Toward Zero-touch Test Amplification

Proefschrift voorgelegd tot het behalen van de
graad van Doctor in de Wetenschappen: Informatica
aan de Universiteit Antwerpen te verdedigen door:

Mehrdad Abdi

Promoter
prof. dr. Serge Demeyer

Faculteit Wetenschappen
Departement Wiskunde-Informatica

Antwerpen, 2022

Photo is AI-generated by DALL.E [`https://openai.com/api/policies/sharing-publication/`].

*The author used the description **A humanoid robot testing software, digital art** to generate this cover photo.*

Toward Zero-touch Test Amplification

Mehrdad Abdi

Promoter:

prof. dr. Serge Demeyer

Proefschrift ingediend tot het behalen van de graad van
Doctor in de wetenschappen: Informatica

This dissertation has been approved by:

Promoter:

prof. dr. Serge Demeyer

Doctoral Jury:

prof. dr. Andy Zaidman	Delft University of Technology, The Netherlands
prof. dr. Stéphane Ducasse	Inria Lille Nord Europe, France
prof. dr. Guillermo Alberto Perez	University of Antwerp, Belgium
prof. dr. Hans Vangheluwe	University of Antwerp, Belgium
prof. dr. Serge Demeyer	University of Antwerp, Belgium

Acknowledgments

First of all, I would like to thank my parents. Thank you for my past, and everything of which I am proud. Then, I would like to thank my wife and son. You are my future. You are the motivation that makes me work hard. Great thanks to my supervisor, Serge Demeyer, for giving me this valuable chance. I learned a lot from you. Thank you for trusting me, supporting me in the project, and giving me the freedom to make a balance between my work and personal life. More importantly, thank you for proposing this exciting project: I really enjoyed working on it. During my Ph.D. study, I had the honor of working with several collaborators. I start with my co-authors. Thanks to Henrique Rocha and Alexandre Bergel for supporting me in my main publications. I also thank Ebert Schoofs, Igor Schittekat, and Haroldas Latonas for helping me expand my research territory through their excellent job in their master's projects. I thank the people who supported me from the Pharo community, especially Stéphane Ducasse, Julien Delplanque, Pavel Krivanek, and Oleksandr Zaitsev. I thank my colleagues at the University of Antwerp (in particular Hans Vangheluwe, Guillermo Alberto Perez, Moharram Challenger, Masoud Ahookhosh, Ali Parsai, John Businge, Sten Vercammen, Brent van Bladel, Mercy Njima, Gustavo Carro, Fons De Mey, Onur Kilincceker, and Mehrdad Moradi) and the SECO-ASSIST project (particularly Tom Mens, Coen De Roover, Anthony Cleve, Eleni Constantinou, Alexandre Decan, Maxime Gobert, Mehdi Golzadeh, Camilo Velazquez Rodriquez, and Pol Benats) for their aids, encouragements, and invaluable feedback. I also thank all other people who gave me time, help, and advice (in particular Benoit Baudry, Benjamin Danglout, Oscar Luis Vera Pérez, Andy Zaidman, Nicolas Anquetil, Anne Etien). I also thank my friends in Swash (in particular Ebrahim Khalilzadeh) and my colleagues in Nokia (in particular Erik Neel) for their understanding and flexibility. Last but not least, I would like to thank my Ph.D. committee, anonymous reviewers in my publications, developers who participated in our experiments, GitHub, and its sponsors for the outstanding service to the open-source community and students, and many others I might have forgotten to mention.

Mehrdad Abdi
Antwerp, Belgium, June 2022

Abstract

Effective testing is essential in today's digital society. Not only do effective tests enhance quality, speed up the development process, and reduce the risk, but ultimately they result in better software. Effective testing is even more important in the context of software ecosystems. These are networks of technical components built by a loosely coupled heterogeneous group of software engineers. The high degree of interdependencies between the components, in combination with the constant evolution within the network, makes effective testing a real challenge. In this dissertation, we investigate the use of test amplifiers in the context of software ecosystems. We exploit the symbiotic relationship between the test amplifier on the one hand and the network on the other hand. We conduct this investigation from the perspective of two feedback loops: (1) The test amplifier is fed by knowledge extracted *out* of the ecosystem, like the source code, development history, interproject dependencies, and developers' activities. (2) The test amplifier provides improvements *in* the available tests to reduce the impact of software defects. In our research, we, therefore, applied test amplifiers in new ecosystems, identified points for improvement, and proposed possible solutions. This includes preliminary results concerning the transplantation of tests from one project to a similar project within the ecosystem. As such, we have made important steps towards the ultimate dream: a "zero-touch" test amplifier that strengthens the tests within a software ecosystem without any human intervention.

Nederlandstalige Samenvatting

Het effectief testen van software is in onze digitale samenleving essentieel geworden. Niet alleen wordt de kwaliteit verhoogd, het proces versneld en het risico verlaagd maar uiteindelijk leidt dit alles tot betere software. Het belang van effectief testen geldt mogelijk nog meer in de context van software-ecosystemen. Dat zijn netwerken van technische componenten die gebouwd worden door losse samenwerkingsverbanden tussen een heterogene groep software engineers. De hoge graad van afhankelijkheid tussen de verschillende componenten in combinatie met de constante evolutie binnenin het netwerk maakt het effectief testen daar een echte uitdaging. Dit proefschrift poneert daarom het gebruik van een zogenaamde testversterkers ("test amplifiers") in de context van software-ecosystemen. Daarbij maken we gebruik van de symbiotische relatie tussen enerzijds de testversterker en anderzijds het netwerk. We gaan daarbij uit van twee feedback lussen: (1) De testversterker wordt gevoed door kennis verkregen *uit* het ecosysteem, zoals de broncode, ontwikkelingsgeschiedenis, afhankelijkheden tussen projecten en acties van de software engineers. (2) De testversterker doet verbeteringen *in* de beschikbare test om zo de impact van fouten te minimaliseren. In ons onderzoek hebben we daarom testversterkers toegepast in nieuwe ecosystemen, verbeterpunten geïdentificeerd en oplossingen voorgesteld. Daarbij hebben we ook eerste resultaten geboekt omtrent het transplanteren van tests uit één project naar een gelijkaardig project binnen hetzelfde ecosysteem. Zo hebben we enkele belangrijke stappen gezet richting de ultieme droom: een robot die tests in software ecosystemen versterkt zonder enige menselijke tussenkomst.

Publications

Papers included in this thesis:

1. Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. **Small-Amp: Test amplification in a dynamically typed language**. In *The International Journal of Empirical Software Engineering* (*EMSE*). Jul, 2022.
URL: https://doi.org/10.1007/s10664-022-10169-8.

2. Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. **AmPyfier: Test Amplification in Python**. In *Journal of Software: Evolution and Process. Special Issue: Automatic Software Testing from the Trenches* (*JSEP*). June, 2022.
URL: https://doi.org/10.1002/smr.2490.

3. Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. **Steps Towards Zero-touch Test Amplification**. (Is submitted to) *International Conference on Software Testing, Verification and Validation* (*ICST 2023*). NA, 2023.
URL: .
This paper was submitted to ASE 2022, but it was rejected. The authors incorporated the comments and submitted the new version to the ICST conference.

4. Igor Schittekat, Mehrdad Abdi, and Serge Demeyer. **Can We Increase the Test-coverage in Libraries using Dependent Projects' Test-suites?**. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering 2022* (*Vision and Emerging Results Track*) (*EASE 2022*). June, 2022.
URL: https://doi.org/10.1145/3530019.3535309.

5. Mehrdad Abdi and Serge Demeyer. **Steps Towards Zero-touch Mutation Testing in Pharo**. In *The 21st Belgium-Netherlands Software Evolution Workshop* (*BENEVOL 2022*)., 2022.
URL: https://www.researchgate.net/publication/362868185.

6. Mehrdad Abdi and Serge Demeyer. **Test Transplantation through Dynamic Test Slicing**. In *The 22nd IEEE International Working Conference on Source Code Analysis*

and Manipulation - New Ideas and Emerging Results (SCAM 2022)., 2022.
URL: https://www.researchgate.net/publication/362868454.

Papers not included in this thesis:

7. Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. **Test Amplification in the Pharo Smalltalk Ecosystem**. In *The International Workshop on Smalltalk Technology (IWST 2019)*. August, 2019.
URL: https://www.researchgate.net/publication/334884478.

8. Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. **Adopting Program Synthesis for Test Amplification**. In *The 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL19 2019)*. November, 2019.
URL: http://ceur-ws.org/Vol-2605/11.pdf.

9. Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. **Reproducible Crashes: Fuzzing Pharo by Mutating the Test Methods**. In *International Workshop on Smalltalk Technology (IWST 2020)*. September, 2020.
URL: https://www.researchgate.net/publication/354117682.

10. Serge Demeyer, Mehrdad Abdi and Ebert Schoofs. **Type Profiling to the Rescue: Test Amplification in Python and Smalltalk**. In *The 5th Workshop on Validation, Analysis and Evolution of Software Tests (VST 2022)*. March, 2022.
URL: https://ieeexplore.ieee.org/document/9825899.

11. Serge Demeyer, Ali Parsai, Sten Vercammen, Brent van Bladel, and Mehrdad Abdi. **Formal verification of developer tests: a research agenda inspired by mutation testing**. In *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles: 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20–30, 2020, Proceedings, Part II. Springer Nature (ISoLA 2020).*, 2020.
URL: https://doi.org/10.1007/978-3-030-61470-6_2.

Contents

Acknowledgments iii

Publications ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Objectives of this Thesis . . . 6
- 1.3 Research Method . . . 7
- 1.4 Summary of Contributions . . . 8
- 1.5 Structure of this Dissertation . . . 9

I Test Amplification in Dynamically-typed Object-oriented Languages 11

2 Small-Amp: Test Amplification in a Dynamically Typed Language 13
- 2.1 Introduction . . . 14
- 2.2 Background . . . 16
- 2.3 Small-Amp Design . . . 23
- 2.4 Small-Amp Extras compared to DSpot . . . 29
- 2.5 Evaluation . . . 34
- 2.6 Threats to Validity . . . 61
- 2.7 Related Work . . . 62
- 2.8 Future Work . . . 64
- 2.9 Conclusion . . . 67

3 AmPyfier: Test Amplification in Python 69
- 3.1 Introduction . . . 70
- 3.2 Background and related work . . . 72
- 3.3 AmPyfier . . . 77
- 3.4 Evaluation . . . 87
- 3.5 Conclusion . . . 97

3.6 Evaluated projects 98
3.7 Evaluation Results 99

II Toward Zero-touch Test Amplification 101

4 Steps Towards Zero-touch Test Amplification 103
4.1 Introduction 104
4.2 Test Amplification 106
4.3 Zero-touch Proof-of-Concept 108
4.4 Evaluation 117
4.5 Threats to validity 122
4.6 Related Work 123
4.7 Conclusion 124

5 Toward Zero-touch Mutation Testing in Pharo 125
5.1 Introduction 125
5.2 Expanding Mutation Operators in MuTalk 127
5.3 Detecting Infinite Loops 128
5.4 Zero-touch MuTalk 129
5.5 Conclusion and Future work 134

6 Test Amplification DevBot 135
6.1 Introduction 135
6.2 Comment Generation 136
6.3 Small-Amp DevBot 136
6.4 Vision: Test Amplification Ecosystem 137
6.5 Conclusion 138

III A Path to Test Transplantation 141

7 Can We Increase the Test-coverage in Libraries using Dependent Projects' Test-suites? 143
7.1 Introduction 144
7.2 Motivating Example 144
7.3 Evaluation 146
7.4 Related work 151
7.5 Conclusions and Future works 151

8 Test Transplantation through Dynamic Test Slicing 153

8.1 Introduction 154
8.2 Background 155
8.3 Dynamic Test Slicing 155
8.4 Proof-of-concept 161
8.5 Related Work 162
8.6 Conclusion 162

IV Conclusion **165**

9 Conclusion **167**
9.1 Final Words 168

Bibliography **189**

List of Figures

1.1 Self-* properties hierarchy 3
1.2 The symbiotic relationship between a test amplifier and the software ecosystem 6

2.1 Improvements on an existing test method submitted to SEASIDE 40
2.2 A new test method submitted to POLYMATH 41
2.3 A new test method sent in a pull-request to the project Pharo-Launcher . . 41
2.4 A new test method sent in a pull-request to the project DataFrame 42
2.5 A new assertion suggested in a pull-request to the project Bloc 43
2.6 A new test method suggested in a pull-request to the project GraphQL . . 43
2.7 A new test method suggested in a pull-request to the project Zinc 44
2.8 A new test method suggested in a pull-request to the project DiscordSt . . 45
2.9 Test methods sent in a pull-request to the project MaterialDesignLite . . . 45
2.10 A test method sent in a pull-request to the project PetitParser2 46
2.11 Changes on an existing test method - OpenPonk 46
2.12 A test method suggested in a pull-request to the project Telescope 47
2.13 The distributions of the number of killed mutants 55
2.14 The distributions of the increase kills 55
2.15 The distribution of absolute execution time (in seconds) 60
2.16 The relative distributions of the time-cost (percentage) 60

3.1 a) Coverage vs Mutation score in Original and Amplified Test Class & b) Methods Added vs Newly Killed Mutants 89

4.1 Activity diagram for a self-aware test amplification in a live system 115

5.1 Hierarchical zero-touch mutation testing or MutationTestingOps for Pharo 130
5.2 A mutant view and its generated issue 133

6.1 An example of the generated comment 136
6.2 An example of the sent pull request 138
6.3 Test amplification ecosystem with human in loop 139

8.1 A model for object representation 156
8.2 An example of versions graph 158

List of Tables

2.1 Transformations in literal amplification 26
2.2 Descriptive Statistics for the Dataset Composed of 13 Pharo Projects and the selected test classes 36
2.3 Pull requests submitted on GitHub 38
2.4 The result of test amplification by SMALL-AMP on the 52 test classes. (Tests with high coverage) 49
2.5 The result of running SMALL-AMP on 10 test class for 10 times 56
2.6 Summary of results in SMALL-AMP and DSPOT 58

3.1 Results after observing the `testDeposit` test method 82
3.2 Results after dynamic type profiling the `SmallFundTest` test class 83
3.3 Comparison of the time-cost using multi-metric selection compared to test selection based on full mutation score 90
3.4 Projects Amplified with AmPyfier 98
3.5 Classes Amplified with AmPyfier 99
3.6 The result of running AmPyfier on 54 test classes 100

4.1 Dataset composed of 5 Pharo projects from GitHub 118
4.2 Descriptive statistics for the test classes. 118
4.3 The result of the quantitative analysis. 119
4.4 Comparison of the number of newly killed mutants when prioritization is enabled and disabled 121

7.1 Descriptive Statistics for the Selected Base Packages and Dependent Projects 147
7.2 Coverage results 149

Chapter 1

Introduction

The world heavily depends on software nowadays, and its failures are costly. Software projects often evolve after release by adding new features or patching some newly discovered faults. Regression testing, consisting of a self-sufficient test suite covering the project and verifying its behaviors as intended, helps developers evolve programs easily. So, testing is indispensable because it enhances software quality and the development process and reduces the cost of failures, which leads to increased customer satisfaction.

Modern software projects do not usually live in isolation, and they interact with each other and form a larger socio-technical unit called *software ecosystems*. Similar to natural ecosystems, in a software ecosystem, a set of computing agents work concurrently on top of common software and hardware platforms. A reward mechanism, either commercial or non-commercial, makes the system stable.

While software ecosystems are gaining more importance gradually, this dissertation steps toward strengthening software tests in the context of software ecosystems.

1.1 BACKGROUND

1.1.1 Software Ecosystems

Manikas and Hansen [1] analyzed different software ecosystems definitions from the previous works and combined them as *the interaction of a set of actors on top of a common technological platform that results in a number of software solutions or services*. The symbiotic relation in these ecosystems incentivizes actors to participate: they gain some benefits from this participation, either commercial or non-commercial. As an example of software ecosystems, we can mention the Google Android ecosystem, or package dependency net-

works like npm for JavaScript projects.

Based on Mens et. al. [2], in order to carry out empirical studies, we need an ecosystem with a sufficient number of projects, an acceptable number of active contributors, and long-lived based on years of activity. It is also important that other researchers be able to reproduce and replicate these studies. Therefore, these requirements lead us to open-source ecosystems. Based on a systematic mapping study accomplished in 2017, the most studied open-source software ecosystems in past papers (before 2016) were the Eclipse and GNOME ecosystems [3].

Testing in software ecosystems. Modern software projects contain a considerable amount of hand-written tests which assure that the code does not regress when the system under test evolves. Indeed, several researchers reported that the test code is sometimes larger than the production code under test [4, 5, 6]. More recently, during a large-scale attempt to assess the quality of test code, Athanasiou et al. reported six systems where test code takes more than 50% of the complete codebase [7]. In these tests, developers try to cover the important cases and use test oracles to verify the intended program behaviors.

Existing tests in the projects are a valuable source of knowledge to improve testing in software ecosystems [8]. For example, these tests can be mutated to generate new test variants covering corner cases [9] or can be transferred between projects (Test transplantation) similar to Software transplantation [10].

1.1.2 Test Amplification

In their survey paper [8], Danglot et al. define test amplification as follows:

> *Test amplification consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (e.g., improve coverage of changes or increase the accuracy of fault localization).*

Test amplification based on unit test synthesis. This test amplification is not a replacement for other test generation techniques and is considered a complementary solution. The main difference between test generation and test amplification is using the existing test suite. Traditional test generators take the program-under-test or formal specifications as the main input and ignore the original test suite, which an expert has written.

A typical test amplification tool is based on two complementary steps.

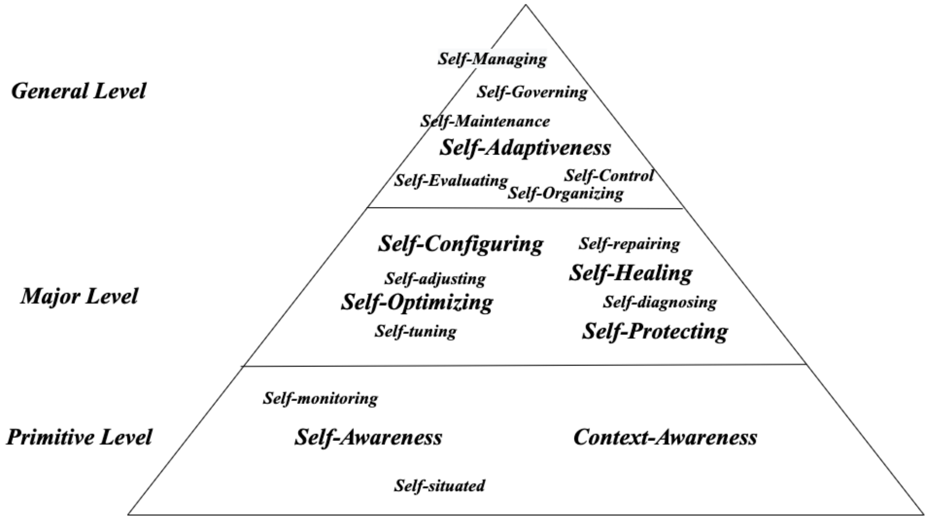


Figure 1.1: Self-* properties hierarchy

(i) *Input amplification.* The existing test code is altered in order to force previously untested paths. This involves changing the set-up of the object under test, providing parameters that represent boundary conditions. Additional calls to state-changing methods of the public interface are injected as well.

(ii) *Assertion amplification.* Extra assert statements are added to verify the expected output of the previously untested path. The system under test is then used as an oracle: while executing the test the algorithm inspects the state of the object under test and asserts the corresponding values.

The input amplification step is typically governed by a series of *amplification operators*. These operators represent syntactical changes to the test code that are likely to force new paths in the system under test. To verify that this is indeed the case, the amplification tool compares the (mutation) coverage before and after the amplification operator.

1.1.3 Zero-Touch Testing

Self-adaptive systems and self-* properties. A *self-adaptive software* monitors the changes in itself and its context and decides how to respond to these changes in order to reduce human supervision. The properties required to achieve such a system are called *self-* properties*. Salehie and Tahvildari introduce a hierarchical model view to classify these properties into three categories [11]. Figure 1.1 illustrates these categories and the related properties. The details about each of these properties are beyond the scope of this thesis.

Test automation model (TAIM). Eldh [12] introduces a test automation model in 6 levels. Level 0 in this model is manual testing, and the highest level (level 5) is called autonomous or Zero-touch testing. In a zero-touch testing tool, most of the self-* properties are implemented, and the human intervention is minimal or zero.

1.1.4 Pharo

Pharo [pharo.org] is a pure object oriented language based on Smalltalk [13, 14]. It is dynamically typed; i.e. there are no type declarations for variables, parameters, nor return values statically, but dynamically, the environment enforces that all objects to have a type and only respond to messages part of the interface. It includes a run-time engine and an integrated development environment with code browsers and live debugging. Pharo users work in a live environment called *Pharo image* where writing code and executing it is tied seamlessly together.

Invoking a method in Pharo is called *message sending*. As a pure language, every action in Pharo is achieved by sending messages to objects. There are no predefined operators, like `+` or `-`, nor control structures like `if` or `while`. Instead, a Pharo program sends the message `#+` or `#-` to a number object, a `#ifTrue:ifFalse:` message to a boolean object, or the message `#whileTrue:` to a boolean returning block object. Any message can be sent to any object. In case the message is not part of the object interface, instead of a compile-time syntax error, the system raises a `MessageNotUnderstood` exception in runtime.

Like Java, all ordinary classes inherit from the class `Object` and every class can add instance variables and methods. Unlike Java, all instance variables are private and all methods are public. Pharo encourages programmers to write short methods with intention revealing names so that the code becomes self explanatory.

Pharo is a live programming environment [15], and offers the notion of *liveness* [16] which greatly impacts how developers work: The system always offers an accessible evaluation of a source code instead of the classical edit-compile-run cycle, and as a consequence, the live programming environment allows for nearly instantaneous feedback to developers instead of forcing them to wait for the program to recompile [17].

1.1.5 Python

Python, one of the most popular (dynamically typed) languages today according to IEEE[1] and on Github[2], is a dynamically typed interpreted language, which supports multiple programming paradigms. Python does not force developers to write their code in an

[1] `https://spectrum.ieee.org/top-programming-languages/` (accessed on 15/1/22)
[2] `https://madnight.github.io/githut` (accessed on 15/1/22)

object-oriented manner, allowing developers to write code following multiple programming paradigms. In the background, however, everything in Python is an object, even the current stack frame.

While everything is an object according to the inner workings of Python, Python has no notion of encapsulation. Private or protected attributes can not be enforced. Every attribute is public and can be accessed from everywhere. It is up to the developer to follow the coding conventions where an attribute should be considered protected if it is prefixed with one underscore (e.g. `_protected_attribute`), and private if it is prefixed with two underscores (e.g. `__private_attribute`).

1.1.6 Program Synthesis

Program Synthesis is the task of automatically creating programs from the underlying programming language that satisfy user intent [18]. This user intent is typically expressed in constraints like input-output examples, demonstrations, natural language, partial programs, and assertions. As we mentioned in Section 1.1.2, one of the typical approaches in test amplification is synthesizing new test methods and adding them to the existing test methods. We can see similar goals between test amplification and program synthesis: users define their intention in a higher-level language, and computers synthesize code snippets that satisfy these intentions.

This section introduces two research areas in program synthesis that can be useful in future works.

Program sketching. In sketching, programmers provide their high-level insights using partial programs, and the synthesizer implements the low-level details. This low-level implementation is generated using counterexample-guided inductiove synthesis (CEGIS). The cegis algorithm relies on an important empirical hypothesis; for most sketches, only a small set of inputs is needed to fully constrain the solution [19].

BigCode. In recent years, academics and practitioners have seen arising the valuable resource of Big code. Big code is the vast amount of code available on the web from open source projects mainly hosted in publicly shared repositories like Github. These projects contain not only source code, but also the history of development, issues, reported bugs and review processes. The availability of big code suggests a new, data-driven approach to developing software [20].

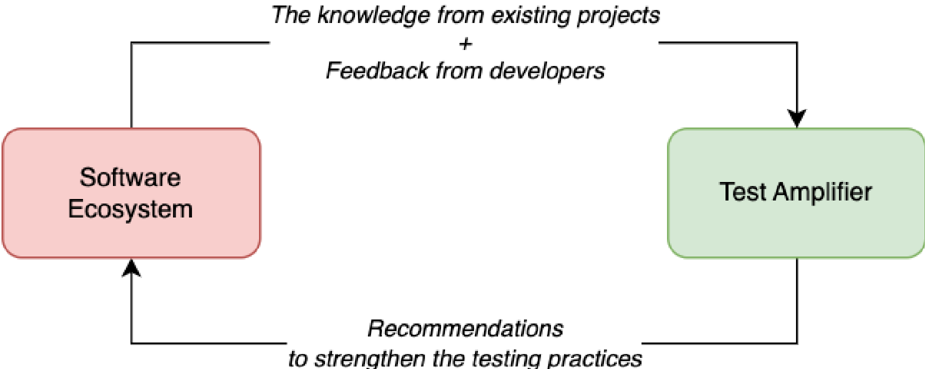


Figure 1.2: The symbiotic relationship between a test amplifier and the software ecosystem

1.2 OBJECTIVES OF THIS THESIS

> ***Thesis:*** *A symbiotic relationship exists between test amplifiers and software ecosystems. This symbiotic relationship is based on two feedback loops: (1) The test amplifier is fed by knowledge extracted *out* of the ecosystem, like the source code, development history, interproject dependencies, and developers' activities. (2) The test amplifier provides improvements *in* the available tests within the ecosystem to reduce the impact of software defects.*

This relationship is illustrated in Figure 1.2. As we mentioned in Section 1.1.1, modern software repositories contain a considerable amount of test code, and this valuable source of knowledge can be exploited to amplify testing practices. This observation leads us to the notion of *test amplification*.

$\Longrightarrow$ ***Goal 1****: We aim to expand the state-of-the-art in test amplification by extending it to new ecosystems. During our exploration, we will identify points for improvement and propose possible solutions.*

Human and AI-based development tools can collaborate in software evolution based on programmer intent [21]. The idea of employing intelligent programs to assist developers is not a new idea. We can find the early attempts in the 80s such as the project of *programmer's apprentice* [22, 23]; and we see this trend is still active nowadays [18, 24]. We need test amplifiers to be available in the ecosystem as recommender systems. Automating all unnecessary developers' involvement is also important to make it more practical. Ideally, a test amplifier should be a full member of the team and serve as a virtual developer, and humans intervene only in identifying engineering goals to be achieved or revising and approving the final results. So, we must identify obstacles that hinder these

tools from being fully autonomous (zero-touch) tools within ecosystems. The vision of employing test amplification tools as bots is also mentioned by Danglot [25] as a long-term perspective.

⟹ ***Goal 2**: We intend to make steps toward a zero-touch test amplifier that strengthens the tests within a software ecosystem without any human intervention.*

Software projects evolve together within a software ecosystem. Based on this coevolution, the efforts in a project may be reused in a similar project. For example, a project that depends on another project tests some parts of it indirectly. So we can reuse the testing inputs/patterns extracted from a project in another related project. We also aim to show the feasibility of test transplantation within a software ecosystem.

⟹ ***Goal 3**: We aim to demonstrate how tests may be transplanted from one project into a similar project within the ecosystem.*

1.3 RESEARCH METHOD

We started this journey with a *replication study*. The state-of-the-art test amplification approach introduced by DSpot [9, 26] had been evaluated only in the statically typed language of Java. By replicating the approach in new ecosystems (1) we acknowledge the validity of the originally conducted research, (2) we gain a deeper understanding of the challenges in test amplification which will lead us to more effective contributions and clearer visions.

We selected two dynamically typed languages of *Pharo*, and *Python* as the replication target ecosystems. Pharo is fully object-oriented with a simpler language model compared to Python. Pharo provides a live programming experience by allowing our tool to modify the language when it runs which makes it a suitable environment for program synthesis tasks. It also benefits from an active and friendly community, which is helpful in the qualitative evaluation of the recommendations. However, its community is small and contains a smaller number of mature projects compared to Java and Python. Therefore we repeated the replication in Python, a more commonly used language and a larger ecosystem. The replication in Pharo —Small-Amp— is evaluated by replicating the quantitative and qualitative studies from DSpot. The replication in Python —AmPyfier— is evaluated by a quantitative study.

Then, we created a proof-of-concept zero-touch test amplification solution for Pharo by integrating Small-Amp with GitHub-Actions. We extended the mutation testing tool in Pharo to make it a zero-touch mutation testing solution. We also created a GitHub bot for providing the amplified tests and interacting with developers.

Finally, we study the feasibility of dependency-based test transplantation and intro-

duce a test-slicing algorithm in two emerging result track papers.

1.4 SUMMARY OF CONTRIBUTIONS

Contributions. This dissertation makes the following contributions:

- SMALL-AMP, a test amplification algorithm and tool, implemented in Pharo Smalltalk. To the best of our knowledge, this is the first test amplification tool for a dynamically typed language. (See Chapter 2)
- Demonstrating the use of *dynamic type profiling* as a substitute for type declarations within a system under test. (See Chapter 2)
- Introducing an approach for *oracle reduction* to remove the unnecessary generated assertion statements. (See Chapter 2)
- Introducing an approach for *test input reduction* to solve the test input explosion problem. (See Chapter 2)
- AMPYFIER, a test amplification tool for Python. To the best of our knowledge, this is the first test amplification tool for Python. (See Chapter 3)
- Introducing a *multimetric selection* approach to speed up the selection process based on mutation testing. (See Chapter 3)
- Introducing an approach for *test method prioritization* to increase the performance in the limited time budget. (See Chapter 4)
- Introducing *test class sharding* that enables test amplifiers to amplify large test classes. (See Chapter 4)
- Introducing an approach for *crash-recovery* that makes test amplifiers crash resilient. (See Chapter 4)
- MUTALKCI as a proof-of-concept solution for *zero-touch mutation testing*. (See Chapter 5)
- SMALL-AMP as a *DevBot*, an attempt to provide an ideal DevBot for test amplification. (See Chapter 6)
- We introduce using *the graph of histories* for slicing tests dynamically and generating fine-grained tests from examples and system-level tests. (See Chapter 8)
- We introduce *dependency-based test transplanting* to exploit the testing knowledge in the dependent projects. To this aim, we propose dynamically slicing the tests in dependent projects and then transplanting them to the base project. (See Chapter 7 and Chapter 8)
- SMALL-MINCE, a test slicing algorithm and tool, implemented in Pharo Smalltalk. (See Chapter 8)

1.5 STRUCTURE OF THIS DISSERTATION

This dissertation is organized into three parts corresponding to our main three goals. In part I, we explain the details about replicating DSpot in two dynamically typed languages, Pharo and Python. In part II, we explain steps toward the zero-touch test amplification and test amplification ecosystem. In part III, we explore the test transplantation, and finally, we conclude the dissertation in part IV.

Part I

Test Amplification in Dynamically-typed Object-oriented Languages

Chapter 2

Small-Amp: Test Amplification in a Dynamically Typed Language

This chapter is a revised version of an originally published paper in the *The International Journal of Empirical Software Engineering* (*EMSE*):

Small-Amp: Test amplification in a dynamically typed language

Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel
In *The International Journal of Empirical Software Engineering* (*EMSE*). Jul, 2022.
URL: https://doi.org/10.1007/s10664-022-10169-8.

ABSTRACT

Some test amplification tools extend a manually created test suite with additional test cases to increase the code coverage. The technique is effective, in the sense that it suggests strong and understandable test cases, generally adopted by software engineers. Unfortunately, the current state-of-the-art for test amplification heavily relies on program analysis techniques which benefit a lot from explicit type declarations present in statically typed languages. In dynamically typed languages, such type declarations are not available and as a consequence test amplification has yet to find its way to programming languages like Smalltalk, Python, Ruby and Javascript. We propose to exploit profiling information —readily obtainable by executing the associated test suite— to infer the necessary type information creating special test inputs with corresponding assertions. We evaluated this approach on 52 selected test classes from 13 mature projects in the Pharo ecosystem containing approximately 400 test methods. We show the improvement in killing new mutants and mutation coverage at least in 28 out of 52 test classes (≈53%). Moreover, these generated

tests are understandable by humans: 8 out of 11 pull-requests submitted were merged into the main code base (≈72%). These results are comparable to the state-of-the-art, hence we conclude that test amplification is feasible for dynamically typed languages.

2.1 INTRODUCTION

Modern software projects contain a considerable amount of hand-written tests which assure that the code does not regress when the system under test evolves. Indeed, several researchers reported that test code is sometimes larger than the production code under test [4, 5, 6]. More recently, during a large scale attempt to assess the quality of test code, Athanasiou et al. reported six systems where test code takes more than 50% of the complete codebase [7]. Moreover, Stack Overflow posts mention that test to code ratios between 3:1 and 2:1 are quite common [27, 28].

Test amplification is a field of research which exploits the presence of these manually written tests to strengthen existing test suites [8]. The main motivation of test amplication is based on the observation that manually written test cases mainly exercise the default scenarios and seldom cover corner cases. Nevertheless, experience has shown that strong test suites must cover those corner cases in order to effectively reveal failures [29]. Test amplification therefore automatically transforms test-cases in order to exercise the boundary conditions of the system under test.

Danglot et al. conducted a literature survey on test amplification, identifying a range of papers that take an existing test suite as the seed value for generating additional tests [30, 31, 32]. This culminated in a tool named DSpot which represents the state-of-the-art in the field [9, 26]. In these papers, the authors demonstrate that DSpot is effective, in the sense that the tool is able to automatically improve 26 test classes (out of 40) by triggering new behaviors and adding valuable assertions. Moreover, test cases generated with DSpot are well perceived by practitioners — 13 (out of 19) pull requests with amplified test have been incorporated in the main brach of existing open source projects [9].

Unfortunately, the current state-of-the-art for test amplification heavily relies on program analysis techniques which benefit a lot from explicit type declarations present in statically typed languages. Not surprisingly, previous research has been confined to statically typed programming languages including Java, C, C++, C#, Eiffel [8]. In dynamically typed languages, performing static analysis is difficult since source code does not embed type annotation when defining variable. As a consequence test amplification has yet to find its way to dynamically-typed programming languages including Smalltalk, Python, Ruby, Javascript, etc.

In this chapter, we demonstrate that test amplification is feasible for dynamically typed languages by exploiting profiling information readily available from executing the

test suite. As a proof of concept, we present SMALL-AMP which amplifies test cases for the dynamically typed language Pharo [13, 14]; a variant of Smalltalk [33]. We argue that Pharo is a good vehicle for such a feasibility study, because it is purely object-oriented and it comes with a powerful program analysis infrastructure based on metalinks [34]. Pharo uses a minimal computation model, based on object and message passing, thus reducing possibilities to experiences biases due to some particular and singular language constructions. Moreover, Pharo has a growing and active community with several open source projects welcoming pull requests from outsiders. Consequently, we replicate the experimental set-up of DSPOT [9] by including a quantitative and qualitative analysis of the improved test suite.

This work is an extension of a previous paper presenting the proof-of-concept to the Pharo community [35]. As such, we make the following contributions:

- *Small-Amp*, a test amplification algorithm and tool, implemented in Pharo Smalltalk. To the best of our knowledge this is the first test amplification tool for a dynamically typed language.
- Demonstrating the use of *dynamic type profiling* as a substitute for type declarations within a system under test.
- *Quantitative evaluation* of our test amplification for the Pharo dynamic programming language on 13 mature projects with good testing and maintenance practices. We repeated the experiment three times. For 28 out of 52 test classes we see an improvement in killing new mutants and consequently the mutation score. Our evaluation shows that generated test methods are focused (i.e. they do not overwhelm the developer) and all amplification steps are necessary to obtain strong and understandable tests.
- *Qualitative evaluation* of our approach by submitting pull requests containing amplified tests on 11 active projects. 8 of them (≈72%) were accepted and successfully merged into the main branch.
- We contribute to *open science* by releasing our tool as an open-source package under the MIT license (`https://github.com/mabdi/small-amp`). The experimental data is publicly available as a replication package (`https://github.com/mabdi/SmallAmp-evaluations`).

The remainder of this chapter is organised as follows. Section 2.2, provides the necessary background information on test amplification and the Pharo ecosystem. Section 2.3 and Section 2.4 explain the inner workings of SMALL-AMP, including the use of dynamic profiling as a substitute for static type information. Section 2.5 discusses the quantitative and qualitative evaluation performed on 13 mature open source projects; a replication of what is reported by Danglot et. al. [8]. Section 2.6 enumerates the threats to validity

while Section 2.7 discusses related work and Section 2.8 lists limitations and future work. Section 2.9 summarizes our contributions and concludes our chapter.

2.2 BACKGROUND

2.2.1 Test Amplification

In their survey paper, Danglot et al. define test amplification as follows:

> *Test amplification consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (e.g., improve coverage of changes or increase the accuracy of fault localization).* [8]

Test amplification is a not replacement for other test generation techniques and should be considered as a complementary solution. The main difference between test generation and test amplification is the use of an existing test suite. Most work on test generation accept only the program under test or formal specifications and ignore the original test suite which is written by an expert.

A typical test amplification tool is based on two complementary steps.

(i) *Input amplification.* The existing test code is altered in order to force previously untested paths. This involves changing the set-up of the object under test, providing parameters that represent boundary conditions. Additional calls to state-changing methods of the public interface are injected as well.

(ii) *Assertion amplification.* Extra assert statements are added to verify the expected output of the previously untested path. The system under test is then used as an oracle: while executing the test the algorithm inspects the state of the object under test and asserts the corresponding values.

The input amplification step is typically governed by a series of *amplification operators*. These operators represent syntactical changes to the test code that are likely to force new paths in the system under test. To verify that this is indeed the case, the amplification tool compares the (mutation) coverage before and after the amplification operator. It is beyond the scope of this chapter to explain the details of mutation coverage; we refer the interested reader to the survey by [36].

We illustrate the input and assertion amplification steps via an example based on `SmallBank`[1] and its test class `SmallBankTest` in Listing 2.1. In this example `testWithdraw`

[1] Available at: `https://github.com/mabdi/smalltalk-SmallBank`

is the original test method while `testWithdrawAll` and `testWithdrawOnZero` are two new test methods derived from it. In the `testWithdrawAll`, the input amplification has changed the literal value of `100` with `30` (line 19), and the assertion-amplification step regenerated the assertions on the balance (line 20) and added a missing assertion on the status of the operation (line 22). The `testWithdrawAll` test method thus verifies the boundary condition of withdrawing by an amount equal to the balance. In the `testWithdrawOnZero`, on the other hand, an input amplifier has removed the call to the `deposit:` method in line 11. This test method now verifies the boundary condition that calling a `withdraw:` with an amount more than zero when the balance is zero is not allowed. This is illustrated by the extra assertions in line 29 and 30.

Code Excerpt 2.1: `testWithdraw` amplified into `testWithdrawOnAll` and `testWithdrawOnZero`

```
1  SmallBank >> withdraw: amount
2    balance >= amount
3      ifTrue: [
4        balance := balance − amount.
5        ˆ true ].
6    ˆ false
7
8  SmallBankTest >> testWithdraw
9    | b |
10   b := SmallBank new.
11   b deposit: 100.
12   self assert: (b balance = 100).
13   b withdraw: 30.
14   self assert: (b balance = 70).
15
16 SmallBankTest >> testWithdrawAll
17   | b success |
18   b := SmallBank new.
19   b deposit: 30.
20   self assert: (b balance = 30).
21   success := b withdraw: 30.
22   self assert: success.
23   self assert: (b balance = 0).
24
25 SmallBankTest >> testWithdrawOnZero
26   | b success |
27   b := SmallBank new.
28   success := b withdraw: 30.
29   self deny: success.
30   self assert: (b balance = 0).
```

2.2.2 Pharo

Pharo [pharo.org] is a pure object oriented language based on Smalltalk [13, 14]. It is dynamically typed; i.e. there are no type declarations for variables, parameters, nor return values statically, but dynamically, the environment enforces that all objects to have a type and only respond to messages part of the interface. It includes a run-time engine and an integrated development environment with code browsers and live debugging. Pharo users work in a live environment called *Pharo image* where writing code and executing it is tied seamlessly together.

