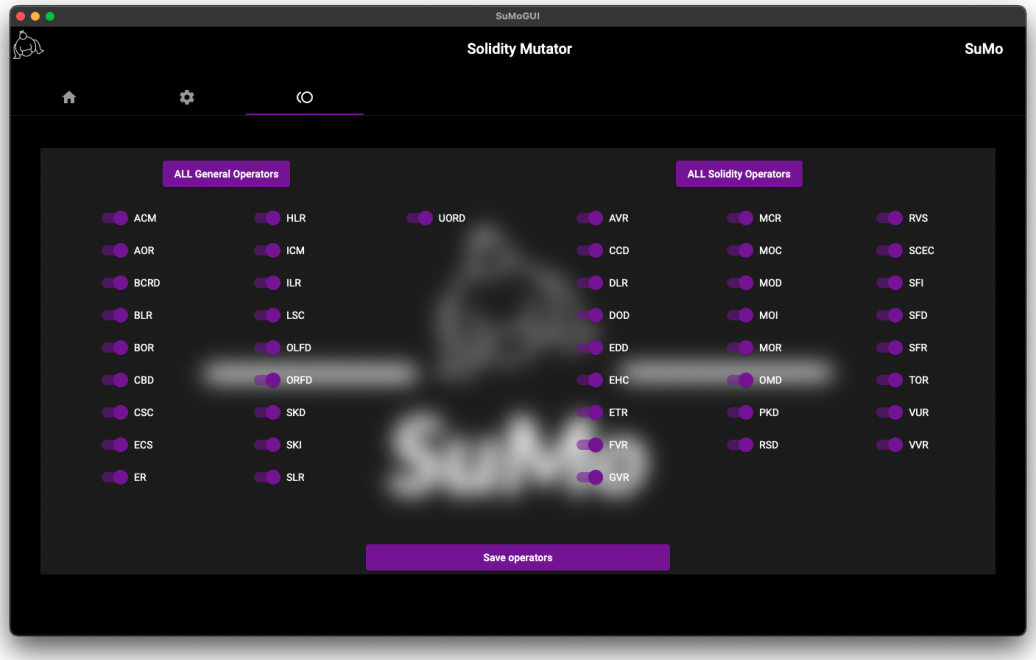


Figura 3.11: The third component of the first page of GUI

- The **mutation testing report** (Fig. 3.12) produce by the *"reporter"* module of SuMo. It contains a summary of the contracts, operators and the mutants generated mutants. When the testing process ends, information about the results are written.
- Another report, is the **Regression Testing report** (Fig. 3.13), it is generated only when Regression testing is active, and it shows information about: total and changed contracts, total and changed tests and also what contracts and tests are involved in the mutation testing.
- The third report shows by ReSuMo is **CVS report** (Fig. 3.5), it contains information related to the mutation testing, in particular, it shows the hashes of mutants involved in the process, the operators and the contracts that permit to create these mutants and the results of testing process in these hashes.
- The last report proposed by ReSuMo is an **XLSX report** (Fig. 3.7), it is in picture and it is a report useful in order to know how many mutants each operator produces and their status, in addiction, it shows the testing-times that each operator has required.

