Steps towards zero-touch mutation testing in Pharo

Mehrdad Abdi[1,2], Serge Demeyer[1,2]

[1]*Universiteit Antwerpen, Middelheimlaan 1, 2020 Antwerpen, België*
[2]*Flanders Make, België*

Abstract
Mutation testing is injecting artificial faults into the code to assess the written test methods. Not surprisingly, this process is time-consuming and may take hours and days to complete. On the other hand, developers, who are busy with different tasks, may find it cumbersome to run mutation testing in their workstations. In this paper, we propose some steps to develop a *zero-touch mutation testing* framework and facilitate employing mutation testing by developers. We extend MᴜTᴀʟᴋ, the mutation testing framework in the live programming environment of Pharo, by (1) adding hierarchical mutation operators, (2) integrating it to Gɪᴛʜᴜʙ-Aᴄᴛɪᴏɴs, (3) visualizing the result in a web-based mutants explorer.

Keywords
Mutation testing Ops, DevOps, Zero-touch testing, Test amplification

1. Introduction

Software is everywhere, and its failures cost. Unit testing is writing small test code snippets that exercise the unit under test and asserts the intended values. In mutation testing [1], some artificial bugs (mutations) are injected into the program under test to evaluate the test suite's strength. We say the test suite kills a mutant when at least one of the tests fails in the mutated program. Alive mutants show that the test suite needs improvements because it is indifferent to the injected faults.

Pharo [2, 3] is a dynamically typed language with a live programming environment focusing on simplicity and immediate feedback. The observations from the experiments in our past work in Pharo motivated us for this work. We developed a test amplification tool, Sᴍᴀʟʟ-Aᴍᴘ [4], that analyzes the program under test and its test suite and suggests new test methods to kill some of the mutants. During the experiment, we noticed that MᴜTᴀʟᴋ, the mutation testing in Pharo, generates too few mutants compared to the mutation testing framework in Java from another work [5]. Mutation testing in Pharo generated 1102 mutants for 52 classes (≈21 mutants per class), while there were 7980 mutants in 40 classes in Java (≈200 mutants per class). To the extent that in one of the cases (`TLLegendTest`), it failed to generate any mutant despite the class under test having 96 lines of code. This observation led us to expand the mutation operator in MᴜTᴀʟᴋ of which the details come in Section 2.

After adding new mutation operators, we witnessed that the number of times Pharo has recovered from freezing has increased, and the main reason was *entering an infinite loop*. Section 3 explains this problem with an example and how we overcome this problem.

In the next step, we created MᴜTᴀʟᴋCI as a *zero-touch mutation testing* for the live programming environment of Pharo. By default, developers are expected to use MᴜTᴀʟᴋ manually by loading it in their Pharo image, running it over their project, and waiting a considerable time to finish. We created a workflow in Gɪᴛ Hᴜʙ-Aᴄᴛɪᴏɴs that loads the project under test and runs a hierarchical mutation testing on it. We call it zero-touch because the burdensome parts of the process are automated, and the developers' attention is needed when the result is ready to be audited. We also call it *MutationTestingOps* because of its similarities to DevOps in running continuous mutation analysis. MᴜTᴀʟᴋCI is explained in Section 4.

The framework also includes a web-based mutant explorer to stash the mutation coverage status over the development time (similar to coveralls.io but for mutation testing). Using this mutant explorer, developers can assess the alive mutants and decide which to kill. We also equip the mutant explorer with a coverage indicator based on the RIPR model [6, 7, 8, 9, 10] which helps developers in their assessment. This web interface is bidirectional and allows the developer to mark the mutants as *to be killed*, which creates an issue in the repository on GitHub. The interactive mutant explorer comes in Section 4.1.

2. Expanding Mutation Operators in MuTalk

2.1. Pharo and MuTalk

Pharo is a pure object-oriented, dynamically typed language based on Smalltalk. It offers a simple language model: every action in the language is accomplished by sending messages to objects. In the context of Pharo, the term *message sending* is used instead of method invocation. As an example, there is no predefined `if` statement in the language: it is implemented as sending the message `ifTrue:` with a block argument to boolean objects. Another significant differences between Pharo and other programming languages are Phaor's live programming environment and its snapshot base nature. Unlike most programming languages, Pharo provides a live programming environment. In Pharo, developers snapshot the state of their image when they exit the environment, and reload the snapshot when they reenter. This nature of Pharo makes it vulnerable to unrecoverable changes in the system by a mutation testing tool unintentionally.