Invoking a method in Pharo is called *message sending*. As a pure language, every action in Pharo is achieved by sending messages to objects. There are no predefined operators, like + or −, nor control structures like `if` or `while`. Instead, a Pharo program sends

the message `#+` or `#-` to a number object, a `#ifTrue:ifFalse:` message to a boolean object, or the message `#whileTrue:` to a boolean returning block object. Any message can be sent to any object. In case the message is not part of the object interface, instead of a compile-time syntax error, the system raises a `MessageNotUnderstood` exception in runtime. Thus, when transforming test code, a test amplification tool should be attentive to not create faulty test codes.

Like Java, all ordinary classes inherit from the class `Object` and every class can add instance variables and methods. Unlike Java, all instance variables are private and all methods are public. Pharo encourages programmers to write short methods with intention revealing names so that the code becomes self explanatory.

Protocols. Pharo, and Smalltalk in general, features *protocols* to organize the methods defined in classes. The notion of protocol is a tag of a method and it acts like a metadata provided by the integrated development environment. As such, classifying a method under a particular protocol has no impact on the behavior.

Since all instance variables are private in Pharo, in order to make them accessible by the external world, accessor methods should be provided which are typically grouped into the protocol `accessing`. In a similar vein, all methods used to set the content of an object upon initialization are grouped into the protocol `instance creation`. Long lived classes that evolve over time, use the `deprecated` protocol, signalling that these methods will be removed from the public interface in the near future. And while all methods are public, Pharo uses the protocol `private` to mark methods which are not expected to be used from the outside. However, as we mentioned earlier, protocols are a tag and Pharo does not block an access to a private method.

The most similar concepts to protocols in other languages are naming conventions, annotations and also access modifiers. For instance, a Java equivalent for methods in *accessing* protocol is following a naming convention like `setVar()` and `getVar()`. In a similar vein, Java uses `@Deprecated` annotation to identify the deprecated methods. An equivalent for methods in *private* protocol in Python is the naming convention of using underscore before the name of private methods, but Java uses *access modifiers* for this purpose.

2.2.3 Coding Conventions in Dynamically Typed Languages

In this section, we describe typical coding conventions that are used by programmers to compensate for the lack of type declarations. When we transform code (like we do when amplifying tests), special care must be taken to adhere to such coding conventions otherwise the code will look artificial and will decrease chances to be adopted by test

engineers. Our perspective comes from Pharo / Smalltalk (as documented in [37]), but similar coding conventions must be adhered to when amplifying tests in Python, Ruby or Javascript.

Parameters with unknown types. In dynamically typed languages, when defining a method which accepts a parameter, the type of the parameter is not specified. However, it is a convention to name the parameter after the class one expects or the role it takes. This is illustrated by the code snippet in Listing 2.2. Line 1 specifies that this is a method `drawOn:` defined on the class `Morph` which expects one parameter. The parameter itself is represented by an unknown type variable `aCanvas` however the name of the variable suggests that the method expects an instance of the class `Canvas`, or one of its subclasses. Line 7 on the other hand specifies that the method `withdraw:` expects one parameter and its role is to be an amount. There is no clue on the type of the parameter (integer, longinteger, float, ...); all we can infer from looking at the code is that we should be allowed to pass it as an argument when invoking the messages >= (line 8) and − (line 10) on balance.

Code Excerpt 2.2: Examples of naming conventions for parameters.

```
1  Morph >> drawOn: aCanvas
2    aCanvas fillRectangle: self bounds
3      fillStyle: self fillStyle
4      borderstyle: self BorderStyle.
5
6
7  SmallBank >> withdraw: amount
8    balance >= amount
9      ifTrue: [
10       balance := balance − amount.
11         ^ true ].
12   ^ false
```

⟹ *When passing a parameter to a method, a test amplification tool has no guaranteed way of knowing the expected type. The name of the parameter only hints at the expected type, hence during input assertion special care must be taken.*

No return types. In dynamically typed languages, there is no explicit declaration of the return type of a method. In Pharo, all computation is expressed with objects sending messages and a message sends always returns an object. By default a method returns the receiving object, which is the equivalent of the `void` return type in Java. However a program can explicitly return another value using ^ followed by an expression.

Code Excerpt 2.3: Example of a void method (left) or function method (right)

```
1  Object >> printOn: aStream
2    | title |
3    title := self class name.
4    aStream nextPutAll:
5      (title first isVowel
6          ifTrue: [ 'an ' ]
7          ifFalse: [ 'a ' ]);
```

```
8         nextPutAll: title
9
10  Object>>printString
11      | aStream |
12      aStream := WriteStream on:
13              (String new: 16).
14      self printOn: aStream.
15      ^aStream contents
```

This is illustrated in Listing 2.3, showing the methods `printOn:` (displays the receiver on a given stream) and `printString` (which returns a string representation of the receiver). `printOn:` is the equivalent of a void call thus returns the receiving object; however the method is declared on `Object` so the receiver object can be anything. `printString` on the other hand returns the result of sending the message `contents` to `aStream`. The exact type of what is returned is difficult to infer via static code analysis. Smalltalk programmers would assume that the return type is a `String` because of the intention revealing name of the method. However, there is no guarantee that this is indeed the case. Thus, when a test amplification tool manipulates the result of a method, it cannot easily infer the type of what is returned.

$\Longrightarrow$ *The lack of explicit return types makes it hard to manipulate the result of a method call while ensuring that no `MessageNotUnderstood` exceptions will be thrown.*

Different return types. In addition to the lack of return type declarations, it is also possible to write a method that can return different types of values. For example, in Listing 2.4 the method `someMethod:` can return an instance of the classes Integer, Boolean or Object (the default return value is `self`).

As a result, removing the return operator (a common mutation operator) will not cause a syntax error yet may cause a change in the return type of a methods. For example, in Listing 2.4 the method `width` (lines 5 and 6), if the return operator is removed in the mutation testing, the type of the return value will be converted from a number to a `Shape` object.

Code Excerpt 2.4: Examples of a changing the return type.

```
1  Example >> someMethod: anInt
2      anInt = 1 ifTrue: [ ^ 1 ]
3      anInt = 0 ifTrue: [ ^ false ]
4
5  Shape >> width
6      ^ width
```

$\Longrightarrow$ *Methods in dynamically typed languages can return various types. Test amplification tools must be aware that a small change in the code may lead to changes in the returned type. Consequently, assertions verifying the result of a method call must be adapted.*

Accessor methods. In Pharo, all instance variables are private and only accessible by the object itself. If one wants to manipulate the internal state of an object one should implement a method for it, as illustrated in Listing 2.5 which shows the setter method `x:` and the getter method `x`. In Pharo, such accessor methods are typically collected in the protocol `accessing` and are a convenient way for programmers to look for ways to read or write the internal state of an object.

Code Excerpt 2.5: Example of a getter (left) and a setter method (right).

```
1 Point >> x
2     ^x
```

```
3 Point >> x: anInteger
4     x := anInteger
```

Such accessor methods are especially relevant for all test generation algorithms [38]. For test amplification in particular, the setter methods are necessary in the input amplification step to force the object into a state corresponding to a boundary condition. The getter methods are necessary in the assertion amplification step to verify whether the object is in the appropriate state. However, there is no explicit declaration for the type of the parameter passed to the setter method `x:` nor for the type to be returned by the getter method `x`.

$\Longrightarrow$ *When manipulating the state of an object one cannot rely on type declarations to infer which parameter to pass to a setter method and which result to expect from a getter method.*

Pass-by-reference. In dynamic languages including Pharo, when sending messages, all arguments are passed by reference. This may imply that sometimes the state is changed and sometimes it is not. This is illustrated by the method `r` in Listing 2.6, which returns the radius in polar coordinates. This involves some calculation (the invocation of `dotProduct:`) which passes the receiver object as a reference. There is no "pass-by-value" type declaration for `dotProduct:`, so one cannot know whether the internal state is changed or not. If `dotProduct:` does not alter the internal state it may be used as a pure accessor method during assertion amplification anywhere in the test. However, if the accessor method does change the internal state the order in which the accessor methods are called has an effect on the outcome of the test.

Code Excerpt 2.6: Is `r` a pure accessor method that does not alter the internal state?

```
1 Point >> r
2     ^(self dotProduct: self) sqrt
```

$\Longrightarrow$ *The pass-by-reference parameter passing makes it difficult to distinguish pure accessor methods. Pure accessor methods can be inserted anywhere during assertion amplification, for*

accessor methods changing the internal state one must take into account the calling order.

Cascading. Listing 2.7 shows the archetypical `Hello World` example. Line 1 specifies that this is a method `helloWorld` defined on a class `HelloWorld`. Line 2 and 4 each sends the message `cr` (a message without any parameters) to the global variable `Transcript` which emits a carriage return on the console. Line 3 sends the message `show:` with as parameter the string `'hello world'` to the global variable `Transcript` which writes out the expected message.

However, a Pharo programmer would never write this piece of code like that. When a series of messages is being sent to the same receiver, this can be expressed more succinctly as a cascade. The receiver is specified just once, and the sequence of messages is separated by semi-colons as illustrated on lines 7—10.

Code Excerpt 2.7: A sequence of messages sent to the same receiver object (left) is written as a cascade (right)

```
1 HelloWorld >> helloWorld
2   Transcript cr.
3   Transcript show: 'hello world'.
4   Transcript cr.
5
```

```
6 HelloWorld >> helloWorldCascading
7   Transcript
8     cr;
9     show: 'hello world';
10    cr.
```

Instance creation. Cascading is frequently used when creating instances of a class as illustrated by the `createBorder` example in the left of Listing 2.8. In line 2 it creates a new `SimpleBorder` object and then initialises the object with color blue (line 3) and width 2 (line 4). During input amplification we need to change the internal state of the object under test, hence it is tempting to inject extra calls in such a cascade. However, because we cannot distinguish between state-changing and state-accessing methods, we risk injecting errors. The code snippet to the right illustrates that injecting an extra `isComplex` call (a call to a state-accessing method) at the end of the cascade erroneously returns a boolean instead of an instance of SimpleBorder. This will eventually result in a run-time type error via a `messageNotUndersood` exception when the program tries to use the result of `createBorderErroneous`.

Code Excerpt 2.8: Injecting extra statements may result in type errors.

```
1 TestBorder >> createBorder
2   ^ SimpleBorder new
3     color: Color blue;
4     width: 2. "returns self"
5
```

```
6 TestBorder >> createBorderErroneous
7   ^ SimpleBorder new
8     color: Color blue;
9     width: 2;
10    isComplex. "returns a boolean"
```

$\Longrightarrow$ *When injecting additional calls during instance creation, one runs the risk of returning an inappropriate value.*

> *Like most dynamically typed languages, Pharo has a lot of coding conventions. When transforming code (for instance, when amplifying tests) we must adhere to these conventions. However, the lack of explicit type information hinders static analysis, needed to identify relevant code constructs.*

2.3 SMALL-AMP DESIGN

In this section, we explain the design of the SMALL-AMP which is an adaption and extension of DSPOT [9, 26] for the Pharo ecosystem. DSPOT is an opensource[2] test amplification tool to amplify tests for Java programs. Our SMALL-AMP implementation is also publicly available[3] on GitHub.

2.3.1 Main Algorithm

The main amplification algorithm is presented in Algorithm 1 and represents a search-based test amplification algorithm. The algorithm accepts a class under test (CUT) and its related test class (TC) and returns the set of amplified test methods (ATM). In addition, the algorithm needs a set of input amplification operators (AMPS) and is governed by a series of hyperparameters:

- $N_{iteration}$ – This parameter specifies the number of iterations and shows the maximum number of transformations on a test input. The default value for this parameter is 3.
- $N_{maxInputs}$ – This parameter specifies the maximum number of generated test inputs that the algorithm keeps. It discards other test inputs. The default value for this parameter is 10.

Initially, the code of CUT and TC is instrumented to allow for dynamic profiling. The test class is executed, all required information is collected and then the instrumentation is removed again. This extra information including the type information allows us to perform input amplification more efficiently and circumvent the lack of type information in the source code (line 2). We discuss about the profiling in Section 2.4.1.

The main loop of the algorithm amplifies all test methods one by one (line 3). V is the set of test inputs, thus test methods without assertion statements. In the beginning, V has only one element which is obtained from removing assertion statements in the

[2] `https://github.com/STAMP-project/DSpot`
[3] `https://github.com/mabdi/small-amp`

Algorithm 1: SMALL-AMP amplification algorithm

```
    input  : CUT: class-under-test
    input  : TC: original test class
    input  : AMPS: a set of input amplification operators
    input  : hyperparameters {N_iteration, N_maxInputs}
    output : ATM: set of amplified test methods
 1  ATM ← {};
 2  extraInfo ← profileCollect(CUT, TC);
 3  for each t ∈ TC do
 4      V ← {removeAssertions(t)};
 5      U ← amplifyAssertions(V);
 6      ATM ← ATM ∪ {x ∈ U| x improves mutation score};
 7      for i ← 0 to N_iteration do
 8          TMP ← {} ;
 9          for each amp ∈ AMPS do
10              TMP ← TMP ∪ amplifyInputs(amp, V, extraInfo);
11          V ← reduce(TMP, N_maxInputs);
12          U ← amplifyAssertions(V);
13          ATM ← ATM ∪ {x ∈ U| x improves mutation score};
14  ATM ← improveReadability(ATM);
15  return ATM
```

original test method (line 4). U is the set of generated test methods which are generated by adding new assertion statements to the elements in V (lines 5). Then the coverage is calculated using the generated test methods accumulated in U and the tests increasing the coverage are added to the final result. SMALL-AMP uses mutation score as a coverage criteria (line 6)

In the inner loop of the algorithm (lines 7 to 13), SMALL-AMP generates additional tests by repeating the following steps $N_{iteration}$ times:

1. SMALL-AMP applies different input amplification operators on V (the current test inputs) to create new variants of test methods accumulated in the variable TMP (line 10). We discuss input amplification in Section 2.3.2.
2. SMALL-AMP reduces TMP by keeping only $N_{maxInputs}$ of current inputs and discarding the rest (line 11). We discuss input reduction in Section 2.4.2.
3. SMALL-AMP injects assertions on the remaining test inputs in V and stores the result in U (line 12). We discuss assertion amplification in Section 2.3.3.
4. SMALL-AMP selects all test methods in U that increase mutation score and adds them to the final result ATM (line 13). We discuss about test selection in Section 2.3.4.

After both loops have terminated, SMALL-AMP applies a set of post-processing steps to

increase the readability of the generated tests (line 14). We discuss these steps in Section 2.4.3.

Algorithm 1 is heavily inspired by DSPOT, but not entirely the same. In other words, we have added a pre-process step (line 2) to collect the necessary information about CUT and TC before entering the main loop. We also have added a post-processing step (line 14) to make the output more readable. We discuss about extras to DSPOT algorithm in Section 2.4.

2.3.2 Input Amplification

During input amplification, the existing test code is altered to force previously untested states. Input amplification involves changing the set-up of the object under test, passing arguments which represent boundary conditions. Additional calls to state-changing methods of the public interface are injected as well. Such changes are bound to fail the original assertions of TC, therefore SMALL-AMP removes all assertions from a test t in TC.

The test code itself is transformed via a series of *Input Amplification Operators*. These change the code in such a way that they are likely to force untested paths and cover boundary conditions. Input amplification operators are based on the genetic operators introduced in Evolutionary Test Classes [39]. Below we explain the *Input Amplification Operators* adopted from DSPOT.

Amplifying literals. This input amplifier scans the test input source code to find literal tokens (numbers, booleans, strings). Then it transforms the literal to a new literal based on its type according to Table 2.1. For example, test input shown in Listing 2.9 is transformed into `testVectorGreater_L` by manipulating the second element from the literal array.

Code Excerpt 2.9: Example Literals Amplification Operator (line 3 vs. line 7)

```
1 testVectorGreater
2   | u w |
3   u := #(−1 0 1) asPMVector.
4   w := u > 0.
```

```
5 testVectorGreater_L
6   | u w |
7   u := #(−1 1 1) asPMVector.
8   w := u > 0.
```

Amplifying method calls. This input amplifier scans the test input source code to find the method invocations on an object. Then it transforms the source code by *duplicating* or *removing* the method invocations. It also *adds new method invocations* on the objects. If the method requires new values as arguments, the amplifier creates new objects. For

Type	Transformation
Numbers	0, increased and decreased values ($+1$ and -1), doubled and halved values ($\times 2$ and $\div 2$), negated value ($\times -1$) replacing with an existing number from the test body
Booleans	negate via not
Strings	add a new random character to a random position remove a character randomly change a character randomly replace by a random string in the same size

Table 2.1: Transformations in literal amplification

primitive parameters, a random value is chosen from the profiled values. For object parameters, the default constructor is used, i.e., it creates a new instance by sending `#new` message to the class. SMALL-AMP ignores *private* and *deprecated* methods (regarding to their protocol) when it adds a new method call. The type information required do safely apply these transformations is obtained in the profiling step explained in Section 2.4.1 – p. 29.

2.3.3 Assertion Amplification

During the assertion amplification step, we inject assertion statements which verify the state of the object under test. The object under test is then used as an oracle: while executing the test the algorithm inspects the state of the object under test and asserts the corresponding values. The assertion amplification step is based on Regression Oracle Checking [40].

Note that assertion amplification is applied twice during the amplification algorithm (Algorithm 1, in lines 5 and 12). There are two reasons for this seemingly redundant design. (1) We assure that the original test method is assertion amplified as well. Since the test inputs are reduced in line 11, there is a possibility that the original test method is discarded and never reaches the assertion amplification in line 12. (2) We can run only assertion amplification by setting the value of $N_{iteration} = 0$. This way no new tests will be generated, but existing tests may become stronger because they check more conditions.

Observing state changes via object serialisation. SMALL-AMP manipulates the test code and surrounds each statement with a series of what we call "observer meta-statements" (see Listing 2.10). Such meta-statements include a surrounding block to capture possible exceptions (lines 19–20 and 24–25) and calls to observer methods to capture the state of

the receiver (line 17 and line 18) and the return value (line 18 and line 23). When necessary, temporary variables are added to capture intermediate return values (tmp1 on line 21 and line 23).

Code Excerpt 2.10: Injection of observer meta-statements

```
1  testDeposit
2    | b |
3    b := SmallBank for: 'JDoe'.
4    b deposit: 100.
5
6
7
8
9
10
11
12
13
```

```
14 testDeposit_instrumented
15   | b tmp1 |
16   [ b := SmallBank for: 'JDoe'.
17     self observe: SmallBank.
18     self observeRetVal: b.
19    ] on: Error do: [ :ex |
20       self observeException: ex].
21   [ tmp1:= b deposit: 100.
22     self observe: b.
23     self observeRetVal: tmp1.
24    ] on: Error do: [ :ex |
25       self observeException: ex]
```

After manipulating the test method, SMALL-AMP runs the test to capture the values by the observer methods. SMALL-AMP serializes objects by capturing the values from its accessor methods. If the return value of an accessor method is another object, it recursively repeats the object serialization up to $N_{serialization}$ times. $N_{serialization}$ is a configurable value (default value is 3). The output of this step of assertion amplification is a set of trace logs which reflect the object states.

Identifying accessor methods. SMALL-AMP relies on the Pharo/Smalltalk coding conventions and therefore selects methods if they belong to protocols `#accessing` or `#testing` or when their name is identical to one of the instance variables. From the selected methods, all methods lacking an explicit return statement and all methods in the protocols `#private` or `#deprecated` are rejected and the remaining are considered as accessors.

Preventing flaky tests via trace logs. A flaky test is a test that may occasionally succeed (green) or fail (red). This may happen if the test is asserting a non-deterministic value. SMALL-AMP tries to detect non-deterministic values before making assertions on them. The assertion amplification module, repeats collecting the trace logs for $N_{flakiness}$ (default value is 10 [41]) times. Then it compares the observed values. If a value is not identical between all collected logs, SMALL-AMP marks it as non-deterministic.

Recursive assertion generation. Based on the type of the observed value, zero, one or more assert statements are generated. If the type is a variant of collection or an object,

which include other internal values, the assertion generator uses a recursive method to build valid assertion statements. For non-deterministic values, the value is not asserted and only its type is asserted. The output of the assertion amplification step is a passing (green) test with extra assertions.

Intended values versus actual values. During assertion amplification, the assertion statements should assert expected values.In SMALL-AMP, we assume that the current implementation of the program is correct, and therefore we deduce the oracle from the current state of the object under test. However, when there is a defect in the method under test, the generated assertions would verify against an incorrect oracle. This is an inherent limitation for both DSPOT and SMALL-AMP, inherited from Regression Oracle Checking [40].

Example. Listing 2.11 shows an example of a trace log collected by line 22 from Listing 2.10 (left) and its recursive assertion statements (right). In this example, we point out that the method `timestamp` is an accessor method in `SmallBank` class which returns a timestamp value. Since this value differs in different executions, it has been marked as a flaky value (line 8) hence only its type is asserted (line 24).

Code Excerpt 2.11: An example of a trace log and its assertion statements

```
1  b:
2    type → SmallBank,
3    accessors:
4      balance:
5        flaky → false,
6        value → 100
7      timestamp:
8        flaky → true,
9        value → 1624
10      user:
11        type → SmallBankUser,
12        accessors:
13          name:
14            flaky → false,
15            value → 'JDoe'
```

```
16
17 testDeposit_withAssertions
18   | b tmp1 |
19   b := SmallBank for: 'JDoe'.
20   "..."
21   tmp1 := b deposit: 100.
22   self assert: b class equals: SmallBank.
23   self assert: b balance equals: 100.
24   self assert: b timestamp class equals: Integer.
       "flaky"
25   self assert: b user class equals:
       SmallBankUser.
26   self assert: b user name equals: 'JDoe'.
27   "..."
```

2.3.4 Test Selection -- Prefer Focussed Tests

During each iteration of the inner loop (lines 7 to 13 in Algorithm 1 – p. 24) SMALL-AMP generates $N_{\text{maxInputs}}$ new tests with their corresponding assertions. In the test selec-

tion step (lines 6 and 13) the algorithm selects those tests which kill mutants not killed by other tests.

First of all, SMALL-AMP performs a mutation testing analysis on CUT and TC and creates a list of live and uncovered mutants. Then SMALL-AMP selects those test methods from U (the set of amplified test methods) which increase the mutation score, thus killing a previously live or uncovered mutant. If multiple tests are killing the same mutant, the shortest test is chosen. If there are multiple short tests, the test with the least changes is chosen. In the DSPOT paper, a similar heuristic is chosen, which the authors refer to as *Focused Test Cases Selecting*.

2.4 SMALL-AMP EXTRAS COMPARED TO DSPOT

While the design of SMALL-AMP was inspired by DSPOT, the lack of explicit type information forced us to make major changes but also permitted us to make improvements. This section describes additional and diverging aspects of SMALL-AMP compared to DSPOT.

2.4.1 Dynamic Profiling to Collect Type Information

At the very beginning of the main algorithm (Algorithm 1 line 2), dynamic type profiling is done only once by executing the original test methods and observing the actual type information of variables.

In dynamically typed languages like Pharo, type annotations are not provided in the source code. So, performing static analysis which depend on types are challenging. In the context of SMALL-AMP, the most important step that relies on static code analysis is input amplification. The other steps are either based on dynamic analysis like assertion amplification, or depend on a third-party library such as selection based on mutation-testing.

In input-amplification, we can group operators into two classes as:

1. *Type sensitive operators.* These operators heavily depend on the type information and without type information they are ineffective or impossible. An important type sensitive input amplifier in SMALL-AMP is method call addition. The types of variables defined in a test method must be inferred when adding a valid method call. In addition, it needs the type information of parameters in the newly called method.

2. *Type insensitive operators.* These are all operators that are still applicable without the type information. An example is the operators amplifying literals. These operators are easy to adapt to a dynamic language because literals are distinguishable from a

token representation of the source code.

To obtain accurate type information we rely on the presence of manually written tests, which should be representative for the normal behaviour of the program under test. We exploit profiling tools (commonly available in modern program environments) to extract accurate type information from the variables present in the program. The profiler is configured to attach hooks to the relevant elements in the code. When these important code elements are executed, the hooks are triggered, the profiler reads the information from the program state and logs it.

In SMALL-AMP, we rely on two distinct profilers:

— *A Method-proxy* profiler, which collects the type of parameters in Class-Under-Test methods.

— *A Metalink* based profiler which collects the type of variables in the test methods

To apply test amplification to other dynamically typed languages one needs comparable profiling technology. Some languages provide reflexive facilities that can be exploited. Python metaclasses for example allow one to transparently hook into the code proxy objects similar to the method-proxies adopted in SMALL-AMP. If such reflexive facilities are not available, one can resort to the debugger APIs to inspect values of variables at run-time.

Profiling by Method-proxies. For gathering the type of parameters in methods, SMALL-AMP uses method proxies [42, 43]. Proxies are methods wrapping the methods in the class under test and trigger instead of the original methods. They first log the arguments and then pass the control to the original method (Listing 2.12).

Code Excerpt 2.12: Wrapper method to log the types of the parameters

```
1 ProfilingProxy >> run: aSelector with: anArray in: aReceiver
2   self logCalled: aSelector withArguments: anArray inType: aReceiver.
3   ^ aReceiver withArgs: anArray executeMethod: method.
```

The main drawback of the Method-proxy profiler is that when a method is not covered by the test class, it will not be profiled. SMALL-AMP reports the list of such uncovered methods as one of its outputs. Using this report, a developer can decide to add new tests for uncovered methods, make them private (using an adequate protocol / method tag), or remove them.

Profiling by Metalinks. Pharo provides Metalinks as a fine-grained behavioral reflection solution [34, 44]. For collecting the type of variables in the test method, SMALL-AMP

uses Metalinks.

A metalink contains an action to perform which is defined by providing a meta-object, a selector, and also a control. Metalinks can be installed on one or more nodes in the abstract syntax tree. Listing 2.13 shows how metalink is defined and installed on all variable nodes in the test method.

Code Excerpt 2.13: Defining a metalink to log the variable node type after execution

```
1  link := MetaLink new
2    metaObject: self;
3    control: #after;
4    selector: #'logNode:context:object:';
5    arguments: #(node context object).
6
7  nodes := testMethod ast allChildren
8    select: #isVariable.
9  nodes do: [ :node |
10   node link: link ]
```

Line 1 to 5 shows how Metalink is initialized. It says that after execution the AST node containing this link, the method `logNode:context:object:` will be called with the following arguments:

— `node`: The static representation of the AST node. It is used to get information such as name and the position in the code.

— `context`: The context of execution including dynamic values of the variables and stack. It is used to access to the values of temporary variables.

— `object`: The state of the object on which the metalink is installed (in this case the test class). It is used to access to the values of instance variables.

In lines 7 to 10, all variable nodes in the test method are selected and then the link is installed on them. After installing the metalinks, the test method is executed. When the execution passes each variable node, the metalink is triggered and the logger method is called. The logger method extracts the type information from the context, logs them and returns. Then, the execution on the test method continues until the end or another metalink is triggered.

How the collected data is used. The collected data from each profiler is stored as a dictionary object mapping the identifier of the profiled data to its type and a list of sample values (only for primitive types). In SMALL-AMP, there are two dictionaries, for the type of method parameters and the type of variable nodes. During the input amplification, when type information is needed, the corresponding dictionary is consulted.

2.4.2 Test Input Reduction

The input amplification step quickly produces a large number of new test inputs with the inner loop of Algorithm 1 – p. 24 – lines 7 to 13. For instance, if the number of inputs in the first iteration is $|v|$, this number in the second iteration grows to $|v| \times |v|$, and in iteration i reaches $|v|^i$. We refer to this problem as *test-input explosion*. Since the number of test inputs grows exponentially, either the number of transformations ($N_{iteration}$) needs to be chosen as small values for being feasible to try all generated test input, or we need to reduce the number of inputs by using a heuristic to select a limited number of them.

SMALL-AMP uses a random selection heuristic which maximises diversity in order to select a maximum number ($\mathrm{N}_{\mathrm{maxInputs}}$) of test inputs. This selection is different from selection by mutation score (Section 2.3.4); we name it *reduction*.

SMALL-AMP reduction considers two techniques:

— *a competitive selection.* a portion of test inputs (by default half of $\mathrm{N}_{\mathrm{maxInputs}}$) are selected completely randomly from the output of all input amplifiers.

— *a balanced selection* in the remaining portion, SMALL-AMP assures that all input amplifiers are contributed by selecting from their outputs regarding an assigned weight. In SMALL-AMP, all input amplifiers are assigned a weight (it is 1 by default for all amplifiers). This maintains a diversity in the selected test inputs.

Why diversity is important? Each input-amplifier algorithm performs transformations based on different considerations. As a result, the number of generated tests is different for input amplifiers. If the test inputs are selected purely random, the result will be dominated by generated tests from amplifiers generating more outputs. Therefore, we need to have a balance between the outputs from each amplifier.

As an example, we compare the number of new test inputs from a *statement-removal amplifier* and a *statement-addition amplifier*. The former has a $O(S)$ complexity where S is the number of statements in the test method. It means that if the number of lines in the test is increased, the number of new test inputs generated by this input-amplifier shows a linear increase. However, the latter has a $O(S * M)$ complexity where M is the number of methods in the class under test. It means that the increase in the outputs depends on not only the numbers of statements in the test, but also the number of methods in the class-under-test. Now, if we select a number of generated test methods randomly, the outputs from the latter operator are more likely to be selected; so the result will be dominated by the result from the second input-amplifier.

2.4.3 Improving Readability Via Post-Processing

In order to make the generated tests more readable, SMALL-AMP adds a few steps after finishing the main loop of the algorithm (line 14 in Algorithm 1 – p. 24). These steps do not have any effect on the mutation score of the amplified test suite; they only make the test cases more readable for SMALL-AMP users.

Assertion reduction. As described in Section 2.3.3, SMALL-AMP generates all possible assertions for all observation points. Consequently, the generated test methods easily include hundreds of new assertions most of which appear redundant. The assertion reducer is a post-processing step that discards all assertions that do not affect the mutation score.

Each amplified test method encompasses the identifier of all newly killed mutants. SMALL-AMP surrounds all assertion statements by exception handling blocks to catch exceptions, especially `AssertionFailure` raised from the assertion statements. Then, the mutation testing framework is run using the newly killed mutants only. When an `AssertionFailure` is caught, the identifier of the assertion is logged as important. Finally, SMALL-AMP keeps only important assertions and remove all other assertion statements.

In some cases, an assertion may call an *impure* accessor methods, i.e., an accessor method that alters the internal state of the object. When such assertions are removed, some of the next assertions may fail. SMALL-AMP runs each test method after removing unnecessary assertions, to confirm that they remain green and the mutants are still killed by the test. If the confirmation failed, the assertion reduction is not successful and all removed assertions are reinserted.

Comply with coding conventions. Before processing a test method, SMALL-AMP breaks complex statements (chains of method invocations and cascades) into an explicit sequence of message sends to permit observing state changes (see Listing 2.10 – p. 27). This is necessary to observe state changes during assertion amplification. In this post-process step, SMALL-AMP cleans up all unused temporary variables, and chooses a better name for the remaining variables based on the type of the variable. If possible, it reconstructs message chains and cascades to make the source code more readable and conform to Pharo coding conventions.

2.5 EVALUATION

To evaluate SMALL-AMP, we replicated the experimental protocol introduced for DSPOT [9]. We adopted a qualitative experiment by sending pull-requests in GitHub for evaluating whether the generated tests are relevant to the developers or not (RQ1). Next, we use a quantitative experiment to evaluate the effectiveness of SMALL-AMP (RQ2, RQ3 and RQ4). The order of RQ1 to RQ4 is exactly the same order in [9] to facilitate the comparing and it does not reflect the importance of the research questions. In RQ5, we make a detailed comparison of our results versus the ones in the original experiment. Finally, in RQ6, we report the time cost of the running SMALL-AMP, with special attention to the performance penalty induced by the additional steps (profiling and oracle reduction).

RQ1: Pull Requests. *Would developers be ready to permanently accept amplified test cases into the test repository?* We create pull-request on mature and active open-source projects in the Pharo ecosystem. We propose the improvement as a pull request on GitHub, comprising improvements on an existing test (typically due to assertion amplification) or new tests (typically the result of input plus assertion amplification). We interpret the statements where extra mutants are killed to provide a manual motivation on why this pull request is an improvement. The main contributors then review, discuss and decide to merge, reject or ignore the pull request. The ratio of accepted pull requests gives an indication of whether developers would permanently accept amplified test cases into the test repository. More importantly, the discussions raised during the review of the pull request provides qualitative evidence on the perceived value of the amplified tests.

RQ2: Focus. *To what extent are improved test methods considered as focused?* We assess whether the amplified tests do not overwhelm developers, by assessing how many extra mutants the amplified tests kill. Ideally, the amplified test method kills only a few extra mutants as then we consider the test *focussed* (cfr. Section 2.3.4 – p. 28). We present and discuss the proportion of focused tests out of all proposed amplified tests. An amplified test case is considered focused if, compared to the original, at least 50% of the newly killed mutants are located in a single method.

RQ3: Mutation Coverage. *To what extent do the amplified test classes kill more mutants than developer-written test classes?* We assess whether the amplified tests cover corner cases by using a proxy — the improvement in mutation score via the mutation testing tool MUTALK [45]. We first run MUTALK on the original class under test (CUT) as tested by the test class (TC) to compute the original mutation score. We distinguish between strong tests and weaker tests, by splitting the set of test classes in half after sorting according to the mutation score. Next, we amplify the test

class and compute the new mutation score. We report the relative improvement (in percentage).

RQ4: **Amplification Steps**. *What is the contribution of input amplification and assertion amplification (the main steps in the test amplification algorithm) to the effectiveness of the generated tests?* Here as well we use mutation score as a proxy for the added value of both the input and assertion amplification step and here as well we distinguish between strong and weak test classes. Therefore, we compare the relative improvement (in percentage) of assertion amplification against the relative improvement of input and assertion amplification combined. We report separately which amplification operators have the most impact, paying special attention to the ones which are sensitive to type information.

RQ5: **Comparison**. *How does* Small-Amp *compare against* DSpot*?* To analyse the differences in result between SMALL-AMP and DSPOT, we compare the qualitative and quantitative results reported in the DSPOT paper against the results we obtained for RQ1 to RQ4.

RQ6: **Time Costs**. *What is the time cost of running* Small-Amp*, including its steps?* To study the applicability of SMALL-AMP, we analyse the time cost of all runs in the quantitative analysis. We compare the relative time cost of each step, paying special attention to the extra overhead of profiling and oracle reduction.

2.5.1 Dataset and Metrics

Selecting a dataset. Firstly, we collected candidate projects under test from different sources: (1) We looked at the projects used in a recent paper focusing on testing in Pharo [46]. (2) We looked at the projects introduced in the "Innovation Technology Awards" section of the ESUG conference from the year 2014. (3) We used GitHub API to find the Pharo projects hosted on GitHub with more than 10 forks and 20 stars.

Then we applied a set of inclusion and exclusion criteria. Our projects need to be hosted in GitHub and written in Pharo. They should include a test suite written in sUnit, and can run in Pharo 8 (stable version). For not being overwhelmed with resolving dependencies, they need to support installation with `Metacello` and not depend on system-level packages like databases or a special installation service. We discarded all libraries that are part of the Pharo system such as collections, or compiler.

Based on the mentioned criteria, we randomly selected 20 projects. Then, we rejected projects having less than 4 green test classes with known class under test and mutation coverage less than 100, and ended up with 13 projects (Table 2.2).

Similar to the experimental protocol in DSPOT [9], we select randomly 4 test classes, 2 high mutation coverage and 2 low, for each project. If a project lacks at least 2 test classes

Table 2.2: Descriptive Statistics for the Dataset Composed of 13 Pharo Projects and the selected test classes

Project	commit Id	#test classes	#test methods	Description (Based on Gihub page)	Test classes in the experiment
Bloc	f69c94b	6	44	UI infrastructure & framework for Pharo	BlLayoutExactResizerTest^l, BlShortcutTest^l, BlInsetsTest^l, BlCompulsoryCombinationTest^l
DataFrame	7b222e2	6	497	Tabular data structures for data analysis	DataFrameCsvReaderTest^h, DataFrameJsonWriterTest^h, DataFrameTypeDetectorTest^h, DataFrameJsonReaderTest^h
DiscordSt	44be9b7	43	532	An API wrapper for Discord	DSUserTest^h, DSDetectChannelCommandTest^h, DSEmbedTest^l, DSSendUserTextMessageItemTest^l
GraphQL	3e57ca8	9	409	A Smalltalk GraphQL Implementation	GQLSchemaGrammarTest^h, GQLSingleAnonimousQueryEvaluatorTest^h, GQLRequestGrammarTest^h, GQLArgumentsTest^l
MaterialDesignLite	45f2f4d	77	484	Binds the google's Material Design Lite project to Seaside and build widgets on top of Material Design	MDLPanelSwitcherButtonTest^h, MDLPaginationComponentTest^h, MDLNestedListTest^l, MDLDialogTest^l
openponk	bfcb84b	19	128	Modeling platform	OPRTElementsConstraintTest^h, OPNullSerializerTest^h, OPProjectTest^l, OPNavigatorAdaptersTest^l
petitparser2	86243ea	47	535	A high-performance top-down parser	WebGrammarTest^h, PP2BufferStreamTest^h, PP2ParsingGuardTest^l, PP2BenchmarkTest^l
pharo-launcher	f1ce748	38	216	Manager for pharo images	PhLAboutCommandTest^h, PhLCopyImageCommandTest^h, PhLDirectoryBasedImageRepositoryTest^l, PhLLocalTemplateTest^l
PolyMath	205dcfc	57	605	Scientific Computing with Pharo	PMBinomialGeneratorTest^h, PMBernoulliGeneratorTest^h, PMFixpointTest^l, PMExponentialDistributionTest^l
Roassal3	69b5645	18	152	Visualization Engine	RSLabelGeneratorTest^h, RSUMLClassBuilderTest^h, RSDraggableCanvasTest^l, RSAthensRendererTest^l
Seaside	3038b49	55	919	Framework for developing sophisticated web applications	WAKeyGeneratorTest^h, WACookieTest^h, WAXmlCanvasTest^l, WAErrorHandlerTest^l
Telescope	a4e128a	11	75	Engine for efficiently creating meaningful visualizations	TLExpandCollapseNodesActionTest^h, TLHideActionTest^h, TLDistributionMapTest^l, TLLegendTest^l
zinc	ee0d071	27	292	HTTP Components to deal with the HTTP networking protocol	ZnEasyTest^h, ZnMessageBenchmarkTest^h, ZnStatusLineTest^l, ZnBivalentWriteStreamTest^l

having high (or low) mutation score, we select from lower (higher) covered classes instead. As result, we have 52 test classes, 27 of them considered strong (high mutation score) and 25 considered weaker (lower mutation score).

Table 2.2 shows the descriptive statistics of the selected projects with a short description, area of usage, number of test classes and test methods and their version based on git commit id, and selected test classes (a superscript ^{h} is used to indicate a test class with high mutation coverage, and ^{l} is used to indicate low mutation coverage).

Detecting the class under test. SMALL-AMP needs a test class and its class-under-test as inputs. Finding a mapping between a test and a class can be challenging. As the default mapping heuristic, we rely on the pattern used by Pharo IDE to detect a test method for a class. The Pharo code browser finds a unit test for a class as follows: it adds the postfix `"Test"` to the name of the class. If there is such class loaded in the system that is a subclass of `TestCase` it is considered as the unit test class. If this heuristic is not followed in a project, one can explicitly define the class-under-test by overriding a hook method in test classes.