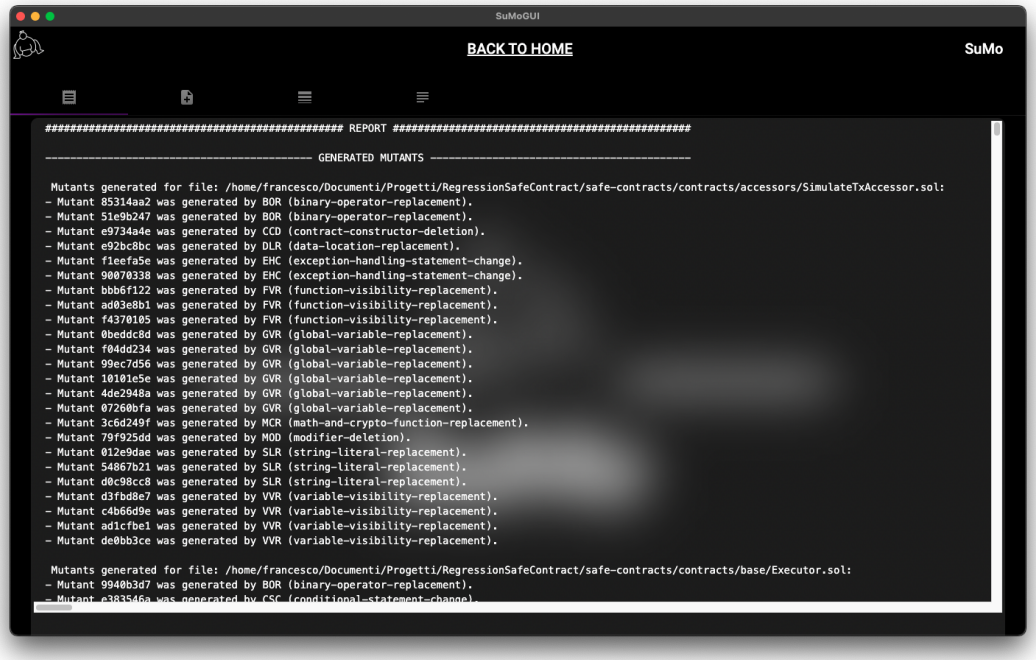


Figura 3.12: Mutation testing report

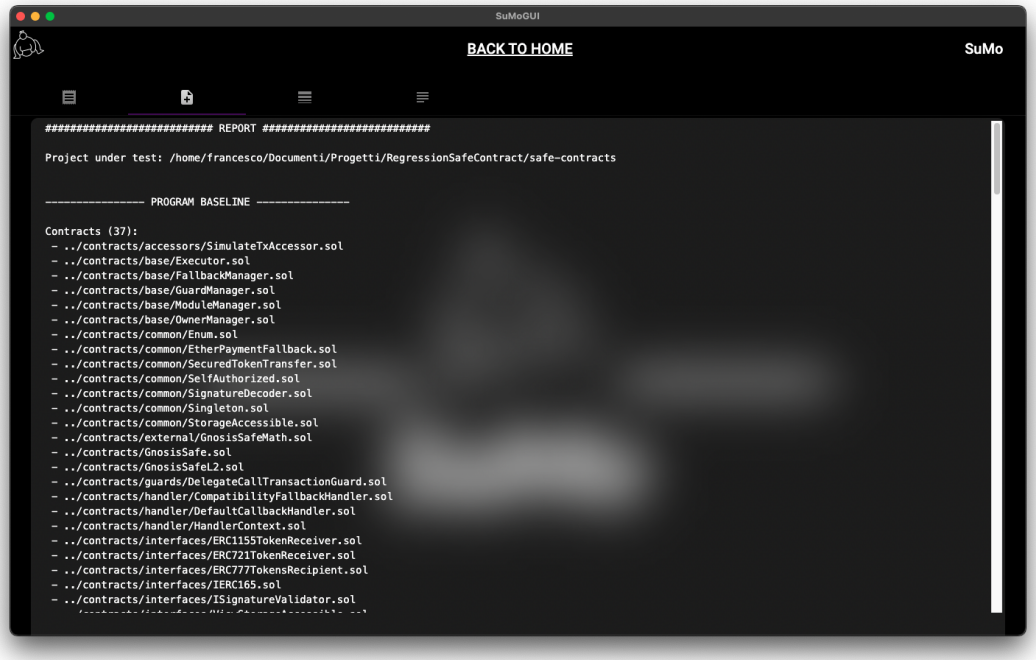


Figura 3.13: Regression testing report

4. Validation

4.1 Introduction

The last part of our work consists in testing and validating the results produced by some projects with the new SuMo implementation (called ReSuMo) in order to prove that the additions made actually affect performance. To perform the tests, we used an Ubuntu machine which mount an Intel Xeon E-2226G at 3.60 GHz and 4.8 GB of RAM.We consider already tested project in the previous SuMo version and we tested them again with TCE and Regression Testing. These projects are:

- CrowdFounding[1]: Is a DApp for crowdfunding campaigns with three main figures: the organizers, the donors, and the beneficiaries.
- Blackjack[2]: A DApp to play Blackjack game on Ethereum
- Safe-Contracts[3]: Is a DApp that use multiple signature functionality that allows secure management of blockchain assets.

The first two projects use the Truffle[4] suites as development and testing framework for Ethereum smart-contract. Instead, SafeContracts use Hardhat[5] with the same purpose. Both suites are supported by ReSuMo that understands automatically which tool is used by the project and changes its operations according to the framework used.

We start TCE validation by testing all the programs in two combination: with and without TCE, they are tested with optimized version of the operators in both combinations. Next, we proceed with the Regression Testing validation, to achieve that we use a lot of commits and we test them in chronological order and save results for each run. Thanks to the regression testing, we notice that some snapshot produce several less mutants than the same project without a regression testing process, so we saved a lot of time. These improvements in Sumo's performance can be proved by the following reports.

[1] https://github.com/giobart/EtherCrowdfunding
[2] https://github.com/0x9060/blackjack
[3] https://github.com/gnosis/safe-contracts
[4] https://trufflesuite.com/truffle/
[5] https://hardhat.org/

4.2 TCE improvements

4.2.1 CrowdFounding

The first project that we test for TCE validation is *"CrowdFounding"* with the time-out set to 5 minutes.

CrowdFounding	TCE-ON	TCE-OFF
Generated mutants	604	604
Equivalent	33	0
Redundant	54	0
Still Born	48	48
Tested mutants	496	556
Killed	233	260
Live	236	296
Timed-out	0	0
Time	296.19 minutes	323.30 minutes
MS	49.68	46.76

Tabella 4.1: *CrowdFounding* table of MT process

From the table above with written results deriving from run without regression testing of the project, it emerges that TCE improves dramatically in several factors, we can see a decrease in the time spent performing mutation testing (for example from 323.30 to 296.19 minutes from Table 4.1). This decrease is justified by the fact that ReSuMo run tests to fewer mutants, for example in Table 4.1 thanks to the TCE 87 mutants are not tested and this represents a time saving of about 8.4%. Using TCE, the testing process will return a more accurate Mutation Score.

4.2.2 SafeContracts

The second project that we test for TCE validation is *"SafeContracts"* with the time-out set to 5 minutes.

SafeContracts	TCE-ON	TCE-OFF
Generated mutants	978	978
Equivalent	33	0
Redundant	0	0
Still Born	152	152
Tested mutants	792	825
Killed	579	579
Live	213	246
Timed-out	1	1
Time	447.18 minutes	463.34 minutes
MS	73.11	70.18

Tabella 4.2: *SafeContracts* table of MT process

As shown in the table of *SafeContracts* results, the TCE system allow us to save time during the mutation testing process. In fact, it was possible to save 16.16 minutes

from the total testing time. Another important improvement of using Trivial Compiler Equivalence is the fact that TCE allows to receive a more accurate mutation score. We can also notice that *SafeContracts* run with TCE presents a better MS (73.11) than the other run without our improvement (70.18).

The TCE also allows, as in this case, to decrease the effort required by the tester to manually analyse and discard mutants equivalent to the end of the process.

4.2.3 BlackJack

The last project that we tested is *"Blackjack"* with the time-out set to 5 minutes.