MᴜTᴀʟᴋ[1] is a mutation testing framework for programs written in Smalltalk. The original mutation operators in MᴜTᴀʟᴋ includes some known patterns related to `Boolean messages`, `Magnitude messages`, `Collection messages`, `Number messages` and `Flow control messages` [11]. Most of the original operators interchange a known messages with other know messages. For example, one of operators replaces `ifTrue:` messages with `ifFalse:`. Other operators may remove the function return operator, remove exception handling blocks, replace a block with an empty block, or replace the `ifTrue:` receiver object with `true/false` objects.

---

[1]https://github.com/pavel-krivanek/mutalk

2.2. Mutation Operators

Learned from previous works and other mutation testing frameworks[2], we added the following new mutation operators to MuTalk[3]. The list is sorted from the most coarse-grained to the finer operators:

- **Extreme transformation.** We adopted an extreme transformation operator [12, 13] that stips the whole body of the test method. In Pharo, these stipped methods always return their object (`^ self`). We use this operator as the most coarse-grained mutation that verifies whether the tests are sensitive to removing all statements from a covered method or not.
- **Disabling invocations.** As we explained earlier, every action in Pharo is achieved by sending messages. The message `#yourself` is a special message that returns the object itself. We implemented a mutation operator that replaces the sent message with `#yourself` to disable an invocation. We use this operator as the second coarse mutation that verifies whether the tests are sensitive to disabling a statement from a covered method or not.
- **Nullifying the arguments.** In this mutation operator, we replace an argument in a message send node with `nil`. This operator also verifies whether the tests are sensitive to disabling an argument in one of the statements.
- **Mutating the literals.** In this mutation operator, we mutate the literal values. We use a negation for the Boolean constants, an increase/decrease or zero for the numerical constants, and replacing with an empty string or a specific predefined string for the string values.

3. Detecting Infinite Loops

After adding the new operators, we witnessed the number of times Pharo freezes has increased so that scarcely an execution finishes. The main freezing reason was *entering an infinite loop*. Here we explain it using an example. The first code in the Listing 1 shows a method in which the factorial of an integer number is calculated recursively. The next code snippets are mutated versions of this method. In these mutants, when MuTalk runs the test to verify mutation detection, an infinite loop happens because the mutation operator disables the conditional statement. Sometimes, the operating system kills the process by an *Out of memory* error.

```
factorial: anInt
  anInt == 1 ifTrue: [ ^ 1 ].
  ^ anInt * (self factorial: anInt –1)
```

```
"Mutant 1: disabled the conditional statement by replacing the message"
factorial: anInt
  (anInt == 1) yourself.
  ^ anInt * (self factorial: anInt –1)
```

```
"Mutant 2: replaced the condition with always false"
factorial: anInt
  false ifTrue: [ ^ 1 ].
  ^ anInt * (self factorial: anInt −1)

"Mutant 3: removed return operator"
factorial: anInt
  anInt == 1 ifTrue: [ 1 ].
  ^ anInt * (self factorial: anInt −1)
```

Listing 1: Examples of an infinate loop after mutation testing.

In a language like Java, the mutation testing framework and the test runner run in two different processes. As a result, the test runner process fails with a *StackOverFlow* error in a similar mutation and is detected effortlessly by mutation testing. However, the story is different in Pharo because it is a live programming environment. The mutation testing framework and the test runner run in a shared process called Pharo image. So, an infinite loop for the test runner means the whole process losses its availability. We explained this problem in [14].

To solve this problem, we need a mechanism similar to *StackOverflow* error in Pharo. We added an auxiliary statement at the beginning of the mutated method that counts the number of its executions and throws an exception that fails the test if it reaches the defined threshold. We exploited the technique used in the class `Halt` in Pharo internals for its implementation. Although this technique significantly decreased the number of freezings, the process still may crash or freeze for other reasons. We leave recovering from other crashes as future work.