Metrics. We adopt the same metrics used in the experimental protocol in [9]:

- **All killed mutants** ($\#Mutants.killed$): The absolute number of mutants killed by a test class in a given class under test.
- **Mutation score** ($\%M.Score$): The ratio (in percentage) of killed mutants over the number of all mutants injected in the class under test.

$$\%M.Score = 100 \times \frac{\#Mutants.killed}{\#Mutants.All}$$

- **Newly killed mutants** ($\#Mutants.killed_{new}$): The number of all new mutants that are killed in an amplified version of the test class.

$$\#Mutants.killed_{new} = \#Mutants.killed_{amplified} - \#Mutants.killed_{original}$$

- **Increase killed** ($\%Inc.killed$): The ratio (in percentage) of all newly killed mutants over the number of all killed mutants.

$$\%Inc.killed = 100 \times \frac{\#Mutants.killed_{new}}{\#Mutants.killed_{original}}$$

Project	Status	Pull request url
PolyMath	Merged	https://github.com/PolyMathOrg/PolyMath/pull/178
Pharo-Launcher	Merged	https://github.com/pharo-project/pharo-launcher/pull/500
DataFrame	Merged	https://github.com/PolyMathOrg/DataFrame/pull/132
Bloc	Merged	https://github.com/feenkcom/Bloc/pull/7
GraphQL	Merged	https://github.com/OBJECTSEMANTICS/GraphQL/pull/12
Zinc	Merged	https://github.com/svenvc/zinc/pull/58
DiscordSt	Merged	https://github.com/JurajKubelka/DiscordSt/pull/75
MaterialDesignLite	Merged	https://github.com/DuneSt/MaterialDesignLite/pull/308
PetitParser2	Open	https://github.com/kursjan/petitparser2/pull/64
OpenPonk	Open	https://github.com/OpenPonk/openponk/pull/35
Telescope	Open	https://github.com/TelescopeSt/Telescope/pull/162

Table 2.3: Pull requests submitted on GitHub

2.5.2 RQ1 --- Pull Requests

In this experiment, we choose an amplified test method for each project and send a pull-request in GitHub. Before the experiment, we sent a pilot pull-request to learn how developers deal with external contributions. Firstly, we explain the pilot pull-request and then all pull-requests are described one by one. Table 2.3 demonstrates the status as well as the url of each pull-request per project.

Pull-requests preparation

Each pull-request contains a single amplified test method[4]. In order to attract the developers' interest, we try to select a test method testing an important class/method. We select the class under test by scanning their name and relating the names to the context of the project. For example, we know the project `Zinc` is a HTTP component, so the class `ZnRequest` should be a core class. We run the tool on the selected test class, and then scan the generated test methods to select one of them. In selecting an amplified test method, we still consider the vocabularies in the name of the original test method. We also prioritize the tests with more mutants killed.

In addition, we need to explain why a test is valuable in each pull-request. Since some developers may not be familiar with the concept of mutation testing, we need to understand the test in advance and explain it in simpler words. We inspect at the selected test method and try to understand the effect of the killed mutant and come up with an easy to understand explanation. Examples of explanations are: increasing branch coverage (`PolyMath`), raising an exception (`pharo-project, DataFrame`), covering new state revealing methods (`Bloc, Zinc`), reducing technical debt (`GraphQL`).

After selecting an amplified test method, we perform small corrections on the gen-

[4]Exception for the project `MaterialDesignLite` where we 2 very similar test methods in a single pull request

erated code, as a normal Pharo developer would do when see an auto-generated code. These corrections include choosing more meaningful names for the test method, variables and string constants, or deleting the superfluous lines, and adding comments for small hints.

All preparation steps are performed by the first author and are reviewed by the second and third authors. In the time of experiment, the familiarity of the first author about the projects was only studying parts of provided readme description in GitHub. So, he was totally unfamiliar with the projects, he had not contributed to any of the projects, and had never reviewed their code. In fact this shows although there might be more interesting tests for experts, a normal Pharo developer with limited knowledge about the projects is able to review the output and detect some useful test methods that are merged to the projects. The preparation of the tests was quite straightforward and normally did not take more than one hour for each project.

Pilot pull-request

Initially, we sent a pull-request[5] to `Seaside` project containing the suggestion for adding a set of new lines into an existing test method. The main goal of this pull-request was to learn more about how developers deal with pull-requests from strangers.

We consider the fact that `Seaside` project is a framework for web application development and we scanned the name of classes and selected `WARequestTest` because we expected this test class is related to a core class-under-test which interacts with Http requests. Then, we amplified the test class and selected the test method with the most mutants killed.

The selected new test method was able to kill 6 new mutants and was the result of a cooperation between assertion amplification (Section 2.3.2) and input amplification (section 2.3.3). We merged the parts of amplified test into the original test method (`#testPostFields`). The test is shown in Figure 2.1. Lines 5 to 10 are produced by assertion amplification on the original test method (`#testPostFields`). Line 15 is added by the *method-call-adder* input-amplifier.

We wrote a description for the pull-request trying to explain why this test is useful. We also expressed that the test method is the output of a tool, because it is important to inform developers in advance that they are participating in an experiment.

After a few days the test was merged by one of the project's developers. Moreover, the developer left a valuable comment containing the following points:

- **The suggestions do not fit this test method**: The developer said "*I expected the test-*

[5] `https://github.com/SeasideSt/Seaside/pull/1215`

```
2      testPostFields
3             | request headers |
4             request := WARequest method: 'POST' uri: '/foo?bar=1'.
5    +        self
6    +                deny: request isXmlHttpRequest;
7    +                assert: request headers class equals: WAHeaderFields;
8    +                assert: request remoteAddress isNil;
9    +                assert: request isPost;
10   +                assert: request sslSessionId isNil.
11            headers := Dictionary new.
12            headers at: 'content-type' put: WAMimeType formUrlencoded greaseString.
13            request setHeaders: headers.
14            request setBody: 'baz=2&bar=3'.
15   +        self should: [ request bodyDecoded ] raise: WAIllegalStateException.
16            request setPostFields: (WARequestFields new at: 'baz' put: '2'; at: 'bar'
       put: '3'; yourself).
17
```

Figure 2.1: Improvements on an existing test method submitted to SEASIDE

Postfields unit test method to focus on testing the postFields". We agree with his remark. If the suggested changes do not have a semantic relation to the original test method, it should be moved to another test or a new one. We considered this advice in the subsequent pull-requests.

- **Usefulness of the result to refactoring the tests**: The developer also stated "*the result of the test amplification makes me evaluate the existing unit tests and refactor them to improve the test coverage and test factorization*". This shows that even if the immediate results of test amplification are not tidy enough, they still help refactor existing tests.

Pull-request details

In the following parts we describe the details on the pull-requests on each project.

PolyMath. We sent a pull-request[6] to this project containing the suggestion for adding a new test method in the test class `PMVectorTest`. The suggested test method is shown in Figure 2.2.

This test method is testing the call of the method `#householder` on two different vectors. Before this test, the method `#householder` was not covered in this test class.

The *method-call-adder* input amplifier adds calls to an existing method in the public interface of a class to the test to force the object under test in a new state. We merged

[6] https://github.com/PolyMathOrg/PolyMath/pull/178

```
73  + PMVectorTest >> testHouseholder [
74  +       | u w |
75  +       u := #(-1 0 1) asPMVector. "`x <= 0` when x = -1"
76  +       w := u householder.
77  +       self
78  +               assert: (w at: 1) equals: 1.7071067811865475;
79  +               assert: (w at: 2) asArray equals: #(1.0 -0.0 -0.4142135623730951).
80  +       u := #(1.00001 2.00007) asPMVector. "`x <= 0` when x = 1.00001"
81  +       w := u householder.
82  +       self
83  +               assert: (w at: 1) equals: 0.5527953485259909;
84  +               assert: (w at: 2) asArray equals: #(1.0 -1.6180158992689828).
85  + ]
```

Figure 2.2: A new test method submitted to POLYMATH

two of them in a new test method that execute two different branches in the test method (based on the condition `x ≤ 0`). The former vector (line 75 in Figure 2.2) forces the ifTrue branch and the latter vector (line 80 in Figure 2.2) forces ifFalse branch. Note that the comment text (line 75) is added manually to increase the readability of the test.

The original test method included two assertions to confirm the type of the returned value of the method (`self assert: w class equals: Array`). The developers asked us to omit these assertion statements. We changed the pull request accordingly and it was merged immediately.

Pharo-Launcher. We sent a pull-request[7] to this project containing the suggestion for adding a new test method in the test class `PhLImportImageCommandTest`. The suggested test method is shown in Figure 2.3.

```
38  + PhLImportImageCommandTest >> testImportNonExistingImage [
39  +       | command |
40  +       command := PhLImportImageCommand new.
41  +       command context: presenter.
42  +       presenter := presenter
43  +               requestAnswer: presenter fileSystem / 'tmp' /
    'does_not_exists.image'.
44  +       self should: [ command execute ] raise: FileDoesNotExistException
45  +
46  + ]
```

Figure 2.3: A new test method sent in a pull-request to the project Pharo-Launcher

This test is produced from the original test method of `testCanImportAnImage`

[7] `https://github.com/pharo-project/pharo-launcher/pull/500`

which verifies an image can be imported using a valid filename. SMALL-AMP applies a literal mutation on the file name (`'foo.image'` changed to `'fo.image'`) that results an invalid filename and consequently raising a `FileDoesNotExistException` error. While preparing the test for the pull-request, we modified the name of the test method and the file to be more meaningfull.

The pull-request was merged in the same day with this comment: "*Indeed, the test you are adding has a value. Good job SmallAmp*".

DataFrame. We sent a pull-request[8] to this project containing the suggestion for adding a new test method in the test class `DataFrameTest`. The suggested test method is shown in Figure 2.4.

```
2874  + DataFrameTest >> testRangeError [
2875  +       self should: [ df range ] raise: MessageNotUnderstood   "Instance of
      Character did not understand #Barcelona"
2876  + ]
```

Figure 2.4: A new test method sent in a pull-request to the project DataFrame

The variable `df` is an instance variable that has been initialized in the `#setUp` method. It includes a tabular data mixed from numbers and texts. The initial amplified test method was generated by adding the method `#range` as the first statement in one of the original test methods. We recognized the remaining statements as superfluous lines and removed all of them. We also added a comment including the exception description.

This test method makes it explicit that calling the method `#range` on a DataFrame object containing non-numerical columns throws an exception. With this new test it becomes an explicit part of the contract for DataFrame.

The pull-request was merged after a few weeks. A developer of the project commented: "*Small-amp seems to be a very valuable tool!*"

Bloc. We sent a pull-request[9] to this project containing the suggestion for adding a new assertion in an existing test method in the test class `BlKeyboardProcessorTest`. The suggested test method is shown in Figure 2.5.

By calling the state revealing method `#keystrokesAllowed`, the assertion verifies the correctness of the object state after an `#processKeyDown:` event. This test is the result of combining assertion-amplification with oracle-reduction. Normally, the assertions-amplification step generates lots of assertions, and the oracle-reduction module removes

[8] https://github.com/PolyMathOrg/DataFrame/pull/132
[9] https://github.com/feenkcom/Bloc/pull/7

```
29     BlKeyboardProcessorTest >> testProcessKeyDown [
30            | eventA |
31
32            eventA := BlKeyDownEvent new.
33            eventA key: BlKeyboardKey a.
34
35            processor processKeyDown: eventA.
36  +         self assert: processor keystrokesAllowed.
37            self assert: (processor buffer hasEvent: BlKeyboardKey a)
38     ]
```

Figure 2.5: A new assertion suggested in a pull-request to the project Bloc

all assertion statements that do not kill any mutant. So, the test code did not need any special preparation and we only need to provide a comment to explain the test method.

The pull-request was also merged after a few weeks with a positive comment.

GraphQL. We sent a pull-request[10] to this project containing the suggestion for adding a new test method in the test class `GQLSSchemaNodeTest`. The suggested test method is shown in Figure 2.6.

```
292  + GQLSSchemaNodeTest >> testDirectives [
293  +        schema := self
294  +                parseSchema:
295  +                        'type A {
296  +                                        id: InternalCount
297  +                                        isB: BooleanType
298  +                                        size: Int
299  +                                        idA: ID_A
300  +                                        values: [ Int ! ]
301  +                                        params (name: StringName, prom:
     FloatingPoint, key: String): [Int]
302  +                                }'.
303  +
304  +        self assert: schema directives class equals: Array.
305  +        self assert: schema directives size equals: 2.
306  +        self
307  +                assert: (schema directives at: 1) class
308  +                equals: GQLSDirectiveNode.
309  +        self assert: (schema directives at: 1) name equals: 'include'.
```

Figure 2.6: A new test method suggested in a pull-request to the project GraphQL

This test method verifies the return value of `directives` in an `schema` object. The re-

[10] `https://github.com/OBJECTSEMANTICS/GraphQL/pull/12`

turned value is generated in the method `GQLSSchemaNode >> initializeDefaultDirectives` and contains technical debt. This test method guards against future evolutions which may break assumptions made by clients. We selected a meaningful name for the test and wrote a comment text. We also added back some of the assertions removed by oracle-reduction step. The pull-request was merged after a few days.

Zinc. We sent a pull-request[11] to this project containing the suggestion for adding a new test method in the test class `ZnRequestTest`. The suggested test method is shown in Figure 2.7.

```
1  + tests
2  + testAcceptsEncodingGzip
3  +       | request |
4  +       request := ZnRequest new.
5  +       request setAcceptEncodingGzip.
6  +       self assert: request acceptsEncodingGzip
```

Figure 2.7: A new test method suggested in a pull-request to the project Zinc

This test method calls the method `#setAcceptEncodingGzip` on an `request` object. Then calls another method `#acceptsEncodingGzip` to verify the change. Both of these methods were not covered in this test class before this test method.

This method is built by the cooperation of different modules of SMALL-AMP: First, *method-call-adder* input amplifier adds a new method call. Then *assertion amplification* inserts a set of new assertions after the added method call. And finally, after the main amplification loop is finished, the oracle-reduction step rejects all superfluous assertion statements. This test method did not need much preparation and we only selected a meaningful name for it. The pull-request was merged in the same day.

DiscordSt. We sent a pull-request[12] to this project containing the suggestion for adding a new test method in the test class `DSEmbedImageTest`. The suggested test method is shown in Figure 2.8.

The method covers the method `#extent` which was not covered in the test class before. This test method did not need much preparation and we only selected a meaningful name for it. The pull-request was merged after a few days.

MaterialDesignLite. We sent a pull-request[13] to this project containing the suggestion for adding two new test methods in the test class `MDLCalendarTest`. The suggested

[11] https://github.com/svenvc/zinc/pull/58
[12] https://github.com/JurajKubelka/DiscordSt/pull/75
[13] https://github.com/DuneSt/MaterialDesignLite/pull/308

```
1  + tests
2  + testExtent
3  +       object := self newObjectToTest.
4  +       object width: 41.
5  +       object height: 42.
6  +       self assert: object extent equals: 41 @ 42
```

Figure 2.8: A new test method suggested in a pull-request to the project DiscordSt

test methods are shown in figure 2.9.

Both of test methods are similar and are created by adding a new method call to the test input. The tests are created for the *Calendar* widget and verify correctness of `#selectPreviousYears` and `#selectNextYears` methods. In these test methods, the oracle-reduction step removed most of the assertions and it only preserved the first assertion killing the mutant: `self assert: calendar yearsInterval fourth equals: 2006`. We replaced the assertions with more human readable assertions (asserting `first` and `last` of the interval). The pull-request was merged the day after.

```
104  + MDLCalendarTest >> testSelectNextYears [
105  +       calendar currentDate: (Date year: 2016 month: 4 day: 10).
106  +       calendar selectNextYears.
107  +       self assert: calendar yearsInterval first equals: 2021.
108  +       self assert: calendar yearsInterval last equals: 2029
109  + ]
120  + MDLCalendarTest >> testSelectPreviousYears [
121  +       calendar currentDate: (Date year: 2016 month: 4 day: 10).
122  +       calendar selectPreviousYears.
123  +       self assert: calendar yearsInterval first equals: 2003.
124  +       self assert: calendar yearsInterval last equals: 2011
125  + ]
```

Figure 2.9: Test methods sent in a pull-request to the project MaterialDesignLite

PetitParser2. We sent a pull-request[14] to this project containing the suggestion for adding a new test method in the test class `PP2NoopVisitorTest`. The suggested test method is shown in Figure 2.10.

The test method tests the value of `currentContext` in `result` object. This test method resulted from assertion amplification combined with oracle-reduction. The test had two assertions: `self deny: visitor isRoot` and `self assert: visitor currentContext class equals: PP2NoopContext`. We added back some of removed assertions relating to `currentContext` and also removed the `self deny: visitor`

[14] https://github.com/kursjan/petitparser2

isRoot to make the test more focused. The pull-request is not merged up to the date of writing (November 21, 2022).

```
20  + PP2NoopVisitorTest >> testCurrentContext [
21  +       parser := $a asPParser.
22  +       result := visitor visit: parser.
23  +       self assert: result currentContext class equals: PP2NoopContext.
24  +       self assert: result currentContext properties isNil.
25  +       self assert: result currentContext node isNil.
26  +       self assert: result currentContext propertiesCopy isNil
27  + ]
```

Figure 2.10: A test method sent in a pull-request to the project PetitParser2

OpenPonk. We sent a pull-request[15] to this project containing the suggestion for adding a set of new lines in an existing test method in the test class OPDiagramTest. The suggested test method is shown in Figure 2.11.

```
101     OPDiagramTest >> testModel [
102         | model project |
103         model := OPTestContainerModel new.
104 +       view := OPDiagram new.
105 +       self assert: view modelType equals: 'UndefinedObject'.
106 +       self assert: view model isNil.
107 +       self assert: view modelName equals: 'UndefinedObject'.
108 +       view model: model.
109 +       self assert: view modelType equals: 'OPTestContainerModel'.
110 +       self assert: view model class equals: OPTestContainerModel.
111 +       self assert: view model name equals: 'container'.
112 +       self assert: view modelName equals: 'container'.
113         self assert: view model equals: model
114     ]
```

Figure 2.11: Changes on an existing test method - OpenPonk

The original test method is presented in Listing 2.14.

```
1 OPDiagramTest >> testModel [
2     | model project |
3     model := OPTestContainerModel new.
4     view := OPDiagram new model: model.
5     self assert: view model equals: model
6 ]
```

Code Excerpt 2.14: Original test method - OpenPonk

[15] https://github.com/OpenPonk/openponk

SMALL-AMP has broken the statement at line 4 in Listing 2.14 (the result is visible in lines 104 and 108 in Figure 2.11) and then added a series of assertions. Since this test is dedicated to test `model`, we kept all assertions reflecting the state of `model` and removed other assertions. So, the assertions in lines 104 to 107 verify the state of a freshly initialized `OPDiagram` object (where model is `nil`), and the assertions in lines 109 to 112 verify the public API through the accessor methods. The pull-request is not merged up to the date of writing this chapter.

Telescope. We sent a pull-request[16] to this project containing the suggestion for adding a new test method in the test class `TLNodeCreationStrategyTest`. The suggested test method is shown in Figure 2.12.

```
32  + TLNodeCreationStrategyTest >> testCopyAsSimpleStrategy [
33  +       | aTLNodeCreationStrategy |
34  +       aTLNodeCreationStrategy := strategy copyAsSimpleStrategy.
35  +       self
36  +               assert: aTLNodeCreationStrategy class
37  +               equals: TLNodeCreationStrategy.
38  +       self assert: aTLNodeCreationStrategy childrenStrategy isNil.
39  +       self assert: aTLNodeCreationStrategy compositeProperty isNil.
40  +       self assert: aTLNodeCreationStrategy childrenSortingStrategy isNil.
41  +       self assert: aTLNodeCreationStrategy compositeChildrenLayout isNil.
42  +       self assert: aTLNodeCreationStrategy nodeStyle isNil
43  + ]
```

Figure 2.12: A test method suggested in a pull-request to the project Telescope

The test method verifies the state of the returned object from calling `copyAsSimpleStrategy`. This method is never covered in the test class. It also contain technical debt. The call to `copyAsSimpleStrategy` is added by method-call-addition amplifier and the state of the returned value is asserted via assertion-amplification. We kept all assertions related to the returned value, and removed all other superfluous lines to make the test more readable. The pull-request is not merged up to the date of writing this chapter.

> ***Answer to RQ1:*** *We submitted 11 pull requests through GitHub to propose amplified test methods to developers. In 8 cases, our request was accepted by the developers and the test has been merged to the code base. In the three remaining cases our pull request was ignored. Moreover, we received qualitative feedback from developers acknowledging the relevance of amplified test methods.*

[16] `https://github.com/TelescopeSt/Telescope`

2.5.3 RQ2 --- Focus

We use the results in Table 2.4 for answering the next research questions. These tables present the result of test amplification on all selected classes selected in our dataset. In Table 2.4, the first 104 rows represent test amplification for test classes with high mutation coverage, while the remaining rows show the test classes with poor mutation coverage. SMALL-AMP algorithm (Algorithm 1) has a stochastic nature, especially test input reduction (Section 2.4.2), which heavily depends on randomness. Therefore, we ran the algorithm three times on each test class to observe the effect of randomness on the results. In addition, we ran the algorithm another time by disabling the profiling and the type sensitive operator for investigating the effectiveness of type profiler (denoted by $\circ\circ\circ$).

The columns in this table indicate:

- *Id*: Used as a reference for the row in the table.
- *Class*: The name of the test class to be amplified.
- # *Test methods original*: The number of test method in the test class before test amplification.
- # *loc CUT*: The number of lines in the class under test.
- % *Mut. score*: The mutation score (percentage) of the test class before test amplification.
- # *New test methods*: The absolute number of newly generated test methods after test amplification.
- # *Focused methods*: The absolute number of focused methods in the generated test methods.
- # *Killed mutants original*: The absolute number of killed mutants by the test class before test amplification.
- # *Newly killed amplified*: The absolute number of newly killed mutants by the test class after test amplification.
- % *Increase killed amplified*: The increase (in percentage) of killed mutants by the test class after test amplification.
- # *Newly mutant A-amp*: The absolute number of newly killed mutants only by Assertion amplification ($N_{iteration} = 0$ in Algorithm 1).
- % *Increase killed only A-amp*: The relative increase (in percentage) of killed mutants only by assertion amplification.
- # *Newly killed type aided*: The absolute number of newly killed mutants by type sensitive input amplifiers.
- % *Increase killed type aided*: The relative increase (in percentage) of increase killed mutants by type sensitive input amplifiers.
- *Time*: The duration of test amplification process in the hours-minutes-seconds (h:m:s)

Table 2.4: The result of test amplification by SMALL-AMP on the 52 test classes. (Tests with high coverage)

Id	Class	# Test methods original	# loc CUT	% Mut. score	# New test methods	# Focused methods	# Killed mutants original	# Newly killed amplified	% Increase killed amplified	# Newly mutant A-amp	% Increase killed only A-amp	# Newly killed type aided	% Increase killed type aided	Time (h:m:s)
1	DataFrameJsonWriterTest	4	51	50	1	1	6	2	33.33	1	16.67	0	0.00	0:00:11
2					1	1	6	2	33.33	1	16.67	0	0.00	0:00:11
3					1	1	6	2	33.33	1	16.67	0	0.00	0:00:11
4	○○○				*1*	*1*	*6*	*2*	*33.33*	*1*	*16.67*	*0*	*0.00*	*0:00:09*
5	DataFrameCsvReaderTest	4	62	85	0	0	17	0	0.00	0	0	0	0.00	0:00:08
6					0	0	17	0	0.00	0	0	0	0.00	0:00:07
7					0	0	17	0	0.00	0	0	0	0.00	0:00:07
8	○○○				*0*	*0*	*17*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:07*
9	DataFrameJsonReaderTest	9	80	72.73	2	2	16	4	25.00	1	6.25	0	0.00	0:08:43
10					2	2	16	4	25.00	1	6.25	0	0.00	0:08:57
11					2	2	16	4	25.00	1	6.25	0	0.00	0:09:11
12	○○○				*2*	*2*	*16*	*4*	*25.00*	*1*	*6.25*	*0*	*0.00*	*0:05:14*
13	DataFrameTypeDetectorTest	15	279	94.12	0	0	64	0	0.00	0	0	0	0.00	0:02:02
14					0	0	64	0	0.00	0	0	0	0.00	0:02:06
15					0	0	64	0	0.00	0	0	0	0.00	0:02:06
16	○○○				*0*	*0*	*64*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:01:48*
17	ZnMessageBenchmarkTest	2	187	80	0	0	24	0	0.00	0	0	0	0.00	0:01:23
18					0	0	24	0	0.00	0	0	0	0.00	0:01:21
19					0	0	24	0	0.00	0	0	0	0.00	0:01:24
20	○○○				*0*	*0*	*24*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:01:05*
21	ZnEasyTest	10	65	66.67	0	0	10	0	0.00	0	0	0	0.00	0:46:25
22					0	0	10	0	0.00	0	0	0	0.00	0:44:20
23					0	0	10	0	0.00	0	0	0	0.00	0:46:35
24	○○○				*0*	*0*	*10*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:28:16*
25	GQLRequestGrammarTest	29	142	96.55	0	0	56	0	0.00	0	0	0	0.00	0:04:07
26					0	0	56	0	0.00	0	0	0	0.00	0:04:09
27					0	0	56	0	0.00	0	0	0	0.00	0:04:02
28	○○○				*0*	*0*	*56*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:04:10*
29	GQLSingleAnonimousQu...	22	56	52.27	2	2	23	2	8.70	0	0	0	0.00	0:05:21
30					2	2	23	2	8.70	0	0	0	0.00	0:05:26
31					2	2	23	2	8.70	0	0	0	0.00	0:05:15
32	○○○				*1*	*1*	*23*	*1*	*4.35*	*0*	*0*	*0*	*0.00*	*0:06:02*
33	GQLSchemaGrammarTest	57	40	93.2	7	7	96	7	7.29	0	0	6	6.25	2:16:15
34					7	7	96	7	7.29	0	0	6	6.25	2:22:55
35					7	7	96	7	7.29	0	0	6	6.25	2:34:02
36	○○○				*0*	*0*	*96*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:11:01*
37	WebGrammarTest	14	28	90	2	2	18	2	11.11	0	0	2	11.11	0:46:51
38					2	2	18	2	11.11	0	0	2	11.11	0:43:21
39					2	2	18	2	11.11	0	0	2	11.11	0:42:40
40	○○○				*0*	*0*	*18*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:16:10*
41	PP2BufferStreamTest	16	188	83.87	0	0	78	0	0.00	0	0	0	0.00	0:00:34
42					0	0	78	0	0.00	0	0	0	0.00	0:00:35
43					0	0	78	0	0.00	0	0	0	0.00	0:00:36
44	○○○				*0*	*0*	*78*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:31*

Table 2.4 (cont.): The result of test amplification by SMALL-AMP on the 52 test classes. (Tests with high coverage)

Id	Class	# Test methods original	# loc CUT	% Mut. score	# New test methods	# Focused methods	# Killed mutants original	# Newly killed amplified	% Increase killed amplified	# Newly mutant A-amp	% Increase killed only A-amp	# Newly killed type aided	% Increase killed type aided	Time (h:m:s)
45	MDLPanelSwitcherBut . . .	6	100	53.85	0	0	7	0	0.00	0	0	0	0.00	0:00:46
46					0	0	7	0	0.00	0	0	0	0.00	0:00:46
47					0	0	7	0	0.00	0	0	0	0.00	0:00:47
48	◦◦◦				*0*	*0*	*7*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:47*
49	MDLPaginationCompo . . .	11	199	71.19	0	0	42	0	0.00	0	0	0	0.00	0:00:27
50					0	0	42	0	0.00	0	0	0	0.00	0:00:27
51					0	0	42	0	0.00	0	0	0	0.00	0:00:26
52	◦◦◦				*0*	*0*	*42*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:27*
53	RSUMLClassBuilderTest	2	23	50	1	1	1	1	100.00	1	100	0	0.00	0:05:00
54					1	1	1	1	100.00	1	100	0	0.00	0:05:34
55					1	1	1	1	100.00	1	100	0	0.00	0:05:23
56	◦◦◦				*1*	*1*	*1*	*1*	*100.00*	*1*	*100*	*0*	*0.00*	*0:03:30*
57	RSLabelGeneratorTest	2	353	82.38	0	0	173	0	0.00	0	0	0	0.00	0:11:19
58					0	0	173	0	0.00	0	0	0	0.00	0:11:07
59					0	0	173	0	0.00	0	0	0	0.00	0:12:21
60	◦◦◦				*0*	*0*	*173*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:12:55*
61	OPNullSerializerTest	3	251	50	0	0	1	0	0.00	0	0	0	0.00	0:00:00
62					0	0	1	0	0.00	0	0	0	0.00	0:00:00
63					0	0	1	0	0.00	0	0	0	0.00	0:00:00
64	◦◦◦				*0*	*0*	*1*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:00*
65	OPRTElementsConstra . . .	2	42	50	1	1	1	1	100.00	1	100	0	0.00	0:00:03
66					1	1	1	1	100.00	1	100	0	0.00	0:00:03
67					1	1	1	1	100.00	1	100	0	0.00	0:00:03
68	◦◦◦				*1*	*1*	*1*	*1*	*100.00*	*1*	*100*	*0*	*0.00*	*0:00:03*
69	PMBernoulliGenerator . . .	4	113	76.47	2	2	13	3	23.08	0	0	0	0.00	0:00:06
70					3	3	13	2	15.38	0	0	0	0.00	0:00:07
71					2	2	13	2	15.38	0	0	0	0.00	0:00:06
72	◦◦◦				*3*	*3*	*13*	*3*	*23.08*	*0*	*0*	*0*	*0.00*	*0:00:05*
73	PMBinomialGenerator . . .	3	164	83.33	1	1	10	1	10.00	1	10	0	0.00	0:01:01
74					1	1	10	1	10.00	1	10	0	0.00	0:01:14
75					1	1	10	1	10.00	1	10	0	0.00	0:01:18
76	◦◦◦				*1*	*1*	*10*	*1*	*10.00*	*1*	*10*	*0*	*0.00*	*0:01:31*
77	TLHideActionTest	3	468	55.56	2	2	5	0	0.00	0	0	0	0.00	0:00:30
78					3	3	5	3	60.00	0	0	1	20.00	0:00:33
79					1	1	5	2	40.00	0	0	0	0.00	0:00:26
80	◦◦◦				*1*	*1*	*5*	*2*	*40.00*	*0*	*0*	*0*	*0.00*	*0:00:14*
81	TLExpandCollapseNod . . .	3	8	66.67	2	2	14	2	14.29	1	7.14	0	0.00	0:01:28
82					2	2	14	2	14.29	0	0	0	0.00	0:00:58
83					2	2	14	2	14.29	0	0	0	0.00	0:01:16
84	◦◦◦				*1*	*1*	*14*	*1*	*7.14*	*0*	*0*	*0*	*0.00*	*0:00:31*
85	DSDetectChannelCom . . .	5	51	50	0	0	3	0	0.00	0	0	0	0.00	0:00:40
86					0	0	3	0	0.00	0	0	0	0.00	0:00:41
87					0	0	3	0	0.00	0	0	0	0.00	0:00:39
88	◦◦◦				*0*	*0*	*3*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:30*

Table 2.4 (cont.): The result of test amplification by SMALL-AMP on the 52 test classes. (Rows up to 104 are tests with high coverage, after 104 are tests with low coverage)

Id	Class	# Test methods original	# loc CUT	% Mut. score	# New test methods	# Focused methods	# Killed mutants original	# Newly killed amplified	% Increase killed amplified	# Newly mutant A-amp	% Increase killed only A-amp	# Newly killed type aided	% Increase killed type aided	Time (h:m:s)
89	DSUserTest	14	62	56.52	1	1	13	4	30.77	1	7.69	0	0.00	0:13:53
90					1	1	13	4	30.77	1	7.69	0	0.00	0:17:13
91					1	1	13	4	30.77	1	7.69	0	0.00	0:14:45
92	○○○				*1*	*1*	*13*	*4*	*30.77*	*1*	*7.69*	*0*	*0.00*	*0:10:15*
93	PhLAboutCommandTest	1	111	75	0	0	3	0	0.00	0	0	0	0.00	0:00:01
94					0	0	3	0	0.00	0	0	0	0.00	0:00:01
95					0	0	3	0	0.00	0	0	0	0.00	0:00:01
96	○○○				*0*	*0*	*3*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:01*
97	PhLCopyImageCommand…	1	25	66.67	0	0	4	0	0.00	0	0	0	0.00	0:00:02
98					0	0	4	0	0.00	0	0	0	0.00	0:00:02
99					0	0	4	0	0.00	0	0	0	0.00	0:00:01
100	○○○				*0*	*0*	*4*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:02*
101	WACookieTest	16	34	50	1	1	33	4	12.12	1	3.03	0	0.00	0:02:33
102					2	2	33	4	12.12	1	3.03	1	3.03	0:02:39
103					2	2	33	4	12.12	1	3.03	1	3.03	0:02:30
104	○○○				*1*	*1*	*33*	*4*	*12.12*	*1*	*3.03*	*0*	*0.00*	*0:01:45*
105	BlShortcutTest	1	113	20	1	1	2	4	200.00	1	50	0	0.00	0:00:04
106					1	1	2	4	200.00	1	50	0	0.00	0:00:04
107					1	1	2	4	200.00	1	50	0	0.00	0:00:04
108	○○○				*1*	*1*	*2*	*4*	*200.00*	*1*	*50*	*0*	*0.00*	*0:00:02*
109	BlCompulsoryCombinati…	4	271	11.76	1	1	4	2	50.00	0	0	1	25.00	0:00:34
110					1	1	4	2	50.00	0	0	1	25.00	0:00:33
111					1	1	4	2	50.00	0	0	1	25.00	0:00:36
112	○○○				*0*	*0*	*4*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:23*
113	BlLayoutExactResizerTest	8	36	40.54	3	3	15	3	20.00	1	6.67	2	13.33	0:01:08
114					3	3	15	3	20.00	1	6.67	2	13.33	0:01:07
115					3	3	15	3	20.00	1	6.67	2	13.33	0:01:06
116	○○○				*2*	*2*	*15*	*2*	*13.33*	*2*	*13.33*	*0*	*0.00*	*0:00:54*
117	BlInsetsTest	10	30	44.14	14	12	64	49	76.56	2	3.12	3	4.69	0:03:10
118					18	14	64	46	71.88	2	3.12	11	17.19	0:03:08
119					18	18	64	50	78.13	2	3.12	4	6.25	0:03:13
120	○○○				*19*	*19*	*64*	*50*	*78.13*	*2*	*3.12*	*0*	*0.00*	*0:02:01*
121	ZnBivalentWriteStreamTest	2	39	44.44	0	0	8	0	0.00	0	0	0	0.00	0:00:04
122					0	0	8	0	0.00	0	0	0	0.00	0:00:04
123					0	0	8	0	0.00	0	0	0	0.00	0:00:04
124	○○○				*0*	*0*	*8*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:04*
125	ZnStatusLineTest	7	220	28.57	1	1	16	1	6.25	0	0	1	6.25	0:00:41
126					1	1	16	1	6.25	0	0	1	6.25	0:00:43
127					1	1	16	1	6.25	0	0	1	6.25	0:00:44
128	○○○				*0*	*0*	*16*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:39*

Table 2.4 (cont.): The result of test amplification by SMALL-AMP on the 52 test classes. (Tests with low coverage)

Id	Class	# Test methods original	# loc CUT	% Mut. score	# New test methods	# Focused methods	# Killed mutants original	# Newly killed amplified	% Increase killed amplified	# Newly mutant A-amp	% Increase killed only A-amp	# Newly killed type aided	% Increase killed type aided	Time (h:m:s)
129	GQLArgumentsTest	16	83	47.06	0	0	8	0	0.00	0	0	0	0.00	0:31:10
130					0	0	8	0	0.00	0	0	0	0.00	0:30:56
131					0	0	8	0	0.00	0	0	0	0.00	0:32:28
132	◦◦◦				*0*	*0*	*8*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:27:17*
133	PP2ParsingGuardTest	3	29	47.06	1	1	8	2	25.00	1	12.5	0	0.00	0:00:08
134					1	1	8	2	25.00	1	12.5	0	0.00	0:00:07
135					1	1	8	2	25.00	1	12.5	0	0.00	0:00:07
136	◦◦◦				*1*	*1*	*8*	*2*	*25.00*	*1*	*12.5*	*0*	*0.00*	*0:00:07*
137	PP2BenchmarkTest	3	24	15.91	1	1	7	6	85.71	1	14.29	0	0.00	0:09:15
138					1	1	7	6	85.71	1	14.29	0	0.00	0:08:48
139					1	1	7	6	85.71	1	14.29	0	0.00	0:08:31
140	◦◦◦				*1*	*1*	*7*	*6*	*85.71*	*1*	*14.29*	*0*	*0.00*	*0:08:51*
141	MDLDialogTest	4	103	14.29	0	0	1	0	0.00	0	0	0	0.00	0:00:00
142					0	0	1	0	0.00	0	0	0	0.00	0:00:00
143					0	0	1	0	0.00	0	0	0	0.00	0:00:00
144	◦◦◦				*0*	*0*	*1*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:00*
145	MDLNestedListTest	11	558	41.04	1	1	55	8	14.55	1	1.82	0	0.00	0:00:14
146					1	1	55	8	14.55	1	1.82	0	0.00	0:00:15
147					1	1	55	8	14.55	1	1.82	0	0.00	0:00:15
148	◦◦◦				*1*	*1*	*55*	*8*	*14.55*	*1*	*1.82*	*0*	*0.00*	*0:00:14*
149	RSAthensRendererTest	1	19	9.88	1	1	48	1	2.08	1	2.08	0	0.00	0:01:08
150					1	1	48	1	2.08	1	2.08	0	0.00	0:01:10
151					1	1	48	1	2.08	1	2.08	0	0.00	0:01:15
152	◦◦◦				*1*	*1*	*48*	*1*	*2.08*	*1*	*2.08*	*0*	*0.00*	*0:01:04*
153	RSDraggableCanvasTest	6	84	35.71	0	0	10	0	0.00	0	0	0	0.00	0:00:34
154					0	0	10	0	0.00	0	0	0	0.00	0:00:35
155					0	0	10	0	0.00	0	0	0	0.00	0:00:36
156	◦◦◦				*0*	*0*	*10*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:32*
157	OPNavigatorAdaptersTest	3	161	24	0	0	6	0	0.00	0	0	0	0.00	0:00:26
158					0	0	6	0	0.00	0	0	0	0.00	0:00:24
159					0	0	6	0	0.00	0	0	0	0.00	0:00:23
160	◦◦◦				*0*	*0*	*6*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:26*
161	OPProjectTest	2	40	28	0	0	7	0	0.00	0	0	0	0.00	0:00:03
162					0	0	7	0	0.00	0	0	0	0.00	0:00:03
163					0	0	7	0	0.00	0	0	0	0.00	0:00:03
164	◦◦◦				*0*	*0*	*7*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:04*
165	PMExponentialDistrib . . .	1	25	17.07	3	3	7	3	42.86	0	0	2	28.57	0:00:23
166					2	2	7	2	28.57	0	0	1	14.29	0:00:15
167					3	3	7	3	42.86	0	0	2	28.57	0:00:11
168	◦◦◦				*1*	*1*	*7*	*1*	*14.29*	*0*	*0*	*0*	*0.00*	*0:00:05*
169	PMFixpointTest	6	18	39.47	5	5	30	12	40.00	3	10	0	0.00	0:03:01
170					4	4	30	9	30.00	3	10	0	0.00	0:03:03
171					5	5	30	12	40.00	3	10	0	0.00	0:04:35
172	◦◦◦				*7*	*7*	*30*	*8*	*26.67*	*3*	*10*	*0*	*0.00*	*0:02:11*

Table 2.4 (cont.): The result of test amplification by SMALL-AMP on the 52 test classes. (Tests with low coverage)

Id	Class	# Test methods original	# loc CUT	% Mut. score	# New test methods	# Focused methods	# Killed mutants original	# Newly killed amplified	% Increase killed amplified	# Newly mutant A-amp	% Increase killed only A-amp	# Newly killed type aided	% Increase killed type aided	Time (h:m:s)
173	TLDistributionMapTest	1	168	37.04	3	3	10	4	40.00	1	10	2	20.00	0:00:26
174					3	3	10	4	40.00	1	10	2	20.00	0:00:26
175					2	2	10	3	30.00	1	10	1	10.00	0:00:28
176	∘∘∘				*1*	*1*	*10*	*2*	*20.00*	*1*	*10*	*0*	*0.00*	*0:00:14*
177	TLLegendTest	2	96	0	0	0	0	0	-	0	-	0	-	0:00:02
178					0	0	0	0	-	0	-	0	-	0:00:02
179					0	0	0	0	-	0	-	0	-	0:00:02
180	∘∘∘				*0*	*0*	*0*	*0*	-	*0*	-	*0*	-	*0:00:02*
181	DSSendUserTextMess...	5	62	33.33	2	2	4	3	75.00	1	25	0	0.00	0:00:34
182					2	2	4	3	75.00	1	25	0	0.00	0:00:36
183					2	2	4	3	75.00	1	25	0	0.00	0:00:34
184	∘∘∘				*2*	*2*	*4*	*3*	*75.00*	*2*	*50*	*0*	*0.00*	*0:00:21*
185	DSEmbedTest	11	33	19.67	7	7	12	20	166.67	1	8.33	6	50.00	0:01:58
186					7	7	12	20	166.67	1	8.33	6	50.00	0:02:02
187					8	8	12	20	166.67	1	8.33	6	50.00	0:02:16
188	∘∘∘				*2*	*2*	*12*	*12*	*100.00*	*2*	*16.67*	*0*	*0.00*	*0:01:15*
189	PhLLocalTemplateTest	11	81	12	1	1	3	2	66.67	1	33.33	0	0.00	0:00:33
190					1	1	3	2	66.67	1	33.33	0	0.00	0:00:32
191					1	1	3	2	66.67	1	33.33	0	0.00	0:00:33
192	∘∘∘				*1*	*1*	*3*	*2*	*66.67*	*1*	*33.33*	*0*	*0.00*	*0:00:39*
193	PhLDirectoryBasedIm...	14	156	39.8	1	1	39	3	7.69	1	2.56	0	0.00	0:01:21
194					1	1	39	3	7.69	1	2.56	0	0.00	0:01:17
195					1	1	39	3	7.69	1	2.56	0	0.00	0:01:19
196	∘∘∘				*1*	*1*	*39*	*3*	*7.69*	*1*	*2.56*	*0*	*0.00*	*0:01:30*
197	WAKeyGeneratorTest	1	51	33.33	0	0	1	0	0.00	0	0	0	0.00	0:00:12
198					0	0	1	0	0.00	0	0	0	0.00	0:00:09
199					0	0	1	0	0.00	0	0	0	0.00	0:00:09
200	∘∘∘				*0*	*0*	*1*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:10*
201	WAXmlCanvasTest	1	142	33.33	0	0	1	0	0.00	0	0	0	0.00	0:00:00
202					0	0	1	0	0.00	0	0	0	0.00	0:00:00
203					0	0	1	0	0.00	0	0	0	0.00	0:00:01
204	∘∘∘				*0*	*0*	*1*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:00:01*
205	WAErrorHandlerTest	11	182	38.46	0	0	5	0	0.00	0	0	0	0.00	0:01:15
206					0	0	5	0	0.00	0	0	0	0.00	0:01:13
207					0	0	5	0	0.00	0	0	0	0.00	0:01:17
208	∘∘∘				*0*	*0*	*5*	*0*	*0.00*	*0*	*0*	*0*	*0.00*	*0:01:19*

format.