Blackjack	TCE-ON	TCE-OFF
Generated mutants	540	540
Equivalent	38	0
Redundant	7	0
Still Born	46	46
Tested mutants	447	492
Killed	137	150
Live	310	342
Timed-out	2	2
Time	154.37 minutes	168.53 minutes
MS	30.65	30.49

Tabella 4.3: *Blackjack* table of MT process

BlackJack also presents the same improvements as the other projects, it has time saving (168.53 versus 154.37) and a small adjustment of the Mutation Score to 30.65 instead of 30.49. Blackjack presents an obvious problem, deductible from the very low MS. The lower is the mutation score, the more a test suite is badly written, and this fact finds evidence in the fact that most mutants are not detected by the tests.

4.3 Regression Testing improvements

In order to validate the improvement that our implementation gave to SuMo, we decided to execute this project: Safe-Contracts. We chose to test Safe-Contracts with regression testing because of the complexity and the great number of contracts and tests in the project itself in fact it acts at file level, so the appliance of the regression testing technique can show up some considerable results. For the project we started testing a certain commit and then we proceeded executing subsequent commits to demonstrate that when changes affect delimited part of the code (like a function, a test case, a block, a module...) is useful to use regression testing.

4.3.1 Safe-Contracts

What we expected from the implementation of regression testing was confirmed by the result we got. Indeed, we gain time from test execution with the activation of regression testing, due to the fact that only the necessary tests were involved on testing process. Of course, testing a project for the first time is as expensive as if the regression testing is off. Profits are evident when the programmer decides to modify just a limited part of the code and, as a consequence of waiting a long time, we only execute those parts that are changed or are involved in these changes.

Safe-Contracts is a Solidity project developed by Gnosis, we run it with ReSuMo[6] following four snapshots in order to validate the regression testing improvements. These commits include changes to the contracts and to the tests as a mean to verify the correct operation made by the regression testing mechanism; selecting contracts and tests that are changed or depends on other changed contracts or tests. From this validation, we can see that no commit except the first produce the total number of mutants. Through this mechanism it is possible to reduce the execution time and have a better overview, thanks to the enhancement of the accuracy of the mutation score, furthermore code changes have impacted on the project reliability.

As we can see from Table 4.4 commit 294 and 295 have the same number of changed contracts (3), but the first one tests 23 contracts using 20 tests, instead the other one despite the 3 changes use 5 contracts in testing phase. That because of the centrality and dependencies of the contract, so, the more dependencies a contract have, more contracts and test will be used. Some snapshots resulting from commits such as commit 295 reports a few number of changes and it could be tested in a few time. If regression testing was not active it would have taken, to run, the same time if not longer than the previous commit, so the time could have been 363.20 minutes or more than 7.05 minutes.

Figure 4.1 is an example of a *"TXT"* file generated during regression testing phase. It contains all information about contracts and tests that are in the baseline, the ones that have been changed and ones that are used, it also includes contracts that are ignored in ReSuMo configuration.

[6]ReSuMo is the final name of SuMo project with TCE, ReSuMe and reports integration