Listing 2: Auxiliary exception for avading infinate loops.

```
factorial: anInt
  RecursionError onCount: 1024. "I will go off if executed 1024 times"
  (anInt == 1) yourself.
  ^ anInt * (self factorial: anInt −1)
```

4. Zero-touch MuTalk

For using MuTalk, developers should perform some tedious tasks, including installing the tool on their Pharo image, initializing it, running it over their programs, and waiting a considerable time to obtain the results. These burdensome steps may hinder MuTalk from being used regularly. In this part, we propose a zero-touch mutation testing solution to automate the unnecessary involvement of developers.

Recently, mutation testing has been employed at scale in Google by integrating it into the build system and using a diff-based probabilistic approach to reduce the number of mutants [15]. Then in the code-review process, alive mutants are shown to developers, and they decide to kill or ignore them. In this part, we try to setup a similar process for Pharo's open-source projects.

Figure 1 illustrates the proposed hierarchical approach for running MuTalk in the CI/CD build servers. This framework is also comparable to *DevOps* [16] frameworks. DevOps provides agility in continuous software delivery by an iterative approach based on automation and
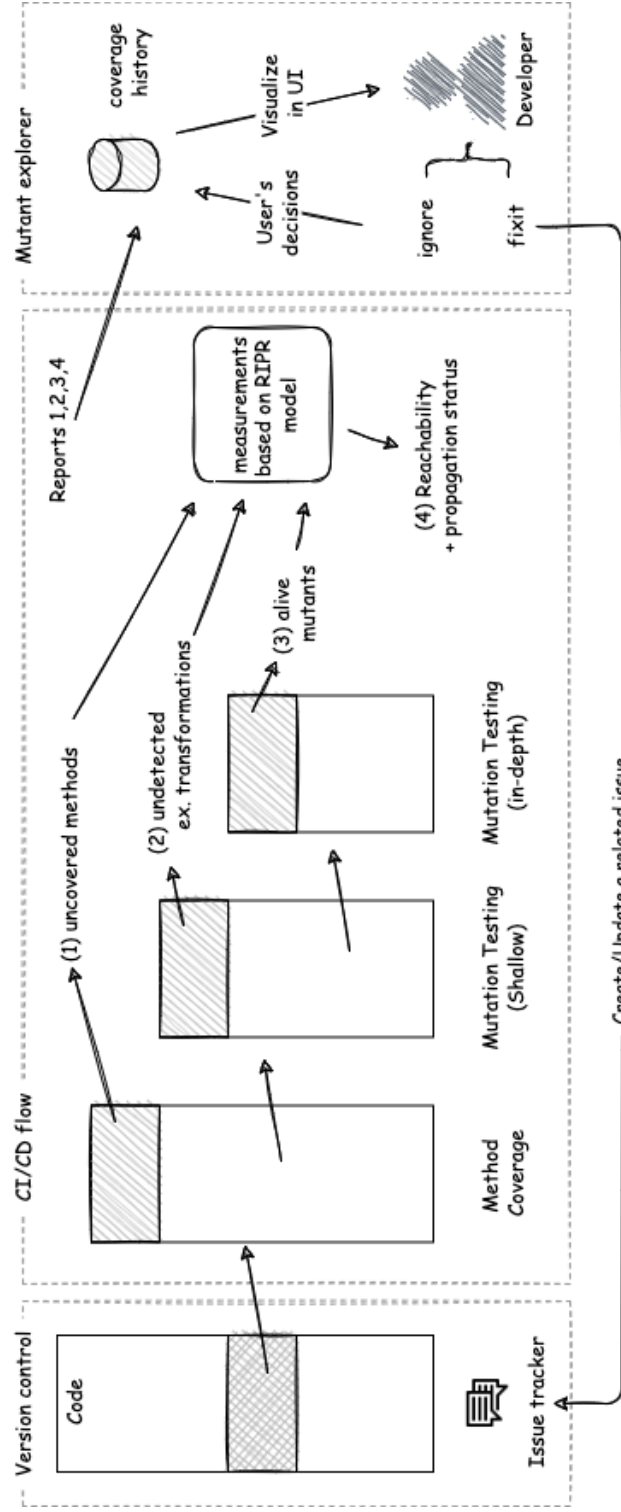
Figure 1: Hierarchical zero-touch mutation testing or MutationTestingOps for Pharo

collaboration. Similarly, we can define Mutation testing Ops (MutOps) as a continuous mutation analysis based on automation and collaboration.