RQ2: To what extent are improved test methods considered as focused?

For answering this research question, we use values in the column # *Focused methods*. We use the same definition for focused methods as DSPOT:

> *Focus is defined as where at least 50% of the newly killed mutants are located in a single method.* [9]

Generating focused tests is important because analysing a focused test is easier (most mutants reside in the same method under test) hence should take less review time from developers. For calculating this value, we use generated annotations by SMALL-AMP on the newly generated test methods which show the details of the killed mutants by the method.

We see that almost all amplified tests are focused. We see only in two cases (`BlInsetsTest`, #117 and #118) that some generated methods are not focused.

> ***Answer to RQ2:*** *We see at least one focused test method in all amplified cases.*

2.5.4 RQ3 --- Mutation Coverage

RQ3: To what extent do the amplified test classes kill more mutants than developer-written test classes?

In 86/156 cases (55.12%), SMALL-AMP successfully amplified an existing test class. The distribution of the number of killed mutants, and increase kill are presented in Figure 2.13 and Figure 2.14 for all test classes, highly covered as well as poorly covered ones. The number of newly killed mutants in these classes (column # *Newly killed amplified*) varies from 1 up to 50 mutants (case #119). In the executions amplifying test classes having high coverage, SMALL-AMP is able to amplify 38 out of 78 (48.7%), and for the test classes having poor coverage this number is 48 out of 78 (61.5%). Therefore, we see more amplification in the classes with poor coverage. The relative increase in mutation score (column % *Increase killed amplified*) varies from 2.08% (cases 149-151) up to 200% (cases 105-107). It is also observable regarding to these metrics that amplification on classes with poor coverage is more successful.

Surprisingly, despite running MUTALK with its all mutation operators, the mutation testing framework did not manage to create any mutant for the class `TLLegendTest` (cases 177-179). MUTALK mutation operators work statically and only a limited set of well-known transformations are provided in the tool.

Figure 2.13: The distributions of the number of killed mutants

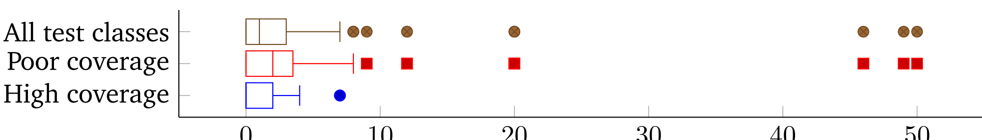
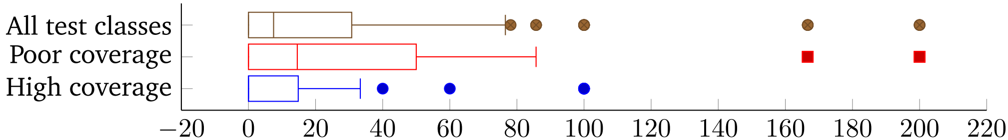


Figure 2.14: The distributions of the increase kills



The effect of randomness. In this section we report the effect of randomness on the results. Based on the Algorithm 1, the main randomness happens during the input-amplification (Line 10) and oracle reduction (Line 11) steps. Therefore, we expect to see the minimum difference in the results generated by assertion amplification (Line 5). The column 11 (# *Newly mutant A-amp*) shows the absolute number of killed mutants only by assertion amplification. These values are identical in executions for all classes except the case 81 (`TLExpandCollapseNodesActionTest`). The reason for this exception is that a specific mutant may be killed by input amplification in a test method, and if it is not killed, it will be killed by assertion amplification in the next test method. Based on the information presented in Table 2.4, regardless of time, the same result are achieved from different executions in 43 out of 52 test classes.

For a deeper investigation, we randomly select 5 test classes from the cases that are affected by randomness, and 5 test classes from the cases without an observable change. Then, we run SMALL-AMP on these classes 10 times (10 class $\times$ 10 times = 100 runs). Table 2.5 shows the results of this experiment. In the column with title *X10*, we report the number of newly killed mutants for each run in order, which is the most important metric for amplification. In the column *X3*, we also echo the values from column # *Newly killed amplified* in Table 2.4 to make the comparison easier. The next two columns compare the *Median* and *Average* values in these two columns. The first 5 rows in this table are the cases affected by randomness, and the next 5 rows are cases without an observable effect.

When we compare the values in columns *X10* with *X3*, we still do not see any visible effect from randomness in the rows 6 to 10. In the first 5 rows, we see the median values and average values in both experiments are similar. In three cases (rows 2, 3 and 4), the values of *X3* did not achieve the maximum number of killed mutants seen in *X10*. In one case (row 1), we see some runs lacking any improvements in *X10* while all of its runs

Table 2.5: The result of running SMALL-AMP on 10 test class for 10 times

	Test class	X10	X3	Median (X10, X3)	Average (X10, X3)
1	PMBernoulliGeneratorTest	3, 3, 0, 0, 2, 2, 2, 3, 2, 3	3, 2, 2	2.0, 2.0	**2.0, 2.33**
2	TLHideActionTest	0, 4, 1, 2, 2, 4, 2, 1, 2, 2	0, 3, 2	2.0, 2.0	**2.0, 1.67**
3	TLExpandCollapseNodes ...	2, 2, 2, 2, 2, 2, 3, 2, 2, 2	2, 2, 2	2.0, 2.0	**2.1, 2.0**
4	PMExponentialDistributi ...	3, 2, 1, 4, 3, 2, 2, 3, 3, 2	3, 2, 3	**2.5, 3.0**	**2.5, 2.67**
5	TLDistributionMapTest	3, 4, 3, 4, 4, 4, 4, 3, 3, 4	4, 4, 3	4.0, 4.0	**3.6, 3.67**
6	PMBinomialGeneratorTest	1, 1, 1, 1, 1, 1, 1, 1, 1, 1	1, 1, 1	1.0, 1.0	1.0, 1.0
7	DSSendUserTextMessage ...	3, 3, 3, 3, 3, 3, 3, 3, 3, 3	3, 3, 3	3.0, 3.0	3.0, 3.0
8	GQLSchemaGrammarTest	7, 7, 7, 7, 7, 7, 7, 7, 7, 7	7, 7, 7	7.0, 7.0	7.0, 7.0
9	BlCompulsoryCombinati ...	2, 2, 2, 2, 2, 2, 2, 2, 2, 2	2, 2, 2	2.0, 2.0	2.0, 2.0
10	ZnMessageBenchmarkTest	0, 0, 0, 0, 0, 0, 0, 0, 0, 0	0, 0, 0	0.0, 0.0	0.0, 0.0

in *X3* shows a successful amplification. To sum up, we see that the randomness has an effect on the result, but the impact is minimal and does not invalidate the findings. In addition, repeating the analysis 3 times is justifiable since running 10 times adds little extra information for a large increase in processing time.

> ***Answer to RQ3:*** Small-Amp *successfully amplified 86 test classes out of 156 cases (55.12%). Even for highly covered test classes,* Small-Amp *improved the mutation coverage in 38 out of 78 cases. In test classes with poor coverage, test amplification becomes even more effective:* Small-Amp *increased the mutation coverage in 48 out of 78 cases and the absolute and relative increase in mutation score was higher.*

2.5.5 RQ4 --- Amplification Steps

RQ4: What is the contribution of input amplification and assertion amplification (the main steps in the test amplification algorithm) to the effectiveness of the generated tests?

As we reported in Section 2.5.4, in 86/156 cases (55.12%), the improvements are achieved from input-amplification and assertion-amplification cooperation. In this research question, we study the results generated only by assertion-amplification, and also generated by the type sensitive operators.

Contribution of the assertion-amplification step. In this section, we filter all amplified test methods that are generated only by assertion amplification. In other words, we only account the improvements from all amplified test methods that are selected from the first assertion amplification (Line 5 in Algorithm 1). The column 11 (# *Newly mutant A-amp*) shows the absolute number of killed mutants only by assertion amplification; column 12 (*% Increase killed only A-amp*) also shows the relative increase. The reported results in Table 2.4 shows that in 61/156 cases (39.1%) at least 1 mutant is killed only by the assertion amplification step. Improvements in four classes (cases 53-55; 65-67; 73-75; 149-151) achieved only by adding new assertions.

Contribution of the type sensitive input amplifiers. Here, we filter all amplified test methods that in at least one of its transformations, a type sensitive input amplifier (in our case *method-call-adder*) is used. While type-sensitive operators benefit the information provided by dynamic profiler step (Section 2.4.1), the contribution of these operators is important because it can show the effectiveness of dynamic profiling.

Column 13 (# *Newly killed type aided*) shows the absolute number of newly killed mutants by the type sensitive input amplifiers. Column 14 (*% Increase killed type aided*) also shows the relative increase. We see that in 30/156 cases (19.2%) the type sensitive input amplifiers contribute to the result. Especially for 2 test classes (`WebGrammarTest` rows 37-39, and `ZnStatusLineTest` rows 125-127), SMALL-AMP was able to amplify the tests only by the type sensitive input operators.

To assess the impact of type profiling, we quantified the effect of the steps that rely on type profiling. We therefore extended the analysis with an additional evaluation step where we disabled the type profiler in the algorithm, as well as the type sensitive input amplifier (method addition amplifier) and ran the tool on all test classes. The results for this experiment are mentioned in the forth row for each test class in Table 2.4 (denoted by ○ ○ ○). We focus on cases in which type-sensitive input amplifiers improved the coverage in at least two of three runs (10 test classes, cases starting with 33, 37, 101, 109, 113, 117, 125, 165, 173, 185). In 8 cases we see that disabling the profiling and also the type sensitive operators decrease the improvements and only in two cases we see no difference (case 101) or a slight improvements (case 117).

Answer to RQ4: *Our study demonstrates that amplifying the tests only using assertion amplification is less efficient than in combination with input amplification. Moreover, the extra information generated by dynamic type profiling helps input amplification in killing more mutants.*

2.5.6 RQ5 --- Comparison

RQ5: How does Small-Amp **compare against** DSpot**?**

We summarize our results from quantitative and qualitative studies and the corresponding results from DSPOT reported in [9] in Table 2.6. An exact comparison is impossible because these studies have been conducted in two completely different ecosystems.

Id	Metric	Small-Amp			DSpot
		#1	#2	#3	
1	Projects	13	13	13	10
2	Test classes	52	52	52	40
3	Test methods	403	403	403	220
4*	New generated test methods	71	76	75	471
5	# Amplified test classes	28	29	29	26
6	% Amplified test classes	53%	55%	55%	65%
7*	# Total killed mutants by original	1102	1102	1102	7980
8*	# Total newly killed mutants	156	151	157	1617
9	% Total increase killed	14.15%	13.70%	14.24%	20.26%

Table 2.6: Summary of results in SMALL-AMP and DSPOT

SMALL-AMP is validated against 52 test classes. It has successfully amplified 28, 29 and 29 of them (≈55%), while DSPOT has been validated against 40 test classes of which 26 cases were improved (65%). The most notable differences between the results from SMALL-AMP and DSPOT are the number of mutants in two ecosystems and consequently the number of newly generated test methods (denoted by * in the Table 2.6). These differences can be attributed to the use of two different mutation testing frameworks in two different languages. SMALL-AMP uses MUTALK which has notably fewer mutation operators than the DSPOT counterpart PITEST. To reduce the effect of the mutation testing framework, we calculate the relative increase in killed mutants within the two ecosystem as follows:

$$\%Total.Inc.killed = 100 \times \frac{\#Total.Mutants.killed_{new}}{\#Total.Mutants.killed_{original}}$$

This value is shown in the row number 12 in the Table 2.6 for two experiments. It is 14.03% in total for SMALL-AMP, and 20.26% for DSPOT.

We have also submitted 11 pull-requests by using SMALL-AMP outputs and 8 of them were merged by developers (≈72%), while Danglot et al. submitted 19 pull requests derived from DSPOT output and 13 of them merged successfully (68%).

Finally, it is worth mentioning that SMALL-AMP is configured as $N_{maxInput} = 10$ which means the reduce step (Algorithm 1, line 11) select 10 test-input in each iteration.

This value for DSpot is not reported in their paper. Increasing this hyperparameter may improve the result, but it also may increase the execution time significantly.

> ***Answer to RQ5:*** *The results from* Small-Amp *and* DSpot *in two different ecosystems are similar.* Small-Amp *and* DSpot *have been successful in amplifying respectively 55% and 65% of their input test classes. We also see ≈72%and 68% merged pull requests in the tests derived from* Small-Amp *and* DSpot *outputs.*

2.5.7 RQ6 --- Time Costs

RQ6: What is the time cost of running Small-Amp, including its steps?

The time cost is important when studying test amplification tools' practicality. Based on Algorithm 1 – p. 24, we know that the complexity of this algorithm is:

$$O(t \times i \times a \times l_t \times m \times l_c)$$

Where

- t is the number of test methods to be amplified
- i is the number of iterations in the main loop
- a is the number of input amplification operators
- l_t is the number of lines in the test methods
- m is the number of mutation testing operators
- l_c is the number of lines in the program under test

Not surprisingly, in some cases, we see that the amplification process takes considerable time. In this research question, we report and compare the execution time of Small-Amp and the relative cost of each step.

Figure 2.15 and Figure 2.16 illustrate a series of box-plots derived from the recorded execution time during the experiments in Table 2.4. In these figures, `Init.` refers to all initializing steps, includes the dynamic profiling to collect type information (Section 2.4.1). Here we also calculate an initial mutation score for the original test suite. `IAmp` refers to the input amplification step. This step loops over all input amplification operators (Section 2.3.2) and afterwards reduces to $N_{maxInputs}$ of current inputs and discarding the rest (Section 2.4.2). `AAmp` represents the assertion amplification step (Section 2.3.3), while `Sel.` selects all test methods that increase the mutation score (Section 2.3.4). `Read.` concerns the post-processing steps to increase the readability of the generated tests, in

Figure 2.15: The distribution of absolute execution time (in seconds)

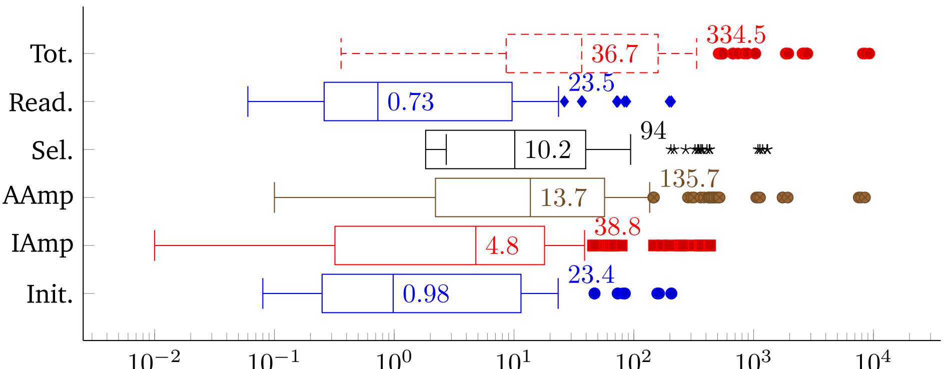
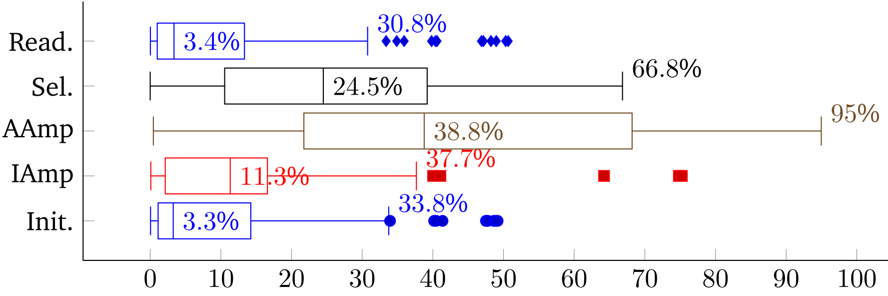


Figure 2.16: The relative distributions of the time-cost (percentage)



particular the oracle reduction (Section 2.4.3). Finally, `Tot.` shows the entire execution time for a class.

Figure 2.15 shows how the execution time is distributed for total execution time and also for each step in seconds. The horizontal axis presents the number of seconds in logarithmic scale. The diagram includes also the values for the median and the upper whisker.

Regarding total execution time (`Tot.`), half of the executions finished in less than 36.7 seconds (the median value). Furthermore, the diagram shows that the majority of amplification (upper whisker) finished in less than 334.4 seconds (5 minutes and 34 seconds). However, we see 25 outliers that refer to the instances that finished in more than 335 seconds. If we set a fixed time budget, for instance a 10 minutes budget for each class, the test amplification process for these classes will not terminate properly. This show the importance of considering time budget management in test amplification tools.

The median value for other steps are: initializing 1 second, input amplification 4.8 seconds, assertion amplification 13.7 seconds, selection by mutation testing 10.1 seconds

and post-processing steps 0.7 seconds.

Figure 2.16 illustrates the relative proportion of time test amplification dedicates to each step. So, the execution time for each step is divided to the total amplification time to compare the steps relatively.

The profiling step and the oracle reductions steps are the fastest steps. The median value for each of these steps are respectively 3.2% and 3.4%. Therefore, we can say profiling and oracle reduction steps do not add much time overhead to the overall process. Next, the input amplification step takes about 11.3%. A considerable portion of execution time is spent during assertion amplification (median 38.8%, upper whisker 95%). The median execution time related to selection step, in which mutation testing is ran, is 24.5%. We suspect the execution time for mutation testing would be more if MuTalk would generate more mutants.

> ***Answer to RQ6:*** *The majority of classes in our dataset has been amplified in less than 5 minutes and 34 seconds. However, in some cases the execution takes longer with a maximum of 2 hours and 34 minutes. Therefore, a time budget management mechanism is needed in the test amplification tools. In the execution time for each steps, we see that the profiling and oracle reductions steps (the extra steps compared to the original* DSpot *algorithm) do not add much overhead to the overall process.*

2.6 THREATS TO VALIDITY

As in any empirical research, we identify factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines provided by [47], we organise them into four categories.

Construct validity. Do we measure what was intended? For RQ1 (Pull Requests), we manually selected test methods which we considered valuable additions to the project. And we provided a motivation for the pull request based on a human interpretation of the extra mutants killed. Thus, the percentage of accepted pull requests is a flattered result. If we would have submitted all amplified test methods the results would be far lower. We consider this risk as minimal, because Small-Amp at the current stage should never be seen as a fully automated code synthesizer tool but rather as a recommender system supporting the human-in-the-loop.

For RQ2 to RQ4 we heavily rely on mutation coverage as a proxy for the corner cases the amplified tests are supposed to cover. There is an ongoing debate in the mutation testing community of whether mutation operators can serve as proxies for actual faults.

Today, there is no alternative so we settled with mutation coverage. But if ever another measure for test effectiveness comes along we need to revise the results.

Internal validity. Are there unknown factors which might affect the outcome of the analyses? For RQ1 (Pull Requests), we don't have any knowledge about the policy the projects had concerning pull requests submitted by outsiders. In the Pharo community, most developers know one another and are likely to trust contributions. However, for our study it was the first author who submitted the pull-requests and at that point in time he was a newcomer in the community. For the three pull requests which were ignored, we don't know whether this newcomer submission played a role.

External validity. To what extent is it possible to generalise the findings? We demonstrated that test amplification is feasible, even for dynamically typed language. We have constructed a proof-by-construction for the Pharo/Smalltalk ecosystem. However, we cannot make any claims regarding other dynamically typed languages (Python, Ruby, Javascript, ...). We are quite confident that type profiling is the key to make test amplification successful in such a context. However, coding conventions are equally important and this may jeopardise the kind of input amplification operators that work.

Reliability (a.k.a. Conclusion Validity). Is the result dependent on the tools? As mentioned earlier, we heavily rely on mutation coverage as calculated by MUTALK. MUTALK lacks several mutation operators compared to the PITEST tool used in the DSPOT paper. This implies that SMALL-AMP will generate fewer test cases and that the newly killed mutants will also be generally lower. We mitigated this threat by always reporting the absolute number combined with the relative increase (in percentage).

The other threat to conclusion validity is the impact of randomness. SMALL-AMP works based on applying random transformations on the tests. Most importantly, the actual amplified tests surviving the Input Reduction step (see Section 2.4.2 – p. 32) may vary from one run to another. We applied the tool on different classes in different projects from various domains, and achieved amplified tests in all circumstances. In addition to the variety in the projects, we ran the tool three times on each test class. Thus, the impact of randomness should be small at best.

2.7 RELATED WORK

Test amplification systems can vary based on the engineering goal. In addition to the amplification of the code coverage [32, 48] or mutation score [49, 50], researchers have used test amplification for other goals like fault detection [51, 52], oracle improvement

[40, 53], fault localization [54, 55], and incompatible environments detection [56].

A test amplification system may use search based techniques [31, 32, 49, 57] or symbolic and concolic execution techniques [48, 58, 59]. The results of the amplification can be added to existing test suite [32, 49] or just modifying the current tests [40]. They also may consider only new changes [57, 58] rather than working on the snapshot of the entire project.

Our work is pretty close to DSpot [9, 26] where Small-Amp is an adaption of DSpot into a dynamic language. This work, same as DSpot, also can be categorised under genetic improvement [60] where it takes advantage of existing test suite as well as an automated search algorithm in order to find an improved version of test code.

Brandt and Zaidman [61] use a lighter version of DSpot to increase the instruction coverage. They also provide an IDE plugin to make the developers interplay with the test amplification tool possible. Nijkamp et al. [62] and Oosterbroek et. al [63] address the readability of the amplified tests by choosing proper names and removing redundant statements.

Dynamically-typed languages. As we argued earlier, test case generation is well studied in statically typed languages [64, 65, 66] but there are only a few academic works that target dynamically typed languages. We list the ones we were aware of below.

Lukasczyk et al. [67] introduce automated unit test generation for Python, and the tool Pynguin which works based on techniques used in statically typed languages more specifically EvoSuite [65] and Randoop [64]. Pynguin circumvents the lack of type information by assuming that the system under test contains type information added by developers in terms of type annotations.

Mirshokraie et al. [68] created a tool names JSEFT, which generate unit tests for javascript functions and events by a record and replay technique. JSEFT relies on web crawling to collect traces from javascript executions. They create test methods by replaying the executions and adding new oracles using a mutation-based process [69].

Wibowo et al. [70] use a genetic algorithm to generate unit test code for the Lua scripting language. The algorithm starts from a random initialized population and then evolve by crossover and mutation operators. The tool only generates assertions for primitive data type values. Small-Amp on the other hand used recursive assertion generation to deal with non-primitive types.

Mairhofer et al. [71] introduce RuTeG, a test generation for Ruby based on genetic algorithms. For each method under test, RuTeG statically processes the parse tree and collects arguments and the list of methods invoked on each argument. Then they use predefined and customized data generators to generate a part of code that is valid based

on data collected from the parse tree. RuTeG does not improve an existing test suite, but rather generates the whole test suite itself.

Mutation testing. Using mutation testing as an actionable target for strengthening an existing test suite is used at large scale at Google [72, 73] and Facebook [74]. These papers are close in spirit to DSpot and Small-Amp as both create new test casess to increase mutation coverage. However, these works use professional developers to generate new tests (manual test amplification) while DSpot and Small-Amp illustrate that a recommender system is feasible.

2.8 FUTURE WORK

In this section, we present some open problems and the ways how Small-Amp and test amplification tools can be improved in the future.

Test amplification ecosystem. In the current implementation of Small-Amp, we run the tool on the whole project. This way of using the tool has some drawbacks: since developers should run it manually, they need a knowledge about the concept, the process and also the tool interface; it may take a long time to amplify all classes in the project; the tool will reamplify some parts of the project on each execution, which will increase the execution time; and finally, developers need to deal with the output manually, they need to understand it, polish the interesting tests, and merge them manually in the code base. Imposing such extra work on developers is likely to make them loose their interest in using the tool often.

In the future, we will integrate the Small-Amp with a build system, for instance GitHub Actions. The build systems will run the tool automatically on the specified events such as on each push, or pull-request or periodically. Additionally, Small-Amp will amplify only the recent changes on each run. It means that only the mutants in the changed parts will be generated which will reduce the cost of amplification significantly. In this case, the amplification will be run in the project level instead of running class-by-class. So, finding an exact relation between the production classes and the test classes will also lost its importance: all test methods covering a changed part can be included in the amplification.

Furthermore, we will build a web-based test editor dashboard to visualize the test amplification outputs, and also a GitHub-Bot to synchronize the Build system output, code base and the dashboard. The developer can use this dashboard to assess the outputs, and also customize the amplified test. The tests after the polishment will be reevaluated automatically, and if it is green, it can be pushed to the code base. So, developers don't

need to overwhelm themselves with tedious tasks and can benefit from test amplification in an ecosystem automated by bots.

Extended use-cases for dynamic profiling. Dynamic profiling is more than merely a type inference solution. It can be generalized to collect various information about unit-under-test based on dynamically running existing test suite. In some cases, statically typed languages also can benefit the profiling mechanism. We enumerate two of these use-cases:

— Pure methods detection: An impure method is a method that looks like an accessor but calling it causes a change in the state of the object [75]. In the scope of SMALL-AMP, identifying pure/impure methods is important during oracle reduction. A new profiler can be implemented using the method proxies to serialize the object state before and after method invocations. If the state is changed, we can infer that the method is impure.

— Providing information for advanced input amplifiers: In this work, we proposed a basic algorithm for test-input reduction (Section 2.4.2). By addressing the test-input explosion problem, test amplification tools can benefit from wider range of input amplifiers. Advanced input-amplifiers can exploit the profiling step to collect useful information dynamically and increase their knowledge about the program under test. For example, a profiler can collect all literal values from the covered methods and use them in literal values amplification operator. Another example can be object transplantation between test methods. A profiler can collect patterns of how objects are created and manipulated and this information can be used in an input-amplifier.

Test method models, best practices and structured strings. Unit tests in object-oriented languages ideally are structured as a sequence of statements that instantiate an object, manipulates its state, and asserts expected values. However, not every test fits this model in real projects. Developers use helper methods, customized assertions, structured strings, some optimizations like grouping the tests or parallelizing them. For instance, a developer may write tests in XML files and load each file in a test method, so these tests heavily depend on parsing structured strings.

If a test does not fit with the ideal test model, the current algorithm of test amplification still can be used, but it may be less successful in producing strong results. We leave identifying and adopting best practices of test methods as an open problem. Additionally, mutating a structured string by understanding its syntax can also be interesting future work.

Using patterns to guide test amplification. As an important future work, we suggest using heuristics to guide the amplification algorithm. Large scale manual test amplification histories like the work at Google [72, 73] can be analyzed to answer questions like: How often do developers write new test methods? Are new test methods similar to an existing test? If they update an existing test method, what is the relation of the updated test method and the mutant to be killed? What transformations are applied to the test?

Answering these questions leads us to find some patterns in how real developers kill mutants. These patterns can help the tool to prioritize some test methods and input amplifiers for killing a particular mutant. Recent advances in deep learning or other program synthesis techniques are promising and can be helpful in making test amplification tools more intelligent [76].

Reducing the mutation testing burden. Test amplification generates tests to optimize mutation coverage, however calculating the mutation score is a time-consuming process. During test amplification, this mutation score is calculated multiple times for each test method: in lines 6, and 14 of Algorithm 1 – p. 24. However, for a test to kill a mutant, it first must reach the statement, then infect the program state, propagate to the output where it must be revealed by an assertion [77, 78].

We can optimize the calculation of the mutation score by using a hierarchical coverage measurement. For example, we can first run a code coverage tool, then we can run an extreme transformation [79], afterwards we only mutate the covered parts.

Another technique for reducing the mutation testing burden is to use mutation operators that are learned from known common bugs like MutationMonkey [74].

Using multiple type-inference mechanisms. The main drawback of using an existing test suite for dynamic profiling is if a method is not covered in the test, we can not collect its type information, hence can not add calls to such methods during input amplification. Static type inference [80] or live typing [81] techniques can be helpful to empower Small-Amp to generate method calls to such uncovered methods.

Involving readability metrics. Based on a previous study [82], the most important aspects for developers in assessing the quality of a test suite are readability and maintainability. Although the code coverage metrics are necessary for identifying the poor test suites, they are limited in distinguishing high-quality tests based on how practitioners perceive the test quality. Since the goal of test amplification is to recommend new test cases ready to be merged into the code base, considering readability and maintainability metrics in the test amplification appears to be a critical next step.

2.9 CONCLUSION

In this chapter, we introduce Small-Amp, an approach for test amplification in the dynamically typed language Pharo/Smalltalk. The main algorithm of Small-Amp is adapted from DSpot, a test amplification technique designed for Java programs. In order to mitigate the lack of type information, we exploit profiling information, readily available by running the test suite. We demonstrate that test amplification is feasible for dynamically typed languages, by replicating the experimental set-up of DSpot, including a qualitative and quantitative analysis of the improved test suite.

In our qualitative analysis, we submitted pull-requests of an amplified test by Small-Amp to the GitHub repositories of the projects in our dataset. From 11 pull-requests we submitted, 8 were merged (≈72%). The developers' comments on the pull-requests illustrate how valuable they perceive the new tests created by Small-Amp. The results from our quantitative study show that Small-Amp succeeds to amplify 28 test classes out of 52, approximately 53% of target classes, in 13 projects from our dataset. The majority of the generated tests are focused, and all test amplification steps (including type profiling step) play a critical role. The results from Small-Amp and the results from DSpot are comparable. We see ≈72% merged pull-requests and 53% successfully amplified test classes in Small-Amp, while for DSpot these values are 68% and 65%. We also see that the value of total increase killed between two tools in two different ecosystems are similar (14% in Small-Amp and 20% in DSpot).

In conclusion, the results of experiments show that by using a profiling step and collecting type information, we can successfully adopt a test amplification approach in a dynamically typed language.

Chapter 3

AmPyfier: Test Amplification in Python

This chapter is a revised version of an originally published paper in the *Journal of Software: Evolution and Process. Special Issue: Automatic Software Testing from the Trenches* (*JSEP*):

AmPyfier: Test Amplification in Python

Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer
In *Journal of Software: Evolution and Process. Special Issue: Automatic Software Testing from the Trenches* (*JSEP*). June, 2022.
URL: https://doi.org/10.1002/smr.2490.

ABSTRACT

Test amplification aims to automatically improve a test suite. One technique generates new test methods through transformations of the original tests. These test amplification tools heavily rely on analysis techniques which benefit a lot from type declarations present in the source code of projects written with statically typed languages. In dynamically typed languages, such type declarations are not available, and therefor, research regarding test amplification for those languages is sparse. Recent work has brought test amplification to the dynamically typed language Pharo Smalltalk by introducing the concept of dynamic type profiling. The technique is dependent on Pharo-specific frameworks and has not yet been generalised to other languages. Another significant downside in test amplification tools based on the mutation score of a test suite is their high time-cost. In this chapter, we present AmPyfier, a tool that brings test amplification and type profiling to the dynamically typed language Python. AmPyfier introduces multi-metric selection in order to increase the time efficiency of test amplification. We evaluated AmPyfier on 11 open-source projects and found that AmPyfier could strengthen 37 out of 54 test classes. Multi-metric selection decreased the time-cost by 17% to 98% as opposed to selection based on full mutation score.

3.1 INTRODUCTION

As proven over and over again, faults are often more expensive to fix than to prevent. In order to prevent regression and detect bugs, an adequate test suite is necessary. To determine the program-based adequacy of a test suite, multiple criteria exist such as, but not limited to, code coverage (method/brach/statement) [83] or the mutation score [36]. The threshold to decide when a test-suite is adequate enough poses a difficult question. As such, striving for a test suite as adequate as possible is necessary, in order to make the test suite as robust. Manually writing a completely adequate test suite is a labour and time-consuming task. This is where research concerning *test amplification* and generation proves to be useful [77, 84].

Test amplification tries to harden a test suite against a given adequacy criterion, criteria, or engineering goal based on the already existing test suite for the project under test [8]. One category of test amplification is closely related to test generation, they both try to automatically improve a test suite, or generate one based on given criteria. Their goals are similar, but the difference between amplification and generation is in the main input it takes. While *test generation* focuses only on the program under test, test amplification generates tests based on modification or extension of the existing handwritten test suite.

In order to automatically improve the mutation coverage of unit tests, Baudry et al. created DSpot, a test amplification tool for the statically-typed language Java [85]. Danglot et al. demonstrated the effectiveness of DSpot, by running it on 10 mature open-source projects. In their experiments, DSpot successfully improved 26 out of the 40 ($\approx$65%) test classes under study [86]. DSpot is able to leverage the fact that Java is a statically-typed language and typing information can be deduced through static analysis of the source and test code.

For dynamically-typed languages, test amplification techniques cannot solely rely on static analysis. Dynamic techniques are needed to deduce the type of variables and values that are to be asserted and to support type-sensitive input amplifiers. Abdi et al. have made a tool, Small-Amp with which they showed that test amplification is also possible for the dynamically-typed language Pharo Smalltalk [35, 87]. They adopted the concept of dynamic type profiling, with which they were able to overcome the lack of type information in the source code. However, their proposed solution is heavily dependent on Metalinks and Pharo's internal design, primarily its live programming environment, and the generality of their approach has not been confirmed on other dynamically-typed languages.

DSpot and Small-Amp both rely on mutation score as the adequacy criterion to improve the test suite against. To determine the mutation score, the code of the project

under test is mutated, and the test suite is executed against the mutated project. If the test suite fails, it is able to detect or kill the mutation and thus is able to prevent a possible regression. Mutation testing, however, is a time-consuming task: the test suite has to be executed the same number of times as there are mutants. When using the mutation score as an adequacy criterion for test amplification, this time cost will multiply with the number of amplified tests.

As dynamically-typed languages, such as Python, become increasingly popular, adequate test suites are needed. In contrast to the popularity of Python, research into automatically improving a test suite for a Python project is still sparse[67, 88], and non-existent in regard to test amplification. Indeed, the popularity of Python is still on the rise[89], due to its ease of learning, its huge amount of external libraries, and the further rise of data-mining and AI research. Python is a dynamically typed interpreted language. Despite Python being an object-oriented programming language, it supports multiple programming paradigms. Python allows a developer to disregard the object-oriented design and write code following the procedural and functional paradigms.[90]

With this chapter, we extend the concept of test amplification to Python, one of the most popular (dynamically typed) languages today according to IEEE[1] and on Github[2], and present **AmPyfier**[3]. With AmPyfier, we generalize the concept of dynamic type profiling using the Python debugging tools. Furthermore, we introduce *multi-metric selection* in order to increase the efficiency of the mutation score as a selection criterion. We demonstrate that it is not only feasible but also effective for existing open-source projects. In our evaluation on 11 open-source projects including 54 test classes, AmPyfier successfully amplified 37 test classes. Furthermore, multi-metric selection decreased the time cost ranging from 17% to 98% as opposed to selection based on the full mutation score. The experimental data is publicly available in our GitHub repository[4] where the generated reports and the amplified test suites can be found.

In the following section, we provide some background, what exactly test amplification is and how it is implemented in DSpot, and Small-Amp. Furthermore, we give some more background about Python, unit-testing in Python, and related work in regard to test generation for Python. In Section 3.3, we explain the main algorithms of AmPyfier itself with the help of a running example, how the implementation for Python differs from Small-Amp and DSpot and introduce multi-metric selection. We evaluated AmPyfier on 11 open-source projects, the results are discussed in Section 3.4, as well as the lessons learnt from these experiments, and the threats to validity. In the last section, we conclude and discuss further work that needs to be done.

[1] `https://spectrum.ieee.org/top-programming-languages/` (accessed on 15/1/22)
[2] `https://madnight.github.io/githut` (accessed on 15/1/22)
[3] `https://ansymore.uantwerpen.be/artefacts/ampyfier`
[4] `https://github.com/SchoofsEbert/AmPyfier_evaluation`

3.2 BACKGROUND AND RELATED WORK

3.2.1 Test Amplification

Test amplification[8] aims at improving a test suite against a given criterion. With test amplification the already existing test suite is taken as the main input. The knowledge gained from the test suite is than exploited to enhance it. Unit testing has become a widely used practice in most software projects. Projects or organizations make use of practices such as minimum requirements for an adequacy criterion (e.g. code coverage) before code is accepted and merged into the project. Others adopt methods such as test-driven development (TDD). In either case, developers write test code during or before the writing of production code. Consequently, the majority of modern software projects include a considerable amount of test code. These test suites often contain meaningful information about the project under test, and what should be tested. Although these test suites may cover most of the main scenarios in the code, some of the corner cases may still be missed and remain untested. In their snowballing literature study, Danglot et al. [8] categorise the research in test amplification as below:

1. Amplification by adding new tests as variants of existing ones (AMP_{add}).
2. Amplification by synthesizing new tests with respect to changes (AMP_{change}).
3. Amplification by modifying test execution at runtime (AMP_{exec}).
4. Amplification by modifying existing test code (AMP_{mod}).