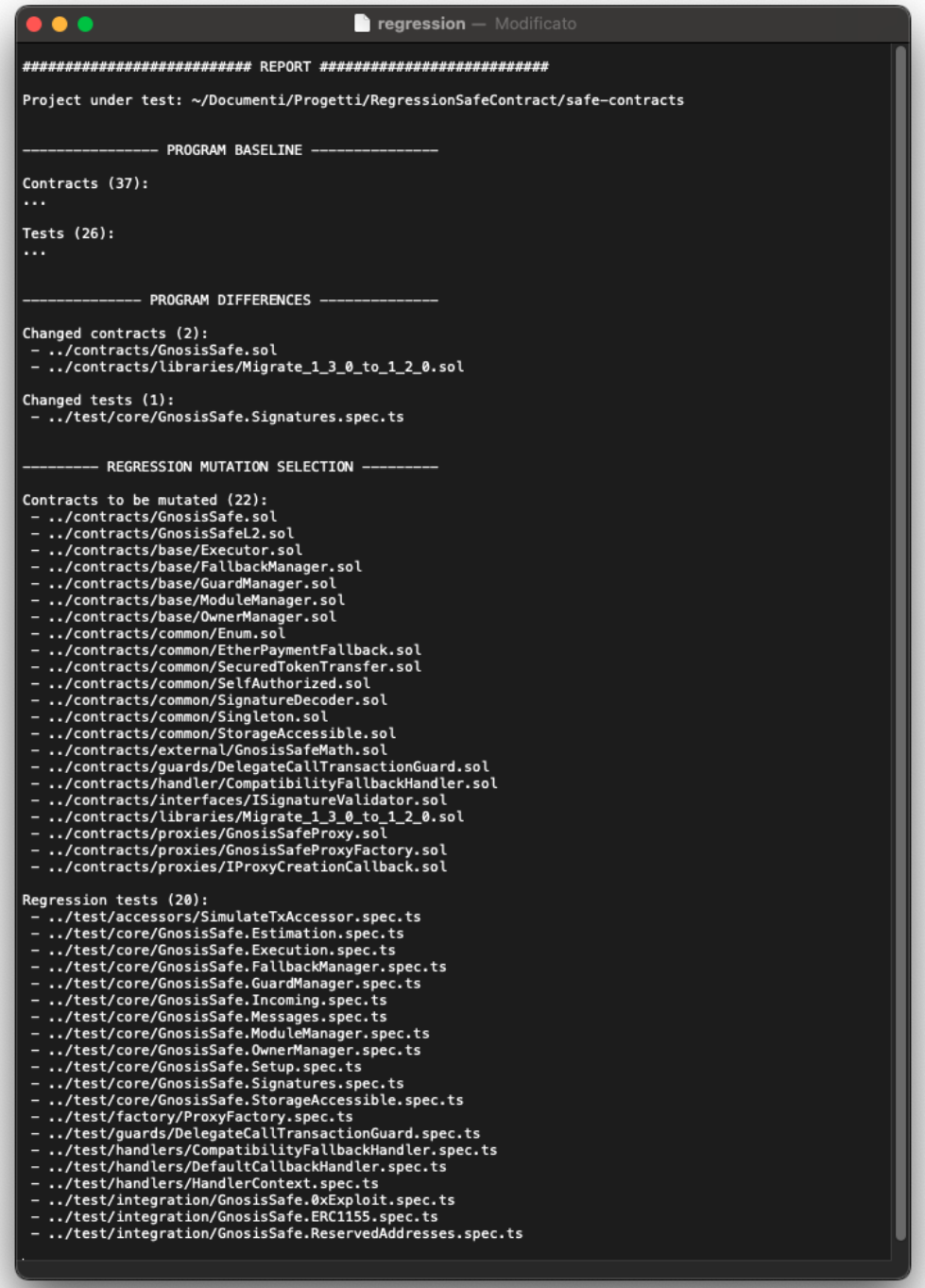


Figura 4.1: Regression report of Commit 294

Safe-Contracts	Commit 291	Commit 294	Commit 295	Commit 298
Total Mutants	943	826	32	893
Mutants tested	773	685	25	735
Changed Contracts	37	3	3	2
Changed Tests	26	0	1	1
Used Contracts	37	23	5	22
Used Tests	26	20	2	20
Time	407.32 m	363.20 m	7.05 m	400.05 m
MS	73.61	73.43	72.00	74.56

Tabella 4.4: *Safe-Contracts* table of RT process.

4.3.2 Safe-Contracts with and without regression

How we can see, in the following tables, the execution data of the same snapshot of *Safe-Contracts* project with and without regression testing. How we can see from the field *"Changed Contracts/Tests"*, contracts and test changed in the no RT run are the same of the RT run because they are the same snapshot coming from the same commit. What makes the difference is the field *"Used Contracts/Tests"*, without regression testing, ReSuMo performs mutation testing on a larger number of contracts. This fact leads to greater number of mutants, a longer testing time and a not reliable Mutations Score.