Mutation testing is a time-consuming process. For a mutation testing analysis in a reasonable time, we reduce the mutation testing surface by (1) employing a diff-based mutation testing works [17, 15] that only considers the changed part in the repository and (2) using a hierarchical analysis to exclude some part of code before a full feature mutation testing.

The continuous mutation testing workflow is triggered when a new code is pushed to the repository. It runs a hierarchical analysis on the selected portion:

1. Firstly, it runs a code coverage tool to find the uncovered parts (report number 1: uncovered methods). If a method is not covered, all its mutants will survive, so we do not need to run mutation testing on it. So, we exclude all uncovered parts from the following analysis.

2. Then, a light mutation testing is executed (report number 2: undetected extreme transformations). In our implementation, we only use the *extreme transformation* operator. Similarly, if an extreme mutation on a method is not detected, we exclude it from the next analysis.

3. In the third step, a more detailed mutation testing, including all remained operators, is executed on the parts detected by the previous step, and report number 3 is formed.

Based on the RIPR model [6, 7, 8, 9, 10], a test method can kill a mutant if it reaches the mutant (reachability); the program state is different from the state in the original version at that point (infection); the infected change is propagated to the state of the test (propagation); finally, the change is revealed by an assertion statement (reveal).

To help developers to kill the mutant manually, we provide two types of coverage status for alive mutants: the list of tests covering each alive mutant and the list of tests having a propagated change (report number 4). The tests covering a mutant are start points for manual investigations on how to kill a mutant. A method with a propagated change is also interesting for developers because it says that they can kill the mutant only by adding an oracle statement to assert the state change caused by the mutation.

We developed a GITHUB-ACTIONS workflow[4] that runs MuTalk, and exports the reports as json outputs. The outputs are sent to the mutants explorer API (See Section 4.1) using GitHub's authenticated account token. We use GITHUB-ACTIONS because most of Pharo's projects currently are hosted on GitHub, and it is freely available for all open-source projects.

4.1. Mutants Explorer

Since interpreting the reports generated in Section 4 may be cumbersome, we designed a web-based mutant explorer[5]. The explorer keeps the history of all builds (similar to coveralls) and visualizes mutants and their coverage status. Furthermore, it is interactive and allows developers to assess the mutants and decide whether they should be killed or ignored. If they decide a mutant to be killed, the explorer adds an item to a GitHub issue related to this build in the repository.

---

[4]https://github.com/mabdi/smalltalk-SmallBank/blob/master/.github/workflows/mutalkCI.yml

[5]https://github.com/harolato/mutation-testing-coverage

Figure 2: A mutant view and its generated issue

Figure 2 shows an example issue to remind the developer how to kill the mutant manually. The left figure is a mutant shown to the developer in which the mutated part is displayed as a diff view on top. Then test methods covering this method are listed with an RIPR indicator. This indicator has three levels:

- ▢▢▢ If none of the levels are active, it means that the test does not reach the mutant. The tests with this degree of coverage do not help developer in killing the mutant manually, so they are excluded from the user interface.
- ▮▢▢ If only the first level is activated, it shows that the test method reaches the mutant.
- ▮▮▢ If there are two active levels, the test reaches the mutants and the change in the program state is propagated to the test state.
- ▮▮▮ If all of levels are activated, it means that the test is killed by this test method. The mutant explorer hides the killed mutants by default.

It is noteworthy that we have three levels in our proof-of-concept because we skip infection level for simplicity.

In this example, we see that `testWithdraw` not only covers the method (the first green block), but the state change from this mutant is propagated to its context (second green block). Using this report, developers understand that they can add an assertion statement to this test method to verify the method's return value `withdraw:` and kill the mutant. They can click the `FIX` button to add an issue (right figure) to the GitHub repository. Using GitHub's REST APIs and the user's token obtained with oAuth, the web interface creates an issue per build and appends all items to fix. Developers can refer to this issue later and amplify their tests manually by adding new test methods or updating their existing tests.