Tools belonging to AMP_{add} will generate new test cases based on the original handwritten test cases. The newly generated test cases try to improve the test suite based on metrics such as mutation score (e.g. DSpot and Small-Amp)[86, 87], code coverage [31, 32, 61], and reproduction of crashes (e.g. MuCrash) [55, 91]. A special case of AMP_{add} is AMP_{change}, these tools and techniques only focus on the changed parts of the project under test compared to its previous version (e.g. DCI)[92, 93]. Instead of modification of the existing test suite, AMP_{exec} will modify the test suites dependencies like the OS file system, libraries, databases, remote services, or access APIs to GPS or Bluetooth (e.g. CAMP) [94, 95]. AMP_{mod} techniques will try to make the test suite more precise by increasing the input exploration (e.g. TAUTOKO) [96] or regenerating the oracles (e.g. Orstra) [97].

3.2.2 Related Work: DSpot and Small-Amp

AmPyfier is the adoption of the algorithms introduced in DSpot and Small-Amp into the Python ecosystem. DSpot and Small-Amp exploit the hand written test classes of a project, to generate new test methods with additional assertions (*assertion amplification*)

and new test input (*input amplification*) in order to increase the mutation score of the test. Whereas DSpot is developed for the statically typed language Java, Small-Amp performs test amplification for the dynamically typed language Pharo Smalltalk.

Components in DSpot and Small-Amp

DSpot and Small-Amp consist of respectively three and four main components. While both DSpot and Small-Amp have assertion amplification, input amplification, and test selection, Small-Amp introduced the dynamic type profiler in order to support dynamically typed languages. In the next paragraphs, we provide short descriptions for each component.

Assertion Amplification. This module is responsible for generating assertions for objects or functions under test. One assertion amplification technique proposed by Xie [97] works based on a dynamic analysis that runs the test to be amplified and captures the object states during the execution. In this technique, the value of getter-methods of an object, or the return values of function calls are captured then extra assertion statements are added.

Input Amplification. Test input are all statements in a test method except the assertion statements. With input amplification, new versions of test input are generated based on the existing test input. The amplified test input may take a different execution path or bring the objects under test into new states. In other words, the input amplification component empowers a test amplification tool to explore the search space of all possible tests.

DSpot uses an input amplification technique inspired by Tonella [39] (ETOC). In this technique, the abstract syntax tree of the test input is modified by a set of input amplification operators, and new versions of the test input are generated. Input amplification operators are divided into two categories. Type-insensitive operators, which range from changing the literal values by simple arithmetic operations on integers and modifications of strings to duplication, or removal of method and function calls. The other category is type-sensitive operators, such as the addition of new method and function calls. Indeed, when passing an argument or when manipulating the return value, we must ensure that we pass a value of the appropriate type. Multiple such amplifiers are applied after each other to combine the different ways in which a test is modified.

Selection. The goal of test amplification is to improve the test suite in regard to a quantifiable engineering goal, expressed as a metric. Possible metrics are: coverage or mu-

tation score improvement, fault detection capability improvement, oracle improvement, and debugging effectiveness improvement [8]. In both DSpot and Small-Amp this metric is the adequacy criterion mutation score. Both tools generate lots of new test methods derived from the existing test suite. The mutation score is calculated for each newly generated test method. If it is able to improve it, the generated test method is selected as an amplified test method otherwise it is discarded.

Type Profiling. In order to support type-sensitive input amplification operators, type-information needs to be deduced from the tests and the project under test. Performing static analysis in dynamically typed languages like Python yields less information than in statically typed languages because the source code in dynamic languages does not include type information. To generate new arguments for new method and function calls, it is typically needed to know which input type to pass to the call. For statical typed languages, such as Java, the types of the input can be derived through a static analysis of the project under test, but for languages such as Python, the types need to be derived dynamically. Since the existing test suite is one of the main inputs in test amplification tools, performing dynamic type profiling by running the existing test suite is possible. The concept, introduced by Small-Amp, is to run the existing test suite and capture (1) the type of arguments in the methods under test (2) the type of variables in the original test methods.

Implementation

The main amplification algorithms used in DSpot and Small-Amp, and now AmPyfier are quite similar. They repeat the input amplification, assertion amplification, and selection steps iteratively for all existing test methods. Mutation Coverage is proposed as a selection criterion in DSpot, and is taken over by Small-Amp.

To know the values to be asserted in the assertion amplification step, DSpot and Small-Amp both use the same technique. First, the statements of interest are encapsulated in observation statements, and the test is executed. The results are collected and, based on the observations, the assertion statements are constructed.

For gathering the type information, Small-Amp introduces a dynamic type profiling step before the main loop of the algorithm. To extract this information Small-Amp adds Metalinks on all variables in a test method. Metalink[34] is a fine-grained reflection mechanism, that allows to install AST node level proxies. Subsequently, Small-Amp runs the test. The Metalinks are fired when the variables are used to capture the type-information in the run-time. This type information is used to support type-sensitive input amplification operators, e.g. generation of random parameter for a newly added method call.

3.2.3 Python

Python is a dynamically typed interpreted language, which supports multiple programming paradigms. Python doesn't force developers to write their code in an object-oriented manner and encapsulate all functionality in classes, allowing developers to write code following multiple programming paradigms. In the background, however, everything in Python is an object, even the current stack frame. This fact is leveraged by the Python Tracer, which allows to dynamically obtain all necessary information about what is happening in the project under test during execution.

While everything is an object according to the inner workings of Python, Python has no notion of encapsulation. Private or protected attributes can not be enforced. Every attribute is public, and can be accessed from everywhere. It is up to the developer to follow the coding conventions where an attribute should be considered protected if it is prefixed with one underscore (e.g. `_protected_attribute`), and private if it is prefixed with two underscores (e.g. `__private_attribute`).

It is possible in Python to pass a custom traceback function to the Python interpreter with the `sys.settrace()` function. Those custom trace functions make it possible to investigate the execution of a Python project, and collect runtime information before certain events occur, such as a new line, a function call, or the occurrence of an exception. The current stack frame, as well as the event type and a possible argument, are passed to the custom trace function and can be inspected. From this stack frame, for example, it is possible to access the variables currently in scope. The `inspect` module , part of the Python Standard Library, elevates this further and provides several useful functions to help get information about live objects such as modules, classes, methods, functions, tracebacks, frame objects, and code objects [98]. For example, it allows to get all members of an object with the `getMembers()` function, or the name of a module some function was defined in using `getModule()`. The `inspect` module can even return the specific source code of an object or method definition with `getsource()`.

Python is an idiomatic language [99], for doing a specific task, there is only one typical, well-known and optimal way. Other ways are not preferred even if they are grammatically correct. Since following the idioms makes the code more readable and easier to maintain, it is important for tools like AmPyfier, which synthesize code portions, to support such guidelines.

Unit Testing in Python

Unit testing in Python has no straightforward conventions. Since Python projects don't have to follow object-oriented design, a unit can be seen as both a module or a class.

Additionally, Python has multiple unit test frameworks, which can function completely differently. Two of the most popular are the default `unittest` framework included in the Python Standard Library, and Pytest[5].

Unittest is inspired by the unit testing framework for Java, JUnit. All test methods should be contained in a class derived from the `unittest.TestCase`. A single test file can consist of multiple such test classes. A test suite can be considered as a single test file, or a collection of test files. In this chapter, we refer to a test suite as one test file, possibly containing multiple test classes.

Test Generation in Python and Related Work

In regard to test amplification, no tools have been developed for Python to the best of our knowledge. However, there is relevant work on test generation, which inspired the design of AmPyfier.

In 2020 Lukasczyk et al. introduced PYNGUIN [67]. A tool that can automatically generate unit tests for Python. However, PYNGUIN implements the test generation techniques of whole-suite generation[84] and feedback-directed random generation[100], established for statically typed languages and uses them on Python. Therefore, PYNGUIN is most effective under the assumption that the system under test contains type information with Python's type annotations. It thus does not completely addresses Python's dynamic nature.

Another tool that automatically generates unit tests for Python is Auger [88], introduced in 2016. Auger leverages the Python tracer and tracks function calls encapsulated with the Auger context manager. If a function is called that is defined in the module under test, Auger keeps track of both the values of the arguments as well as the return values. Based on those values, assertions can be generated. Auger thus does not need to rely on type annotations, and only generates tests for the explicitly called execution of the project under test (no input amplification). Auger needs to be called from a python module with a context manager in order to generate tests for the called functions. If a function is not called, it won't be tested. Furthermore, Auger only supports very trivial projects written in Python 2 without exception handling or decorators.

While both PYNGUIN and Auger could be used as a good starting point to develop a test suite, they both have their limitations. PYNGUIN relies on type annotations, whilst Auger does not alter the execution, can only handle trivial projects and, is in an abandoned state[6].

[5]`https://pytest.org(accessedon15/1/22)`

[6]The last public commit was in february 2019.

3.3 AMPYFIER

In this section, we present AmPyfier. In Section 3.3.1 we present the running example, used to explain the working of AmPyfier in the subsequent subsections. A broad overview of the main algorithm is given in Section 3.3.2. In Sections 3.3.3 and 3.3.5, the main differences in assertion and input amplification for Python are set out. Lastly, the selection of tests is described in Section 3.3.6

3.3.1 Running Example

To explain the workings of AmPyfier, we use a running example, namely a simple fund implementation: SmallFund. See listing 3.1 for the source code of SmallFund. The SmallFund class in our running example has five public methods; `get_balance`, `deposit`, `is_empty`, `get_transactions` and `get_self`. Furthermore, it has two protected attributes (`_balance` and `_transactions`) and one public attribute (`owner`).

```
1  class SmallFund:
2      def __init__(self, owner):
3          self._balance = 0
4          self._transactions = []
5          self.owner = owner
6
7      def get_balance(self):
8          return self._balance
9
10     def deposit(self, amount):
11         if amount >= 0:
12             self._balance += amount
13             self._transactions.append( amount)
14         else:
15             raise Exception("Can not deposit negative amounts", amount)
16
17     def is_empty(self):
18         return self._balance == 0
19
20     def get_transactions(self):
21         return self._transactions
22
23     def get_self(self):
24         return self
```

Code Excerpt 3.1: The SmallFund class

The manually created test suite for SmallFund is presented in listing 3.2. This test suite covers the main scenario of depositing positive amounts. However, it does not cover corner cases like depositing an *amount* less than zero.

```
1  import unittest
2  from SmallFund import SmallFund
3
4  class SmallFundTest(unittest.TestCase):
5      def setUp(self):
6          self.b = SmallFund("Iwena Kroka")
7
8      def testDeposit(self):
9          self.b.deposit(10)
10         self.assertEqual(self.b.get_balance(), 10)
11         self.assertIsInstance(self.b.get_self(), SmallFund)
12         self.b.deposit(100)
13         self.b.deposit(100)
14         self.assertEqual(self.b.get_balance(), 210)
```

Code Excerpt 3.2: Original test suite

3.3.2 Logic

The logic of AmPyfier is shown in Algorithm 2. The main algorithm is inspired by the work of DSpot, with the addition of dynamic type profiling inspired by Small-Amp and multi-metric selection (See Section 3.3.6). The default input is a single Python test file, the Test Suite (TS), the list of amplifiers/mutators (A), the number of subsequent amplifier runs (n) and the module under test (mut).

AmPyfier scores the current test class against the module under test using the multi-level coverage calculator. The variable `score` contains a tuple of absolute coverages. We explain it in more details in Section 3.3.6. Afterwards the test class is passed to the dynamic type profiler in order to dynamically obtain information about the various types used in the method or function calls in the test methods.

The inner loop (lines 6 to 16) loops over each test method in the test class, and assertion amplifies it. If the assertion amplified test (a_test) improves the current score, it is added to the Amplified Test Class (ATC).

The amplified test, the amplifiers to use, the number of input amplification iterations and also the profiled type information are passed to the input amplifier. The input amplifier will return a set of input amplified test methods (ITT). These input amplified test methods are then assertion amplified and sorted based on their modification count.

Finally, the amplified tests are scored, and the current score is updated (line 15). The amplified tests that improved the current score (IIT) are appended to the amplified test class. Once each test method is amplified, the amplified test class is added to the amplified test suite (ATS). At the end, each amplified test class is added to the amplified test suite, which is returned after each test class is amplified.

AmPyfier currently only supports the `unittest` framework. The reason why `unittest`

Algorithm 2: AmPyfier logic

```
   input  : Test Suite TS
   input  : List of amplifiers A
   input  : Number of Input Amplifier runs n
   input  : Module Under Test mut
   output : Amplified Test Suite ATS
1  ATS ← ∅;
2  for TC in TS do
3  |  ATC ← TC;
4  |  score ← Selection(ATC, mut);
5  |  tp ← TypeProfile(TC);
6  |  for test in TC do
7  |  |  a_test ← AssertionAmplify(test, mut);
8  |  |  a_score ← Selection(ATC ∪ {a_test}, mut);
9  |  |  if a_score > score then
10 |  |  |  ATC ← ATC ∪ {a_test};
11 |  |  |_ score ← a_score;
12 |  |  IT ← InputAmplify(test, A, n, tp);
13 |  |  AT ← AssertionAmplify(IT);
14 |  |  AT ← Sort(AT);
15 |  |  score, IIT ← Selection(AT, mut);
16 |  |_ ATC ← ATC ∪ IIT;
17 |_ ATS ← ATS ∪ ATC;
```

is supported first is: (1) it is the standard testing library proposed by the language, (2) it complies with xUnit testing practices similar to JUnit in Java and sUnit in Smalltalk. However, the tool is developed in an extensive way such that support for other frameworks, such as `Pytest`, can be added.

3.3.3 Assertion Amplification

Algorithm 3 shows how assertion amplification is implemented in AmPyfier. It consists of four steps: (1) Firstly, the assertions are stripped from the test. (2) Secondly, the test without assertions is executed and observations are constructed. (3) Then, execution and collection of the observations are repeated `F` times. These observations are compared and all non-deterministic observations are discarded. (4) Finally, (new) assertions are constructed based on those observations.

Algorithm 3: Assertion Amplification

```
    input  : Test Method test
    input  : Unit under Test unit
    output : Assertion Amplified Test Method amplified
1   amplified ← RemoveAssertions(test);
2   observations ← Observe(amplified, unit);
3   for F times do
4       observations2 ← Observe(amplified, unit);
5       if observations ≠ observations2 then
6           observations ← RemoveNonDeterministic(observations,
              observations2);
7   amplified ← AddAssertions(amplified, observations);
```

The first step is to remove the original assertions. The test source is scanned statically and all method calls that are known as an asserting statement are replaced with their asserted expression.

The second step is observing the test method. The test method is dynamically executed and the values of the variables and the state of the objects are captured. For observing the values, DSpot and Small-Amp manipulate the source code to inject observation statements. Python, however, is an interpreted language and lots of information is accessible during runtime. AmPyfier uses Python's `sys.settrace()` function along with a set of custom trace functions to hook in different points during the execution and observe the objects' states and collect the return values of function calls.

The observations of the state of objects are also made possible thanks to the interpreted nature of Python. Python has a built-in module `inspect` enabling the possibility to derive all information about live objects during runtime, such as their state,

the module and file it was defined in, the name, etc. AmPyfier uses a custom method based on `inspect.getmembers()` to get all the public attributes and methods of an object, and stores them in the observation. This custom method is exactly the same as `inspect.getmembers()`, except it adds the possibility to also capture members or variables that cause exceptions instead of simply raising them. This proves its usefulness after input amplification, where AmPyfier may have pushed an object in an erroneous state. If an attribute or the return value of a getter call is itself a complex object, AmPyfier will also observe its attributes and getter methods until a configurable `AE` levels deep. The default value for `AE` is three.

If AmPyfier extracted an `assertRaises` statement, the extracted statement will raise an exception during observation. Exceptions can also occur when the test input is mutated by the input amplification module. As such, it is crucial for AmPyfier to be able to handle the exceptions correctly during observation of the test method. When the observation process is interrupted by an exception, AmPyfier finds the line causing the exception. Then, it wraps this line in a `Try-Catch` block and restarts the observation process. This allows AmPyfier to assert raised exceptions on a per-line basis, and still assert the other statements in a test. Furthermore, AmPyfier generates an object observation for the exception, so it can be asserted that the correct exception is raised.

The results after one observation run of the `testDeposit` test method are shown in table 3.1. Note that method calls with no return value are not stored. Observations of complex objects do not include their protected or private attributes and methods, nor generate new object observations for self-references (e.g., do not generate a new object observation for the `get_self` method call). If a getter call follows after a line with an object observation for the same object, the getter call will be removed from the observation in order to reduce duplicate assertions. This is the case for both `get_balance` calls followed rightly after the `deposit` calls.

To prevent generating flaky tests, AmPyfier excludes all non-deterministic values from the observations. AmPyfier uses a configurable constant `F`, by default equal to 3, and repeats the observing process `F` times. Afterwards, the observation results are compared, and the non-deterministic observations are removed.

After the test has been observed and the values to be asserted collected, the final step is to construct the assertion statements. AmPyfier loops on a line per line basis over the method-body, and constructs the assertions based on the observations originating from that line.

Listing 3.3 shows our running example test method `testDeposit` after assertion amplification.

Line	Observations
9	[{Name: self.b, Type: SmallFund, Object: smallfund.SmallFund object at 0x7f50c0fce400, Members: [(owner:Iwena Kroka)], Methods: [(get_transactions:[10]), (is_empty:False)]}]
10	[{FuncName: get_balance, Value: 10}]
11	[{FuncName: get_self, Type: smallfund.SmallFund}]
12	[{Name: self.b, Type: SmallFund, Object: smallfund.SmallFund object at 0x7f50c0fce400, Members: [(owner:Iwena Kroka)], Methods: [(get_balance:110), (get_transactions:[10, 100]), (is_empty:False)]}]
13	[{Name: self.b, Type: SmallFund, Object: smallfund.SmallFund object at 0x7f50c0fce400, Members: [(owner:Iwena Kroka)], Methods: [(get_transactions:[10, 100, 100]), (is_empty:False)]]}]
14	[{FuncName: get_balance, Value: 210}]

Table 3.1: Results after observing the `testDeposit` test method

```
1  def testDeposit_amp(self):
2      self.b.deposit(10)
3      self.assertEqual(self.b.get_transactions(), [10])
4      self.assertFalse(self.b.is_empty())
5      self.assertEqual(self.b.owner, 'Iwena Kroka')
6      self.assertEqual(self.b.get_balance(), 10)
7      self.assertIsInstance(self.b.get_self(), SmallFund)
8      self.b.deposit(100)
9      self.assertEqual(self.b.get_balance(), 110)
10     self.assertEqual(self.b.get_transactions(), [10, 100])
11     self.assertFalse(self.b.is_empty())
12     self.assertEqual(self.b.owner, 'Iwena Kroka')
13     self.b.deposit(100)
14     self.assertEqual(self.b.get_transactions(), [10, 100, 100])
15     self.assertFalse(self.b.is_empty())
16     self.assertEqual(self.b.owner, 'Iwena Kroka')
17     self.assertEqual(self.b.get_balance(), 210)
```

Code Excerpt 3.3: Assertion Amplified test method

3.3.4 Type Profiler

Since input amplification is only performed statically in AmPyfier, some type-sensitive input amplifiers, such as the addition of a new method call, need type information in advance.

Python is a dynamically typed language and does not force developers to add type information to the source code. Python supports type annotations since Python 3, but there is no guarantee that developers annotate types during development. In addition, annotations defined for functions and variables are not enforced by the python runtime and they are only used by third-party tools like IDEs and linters, so there is no guarantee that they are defined correctly [101]. As a consequence, AmPyfier does not rely on the

annotated types in the source code and uses a dynamic type profiling step to collect type information.

Dynamic type profiling is introduced by Small-Amp for test amplification in the dynamically typed language Pharo Smalltalk (Chapter 2). With AmPyfier we generalized this concept away from Smalltalk specific Metalinks. In simple words, the type profiler exploits the fact that it is possible to extract type information dynamically by executing the existing test suite. As is mentioned in section 3.2.1, the existing test suite is the main input in the algorithm and lots of useful information about the project under test are embedded inside.

The test suite as well as the test class to be amplified are executed in order to detect the type of arguments in the methods and functions under test. This type profiler works similar to the observation step in assertion amplification, and leverages the ability to pass custom tracing functions to the Python interpreter through the `sys.settrace()` function. Allowing AmPyfier to monitor calls executed in the test methods, and discover the possible types used as input for different method and function calls. Getter methods without arguments will not be recorded, as they will be generated during the assertion amplification. For our example test class, the result of the dynamic type profiling can be seen in table 3.2. With this information, new method, or function calls can be generated as well as their random inputs.

function	test	line	caller	caller type	arguments	argument types
deposit	testDeposit	9	self.b	SmallFund	(10)	(int)
deposit	testDeposit	12	self.b	SmallFund	(100)	(int)
deposit	testDeposit	13	self.b	SmallFund	(100)	(int)

Table 3.2: Results after dynamic type profiling the `SmallFundTest` test class

3.3.5 Input Amplification

The algorithm for input amplification in AmPyfier is presented in listing 4. Input amplification starts similar to assertion amplification, namely, stripping the tests of its assertions again. Then we loop over all the amplifiers n times in order to combine different amplifiers. This loop generates loads of new tests, so each time a loop over the amplifiers is completed, only T randomly selected tests are kept. Those T tests are added to the results, and also passed on to the next iteration. Once the n iterations are completed, the selected tests are returned.

In order to input amplify the test, we need to strip the test from any assertions. The statements inside these assertions are extracted, so that the test execution is not changed upon removal of the assertions. If the assertion statements were to be removed com-

Algorithm 4: Input Amplification

input : Test method `test`
input : Number of iterations `n`
input : List of amplifiers `AMPS`
input : Typing information `tp`
output: Input Amplified Tests `AT`

```
1  temp ← { RemoveAssertions(test) } ;
2  results ← ∅;
3  for n times do
4      amplified ← ∅ ;
5      for amp in AMPS do
6          amplified ← amplified ∪ amp.apply(temp, tp);
7      if Size(amplified) > T then
8          amplified ← SelectRandom(amplified, T)
9      results ← results ∪ amplified;
10     temp ← amplified;
```

pletely, the method or functions inside them would not be called. Furthermore, through extraction of the statements, the assertions the original developer deemed important can be restored during assertion amplification. If the assertions were kept in the test, the test would fail on the amplified inputs. We want to construct new assertions based on the amplified inputs.

Once stripped from assertions, AmPyfier loops over the list of selected amplifiers n times. Each time the loop over the amplifiers is completed, new tests are generated, increasing the number of tests exponentially for each loop. To counter this, after each loop, only a configurable amount of tests (T) is randomly selected and passed on to the next loop. The default value of T is 200.

The amplifiers used in input amplification are based on the ones used in DSpot and Small-Amp:

- Literal Mutation:
  - Numerical Values: $0, +1, -1, *2, /2$
  - Strings: add random char on empty string, double the string, random substring of half the size, replace with empty string
  - Booleans: negation
  - Unification of literals
  - Replace with `None`
- Mutation of Method Calls:
  - Removal
  - Duplication

Code Excerpt 3.4: An example of transformations applied during input amplification

```
1  def testDeposit_stripped(self):
2      self.b.deposit(10)
3
4      self.b.get_balance()
5      self.b.get_self()
6      self.b.deposit(100)
7      self.b.deposit(100)
8      self.b.get_balance()
```

```
1  def testDeposit_i_amplified(self):
2      # removed statement
3      self.b.deposit(-45485) # new
           statement
4      self.b.get_balance()
5      self.b.get_self()
6      self.b.deposit(100)
7      # removed statement
8      self.b.get_balance()
```

- Addition of a new method call

In listing 3.4, the test method in listing 3.2 is shown during stages of input amplification. The listing on the left, is the method after stripping the assertions, and the listing on the right is the method after three transformations: 1) One method call has been removed at line 2, 2) at line 3, a new call to a random method is added with a random input value of the correct type, 3) and another call is removed at line 7. Note that, the unit under test used in this example has only three methods, of which two are getters. So, the only interesting method which can thus be added, `deposit()` is generated.

After the input amplifier loop is finished, all newly generated tests are assertion amplified, and sorted depending on the number of modifications:

$$N_{modifications} = N_{all_assertions} + N_{transformations} - N_{original_assertions}$$

Adding a new assertion statement counts as one modification, the same for any modification made through input amplification. Regenerating the original assertions is not counted as a modification. The input and assertion amplified test (listing 3.5), in our example, has a modification count of 7: it has 3 transformations, 4 new assertions are added and 3 assertions are regenerated.

Sorting helps AmPyfier to prefer methods that differ the least from the original test method, i.e. those tests that have the lowest number of transformations. So, if two different amplified tests kill the same mutant, the test with the lowest modification count is selected.

3.3.6 Multi-Metric Selection

Input amplification yields large amounts of new test methods, and most of them do not increase upon the adequacy criterion, thus a method to decide on which tests to add

```
1   def testDeposit_amplified(self):
2       with self.assertRaises(Exception) as excep_info:
3           self.b.deposit(-45485)
4       self.assertEqual(excep_info.exception.args, ('Can not deposit negative 
            amounts', -45485))
5       self.assertEqual(self.b.get_balance(), 0)
6       self.assertIsInstance(self.b.get_self(), SmallFund)
7       self.b.deposit(100)
8       self.assertEqual(self.b.get_transactions(), [100])
9       self.assertFalse(self.b.is_empty())
10      self.assertEqual(self.b.owner, 'Iwena Kroka')
11      self.assertEqual(self.b.get_balance(), 100)
```

Code Excerpt 3.5: Assertions & Input Amplified test method

to the amplified test class is needed. This method scores the amplified test methods based on a given adequacy criterion. AmPyfier mainly uses the same criterion as suggested by DSpot: the number of mutants a test suite is capable to kill and extends this with code coverage and dynamic generation of new mutants.

One of the main downsides of mutation testing is that, naively, a test has to be executed the same number of times as there are mutants. For test amplification, this means that a test has to be executed as many times as the number of mutants multiplied by the number of amplified tests. It comes as no surprise that reducing the number of mutants to test against can drastically decrease the runtime.

AmPyfier reduces the number of mutants based on the Reachability, Infection, Propagation (and reveal) model (RIP) [102, 103, 104]. To kill a mutant: (R) the mutated part needs to be reached by the test (I) for a change in the program state to be visible compared to the original program, (P) the change needs to be propagated to the test, and finally, (reveal) the change needs to be asserted by an assertion statement. Simply said, if a mutant is not covered by a test method, it definitely will not be killed. AmPyfier generates mutants only in the covered lines of code. However, this practice has as a result that amplified tests that reach previously non-covered code will not selected, because they do not kill any mutant. Therefore, AmPyfier introduces multi-metrics selection: the usage of code coverage along with mutation testing and dynamic generation of new mutants for newly covered code.

First, the test is scored against code coverage, if it increases upon the code coverage, the test is added to the amplified test suite. After the tests have been scored against code coverage, new mutants are generated for the newly covered lines. Subsequently, the tests are scored against the alive mutants. If a test kills an alive mutant, it is added to the amplified test suite. A test is thus selected if it increases the code coverage or it kills new mutants.

Python has many capable mutation testing frameworks, but the one used by AmPyfier is `mutaTest`[7]. This framework is chosen because of the extensive API that allows for the introduction of caching. AmPyfier thus does not need to retest for every mutant that is already killed and can throw away mutants that result in time-outs, drastically decreasing the runtime of AmPyfier. To derive the coverage score of a test, AmPyfier makes use of the Coverage.py framework[8].

The return value `score` is a tuple of the absolute number of covered elements. The first element of this tuple shows the number of all covered lines and the second element shows the number of killed mutants. This variable is used in the main algorithm (Algorithm 2) to detect any improvements in the coverage.

3.4 EVALUATION

To evaluate AmPyfier, we ran it using the default configuration on multiple test files from multiple small open-source projects found on GitHub. In total, AmPyfier has been evaluated on 54 test classes belonging to 11 projects. The projects and their characteristics and the evaluated test classes can be seen in Tables 3.4 and 3.5 in the appendix[9]. For our evaluation, all test suites had to have been developed using Python's standard `unittest` framework. We have randomly selected 11 projects with recent commits from varying popularity, size, and coverage and mutation score. The projects range from small one-person projects with no stars and 22 forks to more popular, medium-sized, projects with more than a hundred contributors, a thousand forks, and four thousand stars. This range allowed us to evaluate AmPyfier on both small hobby projects as well as libraries actively used in a multitude of other Python projects. The goal is to test to which extent AmPyfier is still able to increase the mutation and coverage score, even for those test suites with already high adequacy criteria. The results of our evaluation are publicly available in our GitHub repository[10].

The default configuration of AmPyfier is a) the usage of the cache to minimize runtime, b) automatic discovery of the module under test c) 3 observation runs to counter flaky tests d) 200 tests can be collected during each loop inside the input amplification part, e) there are 3 of these loops, and f) all amplifiers discussed in Section 3.3.5 are used.

In our evaluation, AmPyfier detects the module under test for each individual test class. This reduces the need for developers to manually specify which module a specific test class tests. The module under test is then used by AmPyfier to detect and observe interesting method and function calls, objects under test, etc. Furthermore, it allows AmPy-

[7] `https://github.com/EvanKepner/mutatest` (accessed on 15/1/22)
[8] `https://github.com/nedbat/coveragepy` (accessed on 15/1/22)
[9] All repositories accessed on 15/1/2022
[10] `https://github.com/SchoofsEbert/AmPyfier_evaluation`

fier to compute the coverage score for the test class, and keep track of covered mutants to test against. AmPyfier tackles this problem using the following technique:

- First, an inspection of the imports of the test class is performed and imported modules belonging to the project under test are stored.
- Subsequently, all function calls and constructors are observed. The module that is used the most in the test class is considered the module under test.

The results of this experiment are presented in Table 3.6 under Appendix 3.7. The second column (`T`) expresses the execution time it took to amplify the test class. The third column (`LOC`) are the lines of code in the module under test whereas the fourth and fifth columns (`MO` and `MA`) are the number of test methods originally in the test class and the number of amplified test methods added. The other columns are the coverage score (`CS.`) and mutation score(`MS.`) of the original test class (`..O`) and after amplification (`..A`) and the relative increase in scores after amplification (`R..I`), all expressed in %. The relative increase in coverage score is computed as below:

$$\%RCSI = 100 * \frac{\#lines_covered_{Amplified} - \#lines_covered_{Original}}{\#lines_covered_{Original}}$$

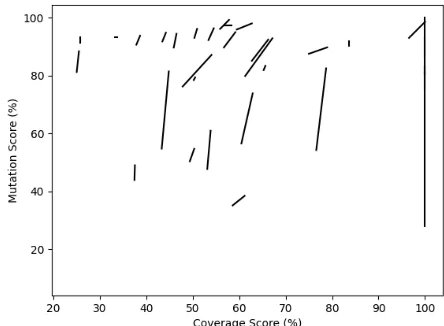
and the relative increase in mutation score in a similar matter:

$$\%RMSI = 100 * \frac{\#mutants_killed_{Amplified} - \#mutants_killed_{Original}}{\#mutants_killed_{Original}}$$

The test classes for which AmPyfier was able to increase a score, the name of the test class, as well as the increased scores are bolded.

Due to space constraints, not all results are represented such as the absolute number of mutants killed, and alive after amplification. For the full results of our experiment, as well as the amplified test suites, we refer to our GitHub repository with the results of the experiment.

AmPyfier is able to increase the coverage score for 28 out of the 46 test classes where coverage improvement was possible (61%). Furthermore, it improves the mutation score for 32 out of the 53 test classes where mutation improvement was possible (60%). In total AmPyfier is able to improve one or both of the adequacy criteria in 37 out of the 53 test classes where improvement in one ore both criteria was possible (70%). Regarding the execution time, amplifying with AmPyfier only took more than 1 hour in 10 out of the 54 test classes thanks to multi-metric selection.



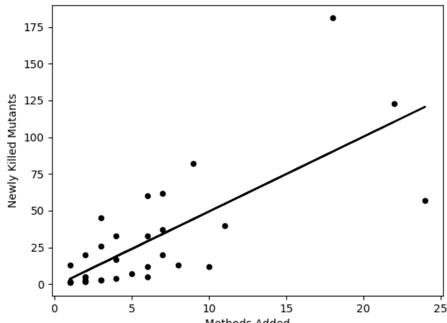


Figure 3.1: a) Coverage vs Mutation score in Original and Amplified Test Class & b) Methods Added vs Newly Killed Mutants

In Figure 1 a) the adequacy criteria improvements are plotted as lines in a line diagram. The starting point of the line is the code coverage and mutation score for the original test class (`CSO`, `MSO`), the end point is the same values for the amplified test class (`CSA`, `MSA`). If a line goes diagonal, this means AmPyfier was able to improve both the coverage and mutation score. A horizontal line means only the coverage score was improved whereas a vertical line means only the mutation score was improved. The diagram shows that for most test classes where AmPyfier improved one adequacy criterion, the other was also improved. Furthermore, the most improvement can be seen for test classes where the original coverage score is higher than 40%. (Note: multiple test classes are hidden together for CSO 100%, this can be seen in the table with the results.)

Figure 1 b) shows the number of methods added versus the number of newly killed mutants (mutants killed by the amplified test class, but not the original test class) and the trend line. The diagram shows that the addition of a single amplified method enables the test class to kill multiple new mutants. The linear trend shows that on average one amplified method kills 5 new mutants. The real number for methods needed to kill the new mutants is even lower, because amplified test methods that only increase code coverage are also counted.

Influence of dynamic type profiling on amplified test methods. In our evaluation, 5 test classes were improved thanks to 6 methods generated by a type-sensitive input amplifier, and thus type profiling. `testSerializers` and `TestTOC` saw their code coverage increase, whereas `TestStringMethods`, `TestSmarty` and `TestMetaData` saw an increase in both adequacy criteria thanks to the addition of a new method call. Although only a small subset of the evaluated test classes were improved thanks to call addition, it shows that type-sensitive input amplifiers are able to increase the coverage and/or the mutation score and that type profiling works outside of Pharo Smalltalk. The reason for

the small subset is for a part implementation-dependent. Amplified test methods with the same number of modifications generated by other input amplifiers are scored (and thus selected) before those generated by type-sensitive input amplifiers.

Influence of Multi-Metric Selection on Time-Efficiency. In order to investigate the influence of multi-metric selection on the runtime of AmPyfier, we have run our evaluation again on 10 test classes with differing original coverage scores, this time selecting the amplified test using classic mutation score (i.e. scoring them against all mutants in the module under test). In our original evaluation, 5 of those test classes saw no improvement in regard to their mutation score, while AmPyfier was able to increase the mutation score for the other 5. As shown in table 3.3, we see that the time AmPyfier took to amplify the test class is drastically reduced thanks to multi-metric selection while the number of newly killed mutants is similar. The reduction in time-cost varies from 17% in `RatioTest` to 98% in `TestConfigParsing`. The differences in the number of killed mutants are due to the effect of randomness during input amplification as also explained in Section 3.4.2.

TestClass TestClass	Runtime Multi-Metric	Runtime Classic	Newly Killed Mutants	Newly Killed Mutants Classic
ProcessTest	0:41:2	1:37:20	123	92
RatioTest	0:44:09	0:53:23	57	55
TestSerializers	0:56:59	11:58:03	26	26
TestBlockParser	0:13:27	2:39:08	17	21
TestConfigParsing	0:07:55	8:12:59	3	2
TestAstRender	0:00:34	0:02:29	0	0
PdfReaderTestCase	1:53:08	8:05:18	0	6
TestAdmonition	0:06:02	3:20:19	0	0
RegistryTests	0:53:20	56:36:50	0	1
TestBlockParserState	0:11:52	8:56:47	0	1

Table 3.3: Comparison of the time-cost using multi-metric selection compared to test selection based on full mutation score

3.4.1 Lessons Learnt

After we collected the results of our evaluation, we took a critical look at the amplified test classes AmPyfier generated. We looked at the test classes where we saw improvement, and more importantly, those with no or minimal improvement. In the following paragraphs, we share what we learned from this investigation and what improvements would be possible.

Amplification of pickles is challenging. Python has the ability to (de-)serialize the structure and states of objects, whereby the object is converted into a bitstream or vice-versa. This makes it possible to read and write objects to files or send them across a network. The process is called Pickling and makes use of the `pickle` module. It is very important when trying to pickle and unpickle an object that it is imported the same way.

Since AmPyfier works based on mutation testing and the unit under test may be mutated, pickling can cause exceptions during the execution of the test method, either during observation or selection.

In our experiments, we encountered this problem in one test class: `ChildDictTests` of the project Addict. However pickling is only tested in one of the 63 methods, so increase in mutation score was still possible for both classes.

If AmPyfier encounters this problem, the following exception will be thrown:
_pickle.PicklingError: Can't pickle <class 'CLASSNAME'>: it's not the same object as CLASSNAME. In the case of `ChildDictTests` the error thus looks as follows:
_pickle.PicklingError: Can't pickle <class 'test_addict.CHILD_CLASS'>: it's not the same object as test_addict.CHILD_CLASS.

Amplification of file-based tests is not effective. Another situation where test amplification is challenging is when the test works based on file inputs. In such a test method, a file is opened and the object under test is initialized based on its content. In these cases, current input amplification operators are not efficient, because they only consider the test source code and generate new inputs by mutating it. They can not manipulate the content of the files to force the test to new states.

For example, in the test class `PdfReaderTestCases`, one of the test methods opens a PDF file and initializes a `PdfFileReader` object from the content of the file. Then it uses the initialized object to assert some values. Literal input amplification operators in this test have low effectiveness because they mutate the string of the file path. Such mutations only generate an incorrect file path, which causes the raising of an exception, and does not permit AmPyfier to explore the search space of possible test inputs.

In our experiments, four test classes were suffering from this problem: the above-mentioned `PdfReaderTestCases` and `AddJsTestCase` of the PyPDF2 project, `TestMPQArchi` of MPyQ and `TestPluginDirective` of Mistune. In only one of those four classes we were able to increase both adequacy criteria: `TestPluginDirective`.

In `TestPluginDirective` we were able to increase both adequacy criteria thanks to a flipped boolean in our amplified test. `test_html_include` (listing 3.6) creates a markdown file which does not escape special characters and then tries to include other

Code Excerpt 3.6: The test_html_include method before amplification, and after amplfication

```
1 def test_html_include(self):
2     md = create_markdown(escape
          =False, plugins=[
          DirectiveInclude()])
3     html = md.read(os.path.join
          (ROOT, 'include/text.md
          '))
4     ...
```

```
1 def
     test_html_include_bool_inv_0_none_4
     (self):
2    md = create_markdown(escape
         =True, plugins=[
         DirectiveInclude()])
3    html = md.read(os.path.join
         (ROOT, 'include/text.md
         '))
4    ...
```

markdown files. Afterwards, it asserts that some of the lines that should have been included are indeed in the generated html.

Thanks to the flipped boolean, the `Markdown` object has its property `_escape` set to `True` which causes, among other lines, the following line to be newly covered: `return '<p>' + escape(html) + '</p>\n'`. The mutation framework will mutate the arithmetic operators, and cause exceptions if the amplified test method is executed. In the original test method, however, the above line was never reached, so those mutations would have never been caught.

Assertion generation for complex objects is not efficient. In assertion amplification, AmPyfier needs to convert the observed object/value into a Python AST node representing the original object. This process is fairly simple for asserting literals, or lists/-dictionaries consisting of literals. However, for complex objects, this is more challenging. To overcome this, AmPyfier records the values from getters and public attributes of each object. If the value is also another object, it repeats the process for it up to a defined depth. However, this technique easily clutters the test with a large number of extra assertions, rendering the test unreadable. A possible way to solve this is to implement assertion pruning. During assertion pruning, only those assertions that improve upon the given adequacy criterion are kept in the test method. However, naively removing assertions and/or statements to declutter the test method, could alter its execution and lead to a failing test. Assertion pruning is currently a work in progress for AmPyfier and is not yet considered in this work.