Safe-Contracts	Commit 294 RT	Commit 294 no RT
Total Mutants	826	943
Mutants tested	685	773
Changed Contracts	3	3
Changed Tests	0	0
Used Contracts	23	37
Used Tests	20	26
Time	363.20 m	402.49 m
MS	73.43	73.61

Safe-Contracts	Commit 295 RT	Commit 295 no RT
Total Mutants	32	941
Mutants tested	25	770
Changed Contracts	3	3
Changed Tests	1	1
Used Contracts	5	37
Used Tests	2	26
Time	7.05 m	404.18 m
MS	72.00	74.03

Safe-Contracts	Commit 298 RT	Commit 298 no RT
Total Mutants	893	947
Mutants tested	735	773
Changed Contracts	2	2
Changed Tests	1	1
Used Contracts	22	37
Used Tests	20	26
Time	400.05 m	423.24 m
MS	74.56	74.13

As already said, the impact of the regression testing in ReSuMo depend on which contracts the changes are made. In the commit #294 and #298 the changes were made on few contracts that are strictly connected to others, in fact during mutation testing are used 23 and 22 contracts. On the contrary, commit #295 changes 3 contracts that are peripheral, so the tool use only 5 contracts during testing process, strongly decreasing the requested time. The same concept can be applied to what tests ReSuMo must use.

5. Conclusion and Future Works

In conclusion, our work on ReSuMo proved to be useful in more than one front. With the integration of the Trivial Compiler Equivalence algorithm, we proved that equivalent and redundant mutants could slow down the time of the mutation testing process and could falter the mutation score. Another important piece of evidence is the fact that the regression testing process can be integrated with the mutation testing and it has many advantages, with certain commits can significantly reduce testing time without compromising the veracity of the results, it also could present a more accurate mutation score.

The introducing of reports containing test information are the means that the user can consult in order to control the progress of the project that is testing, thanks to the report creation system that we programmed it was easy to validate the integration of TCE and regression testing in SuMo.

Our graphic user interface allowed to decrease the learning curve of ReSuMo letting us to easily set all the setting parameters and consult all reports resulting from the mutation and regression process.

In the future, the regression testing tool could be upgraded in order to check regression at the code level. Up to now each single modification on the file is perceived as a code change but this in not functional to the regression testing, for example the addition of a comment or a white space in a contract is considered as a file modification and for this reason regression testing would be still applied on it. As said before, ReSuMo could be released like a modern web-application with a remote back-end and a local front-end in order to decrease the weight of the app and increase its performance, in addiction, with this mode of use it will be accessible by all devices: mobiles, pcs and laptops.

6. Acknowledgements

I would like to express my gratitude to Professor Polini Andrea for believing in me and proposing to me this work.

I also like to thank my supervisor Morichetta Andrea and assistant supervisor Barboni Morena who guided and followed me and my colleagues during these months and helped us to achieve this result.

A special thanks to Camerino University for making it possible and for putting me in connection with so many beautiful people.

Finally, I want to thank my family and my friends that supported me during these years.

Bibliografia

[BMP21] Morena Barboni, Andrea Morichetta e Andrea Polini. ≪SuMo: A Mutation Testing Strategy for Solidity Smart Contracts≫. In: *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE. 2021, pp. 50–59.

[But13] Vitalik Buterin. ≪Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform≫. In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[DS19] Pedro Delgado-Pérez e Sergio Segura. ≪Study of trivial compiler equivalence on C++ object-oriented mutation operators≫. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 2224–2230.

[Ojd+21] Milos Ojdanic et al. ≪Mutation Testing in Evolving Systems: Studying the relevance of mutants to code evolution≫. In: dic. 2021.

[Pap+15] Mike Papadakis et al. ≪Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique≫. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 936–946.

[Won+97] W.E. Wong et al. ≪A study of effective regression testing in practice≫. In: *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 1997, pp. 264–274. DOI: 10.1109/ISSRE.1997.630875.