5. Conclusion and Future work

In this paper, we propose an approach for creating a *zero-touch mutation testing* (or Mutation-TestingOps) framework with: (1) adding new mutation testing operators to MuTalk and use an approach to identify the infinite loops and evade freezings; (2) developing a zero-touch mutation testing to automate burdensome tasks by implementing a GitHub-Actions workflow that loads the project under test and MuTalk, and runs a mutation testing process; (3) the outputs are sent to a mutant explorer in which the history of mutations is recorded and allows developers to assess mutants and mark them as to be fixed. The assessments are collected in a GitHub issue that developers can refer to in the future to amplify the tests manually.

In future work, the system will be run in practice, and a user study will be conducted to evaluate it.

Acknowledgments

the strategic research centre for the manufacturing industry.
Mutant explorer is developed by Haroldas Latonas.

References

[1] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, M. Harman, Chapter six - mutation testing advances: An analysis and survey, volume 112 of *Advances in Computers*, Elsevier, 2019, pp. 275–378. URL: https://www.sciencedirect.com/science/article/pii/S0065245818300305. doi:https://doi.org/10.1016/bs.adcom.2018.03.015.

[2] O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by Example, Square Bracket Associates, c/o Oscar Nierstrasz, 2010.

[3] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep Into Pharo, Square Bracket Associates, 2013. URL: http://books.pharo.org/deep-into-pharo/.

[4] M. Abdi, H. Rocha, S. Demeyer, A. Bergel, Small-amp: Test amplification in a dynamically typed language, Empirical Software Engineering 27 (2022) 128. URL: https://doi.org/10.1007/s10664-022-10169-8. doi:10.1007/s10664-022-10169-8.

[5] B. Danglot, O. L. Vera-Pérez, B. Baudry, M. Monperrus, Automatic test improvement with dspot: a study with ten mature open-source projects, Empirical Software Engineering, Springer Verlag (2019).

[6] N. Li, J. Offutt, Test oracle strategies for model-based testing, IEEE Transactions on Software Engineering 43 (2017) 372–395. doi:10.1109/TSE.2016.2597136.

[7] O. L. Vera-Pérez, B. Danglot, M. Monperrus, B. Baudry, Suggestions on test suite improvements with automatic infection and propagation analysis, arXiv preprint arXiv:1909.04770 (2019).

[8] L. J. Morell, A theory of fault-based testing, IEEE Transactions on Software Engineering 16 (1990) 844–857.

[9] R. A. DeMillo, A. J. Offutt, et al., Constraint-based automatic test data generation, IEEE Transactions on Software Engineering 17 (1991) 900–910.

[10] J. M. Voas, Pie: A dynamic failure-based technique, IEEE Transactions on software Engineering 18 (1992) 717.

[11] H. Wilkinson, N. Chillo, G. Brunstein, Mutation testing, 2009. European Smalltalk User Group (ESUG 09). Brest, France. http://www.esug.org/data/ESUG2009/Friday/Mutation_Testing.pdf.

[12] R. Niedermayr, E. Juergens, S. Wagner, Will my tests tell me if i break this code?, in: 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED), IEEE, 2016, pp. 23–29.

[13] O. L. Vera-Pérez, B. Danglot, M. Monperrus, B. Baudry, A comprehensive study of pseudo-tested methods, Empirical Software Engineering 24 (2019) 1195–1225. URL: https://doi.org/10.1007/s10664-018-9653-2. doi:10.1007/s10664-018-9653-2.

[14] M. Abdi, H. Rocha, S. Demeyer, Reproducible crashes: Fuzzing pharo by mutating the test methods, in: International Workshop on Smalltalk Technologies, IWST, 2020.

[15] G. Petrović, M. Ivanković, State of mutation testing at google, in: Proceedings of the

40th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 163–171. URL: https://doi.org/10.1145/3183519.3183521. doi:10.1145/3183519.3183521.

[16] L. Leite, C. Rocha, F. Kon, D. Milojicic, P. Meirelles, A survey of devops concepts and challenges, ACM Computing Surveys (CSUR) 52 (2019) 1–35.

[17] W. Ma, T. Laurent, M. Ojdanić, T. T. Chekam, A. Ventresque, M. Papadakis, Commit-aware mutation testing, in: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 394–405.