An example of this can be seen in listing 3.7. The left column shows the the `test_create` method of the `TestMIDIFile` in the python-twelve-tone project. After assertion amplification, 85 new assertions are generated for the complex MIDIFile object; 39 after initialization and 46 after the `create(notes)` method call. Upon investigation, only one

Code Excerpt 3.7: The test_create method before amplification, and the assertion pruned version

```
1    def test_create(self):
2        notes = [1, 2, 3, 4, 5, 6,
             7, 8, 9, 10, 11]
3        path = 'tmp'
4        os.makedirs(path, exist_ok=
             True)
5        os.chdir(path)
6        m = MIDIFile(filename='test
             .mid')
7        m.create(notes)
8        self.assertTrue(os.path.
             exists(os.path.join(os.
             getcwd(), 'test.mid')))
9        os.chdir(os.pardir)
10       shutil.rmtree('tmp',
             ignore_errors=True)
```

```
1    def test_create(self):
2        notes = [1, 2, 3, 4, 5, 6,
             7, 8, 9, 10, 11]
3        path = 'tmp'
4        os.makedirs(path, exist_ok=
             True)
5        os.chdir(path)
6        m = MIDIFile(filename='test
             .mid')
7        m.create(notes)
8        self.assertEqual(m.pattern.
             tracks, [[[0, 60, 200,
             1], [1, 61, 200,
9            1], [2, 62, 200, 1],
                 [3, 63, 200, 1],
                 [4, 64, 200, 1],
                 [5, 65,
10           200, 1], [6, 66, 200,
                 1], [7, 67, 200,
                 1], [8, 68, 200,
                 1], [9,
11           69, 200, 1], [10, 70,
                 200, 1]]])
12       self.assertTrue(os.path.
             exists(os.path.join(os.
             getcwd(), 'test.mid')))
13       os.chdir(os.pardir)
14       shutil.rmtree('tmp',
             ignore_errors=True)
```

assertion generated after the `create(notes)` method call alone kills the 13 mutants needed to reach 100% mutation score. The test should, after assertion pruning, look like the right-hand side of listing 3.7.

Additionally, some developers define customized assertions or helper methods that contain a set of assertions for specific types. The test generator should be consistent with such a coding style, which it currently doesn't do. A possible solution is using the profiling module to profile how an object is asserted in the existing test suite. Using this mechanism, the assertion amplification module would learn from developers how to assert objects and would produce similar code.

Helper methods needs to be considered. Developers frequently use helper methods in their tests. A helper method is another method defined in the test suite, but it is not a test method. It can be defined for different reasons like grouping a set of assertions to

reuse them in different test methods, or setting up an object and bringing it to a particular state.

If a helper method includes assertion statements, it is necessary to consider stripping them before input amplification. In our experiments, for example the class `AddJsTestCase` uses such helper method. As a result, the amplification of this class has not been successful, because the majority of input amplified tests were failing. Helper methods that are defined for altering inputs also need to be considered in input amplification for increasing space exploration.

In our experiments' dataset, helper methods were only used in the above mentioned `AddJsTestCase`.

The presence of a helper method is one of the main differences between a test generation and a test amplification tool. In test generation, the tool starts to generate the tests from scratch and the structure of a test is defined by the tool. However, in test amplification, the tool needs to adopt the style of the existing test suite and consider its elements.

Idiomatic Python code. Test amplification is a program synthesis task which generates new test methods based on existing ones. The generated test methods needs to be merged into the code base to make the effect of improved tests permanent. Therefore, it is important that the generated code is readable and adheres to the idiomatic coding conventions [99].

In the current implementation of AmPyfier, the test method are changed via input amplification operators which make subtle changes to the original test code. So, if the original test code follows Python coding idioms, the transformed code is likely to follow the same idioms. Additionally, when generating new method calls and asserting values, AmPyfier respects the Python conventions in accessing private and protected members. If we decide to extend AmPyfier with more complex input amplifiers, then the idiomatic nature of the code transformation should be taken into account.

Nevertheless, AmPyfier currently does not choose meaningful names for the generated test methods, as for instance done by Nijkamp et. al. [62]. The tool also adds plenty of assertions, some of which are irrelevant, which has a negative effect on the readability of the generated code. So, the produced test code is not immediately ready to be merged to the code base and it need a revision by a developer.

Continuous test amplification may help. Lastly, as already briefly touched upon, the main downside to AmPyfier, and test amplification in general, is its complexity. It is a time-, and resource-consuming task. At worst, each test has to be executed the amount of mutants times the amount of amplified test generated times, to score them all. Further-

more, multiple runs of the test are needed during assertion amplification to evade flaky tests, or to observe tests after an exception occurred. This makes it difficult to introduce test amplification into realistic development environments.

Therefore, we suggest employing AmPyfier in the Continuous Integration pipeline to be triggered periodically or in each code push to amplify the recent changes. In this case, the mutants would be generated only for the changed portions of code, and consequently the amplified tests should cover the changes. This can significantly reduce the load of execution. The amplified tests and other reports can be exported as artifacts, or they can be automatically sent as pull requests to be reviewed by the developers.

Time-based flaky tests can render faulty test methods. AmPyfier counters the generation of flaky tests through observing the test method F times as explained in subsection 3.3.3 and removing the non-deterministic results. However, sometimes a function returns the same results for a certain amount of time and then changes its return value (e.g. an API call to a server to ask when it was last updated). If this update does not happen during the F observations, AmPyfier will generate assertions based on the value of when it observed the test method. On a later execution of the test method, either during the mutation testing stage of AmPyfier, or once the test has been added to the amplified test class, the test will fail. AmPyfier tries to reduce this threat through the removal of faulty test methods before the selection process, however, the test method could still be valid at this time.

Impure getter functions or attributes during observation. When building object observations improper getter, or `getattr` and `__getattribute` methods could alter the state of the observed objects and thus cause unexpected (for the developer) results in the generated assertions. While unexpected for the developer, the amplified test methods would not fail, as the assertions are generated in the same order as the observations. The improper functions or methods would have the same effect in a handwritten test method. This can be seen as related to the “main pain point” for test amplification/generation. The techniques expect a properly implemented project. Still, the amplified tests could prove useful, because a developer, upon inspection of the amplified tests, could notice the unexpected behavior.

3.4.2 Threats to Validity

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [105, 106]), we organise them into four

categories.

Construct validity: do we measure what was intended? We measure the improvement of an amplified test class based on the increase in code coverage and mutation score. While there is debate around the effectiveness of these metrics, both are often used as adequacy criteria for unit test suites. These metrics provide quantitative evidence for the quality of the amplified tests, hence are sufficient at this stage of the research. Yet to truly measure whether the amplified tests add value, we should collect qualitative evidence from actual developers maintaining these projects.

Internal validity: are there unknown factors which might affect the outcome of the analyses? To evaluate the performance of the multi-metric selection, we use the execution time of the tool. However, the execution time may depend on different factors such as the number of CPU cores and the available RAM on the system. To reduce these factors, we ran our experiments in the same system (An AWS server with 2 CPUs and 8 GB RAM) with different configurations.

Another aspect influencing the results is the automatic selection of the module under test. As explained in Section 3.4, AmPyfier will score the test class against the project module most used in the test class. AmPyfier however has no way to ensure that this is indeed the intended module under test. This implies that during test selection, the coverage and mutation scores might be computed against the wrong module. In a realistic setting, this risk can be addressed by configuring AmPyfier with the appropriate traceability links.

External validity: to what extent is it possible to generalise the findings? We have demonstrated that test amplification is feasible for Python thanks to the generalisation of dynamic type profiling. AmPyfier has been evaluated on 54 test classes belonging to 11 projects with sufficient diversity in project characteristics (see Tables 3.4 and 3.5). However, these projects are mainly libraries and components and the tests were typical unit tests thereof. Whether these results hold for other projects (with a database, with heavy math, ...) remains to be seen.

AmPyfier has shown that the runtime efficiency of test amplification can be increased thanks to the introduction of multi-metric selection. While this is only tested for Python, similar results are expected for other languages and mutation frameworks.

Reliability: is the result dependent on the tools? To compute the coverage, and mutation score AmPyfier is dependent on specific tools. For mutation testing tools, the

mutation operators they support can differ greatly. Furthermore, not all tools are able to only create mutations for covered code, thus minimizing the effect of multi-metric selection on the time efficiency.

AmPyfier further makes use of randomness multiple times during input amplification. Some input amplifiers depend on randomness, such as the `add-call-amplifier`. It will randomly generate the parameters for a newly added method or function call. When it needs to generate a number as parameter, for example, this could be a large or small, positive or negative number. Dependent on that number, a new mutant could be killed or missed. Furthermore, if during a round of input amplification, more than N test methods are generated, as explained in subsection 3.3.5, a random subsection of those test methods is selected. Those tests are then passed to the next input amplification round and are scored at the end. As such, the possibility exists that interesting test methods are disregarded before scoring.

It could thus be that if we rerun the evaluation (Table 3.6), a slightly higher, or smaller, number of test classes will see improvements in regard to their coverage or mutation score. However, the goal of the evaluation was to show that test amplification, thanks to the generalisation of dynamic type profiling, is feasible for Python, and as such the randomness does not deny our results.

In our experiment regarding multi-metric selection, as shown in Table 3.3, we see the influence of randomness between both runs, but the amount of newly killed mutants is similar in both runs, sometimes in favour of one run, other times in favour of the other. The experiment shows that multi-metric selection drastically reduces the time-cost, and upon investigation, the differences in newly killed mutants can all be traced back to randomness. As such, our conclusion still holds. Multi-metric selection reduces the time-cost without affecting the effectiveness.

3.5 CONCLUSION

In this chapter, we introduced AmPyfier, a test amplification tool for Python, a dynamically typed and interpreted language. To build the tool we heavily relied on the design of DSpot, a test amplification tool for the statically typed language Java. Yet, to overcome the shortcomings of a dynamically typed language, we incorporated type profiling as used by Small-Amp for the dynamically typed language Smalltalk. However, type profiling in Small-Amp heavily relied on Pharo specific Metalinks, with AmPyfier we have generalised this concept. For Python specifically, we leveraged the fact that it is an interpreted language, where lots of information is available at runtime. Furthermore, we have introduced multi-metric selection in order to increase the time efficiency of test amplification. With multi-metric selection, mutants are only generated for covered code

in the project under test and amplified test methods are scored against both code coverage and mutation score. If an amplified test method covers new code, new mutants will be dynamically generated.

We evaluated AmPyfier on 11 open-source projects, and found that the tool could successfully strengthen 37 out of 53 test classes in regard to code coverage and mutation score. Furthermore, multi-metric selection decreased the time-cost by 17% to 98% as opposed to selection based on full mutation score. We collected qualitative evidence from the cases where AmPyfier failed to strengthen the test suite, deriving lessons learned and sketching areas for further improvement. Despite these shortcomings, we conclude that test amplification is feasible for one of the most popular programming languages in use today.

3.6 EVALUATED PROJECTS

Project (GitHub)	**Age (year)**	**Stars**	**Forks**	**Contributors**	**LOC**	**Test LOC**
mewwts/addict	8	2.2k	133	27	142	443
Python-Markdown/markdown	12	2.8k	713	134	3855	2308
lepture/mistune	8	2k	210	36	1366	308
eagleflo/mpyq	12	87	28	6	289	43
appditto/pippin_nano_wallet	3	52	17	8	2107	47
eatonphil/pj	4	71	11	2	167	25
dhagrow/profig	8	22	0	1	814	498
mstamy2/PyPDF2	10	4k	1k	68	3643	52
accraze/python-twelve-tone	6	68	5	8	101	60
seatgeek/thefuzz	1	431	31	1	357	325
richardpenman/whois	3	172	99	30	1273	255

Table 3.4: Projects Amplified with AmPyfier

Project	Test Classes
Addict	DictTests, ChildDictTests
markdown	RegistryTests, TestAbbr, TestAdmonition, TestAncestorExclusion, testAtomicString, TestBlockAppend, TestBlockParser, TestBlockParserState, TestCaseWithAssertStartsWith, TestCliOptionParsing, TestConfigParsing, TestConvertFile, testElementTailTests, TestErrors, TestEscapeAppend, testETreeComments, TestExtensionClass, TestGeneralDeprecations, TestHtmlStash, TestMarkdownBasics, TestMetaData, testSerializers, TestSmarty, TestTOC, TestVersion, TestWikiLinks
mistune	TestAstRenderer, TestMiscCases, TestPluginAdmonition, TestPluginDirective
MPyQ	TestMPQArchive
Pippin Nano Wallet	TestAESCrypt, TestNanoUtil, TestRandomUtil, TestValidators, TestWalletUtil
PJ	TestStringMethods
profig	TestBasic, TestStrictMode
PyPDF2	PdfReaderTestCases, AddJsTestCase
Python Twelve Tone	TestMatrix, TestMIDIFile
TheFuzz	ProcessTest, RatioTest, StringProcessingTest, TestCodeFormat, UtilsTest, ValidatorTest
Whois	TestExtractDomain, TestParser, TestNICClient

Table 3.5: Classes Amplified with AmPyfier

3.7 EVALUATION RESULTS

Legend: T = Execution time of AmPyfier; MO = Number of methods originally in the test class; MA = Number of amplified methods added to the amplified test class; LOC = Number of lines of code; CSO = Coverage score of the original test class; CSA = Coverage score of the amplified test class; RCSI = Relative increase in coverage score after amplification; MSO = Mutation score of the original test class; MSA = Mutation score of the amplified test class; RMSI = Relative increase in coverage score after amplification

TestClass	T	MO	MA	LOC	CSO	CSA	RCSI	MSO	MSA	RMSI
TestRandomUtil	00:00:00	1	0	7	100.00%	100.00%	0.00%	100.00%	100.00%	0.00%
TestVersion	00:01:50	2	1	13	100.00%	100.00%	0.00%	**91.53%**	**93.22%**	**1.85%**
TestValidators	00:02:14	2	3	39	100.00%	100.00%	0.00%	**86.57%**	**91.04%**	**5.17%**
ChildDictTests	01:10:04	63	2	130	100.00%	100.00%	0.00%	**80.52%**	**83.12%**	**3.23%**
DictTests	00:53:11	63	2	130	100.00%	100.00%	0.00%	**80.52%**	**83.12%**	**3.23%**
TestCliOptio ...	00:10:01	16	2	41	100.00%	100.00%	0.00%	**75.44%**	**80.70%**	**6.89%**
StringProcess ...	00:00:21	2	0	15	100.00%	100.00%	0.00%	57.14%	57.14%	0.00%
TestMIDIFile	00:00:13	2	1	18	100.00%	100.00%	0.00%	**27.78%**	**100.00%**	**260.00%**
TestAESCrypt	00:00:31	1	4	29	**96.55%**	**100.00%**	**3.57%**	**93.06%**	**98.61%**	**5.96%**
TestNanoUtil	00:00:00	1	0	40	87.50%	87.50%	0.00%	29.27%	29.27%	0.00%
TestMatrix	00:03:58	5	1	55	83.64%	83.64%	0.00%	**90.32%**	**91.94%**	**1.79%**
ProcessTest	00:41:28	14	22	329	**76.60%**	**78.72%**	**2.78%**	54.25%	**82.53%**	**52.12%**

TestClass	T	MO	MA	LOC	CSO	CSA	RCSI	MSO	MSA	RMSI
TestStringM ...	00:08:26	10	6	152	**75.00%**	**78.95%**	**5.26%**	**87.56%**	**89.78%**	**2.54%**
TestMPQArc ...	00:10:04	5	2	262	65.27%	65.65%	0.58%	**81.92%**	**83.38%**	**1.78%**
TestMiscCases	00:17:15	9	3	679	**62.74%**	**66.27%**	**5.63%**	**85.13%**	**92.48%**	**8.64%**
TestTOC	02:29:50	15	18	1,672	**61.30%**	**67.17%**	**9.56%**	**79.88%**	**92.93%**	**16.39%**
RatioTest	00:44:09	26	24	329	**60.49%**	**62.92%**	**4.02%**	**56.53%**	**73.86%**	**30.65%**
TestPluginD ...	00:05:22	3	6	679	**59.50%**	**62.74%**	**5.45%**	**95.96%**	**98.07%**	**2.20%**
TestStrictMode	00:13:26	4	8	737	**58.62%**	**61.19%**	**4.40%**	**35.11%**	**38.42%**	**9.42%**
TestPluginA ...	00:07:57	8	5	679	**56.85%**	**58.32%**	**2.59%**	97.27%	97.27%	0.00%
TestAncestor ...	00:22:42	2	6	1,713	**56.74%**	**59.19%**	**4.32%**	**89.77%**	**95.02%**	**5.84%**
TestAbbr	00:08:15	2	6	1,672	**55.92%**	**57.83%**	**3.42%**	**96.18%**	**99.31%**	**3.26%**
TestWikiLinks	00:33:50	6	7	1,672	**53.41%**	**54.55%**	**2.13%**	**92.32%**	**96.38%**	**4.40%**
TestParser	01:08:56	10	11	762	**53.15%**	**53.87%**	**8.89%**	**47.70%**	**60.94%**	**28.37%**
testSerializers	00:56:59	12	3	1,713	**50.38%**	**50.96%**	**1.16%**	**93.08%**	**96.18%**	**3.33%**
TestSmarty	00:12:41	1	5	1,672	**50.24%**	**50.54%**	**0.60%**	**78.48%**	**79.40%**	**1.17%**
TestBasic	01:14:35	18	10	737	**49.39%**	**50.34%**	**1.92%**	**50.36%**	**54.68%**	**8.57%**
TestMetaData	01:27:13	5	9	1,672	**47.85%**	**54.07%**	**13.00%**	**76.23%**	**87.18%**	**14.36%**
TestConvertFile	00:14:46	3	1	1,783	**46.12%**	**46.18%**	**0.13%**	94.42%	94.42%	0.00%
TestMarkdo ...	00:19:34	6	4	1,713	**45.94%**	**46.53%**	**1.27%**	**89.73%**	**94.50%**	**5.32%**
testAtomicSt ...	00:18:17	3	7	1,713	**43.49%**	**44.25%**	**1.74%**	**91.83%**	**94.86%**	**3.29%**
TestNICClient	00:08:28	1	7	263	**43.35%**	**44.87%**	**3.51%**	**54.74%**	**81.47%**	**48.82%**
TestBlockParser	00:13:27	2	4	1,713	**37.89%**	**38.70%**	**2.16%**	**90.71%**	**93.81%**	**3.41%**
TestExtractD ...	01:12:52	7	2	1,107	**37.49%**	**37.58%**	**0.24%**	**43.88%**	**48.98%**	**11.63%**
TestAstRenderer	00:00:34	13	0	679	34.61%	34.61%	0.00%	99.30%	99.30%	0.00%
PdfReaderTe ...	01:53:08	2	0	3,102	34.46%	34.46%	0.00%	89.27%	89.27%	0.00%
testETreeCo ...	00:04:44	4	0	1,713	33.57%	33.57%	0.00%	92.37%	92.37%	0.00%
TestErrors	00:06:40	6	1	1,713	**33.33%**	**33.80%**	**1.40%**	93.21%	93.21%	0.00%
TestAdmonition	00:06:02	1	0	1,672	33.25%	33.25%	0.00%	90.16%	90.16%	0.00%
testElement ...	00:04:40	1	1	1,713	**32.69%**	**32.75%**	**0.18%**	91.41%	91.41%	0.00%
AddJsTestCase	03:19:31	2	1	3,102	**32.14%**	**32.17%**	**0.10%**	86.44%	86.44%	0.00%
TestEscapeA ...	00:05:13	1	0	1,713	31.93%	31.93%	0.00%	91.62%	91.62%	0.00%
TestBlockAp ...	00:05:17	1	0	1,713	31.39%	31.39%	0.00%	91.62%	91.62%	0.00%
UtilsTest	00:01:26	4	0	329	29.79%	29.79%	0.00%	19.23%	19.23%	0.00%
ValidatorTest	00:03:30	2	0	329	28.27%	28.27%	0.00%	34.65%	34.65%	0.00%
RegistryTests	00:53:20	12	0	1,713	27.32%	27.32%	0.00%	92.44%	92.44%	0.00%
TestCodeFormat	00:09:34	1	0	329	26.44%	26.44%	0.00%	8.45%	8.45%	0.00%
TestConfigPa ...	00:07:55	3	3	1,713	25.80%	25.80%	0.00%	**92.18%**	**93.20%**	**1.11%**
TestHtmlStash	00:06:07	3	1	1,713	25.74%	25.74%	0.00%	**91.36%**	**91.76%**	**0.41%**
TestBlockPar ...	00:11:52	4	0	1,713	25.39%	25.39%	0.00%	91.82%	91.82%	0.00%
TestGeneralD ...	00:04:25	1	0	1,713	25.10%	25.10%	0.00%	91.47%	91.47%	0.00%
TestCaseWit ...	00:01:25	1	0	1,672	25.06%	25.06%	0.00%	91.09%	91.09%	0.00%
TestExtensio ...	00:14:21	7	2	1,672	**25.06%**	**25.54%**	**1.91%**	**81.23%**	**88.45%**	**8.89%**
TestWalletUtil	00:18:44	3	0	187	23.53%	23.53%	0.00%	12.90%	12.90%	0.00%

Table 3.6: The result of running AmPyfier on 54 test classes

Part II

Toward Zero-touch Test Amplification

Chapter 4

Steps Towards Zero-touch Test Amplification

Steps Towards Zero-touch Test Amplification

Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel

This chapter is submitted to: *International Conference on Software Testing, Verification and Validation* (*ICST 2023*).

ABSTRACT

Test amplification exploits the knowledge embedded in an existing test suite to strengthen it. A typical test amplification technique transforms the initial tests into additional test methods that increase the mutation coverage. Although past research demonstrated the benefits, additional steps need to be taken to incorporate test amplifiers in the everyday workflow of developers. This chapter explains how we integrate Small-Amp *with* GitHub-Actions *to introduce* zero-touch test amplification*: a test amplifier that decides for itself which tests to amplify and does so within a limited time budget. To attain zero-touch test amplification, we incorporate three special-purpose features: (i) prioritization (to fit the process within a given time budget), (ii) sharding (to split lengthy tests into smaller chunks), and (iii) sandboxing (to make the amplifier crash resilient). We evaluated our approach by installing a zero-touch extension of* Small-Amp *on five open-source Pharo projects deployed on GitHub. Our results show that zero-touch test amplification is feasible at a project level by integrating it into the build system. Moreover, we quantify the impact of prioritization, sharding and sandboxing so that other test amplifiers may benefit from these special-purpose features.*

4.1 INTRODUCTION

Unit testing is writing small pieces of executable code to exercise the program's units and ensure they work as intended. Even though writing these unit tests is initially a tedious process, it prevents the system under test from regressing in the long term. A common way to evaluate the strength of a test suite is to measure code coverage or mutation coverage [36]. Since manually covering all corner cases of a program is a challenging task, automated test generation [64, 84, 107] and test amplification [35, 78, 85, 91, 108, 109, 110] tools were investigated to create stronger test suites. These tools analyze the program under test and produce new test methods that permanently increase coverage if merged into the codebase.

SMALL-AMP (introduced in Chapter 2) is the state-of-the-art test amplification tool in the Pharo language. It extends DSPOT [85] by bringing test amplification to dynamically typed languages. Both tools work as a recommender system that synthesize new test methods and presents them to developers, which then decide whether these tests are worthwhile to be merged into the codebase. Qualitative studies on DSPOT and SMALL-AMP illustrate that developers value the generated tests and accept the corresponding pull requests [87, 108].

Although the results of test amplification tools are promising, their practical application is still questionable. Past research shows that test amplification tools are cumbersome [111]; not only are they complex and hard to configure, but their execution time is unpredictable and sometimes even unacceptable. For instance, considering test amplifiers employing mutation testing, the amplification of some test classes require 5+ hours in DSPOT [108], 5+ hours in DCI [109], 2+ hours in SMALL-AMP [87], and 3+ hours in AMPYFIER [110].

$\Longrightarrow$ *Although test amplification tools emerged to support developers, the complexity and long execution times hinder their adoption.*

Brandt and Zaidman [61] employ a lighter version of DSPOT in an IDE and introduce developer-centric test amplification. Because of the time cost consideration, they restrict test amplification to increase the instruction coverage and skip amplifying the mutation coverage. This confirms that developers' workstations are unsuitable for comprehensive mutation-based test generation: it is impossible to provide instantaneous feedback.

$\Longrightarrow$ *Executing mutation-based test amplifiers on a developer workstation is seldom feasible due to the computational overhead.*

In contrast, the work by Campos et al. [112] and Danglot et al. [109] employ *continuous integration servers* to exploit automated tests. The former integrates EvoSuite [84] (a test generation tool for Java) within a continuous integration setting to optimize the test

generation. The latter runs a variation of DSpot (named DCI) to detect the behavioral changes on each commit in continuous integration.

$\Longrightarrow$ *Continuous integration servers, running on powerful servers configured in build farms, open up possibilities for improved test synthesis.*

A long term possibility is to allow for fully autonomous —*"zero-touch"*— testing, as envisioned in Level 5 of the Test Automation Improvement Model [12]. In this vision, a test amplifier will decide for itself which tests to amplify, incorporate the synthesized tests in a separate branch, execute the strengthened test suite and —if all steps pass— push the strengthened test suite onto the main branch. All without any intervention of a software engineer.

$\Longrightarrow$ *Zero-touch test amplification, where amplified tests are automatically added, is the ultimate vision for fully autonomous test synthesis.*

One issue preventing fully autonomous test synthesis is that mutations in the code may result in system crashes [91]. Especially in live systems such as Pharo, system crashes corrupt the system image beyond repair. If a crash happens, the system must revert back to a state where the system is known to be pure.

$\Longrightarrow$ *A crash resilient test synthesis process is a necessary prerequisite for zero-touch test amplification.*

In this chapter, we explore the feasibility of zero-touch test amplification. We present a proof-of-concept tool that integrates Small-Amp with GitHub-Actions to fully automatically strengthen the existing test suite within a limited time budget. To this end, our proof-of-concept incorporates three special-purpose features: (i) prioritization (to fit the process within a given time budget), (ii) sharding (to split lengthy tests into smaller chunks), (iii) and sandboxing (to make the amplifier crash resilient).

We evaluated our approach by installing a zero-touch extension of Small-Amp on five open-source Pharo projects deployed on GitHub. Our results show that zero-touch test amplification is feasible at a project level by integrating it into the build system. Moreover, we quantify the impact of sharding, prioritization, and sandboxing so that other test amplifiers may benefit from these special-purpose features. Our experiments show that prioritization has better performance (up to a 34% increase), crashes occurred in about 17% of the cases, and are restored successfully by sandboxing mechanism, and sharding allowed for large classes to fit into our time budget but came at a cost of 30% more duplicated mutants. Additionally, our new time budget-aware process was able to finish the amplification in an acceptable period of 30 to 90 minutes.

The remainder of the chapter is organized as follows. Section 4.2 provides the necessary background to understand the challenges of zero-touch test amplification. Sec-

tion 4.3 explains how we integrate SMALL-AMP with GITHUB-ACTIONS, and details the sharding, prioritization, and crash recovery features. Section 4.4 presents the quantitative results of the evaluation on five projects. Section 4.5 enumerates the threats to validity. Section 4.6 provides an overview of the related work which inspired this proof-of-concept. Finally, we summarise the main conclusions in Section 4.7.

4.2 TEST AMPLIFICATION

Modern software repositories contain a considerable amount of tests. These tests are written mostly by developers who have deep knowledge and understanding of the program. The main idea in *test amplification* [8] is exploiting this valuable resource of knowledge to improve the test suite.

In SMALL-AMP (Chapter 2), the state-of-the-art test amplifier in the Pharo ecosystem, this improvement is achieved by synthesizing new test methods which will permanently increase the mutation coverage when merged into the codebase. SMALL-AMP is a replication of DSPOT [108] in the dynamic language of Pharo [113, 114].

4.2.1 Amplification Algorithm

SMALL-AMP (as well as DSPOT) iterates over all test methods in a test class and applies the following operations.

Input amplification transforms the original test method using a set of input amplifiers and generates new versions of the test method. Usually, some of these versions of the test method bring the program under test to an untested state or take a different execution path from the original test method, leading to killing the live mutants. However, the original test method usually contains some assertion statements to verify the intended state. Since these assertion statements are no longer valid in the transformed versions, SMALL-AMP removes the original statements before the transformation.

Assertion amplification regenerates appropriate assertions to verify the actual state of the program by manipulating the generated test method and inserting observing statements. In this step, the test method is executed, and SMALL-AMP logs the actual state of the program using the object inspection. SMALL-AMP generates new assertion statements using these logs and adds them in place of observer statements. The new assertions should all pass for the version of the code they were generated for.

Selection by mutation score. Up to this step, we have new versions of the original test method that were transformed by input amplifier and equipped with new assertion statements by assertion amplifier. In this step, mutation testing is run on the program under test using these generated test methods. Test methods that increase the mutation cov-

erage by killing new mutants are kept, and the remaining test methods are discarded. SMALL-AMP relies on Mutalk [45], a test amplification platform for Pharo.

4.2.2 Challenges For Test Amplification

In this section, we identify the main challenges faced by test amplification tools, SMALL-AMP in particular, to be more practical and incorporate them into the daily workflow of developers.

Using test amplification tools is cumbersome. In addition to writing code and tests, developers are usually busy with other activities like meetings, bug fixing, emails, networking, learning, documentation, helping others, administration tasks, and others [115, 116, 117]. Test amplification tools are complicated and hard to configure, and using them needs deep knowledge about different topics like mutation testing [111]. If we expect developers to run the tool on their workstations, each developer will need to deal with some extra tedious tasks.

Current test amplification tools do not support time budget management. Test amplification execution time varies from test class to test class, and estimating it in advance is difficult; amplification tools usually have long execution times and need considerable processing resources. It is inconvenient for developers to employ these tools in their workstations, dedicating the entirety of their resources to test amplification and waiting hours or days for a test amplification run to completion. Setting a time limit is necessary for such a long process. On the other hand, the current test amplification tools lack a mechanism to prioritize their tasks for gaining the maximum benefit when running on a time budget.

Test amplification in live systems is more challenging. SMALL-AMP amplifies programs written in Pharo, and Pharo is a live programming environment [15]. Pharo offers the notion of *liveness* [16] which greatly impacts how developers work: system always offers an accessible evaluation of a source code instead of the classical edit-compile-run cycle, and as a consequence, the live programming environment allows for nearly instantaneous feedback to developers instead of forcing them to wait for the program to recompile [17].

Ducasse et al. [118] identify the challenges of supporting automated testing tools in Pharo, and they mention executing destructive methods in random testing as a challenge and emphasize the need for sandboxing. During the amplification process, SMALL-AMP works with two different kinds of mutations: the mutation on the production code (mutation testing), and the mutation on test methods (input amplification); and each mutation applies a random change to the code. Executing such random code in a live system introduces two major challenges:

- Random code easily leads to infinite loops/recursions and deadlocks. Worse, it is possible to call critical methods (terminating the virtual machine and unloading

class), leaving the live system in an unsafe state. Consequently, an image crash or freeze is more prone to happen during amplification [91]. In a live system, a crash means we no longer have the amplification state in which the previously amplified results were stored.
- Random code may pollute the internal state of a system, resulting in flaky tests [119, 120]. For example, suppose an object is cached in a class side variable (static variable). Developers expect this cached value to be immutable, but it may be altered unexpectedly during mutation testing. As a result, all tests depending on this cached value will fail after the mutation testing while passing before. This pollution will remain in the live system forever and may cause side effects on the generated tests.

⇒ **Zero-touch Test Amplification** may alleviate these challenges. Instead of asking developers to run a completely configured tool on a desktop computer, we can embed the tool in a fully autonomous process on the continuous integration servers. Instead of running the tool until completion, no matter how long it takes, we can change the base algorithm to run in a given time-budget and optimize accordingly. Instead of restarting the process after a crash in a fresh unpolluted state, we can run the tool in a sandbox environment to be able to circumvent (even reproduce) the crash.

4.3 ZERO-TOUCH PROOF-OF-CONCEPT

In this chapter, we explore the feasibility of zero-touch test amplification. We present a proof-of-concept tool that integrates SMALL-AMP with GITHUB-ACTIONS to fully automatically strengthen the existing test suite within a limited time budget.

4.3.1 SmallAmp and Pharo

Why Small-Amp? In principle, we could have chosen any test amplification tool for our proof-of-concept. We decided to focus on tools used in dynamic languages given the popularity of such languages among practitioners. For instance, JavaScript was the most popular language on StackOverflow.[1] At the time of the research, we were not aware of any test amplification tool for JavaScript. Therefore, we chose SmallAmp [87] because it was a recent tool for the dynamic language Pharo Smalltalk. Moreover, Pharo presented more challenges due to its live programming environment that we deemed interesting to investigate.

[1] https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

What is interesting about Pharo? Pharo is a Smalltalk-based object-oriented dynamic-typed language. Pharo also includes a programming environment, integrated with development tools, a run-time virtual machine, and live debugging features. Pharo is not "file-based" as programmers work directly in an Image which is a live environment that stores the code, the states manipulated by the code, and the current execution [114].

As a simple analogy, we can think of the Pharo image as an Operating System and IDE rolled into one container that becomes a live programming environment. This *liveness* offers more challenges for test amplification (as we previously explained in Section 4.2.2) which we consider interesting to explore in this research. Moreover, if our proof of concept works for a more challenging scenario, it will be possible to adapt it to simpler situations.

4.3.2 Integration with GitHub-Actions

Why GITHUB-ACTIONS? We adopted the GITHUB-ACTIONS build system as a suitable platform to build a proof of concept automated test amplification tool for several reasons. (1) A build system can be configured once and used by all contributors in a project or even multiple projects. (2) A build system can trigger the test amplification based on relevant events like each pull request, scheduled like running per week, or manually when needed. (3) A build system executes on the Continuous Integration Servers, freeing developers' machines from the computation. (4) GITHUB-ACTIONS defines a language for defining workflows and which allows for parallelization. (5) Build system has become more popular in recent years [121, 122, 123]. (6) Most well-known open-source Pharo projects are hosted on GitHub, and GITHUB-ACTIONS is freely available for open-source projects [124].

How does GITHUB-ACTIONS work? GITHUB-ACTIONS is based on workflows, and each workflow contains one or more jobs that can be run in parallel or sequential. Each job starts a new operating system instance in a virtual machine or container and performs some steps. Each step may run a terminal command or use a private or public custom action [125]. Workflows can be triggered by predefined events like when a new code is pushed, merged, or based on a schedule. By default, the return value from a workflow run is only the state of success or failure. However, GITHUB-ACTIONS supports creating artifacts to persist additional data [126]. GITHUB-ACTIONS also allows defining reusable workflows [127], which facilitate the workflow maintenance on the users' side.

Small-Amp **integration to GitHub-Actions** For integrating Small-Amp, we define a reusable workflow[2], and also a GitHub-Actions custom action[3] to setup a Pharo instance and run Small-Amp in it. Developers in the user projects need to define a workflow that calls the reusable workflow and pass some main configuration parameters. Some of the essential parameters required to be configured by the users are the number of parallel jobs and the project loading parameters. If the workflow is triggered by a push or pull request, the test amplification tool will consider all changes in the commit; but if triggered manually or by schedule, it amplifies the entire project or the specified classes.

The workflow contains three sequential phases. Each phase is composed of a job or a set of similar jobs that run in parallel. Since each job starts on a clean operating system, Pharo is installed first, and then Small-Amp and the project-under-test are loaded in Pharo.

The first phase is *prescreening*, which consist of a single job with the following steps:

1. Small-Amp scans all defined test classes in the project and attempts to detect the class under test by its default heuristic. (Details are in Chapter 2).
2. If a class contains too many test methods, the test optimization will perform poorly. Small-Amp therefore shuffles its test methods and breaks it into smaller temporary test classes (i.e. *sharding* Section 4.3.4).
3. Small-Amp assigns test classes different job identifiers to be distributed over jobs in the next phase.

The second phase is *amplification*, which consists of multiple parallel jobs. Each job iterates over its assigned test classes and performs the following steps:

1. It creates a sandbox for each test class. The amplification tool is executed within a sandbox to make it *crash resilient* (see Section 4.3.5).
2. It enforces a maximum time budget for each test class (like 15 minutes) to ensure that the amplification terminates in a predictable time (see Section 4.3.3).

The final phase is *merging*, in which a single job collects all output files from the amplification jobs, merges them, and exports the tool's outputs as *artifacts*. Developers should collect these artifacts after the workflow is finished and incorporate the amplified test methods. Since we validate our approach on popular Pharo projects, we explicitly opted to exclude this step from the fully automatic workflow, although it is straightforward to do so. In that sense, the proof-of-concept is not truly "zero-touch": we still need a human in the loop to accept the synthesized test.

[2] github.com/mabdi/small-amp/blob/master/.github/workflows/SmallAmpCI.yml
[3] github.com/mabdi/smallamp-action

4.3.3 Test-Method Prioritization

Since the execution time of the test amplification tool varies from test class to test class, we set a time limit on each amplification process to make it more practical. The test amplification algorithm introduced in DSpot and Small-Amp does not provide time budget management. In this section, we extend the Small-Amp algorithm by proposing a test-method prioritization heuristic to increase the algorithm's efficiency when executed within a limited time budget.

This heuristic is based on the intuition that a test method covering more live mutants has a better chance of killing them. Therefore, we count the number of live mutants in all covered methods by a test and compute a mutant coverage score for each test method. We also prefer to emphasize the immediate mutants because they are directly covered by the test, and auditing them is easier for developers. In object-oriented unit testing, a test method normally initializes an instance of class-under-test and invokes some of its methods. So, killing shallower mutants can be interesting because they are more likely to be in method-under-test or other important methods.

Finally, based on these scores, we calculate a weight for each test method and select one of them using the roulette wheel method [128]. We select an individual randomly in a roulette wheel selection, but the probability of this selection corresponds to its weight. The benefit of using this selection mechanism is to increase diversity in the selected methods by giving a chance to less-favored test methods of being selected.

Setting scores and weights We suppose that we have a function μ that returns the number of live mutants in each method under test:

$$\mu = \{m_1 \mapsto a_1, m_2 \mapsto a_2, ..., m_n \mapsto a_n\}$$

In addition, we have a directed graph $G = (V, E)$ for the method invocations. The vertices correspond to all test methods (T) and methods under test (M). There is also a directed edge from the node v to node v' if v invokes v'.

$$V = T \cup M$$
$$E = \{v \rightarrow v' | v, v' \in V \land v' \text{ is invoked from } v\}$$

We define the coverage set of the test method t as the set of all methods under test covered by t:

$$C_t = \{m | m \in M \land \exists\, p = (t \rightarrow ... \rightarrow m) \in \mathcal{P}(G)\}$$

In this relation, $\mathcal{P}(G)$ is the set of all paths in the graph G, and p is a path starting from t and ending in m. Similarly, we define the immediate coverage set (path length is 1) as:

$$I_t = \{m | m \in M \land \exists\, p = (t \rightarrow m) \in \mathcal{P}(G)\}$$

Now, the scoring function s is:

$$s(t) = \alpha + \beta \sum_{m \in I_t} \mu(m) + \gamma \sum_{m \in C_t - I_t} \mu(m)$$

The first part of this equation is the scoring offset. If $\alpha = 0$, all test methods not covering any mutant will be excluded from the amplification process. The second part of the equation is the immediate coverage score. As a result, we expect the mutants in these methods to be killed faster than deeper mutants. The third part of the equation is the coverage score. In this part, we consider all remaining mutants in other covered methods.

The variables α, β and γ are tuning parameters: For a default value, we choose $\alpha = 1$ to prevent excluding the test methods with no mutant coverage because these tests may be able to kill new mutants after some transformations. Since we prefer to prioritize the mutants in instantly covered methods, so we choose $\beta = 3, \gamma = 1$.

After calculating the score for each test methods, we set a weight for each test method as:

$$w(t) = \frac{s(t)}{S} \text{ ; where } S = \sum_{t \in T} s(t)$$

We use these weights to select a method to be amplified using a selection method called roulette wheel [128]. The scores and weights need to be updated in each cycle because the number of live mutants in the methods changes after each test amplification loop. Recalculating the weights does not have much overhead because the coverage graph does not need to be regenerated each time. We only need to update the μ function, and recalculate $s(t)$ and $w(t)$ for all remaining tests.

Changes in the Algorithm First of all, we update the *input amplification* and *assertion amplification* steps in SMALL-AMP to make them time budget observant: If the time limit is due, new test inputs will not be input/assertion amplified, and all the currently amplified instances will be returned. We also added a test method selection based on the weight assignment heuristic, and the roulette wheel method described earlier in this section. We present the updated time budget observant algorithm in Algorithm 5.

Algorithm 5: Updates in amplification algorithm to support time budget management

```
    input  : class-under-test CUT
    input  : set of test methods T
    input  : hyperparameters {N_iteration}
    output : set of amplified test methods ATM
 1  ALV ← mutationTesting(CUT, T);
 2  U ← amplifyAssertions(T);
 3  ATM ← {x ∈ U| x improves mutation score};
 4  ALV ← ALV − {x ∈ ALV| x is killed in ATM};
 5  t ← rouletteWheel(CUT, T, ALV);
 6  while t ≠ null do
 7  |   V = {t};
 8  |   for i ← 0 to N_iteration do
 9  |   |   V ← amplifyInputs(V);
10  |   |   U ← amplifyAssertions(V);
11  |   |   ATM ← ATM ∪ {x ∈ U| x improves mutation score};
12  |   |_  ALV ← ALV − {x ∈ ALV| x is killed in ATM};
13  |   T ← T − {t};
14  |_  t ← rouletteWheel(CUT, T, ALV);
15  return ATM
```

In the new algorithm, initially, we run mutation testing to calculate the live mutants (ALV). Then, we execute assertion amplification on all test methods to kill those mutants that can be killed only by expanding the assertion statements. Since a single assertion amplification is faster than the combination of input amplification and assertion amplification, this step does not need any selection based on the scores. We remove the newly killed mutants from ALV and select a random test to be amplified using the roulette wheel method (line 5). The main amplification loop runs on the selected test method t (lines 8 to 12). Then, the method is removed from the list of all test methods to be amplified T (line 13), and the weights will be recalculated based on the current live mutants. Then, a random test from the remaining unamplified tests will be selected considering their weight (line 14). If all test methods are visited or the time budget is due, the roulette wheel will return a null value, and then the final amplified test methods (ATM) will be returned.

4.3.4 Sharding

While experimenting on real projects with the time budgets, we witnessed that the number of test methods skipped in some classes is unacceptable because they include tens/hundreds of test methods. Using a variable time budget based on the test method numbers does not solve the problem thoroughly because amplifying some of these large classes may need more time than the jobs' allowed time (6h in GitHub-Actions). Therefore, we split long test classes into smaller temporary test classes. We call this action

Sharding.

We use a threshold (default 15 test methods) as a sharding factor. If a class contains more test methods than the defined threshold, SMALL-AMP shuffles its test methods and distributes them into smaller temporary test classes. In parallel jobs, all jobs must produce the same shards, so a shared randomization seed is required. We derive this seed from the *workflow run id* in our proof-of-concept.

Predicting the number of jobs The following formula estimates the minimum number of parallel jobs required for a successful test amplification:

$$J_{min} = \lceil \frac{b \times \sum_{c \in TC} \lceil \frac{t_c}{S} \rceil}{M} \rceil$$

where J_{min} is the minimum number of jobs required, b is the time budget per class, M is the maximum allowed execution time for each job by platform, t_c is the number of tests in the class c, S is the sharding factor, and finally, TC is the set of test classes to be amplified.

Therefore, the expression $\sum_{c \in TC} \lceil \frac{t_c}{S} \rceil$ shows the number of classes after sharding. We expect that all shards finish in the defined budget (b), so the fraction's numerator calculates the minimum time needed to amplify all tests in a single job. However, we suppose that the build system defines an execution limit on each job (M). Dividing the time needed to amplify all classes by the maximum job execution achieves the minimum number of required jobs.

4.3.5 Crash Resilience

SMALL-AMP is designed to run within a Pharo image, representing the complete state of the live system. Besides the system under test, the tool and its necessary components (the compiler, test runner, mutation testing framework, ...) run in the same Pharo image and therefore the same memory space. This architecture introduces a serious risk: if a crash happens, the whole Pharo process is lost, including the crashed component as well as the SMALL-AMP core.

A reliable test amplification tool running in a live environment, like Pharo, should be able to recover from these crashes without losing the entire state of the amplification process. Without a crash recovery mechanism, integration into build systems is impossible because any crash in SMALL-AMP will fail the entire workflow.

We enumerate some common reasons for an unexpected termination:

- *Killed by the operating system.* The Operating System may kill the Pharo process with an *Out of memory* error. This issue commonly happens in the mutation testing step

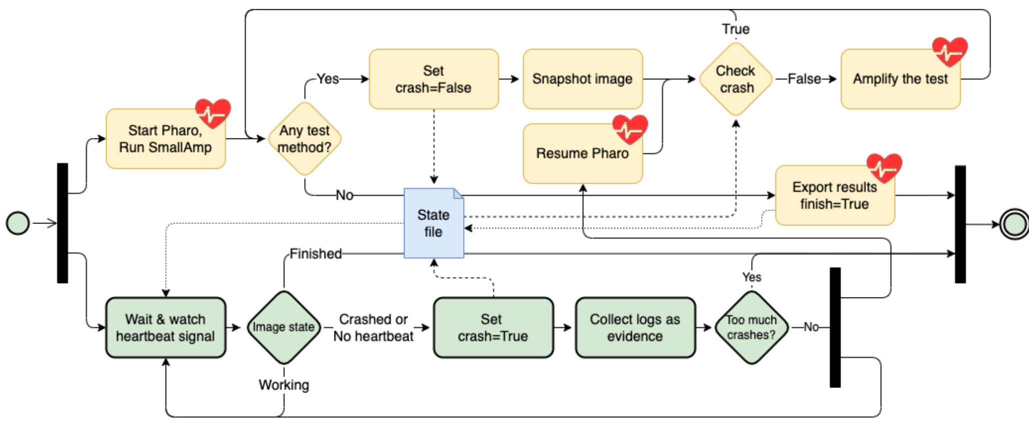


Figure 4.1: Activity diagram for a self-aware test amplification in a live system

when the process creates infinite recursion because of the injected fault.

- *Pharo process crashes.* The Pharo process is terminated unexpectedly with errors like *Segmentation fault* or *Assertion failed*. For example, in one case (`github.com/ObjectProfile/Roassal3/issues/142`)1 the Pharo process crashes because one of its native libraries has aborted.
- *Pharo process freezes or waits forever.* Pharo freezes because it executes a mutated test method that enters a deadlock and waits forever (`github.com/pharo-project/pharo/issues/6754`2, `github.com/feenkcom/gtoolkit/issues/1454`)3. Note that similar problems also happen in other tools such as DSpot (`github.com/STAMP-project/dspot/issues/994`)4.
- *Unwanted process termination.* A mutated test calls a critical method in the system API. For example in one case (`github.com/pharo-project/pharo-launcher/issues/454`)5, during input amplification, a method call is added that snapshots the image and exits with return code 0.

Consistent with previous studies [54, 55, 91], these crashes are interesting from a reliability perspective, and the pieces of code that broke Pharo can be used to reproduce the crash. So, besides recovering from the crashes, we collect sufficient information to allow developers to reproduce them.

How to recover from crashes? We use application heartbeats [129] to make test amplification self-aware [130] by detecting crashes and recovering from them. To this aim, we use a separate process (called the *runner script*) that initiates the amplification process in Pharo and watches its status. Figure 4.1 shows the activity diagram for describing how these two processes interact to detect the crashes and recover from them. The compo-

nents on top, which are colored in yellow and have a light border, are steps executed in the Pharo image by SMALL-AMP. The components on the bottom, which are colored in green and have a bold border, are steps executed in the runner script.

The runner script runs a Pharo process (child process) and initiates SMALL-AMP. SMALL-AMP regularly creates heartbeat signals by updating the content of a heartbeat file to notify that it is still functioning. The runner script also periodically watches the update time of the heartbeat file to make sure that the child is alive and works as expected.

Before starting to amplify a test method, SMALL-AMP sets a flag *crash* to False in a shared file (state file) and snapshots the current state of the Pharo image. Then, it loads the state file and checks the value of the *crash* flag. If it is the same stored value (False), it starts to amplify the test method.

If a crash happens while amplifying a test method, the Pharo image will be terminated unexpectedly or cease to produce heartbeat signals. In this case, the runner will notify that there is a problem in the child process. It will kill the child process if it is still running, then will update the state file by setting the *crash* flag value to True. Then it will collect the available logs, including the last generated test method causing the crash. Finally, it will resume the Pharo image and watch its status again.

Pharo is a live system; when we snapshot its state, the next time we start the image, it will continue from the snapshot point. In our case, it will resume from the decision activity labeled *Check crash* (Figure 4.1), not from the beginning of the SMALL-AMP algorithm. After recovering from a crash, SMALL-AMP will load the state file and check the value of the *crash* flag. Since the runner has flipped its value, it will conclude that a crash may happen if it continues to amplify the current test method. Therefore, it skips this test method and continues to amplify other methods.

The runner also keeps track of the number of recovered crashes. If it is more than a fair number `MAX_CRASH` (default is 10), it will understand that there is a severe problem in amplifying this class, so it will stop trying.

If all test methods are amplified in the Pharo process, it will finalize and export the results. Then it will update a flag *finish* in the state file and exit the process. The runner will realize that the child process has exited with a *SUCCESS* return code. It will check the *finish* flag; if it is set correctly, it will stop watching and finish the process; otherwise, it will consider this termination as a crash. As explained earlier, we do not trust the return value because the exit API can be called indirectly during test amplification.

4.4 EVALUATION

To evaluate whether zero-touch test amplification is indeed feasible and to quantify the impact of prioritization, sharding, and sandboxing, we formulate the following research questions.

RQ1 – Is it possible to fully automatically amplify a test suite using GitHub-Actions? This is the primary research question for this feasibility study. To evaluate whether zero-touch test amplification is feasible, we install the proof-of-concept extension of SMALL-AMP on five open-source Pharo projects deployed on GitHub. We collect quantitative evidence on the execution times when ran on the GitHub platform.

RQ2 – How does the prioritization heuristic affect the test amplification performance? To quantify the impact of the test prioritization, we compare the number of killed mutants with or without test prioritization and the execution time with and without the time budget.

RQ3 – How many duplicated mutants are created after sharding? The sharding step splits large test classes (more than 15 test methods) to avoid that the optimization step is forced to ignore relevant test methods. However, the same mutant may then be killed by more than one of the shards, thus results in duplicated mutants. Duplicated mutants designate wasted computations in the test amplification algorithm, hence should be minimized.

RQ4 – Does sandboxing circumvent crashes? To illustrate the necessity of sandboxing, we count how often the sandbox recovered from (a) a system freeze, (b) a system crash, and (c) a polluted image.

4.4.1 Dataset

We used the GitHub search API to sort the Pharo projects based on the number of stars. Then we discarded the system projects and projects without a smalltalkCI configuration file (`.smalltalk.ston`). *smalltalkCI* is a framework to integrate the Pharo projects with continuous integration platforms. We prefer the projects to include `.smalltalk.ston` file because it contains all project-wise configurations, as well as it shows that the project is CI friendly. After filtering, we selected the five projects with the most stars. We had to limit the evaluation to five projects because running SMALL-AMP at the project level takes considerable time, and we had to repeat the evaluation multiple times. The final selected projects and their number of stars and a short description are shown in Table 4.1.

Table 4.2 shows the descriptive statistics for the test classes in these projects. In this evaluation, we exclude the test classes without any passing (green) test method or test

Table 4.1: Dataset composed of 5 Pharo projects from GitHub

Project	Stars	Description
Seaside	389	Web-application Framework
PolyMath	148	Scientific Computing for Pharo
NovaStelo	113	Block-style programming environment
Moose	103	Platform for software and data analysis
Zinc	69	HTTP networking protocol framework

Table 4.2: Descriptive statistics for the test classes.

	Seaside	PolyMath	NovaStelo	Moose	zinc
# before sharding	82	68	50	8	35
# Large test classes	10	3	7	0	2
# after sharding	99	87	82	8	40
# with 100% coverage	29	3	16	0	2
# without any green test	0	0	3	0	0
# Classes to be amplified	70	84	63	8	38

classes with 100% mutation coverage. We also consider test classes with more than 15 test classes as large test classes and break them down into shards.

4.4.2 Evaluation

We forked all projects in our dataset in GitHub and set up the test amplification GitHub-Actions workflow. In setting the default values for the workflow we exploited the content of `.smalltalk.ston` for loading the project, and the default values defined in Chapter 2 for running Small-Amp ($N_{maxInput} = 10$, $N_{iteration} = 3$). Since they report that the majority of executions are finished in less than 6 minutes, we set the time budget to 12 minutes to have some leeway. We consider a crash if the heartbeat file is not updated for 4 minutes. We used eight parallel jobs for each workflow, plus one initial and one finalizing job, totaling ten jobs for each workflow run. Since GitHub-Actions offers up to 20 jobs to run simultaneously for free accounts and open source projects at the time of writing this chapter, this number of jobs is acceptable for open source projects. GitHub also allows each job to run for a maximum of 6 hours [124].

Then we manually ran the workflow six times. We enabled the test method prioritization mechanism in the first three runs and disabled it in the subsequent three runs. We collected the generated artifacts and analyzed them to answer our research questions. To

Table 4.3: The result of the quantitative analysis.

		Prioritization Enabled			No Prioritization		
		#1	#2	#3	#1	#2	#3
1	# All test classes executions	263	263	263	263	263	263
2	# Finished executions	221	216	217	213	209	216
3	# Image pollution	14	18	17	16	23	18
4	# Unfinished	28	29	29	34	31	29
5	# Recovered freezings/crashes	37 (16.7%)	37 (17.2%)	37 (17.0%)	43 (20.1%)	44 (21.0%)	37 (17.1%)
6	# Executions having improvement	105	98	97	92	96	102
7	% Executions having improvement	47.51%	45.37%	44.70%	43.19%	45.93%	47.22%
8	# Test methods	1805	1766	1757	1703	1677	1762
9	# Generated tests	223	213	194	165	166	199
10	# All mutants in finished cases	9758	9670	9713	9094	9029	8761
11	# Mutants live original	3984	3957	3972	3814	3315	3292
12	# Mutants killed original	5774	5713	5741	5280	5779	5469
13	# Newly killed mutants	561	561	533	483	499	538
14	% Increased kills	9.71%	9.81%	9.28%	9.14%	8.63%	9.83%
15	# Mutants killed in Large test classes	198	176	167	115	144	157
16	# Duplicated killed mutants in Large classes	56	56	48	34	51	58
17	% Duplicated killed in Large classes	28.28%	31.82%	28.74%	29.57%	35.42%	36.94%
18	# Time budget finished	18	18	18	17	19	21
19	# Test methods skipped	148	133	123	118	116	119
20	Workflow duration: All	4:33:41	4:42:09	4:35:45	5:00:54	4:19:13	4:37:20
21	Seaside	1:28:14	1:32:34	1:24:47	1:31:14	1:28:06	1:25:23
22	PolyMath	1:04:57	1:07:08	1:06:43	1:12:21	1:10:11	1:07:19
23	NovaStelo	1:00:58	1:01:08	1:04:21	1:12:24	0:41:16	1:03:53
24	Moose	0:26:45	0:27:37	0:27:52	0:27:18	0:27:59	0:26:29
25	Zinc	0:32:47	0:33:42	0:32:02	0:37:37	0:31:41	0:34:16

conclude, we manually ran 30 workflows in total (6 for each of the five projects), resulting in 300 jobs on GitHub servers.

Table 4.3 shows the results from this analysis. In each run, 263 test classes are executed (row 1) in which between 209 to 221 cases are finished (row 2), and 28 to 34 cases are unfinished (row 4).[4] In 92 to 105 cases, SMALL-AMP is able to successfully amplify the test class (row 6) and generate 165 to 223 new test methods (row 9). In 17 to 21 cases, the time budget was in effect (row 10), so 116 to 148 test methods are skipped during amplification (row 11). The execution for each run, the sum of all projects, takes between 4:19 to 5 hours (row 12). We have included the artifacts and workflow run logs in the replication package.[5]

4.4.3 RQ1: Is It Possible to Fully Automatically Amplify A Test Suite Using GitHub-Actions?

Table 4.3 shows that for the 263 test classes, the test amplification finished successfully in 209 — 221 cases. The maximum execution time for an entire project was about 90 minutes for *Seaside*, while the minimum time was around 27 minutes for *Moose*. Since each

[4]The cases finished without results were due to a timeout in the post-processing step, which was a bug in the implementation only discovered during the analysis

[5]`github.com/mabdi/SmallAmp-evaluations`

workflow can run up to 6 hours in GitHub-Actions, these values show that there is room for optimizing the configurations. This may be done by reducing the number of parallel jobs from 8 to a lower value; by increasing the Small-Amp parameters ($N_{maxInput}$, $N_{iteration}$); by increasing the time budget. This illustrates that fine-tuning the configuration will be needed when zero-touch testing is adopted for a given project but there is sufficient room to do so.

In addition, if we consider the execution time per project in all 6 runs, we see that the results are similar and do not vary a lot. This similarity confirms that setting a time budget makes the execution time of test amplification indeed more predictable.

Answer to RQ1: *Our proof-of-concept demonstrates that integrating a test amplification tool within a continuous integration server allows for a fully autonomous process. Moreover, the time budget allows for doing so in an acceptable period of 30 to 90 minutes in our analysis.*

4.4.4 RQ2: How Does the Prioritization Heuristic Affect the Test Amplification Performance?

By design, if the test amplification hits the time limit, it will kill fewer mutants. However, the test prioritization should dampen this effect. We expect more killed mutants with test prioritization than without. On the other hand, when the time limit is not reached, the impact of test prioritization should be negligible. We, therefore, compare the value of the increased kill based on their timeout status in two configurations (prioritization enabled in the first three runs and disabled in the next three). The value of increased kills is calculated as follows:

$$100 \times \frac{\#\textit{killed mutants with prioritization}}{\#\textit{killed mutants without prioritization}}$$

Table 4.4 compares the number of newly killed mutants for these test classes and their increase.

As expected, we see the cases with prioritization have better performance (34.09%) when they run out of time. For the classes that finished within time, the increase in the killed mutants is negligible (0.70%).

Answer to RQ2: *Test prioritization is an effective way to impose a predictable time budget on the test amplification. In those cases where the time limit is reached, test prioritization kills more mutants. When the time limit is not reached, test prioritization has negligible impact.*

Table 4.4: Comparison of the number of newly killed mutants when prioritization is enabled and disabled

	Timeout	**In time**
Number of classes	7	156
Disabled (killed mutants)	44	1267
Enabled (killed mutants)	59	1276
Increase	**34.09%**	**0.70%**

4.4.5 RQ3: How Many Duplicated Mutants Are Created After Sharding?

To quantify how much waste is induced by the sharding step, we calculate the number of duplicated mutants killed due to splitting overly large (more than 15 test methods) classes.

First of all, Table 4.2 illustrates that such large test classes actually exist, although it depends a lot on the project. The **Seaside** project has 10 large classes; hence the sharding increased the number of test classes from 82 to 99. The **Moose** project, on the other hand, had no large test classes.

For those cases with large test classes, we found 198, 176, 167, 115, 144, and 157 killed mutants (Table 4.3, row 15). Consequently, the number of duplicated mutants in the shards is 56, 56, 48, 34, 51, and 58 (Table 4.3, row 16). Therefore, about 28% to 37% of the killed mutants are duplicated when we employ sharding, which is considerable. Further research is warranted to see whether we can decrease these duplicates, for instance, by clustering the shards based on their coverage. Finding a balance between the sharding factor and the time budget (in our analysis, we used 15 and 12 minutes) may also decrease the number of duplications.

> ***Answer to RQ3:*** *Sharding allows to run the test amplification in a given time budget, even with overly long test classes. However, it comes at a cost: in our analysis, we see around 30% duplication in the killed mutants when large test classes get split into distinct shards.*

4.4.6 RQ4: Does Sandboxing Circumvent Crashes?

To assess the effectiveness of sandboxing mechanism, we process the job logs in all six runs to collect quantitative evidence of crashes, freezes, and polluted images. In the finished execution, we see 37 to 44 classes recovered from a crash (Table 4.3, row 5). Overall, the crash-recovery mechanism recovered the amplification process 235 times in all six runs. After investigating the reasons for these crashes, we found that most cases (about

95%) are recovering from a system freeze. In one case, the crash is because of a *Segmentation fault* error.

Table 4.3 row 3 shows in 14 to 23 cases of classes, image-state pollution occurs; note that the numbers vary because of sharding. However, this only occurred in one of the projects (*NovaStelo*), where 12 test classes are green before mutation testing and become red after.

We conclude that image crashes, freezing, and state pollution frequently happen when running the test amplification in Pharo. When these are not appropriately handled, test amplification integration in continuous integration will fail. Nevertheless, the proposed crash-recovery mechanism allows SMALL-AMP to overcome the problems and skip the failing cases.

> ***Answer to RQ4:*** *Image crashes, freezing, and state pollution frequently happen when running the test amplification in Pharo. The results of the evaluation show that the proposed sandboxing mechanism is effective in overcoming these problems.*

4.5 THREATS TO VALIDITY

Did we measure what was intended? (construct validity) We use quantitative metrics (the number of newly killed mutants, the number of recovered crashes, and execution time) to quantify the impact of zero-touch test amplification. However, a qualitative study will be needed to evaluate whether the recommended tests add value. This qualitative study is considered future work and is beyond the scope of this chapter.

Are there unknown factors that might affect the outcome of the analyses? (internal validity) The test amplification tool, the prioritization mechanism, and the GITHUB-ACTIONS workflow include various parameters. For configuring SMALL-AMP, we used the parameters from Chapter 2, in which the authors have not claimed that values are optimal. Similarly, the prioritization mechanism parameters (α, β and γ in Section 4.3.3) are also configured by preliminary values based on authors' insights. We see this risk as a minimum because optimizing the parameters should not invalidating the findings.

For identifying a pollution (Section 4.2.2), we run the test class after the early mutation testing (Algorithm 5 line 1). If the test is green, we assume the state is not polluted and continue the algorithm. However, there might be some pollution undetected by the tests. Since we use a fresh Pharo image for amplifying each test class, it stops propagating the possible pollution to the following process.

To what extent is it possible to generalize the findings? (external validity) We expect the overall finding, like the applicability of project-level test amplification (RQ1), the impact of test method prioritization (RQ2) and sharding (RQ3), and the relevance of the sandbox mechanism (RQ4) to be valid in other tools. However, we cannot claim that the numbers and other details are valid for other ecosystems, and separate studies should be conducted in other ecosystems.

Is the result dependent on the tools? (conclusion validity) Our work depends on SMALL-AMP as the test amplification tool. In RQ2, we compare the results from SMALL-AMP in two different configurations to assess the impact of prioritization. So, we expect the differences in the results are mainly because of the configuration, not the tool itself. Another critical factor is randomness, which we tried to diminish by repeating the experiment three times for each configuration on five different mature projects with a high number of test classes.

4.6 RELATED WORK

This work extends SMALL-AMP (Chapter 2), test amplification in Pharo by integrating it into GITHUB-ACTIONS, providing a prioritization heuristic, sharding, and sandboxing. Test amplification for crash reproduction has also been reported in other papers [54, 55, 91]. To the best of our knowledge, sharding (i.e. splitting test cases to fit in a time budget) is a novel concept in test amplification. However, some works in parallel test case prioritization also split a test suite to fit a time budget (running different portions of the test suite in different machines) [131].

There are several works in the Pharo community related to our sandboxing solution: Polito et al. study the bootstrapping problem in Smalltalk and provide the Hazelnut model for bootstrapping reflective systems [132]. The work by Béra et al. introduces Sista, a fast snapshotting and restoring solution to increase the warmup time of Pharo images [133]. Epicea [134] records the changes in the code and some IDE events in logs, which can be used to recover the lost state.

We consider our extension of SMALL-AMP as a proof of concept for "zero-touch" test amplification. Chapter 5 introduces a zero-touch mutation testing framework for Pharo to alleviate the extra efforts from the developers' side in mutation testing analyses. Relevant related work for true zero-touch test amplification will be techniques for increasing the readability of the generated tests, i.e. intention revealing names [62] and removing the redundant statements [63].

Repairnator [135], on the other hand, is a complementary attempt at zero-touch test automation, this time for automated program repair. Other complementary work on

zero-touch test automation is the work by Campos et al. [112]. They introduce Continuous Test Generation (CTG) by incorporating EvoSuite in a continuous integration setting. In a similar vein, Danglot et al. [109] investigated ways to exploit test amplification in a continuous integration setting.

4.7 CONCLUSION

In this chapter, we argued that "zero-touch" test amplification may alleviate the challenges that prevent widespread adoption of test amplification tools. In this vision, a test amplifier will decide for itself which tests to amplify, incorporate the synthesized tests in a separate branch, execute the strengthened test suite and —if all steps pass— push the strengthened test suite onto the main branch. All without any intervention of a software engineer. To demonstrate the feasibility of this vision, we present a proof-of-concept tool that integrates SMALL-AMP with GITHUB-ACTIONS to automatically strengthen the existing test suite within a limited time budget.

We validated the proof-of-concept tool on five popular open-source Pharo projects. The results show that integrating a test amplification tool within a continuous integration server indeed allows for a fully autonomous process. Moreover, the time budget allows to do so in an acceptable time span; 30 to 90 minutes in our analysis. Test prioritization was able to improve the performance of the tool when the time budget was exceeded by up to 34%. Test sharding was needed to run the test amplification in a given time budget, even with overly long test classes. However, it comes at a cost: in our analysis, we see around 30% duplication in the killed mutants when large test classes get split into distinct shards. Last but not least, we demonstrated that sandboxing is an effective way to make the test amplification crash resilient.

Even though a zero-touch approach will facilitate the adoption of test amplification, more qualitative research is needed to make the results acceptable. This ranges from improving the readability of the generated test cases (intention revealing names, meaningful comments, ...) to usability studies assessing the added value of the amplified tests.

4.7 REPLICATION PACKAGE

Our supplementary resources are available in our replication package[6] which is anonymous. The reviewers may access them without breaking the double-blind process.

[6] https://zenodo.org/record/6482867#.Y1CxluzMJxi

Chapter 5

Toward Zero-touch Mutation Testing in Pharo

This chapter is a revised version of an originally published paper in the *The 21st Belgium-Netherlands Software Evolution Workshop* (*BENEVOL 2022*):

Steps Towards Zero-touch Mutation Testing in Pharo

Mehrdad Abdi and Serge Demeyer
In *The 21st Belgium-Netherlands Software Evolution Workshop* (*BENEVOL 2022*). , 2022.
URL: https://www.researchgate.net/publication/362868185.

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* MuTalk, *the mutation testing framework in the live programming environment of Pharo, by (1) adding hierarchical mutation operators, (2) integrating it to* GitHub-Actions, *(3) visualizing the result in a web-based mutants explorer.*

5.1 INTRODUCTION

Software is everywhere, and its failures are costly. Unit testing is writing small test code snippets that exercise the unit under test and asserts the intended values. In mutation testing [136], 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 [14, 113] 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, SMALL-AMP [87], 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 MUTALK, the mutation testing in Pharo, generates too few mutants compared to the mutation testing framework in Java from another work [9]. 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 MUTALK of which the details come in Section 5.2.

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 5.3 explains this problem with an example and how we overcome this problem.

In the next step, we created MUTALKCI as a *zero-touch mutation testing* for the live programming environment of Pharo. By default, developers are expected to use MUTALK 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 GITHUB-ACTIONS 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. MUTALKCI is explained in Section 5.4.

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 [78, 102, 103, 104, 137] 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 5.4.1.

5.2 EXPANDING MUTATION OPERATORS IN MUTALK

5.2.1 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.

MuTalk[1] is a mutation testing framework for programs written in Smalltalk. The original mutation operators in MuTalk includes some known patterns related to `Boolean messages`,`Magnitude messages`,`Collection messages`,`Number messages` and `Flow control messages` [45]. 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.

5.2.2 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 [138, 139] 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

[1]https://github.com/pavel-krivanek/mutalk
[2]PIT: https://pitest.org/quickstart/mutators/
[3]https://github.com/mabdi/mutalk

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.

5.3 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 5.1 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.

```
1   factorial: anInt
2     anInt == 1 ifTrue: [ ^ 1 ].
3     ^ anInt * (self factorial: anInt -1)
4
5   "Mutant 1: disabled the conditional statement by replacing the message"
6   factorial: anInt
7     (anInt == 1) yourself.
8     ^ anInt * (self factorial: anInt -1)
9
10  "Mutant 2: replaced the condition with always false"
11  factorial: anInt
12    false ifTrue: [ ^ 1 ].
13    ^ anInt * (self factorial: anInt -1)
14
15  "Mutant 3: removed return operator"
16  factorial: anInt
17    anInt == 1 ifTrue: [ 1 ].
18    ^ anInt * (self factorial: anInt -1)
```

Code Excerpt 5.1: 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 [91].

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.

Code Excerpt 5.2: Auxiliary exception for avading infinate loops.

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

5.4 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 [140]. 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 5.1 illustrates the proposed hierarchical approach for running MuTalk in the CI/CD build servers. This framework is also comparable to *DevOps* [141] frameworks.

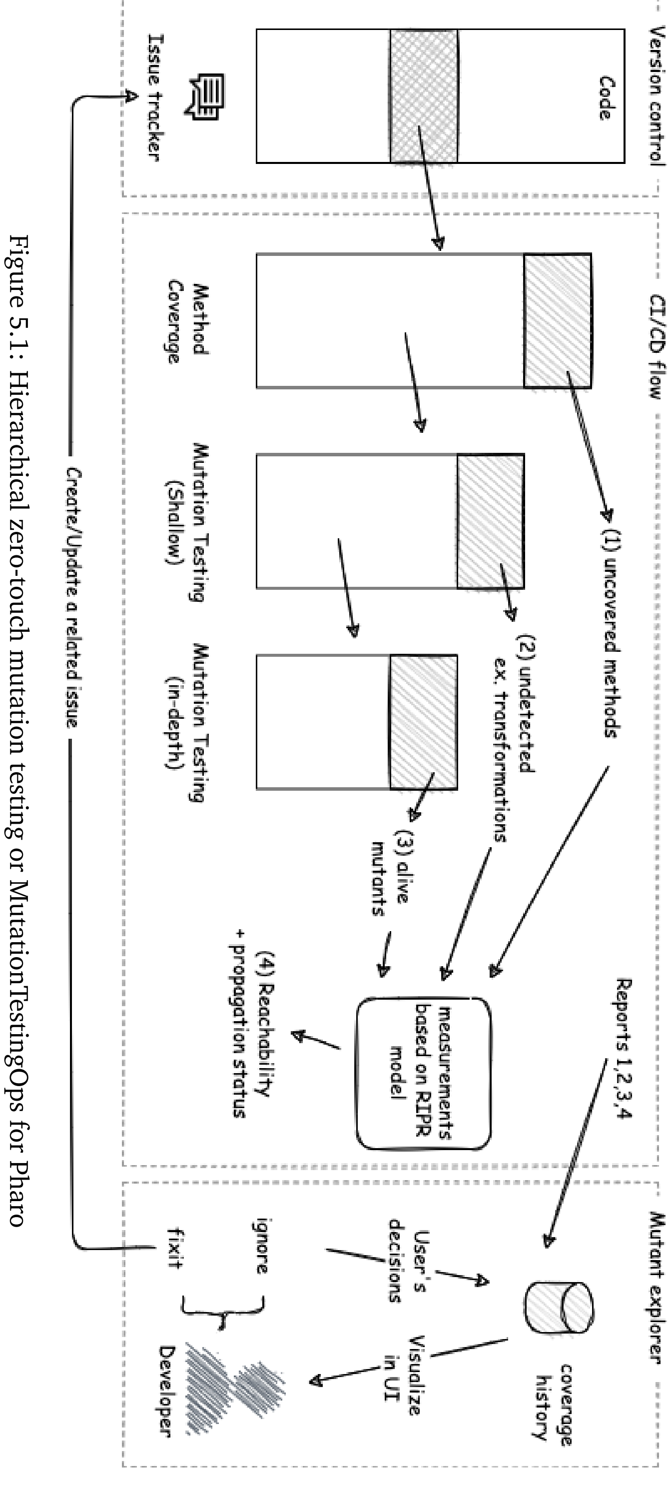


Figure 5.1: Hierarchical zero-touch mutation testing or MutationTestingOps for Pharo

DevOps provides agility in continuous software delivery by an iterative approach based on automation and 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 [140, 142] 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 [78, 102, 103, 104, 137], 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 starting 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 5.4.1)

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

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.

5.4.1 Mutants Explorer

Since interpreting the reports generated in Section 5.4 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.

Figure 5.2 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] https://github.com/harolato/mutation-testing-coverage

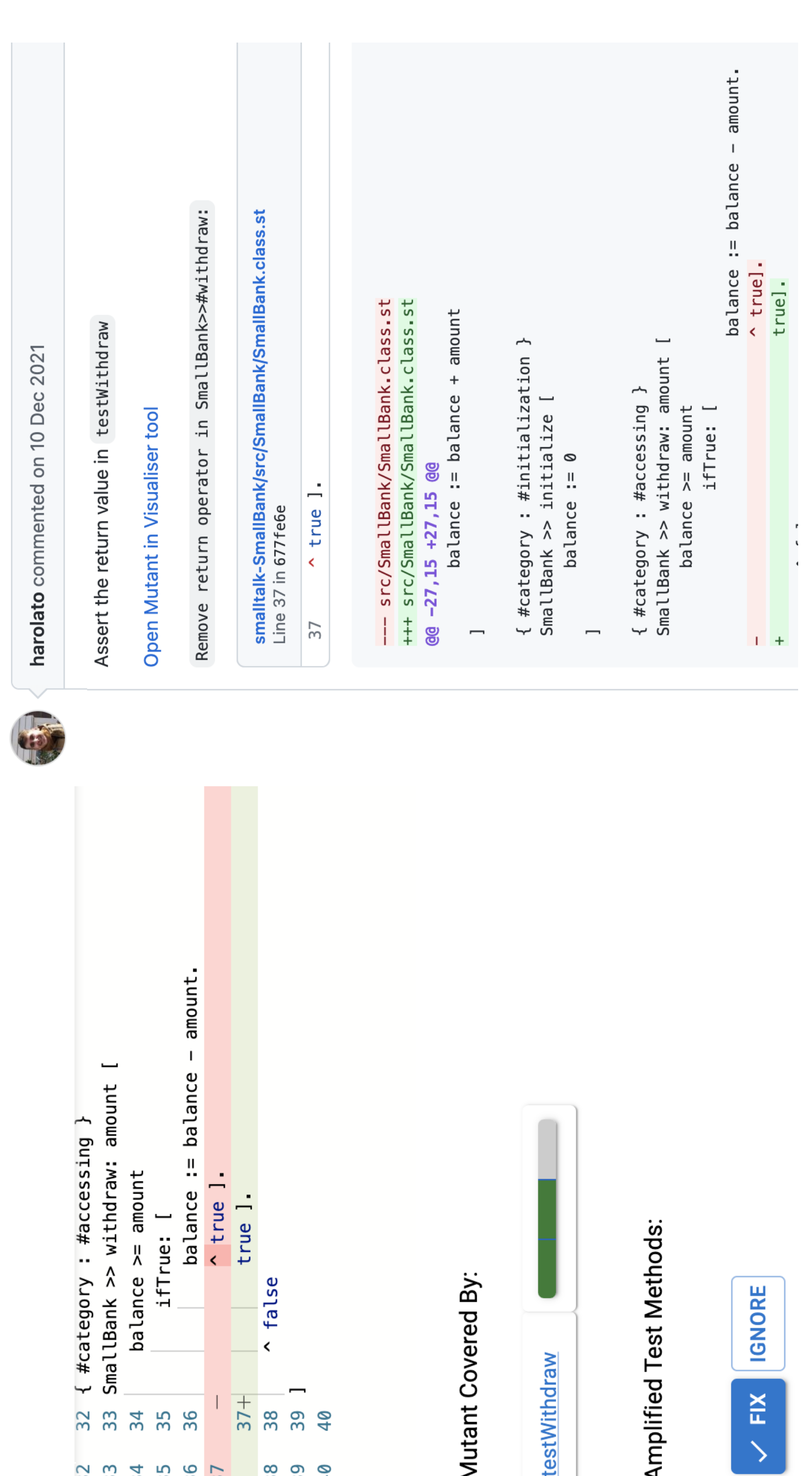


Figure 5.2: A mutant view and its generated issue

5.5 CONCLUSION AND FUTURE WORK

In this paper, we propose an approach for creating a *zero-touch mutation testing* (or MutationTestingOps) 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.

Chapter 6

Test Amplification DevBot

ABSTRACT

This chapter explains the extension of the work described in Chapter 4. We employed GitHub bots to process the workflow output and push the results into the repository. ***This part of the work is not included in any of the publications.***

6.1 INTRODUCTION

As we mentioned in Section 4.7, it can still be cumbersome for developers to process the GitHub-Actions workflow artifacts manually. We employed GitHub bots (or GitHub Apps [143]) for smoothing this process. GitHub apps are first-class GitHub actors: they have their identity and act on their behalf.

Our goal in this project is to offer the newly generated tests by test amplification tools to software engineers by pull requests. Software engineers can assess the pull requests and accept (a part of) them to be merged into the repository code. Therefore, it is important that the offered tests encompass a readable comment explaining a reason why Small-Amp has offered it.

In this chapter, first, we explain the comment generation step, which is a new Small-Amp post-process (post-processing is explained in Section 2.4.3). Then, the details about employing the pre-authenticated GitHub-Actions bot and providing the result as a pull request are illustrated by examples. Finally, we talk about the future direction of a *test amplification ecosystem*.

```
74  + DataFrameJsonReaderTest >> testAmplified_5_1 [
75  +
76  +         "SmallAmp has derived this test from
      `DataFrameJsonReaderTest>>#testReadFromJsonOrient` by applying some transformations
      and regenerating its assertions.
77  +         This test can:
78  +         * It detects the injection of an artificial fault (Replace #ifTrue:
      receiver with true) in `DataFrameJsonReader>> #read:`:
79  +                 original code snippet: `orient = 'auto' ifTrue: [ self
      inferOrientFromJson: json ]`
80  +                 Mutated code snippet: `true ifTrue: [ self inferOrientFromJson:
      json ]`
81  +                 Dynamic state: {#orient->'Sm`:%'}
82  + "
83  +
84  +         <madeBySmallAmp: 'Amplified_5_1'>
85  +         | output |
86  +         self
87  +                 should: [
88  +                         output := DataFrame
89  +                                         readFromJson: directory / 'split.json'
90  +                                         orient: 'Sm`:%' ]
91  +                 raise: KeyNotFound
92  + ]
```

Figure 6.1: An example of the generated comment

6.2 COMMENT GENERATION

Recent works in employing mutation testing in industry [73, 74] show that the majority of the practitioners have limited knowledge about *mutation testing*. So, we added some extra human-readable explanations about the mutation process including the mutated part of the code, the code after mutation, and also the dynamic state of the program in that position after the mutation is executed. Figure 6.1 shows an example of the generated comments.

6.3 SMALL-AMP DEVBOT

6.3.1 Bots For Supporting Software Development

The idea of employing intelligent programs to assist developers is not a new idea. We can find the early attempts in the 80s such as the project of *programmer's apprentice* [22, 23]; and we see this trend is still active nowadays [18, 24].

Erlenhov et. al. [144, 145] coined terms *DevBots* for referring to bots for software development. They also define *ideal DevBots* for referring to artificial software developers

which are autonomous, adaptive, and hold technical and social competence. These bots are self-sufficient, monitoring continuously and deciding when to act. They learn from previous experiences and the feedback from other entities, so they adapt themselves to reduce their mistakes or increase their accuracy. They focus on solving a specific problem, preferably using different techniques. Finally, since some entities in the ecosystem may distrust bots, ideal DevBots need to have social abilities such as human-like communications to reduce such side effects.

6.3.2 Small-Amp DevBot

We extended the job in the merging phase (See Section 4.3.2) of our zero-touch test amplification process. In this extension, the project-under-test is cloned in the running machine in GitHub-Actions job and the amplified methods are committed in it in a new branch (*amplificaion branch*). Then the new branch is pushed into the repository, and a new pull request from it to the main branch is sent. In order to facilitate test methods exclusion from the pull request (cherry picking), each commit only includes a single test method. We use the *checkout*[1] action that provides a pre-authenticated git instance by the *github-actions bot* identity. However, for more advanced use cases (see Section 6.4), the approach can be extended by devising a standalone GitHub bot.

We still expect the tests need some polishments such as removing redundant lines and renaming the tests so normally developer will not accept the pull request without some corrections. Developers can load the offered code in the amplification branch into their favorite IDE and use debuggers and polish the tests. They can also use GitHub's web interface for some quick corrections.

6.4 VISION: TEST AMPLIFICATION ECOSYSTEM

In this section, we introduce the test amplification ecosystem. In this ecosystem, human involvement is reduced as possible and they are involved only in important tasks such as auditing the amplified tests.

Figure 6.3 illustrates the test amplification ecosystem. The left part is only controlled by a human, and all other works (the right side) will be handled by *a devbot*. The first flow shows the developers' traditional duty of writing code which they stash their code in a version control such as GitHub. Number 2 refers to the zero-touch running of the test amplification tool on the original test suite. Number 3 also refers to exporting the result as artifacts. We have covered flows number 2 and 3 in Chapter 4. Section 6.3 explained how we can employ GitHub bots to provide the output as pull requests (number 4.1).

[1] `https://github.com/actions/checkout`

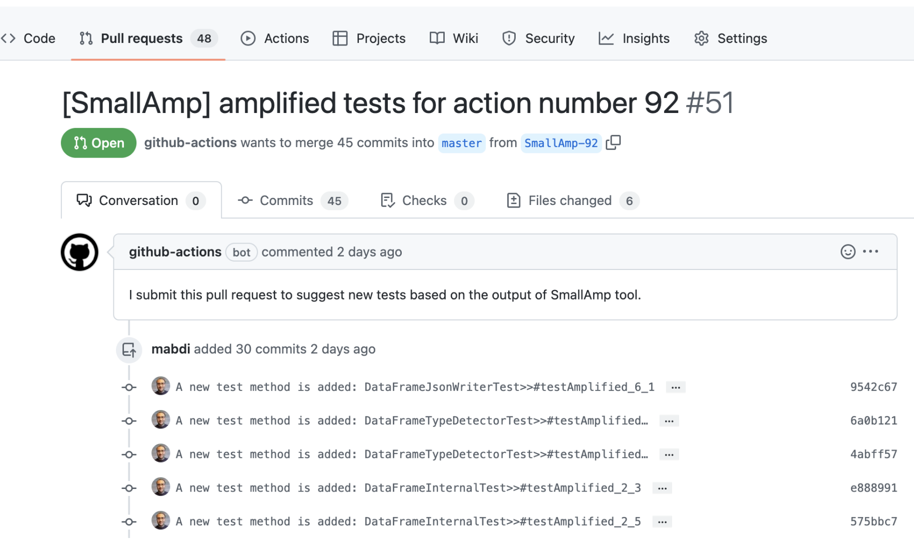


Figure 6.2: An example of the sent pull request

However, some developers may prefer not to mess up their projects with a tool output. As an alternative, we can employ a web-based test explorer (similar to Section 5.4.1) to index the result and visualize it for developers (Flow 4.2). An IDE plugin can be employed on the developer side to fetch the latest result in IDE and help the developer in auditing them (Flows 5, and 6.1). This part is similar to TestCube [61], but the IDE plugin in this solution is much lighter because it does not include any test amplification tool inside. Additionally, in this architecture, the developers are allowed to exploit a web-based test editor and fulfill their revisions (Number 6.2). The revised version of the tests is returned to the test explorer (Number 7), but they are not ready for being merged into the code yet. The DevBot will start the amplification tool again to verify the edited tests (Numbers 8 and 9). If the tests are not green or they do not cover the intended parts, the bot will warn developers about their revision. If there is no problem, or the developer forces the changes, it will push the changes to the code base (Number 10).

6.5 CONCLUSION

In this chapter, we extended our recent work on zero-touch test amplification by using GitHub bots to submit automated pull-request from the GitHub-Actions workflow artifacts. DevBot includes the amplified test methods in separated commits in the submitted pull requests to make it easier to be cherry-picked by developers during the assessment. Each amplified test method encompasses a human-readable comment that explains the intention of the recommended tests.

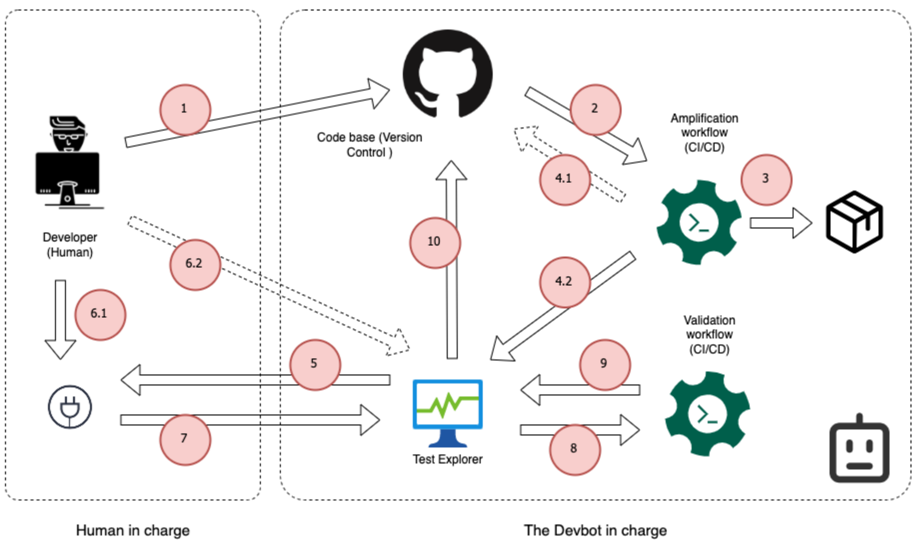


Figure 6.3: Test amplification ecosystem with human in loop

We drew a sketch of a test amplification ecosystem in which humans are kept in the loop and all trivial tasks are done by DevBots. Evaluating the usability of the system by conducting a user study is the future work.

Part III

A Path to Test Transplantation

Chapter 7

Can We Increase the Test-coverage in Libraries using Dependent Projects' Test-suites?

This chapter is a revised version of an originally published paper in the *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering 2022* (*Vision and Emerging Results Track*) (*EASE 2022*):

Can We Increase the Test-coverage in Libraries using Dependent Projects' Test-suites?

Igor Schittekat, Mehrdad Abdi, and Serge Demeyer
In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering 2022* (*Vision and Emerging Results Track*) (*EASE 2022*). June, 2022.
URL: https://doi.org/10.1145/3530019.3535309.

ABSTRACT

Modern software systems increasingly depend on packages released on code sharing platforms such as GitHub, Bitbucket, and GitLab. To minimize the risk of lurking defects in such packages, strong test suites covering the normal as well as the exceptional paths are needed. In this chapter, we explore the potential of using tests from dependent projects to increase the code coverage of base packages. We extracted 4 popular Python packages from GitHub together with 14 dependent projects and analyzed the code coverage of the available tests. We observed that adopting the tests of the dependent projects in the test suite of the base library, would increase the line coverage in 9 out of 14 (64%) of the cases and the mutation coverage in all of them (100%). Our results suggest that a tool which would generate tests for the base package based on the tests in the dependent projects, would help to strengthen the test suite.

7.1 INTRODUCTION

Software testing is an important part of software development. If the software is tested poorly, defects will go unnoticed and slip into production. *Code coverage* is the commonly adopted way of quantifying the strength of a test suite [146]. Higher coverage implies that more of the code under test is executed by the test suite, raising the confidence in its fault detection capacity. However, code coverage tells only half of the story. To verify whether a test suite is actually capable of detecting defects *mutation coverage* is widely acknowledged as the state-of-the-art [147].

With the advent of code sharing platforms such as GitHub, Bitbucket, and GitLab, *software ecosystems* are gaining more and more importance [1]. As an example, in 2017 the `npm` ecosystem for building Javascript based web applications was reportedly containing over 300,000 interdependent software packages [148].

From a testing perspective, the scale of these ecosystems is both a blessing and a curse. On the one hand, it illustrates the importance of a strong test suite, as a lurking defect in one package may impact all dependent packages. Unfortunately, many projects in the Python ecosystem exhibit poor coverage [149]. On the other hand, the sum may be more than the parts: tests exercising a given package will also cover the packages it depends upon.

In this chapter, we will assess to which extent the coverage of a given package can be increased by not only looking at the tests from the package itself, but also from the tests of dependent projects. We distinguish between the *base package* (the package where more coverage would be desirable) and the *dependent projects* (where we copy additional tests). We will focus on projects written in Python. A survey conducted in 2021 reported that Python is one of the fastest growing communities [150]. At the time of writing (March 2022), the Python Package Index (PyPI) has more than 36000 packages which are actively maintained and evolve continuously.

In Section 7.2 we will start with an example to show the potential of our research. Next, in Section 7.3 we evaluate the idea on realistic sample of actual Python packages, illustrating the potential increase in line coverage as well as mutation coverage. In Section 7.4 we discuss related work, and in Section 7.5 we conclude the chapter and discuss some future work.

7.2 MOTIVATING EXAMPLE

We start with an illustrative example of how tests from dependent projects can increase the coverage of the base package. Suppose you have a package `Collections` which implements a basic Stack, implementing `push`, `pop`, `getTop` and `isEmpty`. The

test suite of `Collections` implements a test for the stack, but only for the `push` and `getTop` functions. The `pop` and `isEmpty` are not tested within the test suite of `Collections` (Listing 7.1).

```
1  from Collections import Stack
2  def test_push():
3      s = Stack()
4      s.push(1)
5      assert s.getTop() == 1
6      s.push(2)
7      assert s.getTop() == 2
8      s.push(3)
9      assert s.getTop() == 3
```

Code Excerpt 7.1: StackTest

Now suppose a project, `Calculator`, uses the Stack from the `Collections` package to check if parenthesis match. (Listing 7.2)

```
1  from Collections import Stack
2  def match(string):
3      s = Stack()
4      for char in string:
5          if char == '(':
6              s.push(char)
7          if char == ')':
8              if s.getTop() == '(':
9                  s.pop()
10             else:
11                 return False
12     return s.isEmpty()
```

Code Excerpt 7.2: Calculator

In this example we can see that the functions `pop` and `isEmpty` are used as well. If this function is tested using the test suite of the `Calculator` project, it will indirectly test the Stack from the `Collections` package. Listing 7.3 shows the tests for the `match` function, and is located within the test suite of the `Calculator` project.

```
1  from Calculator import match
2  def test_match():
3      assert match('(1+2)*3+(3-1)')
4      assert not match('(1+2)*3+)3-1)')
5      assert not match('(1+2)*3+(3-1')
```

Code Excerpt 7.3: MatchTest

When combining the test from the `Calculator` package with the test from the `Collections`

package, the functions `pop` and `isEmpty` are tested as well, increasing the overall code coverage in the base package.

7.3 EVALUATION

In this section we evaluate quantitatively the impact tests from dependent projects would have on the coverage of the base project.

7.3.1 Project Selection

For our study, we use projects that we selected from libraries.io, a website with a wide variety of packages. We specified some prior criteria for the search of the packages. First of all, the projects are Python packages, so we looked for projects available in PyPI. Dependency of packages is shown by libraries.io, so we could select the *dependent projects* based on that data. Secondly, the source code of all *base packages* and *dependent projects* had to be on Github. Next, the *base package* needed to include a test suite. We exclude packages with full coverage because fully covered libraries can not increase any further. For this we relied on the information we got from the package maintainers. If no such information was present, we ran the testing framework ourselves. Next, we processed *dependent projects* which needed to include a test suite which covered parts of the code executing the base package.

With these criteria in mind, we selected 4 base packages and a total of 14 dependent projects. Table 7.1 shows the descriptive statistics for selected projects.

Failing and erroring tests were ignored in the evaluation, as all tests needed to succeed for mutation testing to run. The mutation score and number of mutants on the dependent projects were not computed as they do not affect the study.

An interesting remark is that the tests of *pdf-redactor* do not test the project itself, but it tests the imported packages. That's why the line coverage is 0%.

7.3.2 Impact On Line Coverage

RQ1: *To what extent can the line coverage of a base package increase by incorporating tests from dependent projects?*.

Line coverage was calculated using the python package `coverage`. `Coverage` is a convenient tool that measures the coverage from any python source code, independent of the location of the code on the file system. This implies that tests from one project could be run while only looking at the coverage of the code from another project. It also allows combining results of different runs. For our evaluation this implies that the combined

Table 7.1: Descriptive Statistics for the Selected Base Packages and Dependent Projects

BP/DP	Source	#ST	LoCP	LoCT	LC	MS	#M
flair	https://github.com/flairNLP/flair	89	34942	2185	33 %	0,077 %	22825
textattack	https://github.com/QData/TextAttack	35	20038	650	35 %		
nlp-gym	https://github.com/rajcscw/nlp-gym	6	1905	146	50 %		
pydata-wrangler	https://github.com/ContextLab/data-wrangler	23	1783	446	75 %		
textwiser	https://github.com/fidelity/textwiser	43	1994	785	82 %		
seqal	https://github.com/tech-sketch/SeqAL	8	590	194	77 %		
pdfrw	https://github.com/pmaupin/pdfrw	29	3686	799	41 %	18 %	2207
pdf-annotate	https://github.com/plangrid/pdf-annotate	120	3098	1795	95 %		
dungeon-sheets	https://github.com/canismarko/dungeon-sheets	105	61875	2029	94 %		
PyPDFForm	https://github.com/chinapandaman/PyPDFForm	132	1956	4063	100 %		
pdfconduit	https://github.com/sfneal/pdfconduit	28	1984	668	65 %		
pdf-redactor	https://github.com/vitalbeats/pdf-redactor	29	854	813	0 %		
sacred	https://github.com/IDSIA/sacred	618	9527	8018	66 %	31,7 %	4568
imitation	https://github.com/HumanCompatibleAI/imitation	260	10526	2514	48 %		
seml	https://github.com/TUM-DAML/seml	23	4257	407	29 %		
scikit-datasets	https://github.com/daviddiazvico/scikit-datasets	37	2375	608	82 %		
casanova	https://github.com/medialab/casanova	51	1532	1026	88 %	58 %	678
minet	https://github.com/medialab/minet	40	16255	1567	19 %		

BP/DP : Base packages and their dependent projects.
Source : The Github address for the project.
#ST : The number of successful tests.
LoCP : The lines of code in the project.
LoCT : The lines of code in the test suite.
LC : The line coverage on the project.
MS : The initial mutation score of the base package.
#M : The number of mutants on the base package.

results are the union of the covered lines in the different runs.

First, the line coverage was measured for each of the *base packages*. Next, the coverage was measured using the tests of the *dependent projects*. By combining the coverage results of those runs, we obtain the increase in coverage. Combining all coverage results from the different dependent projects resulted in the final coverage result.

7.3.3 Impact On Mutation Coverage

RQ2: *To which extent can the mutation coverage of a base package increase by incorporating tests from dependent projects?*.

Mutation coverage was calculated using the python package `mutmut`. This package provides a feature to select specific code to mutate, meaning it is possible to only mutate code from the base package, while running the tests from a dependent project. Another convenient feature is the option to include a coverage file. For this we used the coverage file provided by the package `coverage`, as we already had access to it from the first analysis (Section 7.3.2). This coverage file is used by `mutmut` to mutate only covered parts of the code. Mutants in parts of the code that are not covered by any test will never be killed, so even before running mutation coverage we can already predict which mutants will definitely survive. This means we didn't have to run the tests with those mutants, sparing a lot of computation time.

`mutmut` provided a file storing the results of each run. Using this file we could com-

bine the results of different runs. The combined killed mutants is measured as the union of the killed mutants over the different runs.

7.3.4 Results

Table 7.2 shows the results of the evaluation. The table columns are interpreted as follows.

— BP/DP : All base packages and their dependent projects. In each group, the base package is listed in bold.

— LC : The line coverage on the base package. Here we don't combine the tests with the test suite of the base package.

— ILC : The increase in line coverage on the base package. Here we combined the tests with the test suite of the base package. The increase shown in the base package is the result of combining the results of all dependent projects.

— TLC : The time for running line coverage.

— #KM : The number of newly killed mutants on the base package. In the rows of the base packages this indicates the newly killed mutants when combining the results of all their dependent projects.

— IMS : The increase in mutation score on the base package. As before, the rows with base packages list the increase when combining the results all their dependent projects.

— TMC: The time for running mutation coverage.

We included the resulting data of our evaluation on Figshare[1]. This data can be used to replicate our findings.

Answering RQ1. In 9 out of the 14 dependent projects we can see an increase in line coverage. The increase shown in the rows of the base packages is the increase we get when we combine the coverage results of all dependent projects. Here we see that 2 out of 4 base packages get an increase.

Answering RQ2. For mutation testing we can see that every project is awarded with an increase in mutation score. Even those projects were we did no see an increase in line coverage.

[1] https://figshare.com/s/304e0f741c3879b6e068

Table 7.2: Coverage results

BP/DP	LC	ILC		TLC	#KM	IMS		TMC
flair	33 %	+ 4%	↗	5m	467	+ 0,022 %	↗	2w
textattack	19 %	+ 2 %	↗	20m	265	+ 0,012 %	↗	6d
nlp-gym	17 %	+ 0 %		30s	81	+ 0,003 %	↗	12h
pydata-wrangler	19 %	+ 1 %	↗	30s	101	+ 0,004 %	↗	2d
textwiser	17 %	+ 2 %	↗	5m	87	+ 0,004 %	↗	12h
seqal	22 %	+ 3 %	↗	1m	363	+ 0,016 %	↗	1w
pdfrw	41 %	+ 28 %	↗	2s	674	+ 30 %	↗	1h
pdf-annotate	52 %	+ 15 %	↗	20s	371	+ 17 %	↗	30m
dungeon-sheets	50 %	+ 13 %	↗	1m	41	+ 2 %	↗	30m
PyPDFForm	60 %	+ 24 %	↗	1m	572	+ 26 %	↗	1h
pdfconduit	56 %	+ 21 %	↗	10s	440	+ 20 %	↗	45m
pdf-redactor	46 %	+ 10 %	↗	2s	50	+ 2 %	↗	15m
sacred	66 %	+ 0 %		30s	987	+ 21,6 %	↗	2w
imitation	25 %	+ 0 %		5m	18	+ 0,4 %	↗	1d
seml	25 %	+ 0 %		1s	22	+ 0,5 %	↗	1d
scikit-datasets	47 %	+ 0 %		5m	987	+ 21,6 %	↗	2w
casanova	88 %	+ 0 %		1s	31	+ 4 %	↗	1m
minet	34 %	+ 0 %		6s	31	+ 4 %	↗	1m

BP/DP : Base packages and their dependent projects.
LC : Line coverage on the base package.
ILC : Increase in line coverage on the base package.
TLC : Time for running line coverage.
#KM : Newly killed mutants on the base package.
IMS : Increase in mutation score on the base package.
TMC: Time for running mutation coverage.

7.3.5 Discussion

The results of our evaluation show that using test suites from dependent projects can significantly increase the coverage on the base package. In this section, we explain how to make these improvements permanent, and look deeper into the execution time of the evaluation.

How to make these improvements permanent? Now that we have shown there is a possibility to improve the code coverage by looking at tests of the dependent projects, an important question still needs to be answered: How to make this increase permanent? *Dependency based test transplantation* is a solution: We can generate new tests to be included in the base package's test suite by exploiting the source of knowledge gained from the dependent projects. Therefore, test suites in the dependent projects are not required to be run every time.

To automate this, a tool needs to be created. First, it needs to filter out relevant tests within the test suite of the dependent projects. Not all tests in these test suites will cover parts of the base package, and only tests that cover those parts can be used to generate relevant tests.

Next, a technique similar to *capture and replay* [151, 152, 153] can be used to generate isolated new tests. An isolated test only depends on the code from the base package and does not require the units in the dependent project. Tracers should be installed on the public APIs in the packages, then the tests in the dependent projects should be executed to capture the execution traces. By looking at execution traces of the tests, new tests can be generated for the base package.

Creating a tool that holds up to our expectations is still future work, but in this chapter, we have shown that it can help software developers to test their packages even better.

Execution time. One drawback at this point is the execution time. For mutation testing, we had runs taking over a week; despite the optimisation of only mutating the covered parts of the code. One solution would be to only consider tests that indirectly execute the base package, instead of all tests in the dependent projects. This would require some static or dynamic source code analysis to filter out these tests.

7.3.6 Threats to Validity

Generalization. We performed our experiment on four base packages and a total of fourteen dependent projects, to prove the potential. The increase in coverage for both line coverage and mutation coverage can change depending on the selection of packages.

In our experiments, we saw an increase in mutation coverage for 100% of the packages. However, we can not guarantee the rate of 100% for all other projects upfront, but we still expect a considerable rate.

Exclusion of failing tests. The package used for mutation testing, `mutmut`, only works on successful tests. For consistency, failing tests were excluded for both line coverage and mutation coverage. We see this effect on our experiment as minimal because our goal is to show the positive impact instead of the precise numbers. We expect similar results when failing tests would be included.

7.4 RELATED WORK

The main inspiration for our work is a paper by Křikava et al. [153]. This paper describes unit test generation in R, based on executing examples in the base package and dependent projects. We extend their work by incorporating mutation coverage and applying it to the Python programming language.

Another inspiration came from a paper by Hejderup et al. [154]. These authors studied the impact of version changes within base packages of Java projects and which effect they can have on the dependent projects. The first research question in particular was relevant: *Do test suites cover the use of third-party libraries in projects?* Their findings show that only 13% of tests have less than 10% of dependency coverage. This suggests that the majority of projects have some tests exercising at least one dependency use.

The last source of inspiration stems from the work on *test amplification* [8]. Test amplification takes existing manually created unit tests, adding extra inputs and extra assertions with the goal to increase the coverage. Chapter 3 demonstrated the feasibility of test amplification for Python with a tool named AmPyfier.

7.5 CONCLUSIONS AND FUTURE WORKS

In this chapter, we assessed the potential of extending the test suite of a base package by looking at tests of dependent projects. We quantified the impact of these extra tests by calculating the increase in both line coverage and mutation coverage. We observed an improvement in line coverage for 64% of the projects; mutation coverage improved in all (100%) of the projects. We conclude that copying tests from dependent projects indeed strengthens test suites for base packages. This indicates that a tool which would create tests based on tests of dependent projects is a viable option. In future works the issue with execution time needs to be addressed. Selecting up front the tests of the dependent projects that indirectly access parts of the base package can have a major impact on this.

Chapter 8

Test Transplantation through Dynamic Test Slicing

This chapter is a revised version of an originally published paper in the *The 22nd IEEE International Working Conference on Source Code Analysis and Manipulation - New Ideas and Emerging Results* (*SCAM 2022*):

Test Transplantation through Dynamic Test Slicing
Mehrdad Abdi and Serge Demeyer
In *The 22nd IEEE International Working Conference on Source Code Analysis and Manipulation - New Ideas and Emerging Results* (*SCAM 2022*). , 2022.
URL: https://www.researchgate.net/publication/362868454.

ABSTRACT
It is proven that the test coverage of libraries can be expanded by using the existing test inputs from their dependent projects. In this chapter, we propose an approach for test slicing *that allows us to extract these test inputs, isolate them, and transplant them into the test suite in libraries. We dynamically execute the tests in the dependent project and create its graph of histories. Then, we traverse back from the interesting object state and collect the taken edges. Finally, we reverse the collected edges and create a sequence of method calls to reconstruct the same object state. We are still implementing a proof-of-concept in Pharo, but we mention some of the lessons learned so far.*

8.1 INTRODUCTION

Modern software repositories contain a test suite covering some of its code. In a software ecosystem, projects usually import modules from other libraries and invoke their public interfaces to fulfill their tasks. Our recent study (Chapter 7) illustrated that the tests in the user projects (source) indirectly cover some new parts in libraries (destination). This shows the opportunity of exploiting these test suites to amplify libraries' test coverage.

One easy solution to generate new unit tests is taking snapshots of the interesting object states during test execution and restoring them in test methods. However, these tests may not comply with the unit testing pattern in object-oriented programming languages and be less readable. In object-oriented programming languages, a *unit test* typically is initializing an instance of the class-under-test, updating it to the desired state, and finally, asserting the expected states. In addition to readability, the snapshots may depend on some classes from the source project that do not exist in the target project.

This chapter introduces a method to synthesize valid sequences of method calls to reconstruct the state of the object-under-test and other necessary objects based on tests in the source project. The method-call sequence will be installed as a unit test in the destination project. We call this process *dependency-base test transplantation*. In this chapter, we adopt the terminology from the original paper positioning the idea of code transplantation [10]. Hence, we refer to the dependent project (source) as *donor project*, the library (destination) as *host project*, and the object state to be transplanted as *organ*. We also refer to the test method containing the organ in the donor project as *donor test*.

We propose an algorithm to slice the donor test dynamically. First, we execute it and collect execution traces, including method calls. We form a *graph of histories* using these traces. Then, we spot the interesting object state (organ) in the graph and extract a subgraph. In extracting this subgraph, we traverse the graph backward, starting from the organ, and collect the list of edges. Finally, we reverse the collected edges and synthesize a sliced test method. In this process, we isolate the slices by mocking those classes not belonging to the host project. Once we obtain a precise slice containing the organ, we can transplant the test into the host project.

To conclude, we reduce the test transplantation problem into a test slicing problem and consequently a test slicing problem into a graph traversal problem. We also write about our learned lessons from implementing (work in progress) this approach in a proof-of-concept tool called SMALL-MINCE in Section 8.4.

8.2 BACKGROUND

Test amplification. Software repositories contain a considerable amount of test code written by developers to prevent regression in software evolution. Exploiting this source of knowledge to improve software testing is called *test amplification* [8]. SMALL-AMP (Chapter 2) is a test amplifier in Pharo that synthesizes new unit tests that increase the mutation coverage. SMALL-AMP is based on four main components: (1) a *profiler* which captures variables' type information, (2) an *input amplifier* which transforms existing test methods and generates new test inputs, (3) an *assertion amplifier* which regenerates the assertion statements, and (4) a *selector* that runs mutation testing and identifies the tests introducing new coverage.

Program slicing. Program slicing is finding a smaller set of statements from a program based on a slicing criteria [155]. A slicing criterion is defined as a target statement and a variable. So, the goal is to find a program slice in which the value of the target variable in the specified statement is identical to the same variable value in the same statement in the original program. The slicing algorithm uses the statements dependency graph and computes a slice by a backward graph traversing.

Static slicing of the program considers all possible inputs in a program and may produce slices with unnecessary statements. Dynamic program slicing [156] uses a specific input and performs the slicing based on the executed statements. As a result, it produces more precise slices, which help debug the program based on specified inputs.

Pharo. Pharo[1] is a pure object-oriented, dynamically typed language. In Pharo, all actions are done by sending *messages* to objects which is the equivalent to method invocation in other languages. The instance variables are private and can only be updated by the methods. The Pharo environment offers several facilities for dynamically inspecting the internal state of execution, making it well suited for dynamic program analysis.

8.3 DYNAMIC TEST SLICING

The traditional program slicing technique models the program as a set of statements and their relations. In this chapter, we model a program as a set of object states and their relations and create a *graph of histories*. The result of the slicing is a sequence of method calls that produces the same state of objects in the defined location. In this section, we introduce the object representation and the graph representation methods inspired by related work on the subject [40, 152, 157].

[1] `https://pharo.org/`

```
state := primitive | history | self | special
primitive := int | str | null | ...
history := uid, event*
event := <message,
            state:= version,
            state':= version,
            args:= state*,
            args':= state*,
            returns:= state>
version := A set of var_name ↦ state pairs
special := uid, predefined representations (lists, streams,...)
```

Figure 8.1: A model for object representation

8.3.1 Model

Language Model. We use an object-oriented language model, similar to what Pharo provides: Everything in the language are objects, all objects have a default constructor (`new`), and the instance variables in it are private and can be updated only by its methods. All objects are passed by their reference as arguments.

Object representation. Values in this language are either *primitive* values (integers, strings, ...) or objects. Each object is initiated by its constructor (`new`), and its state is updated by sending different messages. We refer to sending messages as *events* that create a new *version* of the receiver object. The set of all object's versions is its *history*. We adopt two representations for object values: (1) as a *history* which is a unique identifier uid and a list of *events* (2) as *versions* which are concrete state of objects: it includes a mapping of instance variables to their values; if the value is not a primitive, we represent it as an object history (Figure 8.1).

An event shows that the *message* with the *args* has been sent to the receiver object (identified by uid in history) when it had the internal state of *state*. This call has led it to the internal state *state'*, produced the return value of *returns*, and the states of arguments are changed to *args'* after the method call.

8.3.2 The Graph of Histories

Each event on an object creates a new version. We create an acyclic graph using the versions of all objects as nodes, and their relations as edges ($G =< V, E_{events}, E_{args}, E_{rets}, E_{argrets} >$). For simplicity, we skip showing the primitive values in the graph. The nodes are connected with four types of edges in this graph:

1. **Event edges:** The messages sent to the object which have updated its version. We use solid arrows to represent these edges and annotate them with the message name in Figure 8.2.

2. **Argument edges:** Shows that an object version is used as an argument in an event. Figure 8.2 uses hollow arrows to represent these edges.

3. **Return edges:** Shows that an object version is returned from a method call. We use dashed arrows to represent these edges in Figure 8.2. We also use the node after the method call as the source of these edges. For example, the event `md1` on the object d version 1 leads it to its version 2 and meanwhile returns the object o version 2. We draw a dashed edge from d version 2 to o version 2.

4. **Argument return edges:** Shows the state of arguments after a method call. We use dotted arrows to represent these edges. We also omit these edges when the state of the argument is not updated within an event. Similar to return edges, we use the node after the method call as the source of these edges.

Code Excerpt 8.1: Example code for dynamic test slicing

```
1  DriverTest >> test1
2    d := Driver new.
3    o := d md1: #ClassO.
4    o mo2.
5    o mo3.
6    d md2.
7    d md3: o
8
9  Driver >> md1: aSymbol
10   "..."
11   aSymbol = #ClassO ifTrue: [
12     retVal := ClassO new.
13     a := ClassA new.
14     x := ClassX new.
15     retVal mo1A: a X: x
16     ^ retVal ]
17   "..."
18
19 ClassO >> mo1A: aObj X: xObj
20   xObj mx1.
21   y := ClassY new.
22   xObj mx2Y: y.
23
24 ClassO >> mo2
25   r := ClassR new.
26   r mr1.
27   ^ r
28
29 ClassX >> mx2Y: yObj
30   z := ClassZ.
31   z mz1.
32   yObj my1Z: z.
33   ^ 1
```

Figure 8.2 shows the graph related to the code in Listing 8.1. As an example, we focus on the history of the object o: We see it at line 3, but looking deeper, this object is initialized at line 12 as the variable name $retVal$. Both variables o (line 3) and $retVal$ (line 12) refer to the same object in execution time. The message `#mo1A:X:` is sent to it using the arguments a version 1 and x version 1 (lines 15). This message brings this object to its second version. From the outcoming dotted edge (argument return), we understand

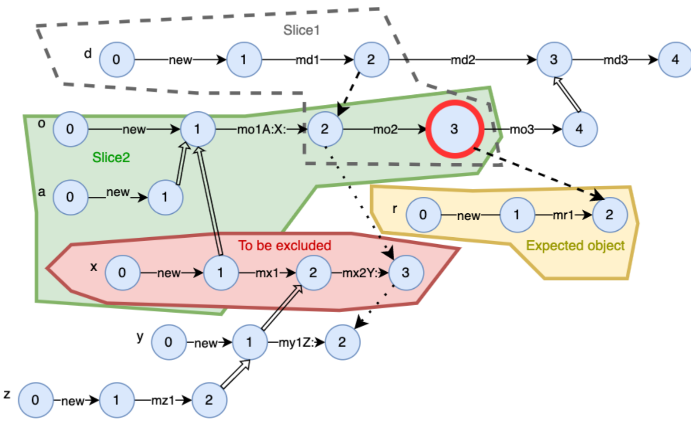


Figure 8.2: An example of versions graph

that the state of the object x is updated from version 1 to 3 inside `#mo1A:X:` (lines 19 to 22). The incoming dashed edge shows that version 2 of the object o is returned when the message `#md1:` is sent to the object d (line 3). The event edge of `#mo2` is the transition to version 3 for object o (line 4). The outgoing dashed edge shows that an object r version 2 is returned from this event (line 27). The object o is updated to version 4 by accepting `#mo3` (line 5). We also understand that the version 4 of o is used in calling `#md3` on the object d.

8.3.3 Slicing Tests

A *unit test* in an object-oriented language typically consists of initializing an instance of class-under-test, updating it to the desired state, and finally, asserting the expected states. In other words, a unit test is the history of the object-under-test. For example, the method `#test1` in Listing 8.1 is the history of the object d (the assertion statements are removed for simplicity).

If we recognize that a version of an object is interesting from the testing point of view, we can synthesize a unit test to regenerate the same state based on the graph of histories. We start from the target node, traverse the graph backward, and collect all events in order to reconstruct the same order.

In this method, we use a mapping V to store the list of objects to be traversed and their latest visited version. We also use the list of S including visited edges. Because of the space considerations, we only use an example to describe the algorithm. Our example graph considers version 3 of the object o as the slicing criteria. We start traversing from this node ($V = \{o \mapsto 3\}, S = []$):

There is only one node to be traversed in V. We pick it and see that it has only one incoming event edge and does not have any other incoming or outgoing edges to other visited objects, so it is safe to visit (There is an outgoing dashed edge to r version 2, but r is not in the list of nodes to be traversed: $r \notin V$). We update the version of o in V and also add the visited edge to S: $V = \{o \mapsto 2\}, S =$ [call o.mo2].

Again, we have one node in V. We see that it has two incoming edges: an event edge and a return edge. A version of the object can be obtained either by sending a message to its previous version or being returned from another event as a return value or argument. So we can take either of them. We explore both scenarios, which will create two different slices.

— We take the return edge. It says that the current state is the return value from calling `#md1` on the object d version 1. We discard o from V and add d: $V = \{d \mapsto 1\}, S =$ [call o.mo2, return of d.md1]. Traversing the remainder of this path is straightforward ($V = \{d \mapsto 0\}, S =$ [call o.mo2, return of d.md1, call d.new]). When all objects in V have reached their version 0, the algorithm finishes.

— We consider the event edge (calling `#mo1A:X:`). This edge will require two other objects a and x with version 1 as arguments. We check V to check if these objects are in the list of nodes to be traversed. If we found them in V, we check that their version is 1. If they exist in V and have a version higher than 1, we postpone traversing the current edge (`#mo1A:X:`) and continue traversing argument objects to bring their version to 1. Finally, if they do not exist in V, we add them to the list. In our case, we take the event edge and add a version 1 and x version 1 to V: $V = \{o \mapsto 1, a \mapsto 1, x \mapsto 1\}, S =$ [call o.mo2, call o.mo1A:X:].

  In the next step, we select one of the nodes from V. Let us take a and traverse its edge: $V = \{o \mapsto 1, a \mapsto 0, x \mapsto 1\}, S =$ [call o.mo2, call o.mo1A:X:, call a.new]. Traversing nodes x and o is also straightforward: $V = \{o \mapsto 0, a \mapsto 0, x \mapsto 0\}, S =$ [call o.mo2, call o.mo1A:X:, call a.new, call o.new, call x.new]).

We synthesize a test by reversing the traversed edges and synthesizing each event. Listing 8.2 shows the two sliced tests from this example.

Code Excerpt 8.2: Example of sliced tests

```
1 testSlice1
2     d := Driver new.
3     o := d md1: #ClassO.
4     o mo2.
5
6
7
```

```
8  testSlice2
9      x := ClassX new.
10     o := ClassO new.
11     a := ClassA new.
12     o mo1A: a X: x.
13     o mo2.
```

Method call sequence minimizing. In real traces, the number of events on the objects is considerable, and synthesizing all events may result in a lengthy unreadable test. We can analyze the $state$ changes in objects to reduce some extra events.

As an example, consider state preserving methods (getter) are called inside a loop. In the generated test slice, we will see plenty of unnecessary calls to these getter methods. We can minimize the slice length by detecting and removing these invocations. They can be detected by evaluating the difference between the state after the event and the state before each event: $state' - state == \varnothing$.

Assert generation. Generating the assertion statements to assert the primitive types is straightforward because the actual value of the primitive can be found in the event traces. However, there are opportunities for generating advanced assertions, such as asserting the *expected objects*. In the example graph in Figure 8.2, the object r is returned from sending `mo2` to o. Listing 8.3 shows a sliced test that reconstructs the expected object $r_expected$ and asserts it is equal to the returned value.

Code Excerpt 8.3: Asserting expected objects

```
1  testSlice2_withAsserts
2    o := ClassO new.
3    x := ClassX new.
4    a := ClassA new.
5    o mo1A: a X: x.
6    r_expected := ClassR new.
7    r_expected mr1.
8    r_actual := o mo2.
9    self assert: r_actual equalsTo: r_expected
```

8.3.4 Test Isolation

In test isolation, our goal is to exclude some classes from the sliced tests. We replace the excluded classes with mock objects.

In our example in Listing 8.1, we deduce that the class $ClassX$ needs to be excluded. This exclusion will invalidate the sliced test `testSlice2` because it depends on $ClassX$. We can see from the graph that the object x, which is an instance of this class, is passed as an argument to the method `mo1A:X:`. We also see that two other messages `mx1` and `mx2` are sent to x when it was being processed in the method `mo1A:X:` (the version of x is updated from 1 to 3 when it is returned). We create the mock object $xMocked$ that simulates the required behaviors. Listing 8.4 illustrates a test method with the mocked

$ClassX$ based on `Mocketry` mocking library[2].

Code Excerpt 8.4: Isolated sliced test

```
1  testSlice2_mocked
2      o := ClassO new.
3      xMocked := Mock new.
4      xMocked stub mx1 willReturn: nil.
5      (xMocked stub messageWith: (Instance of: ClassY)) willReturn: 1.
6      a := ClassA new.
7      o mo1A: a X: xMocked.
8      o mo2
```

8.4 PROOF-OF-CONCEPT

We implemented our algorithm in a proof-of-concept tool called SMALL-MINCE[3] in the Pharo language (work still in progress). The tool consists of three main components: (1) tracer, (2) slicer, and (3) synthesizer.

The tracer component manipulates the classes in the project to enable them to log the details of message invocations. We employed *method proxies* to capture the receiver and arguments state before and after an invocation. We use an integer instance variable as the object's version (increases by each event), and also a stack to reject the internal method invocations. After the manipulation, the donor test is executed, and traces are collected.

The slicer component creates the graph and extracts some subgraphs based on the identified target organ (as the program input). The graph does not need to be entirely loaded in the memory at this stage, and we can mine the logs to traverse it. After traversing the graph and obtaining the list of events, we minimize it by skipping the state-preserving events.

The synthesizer module converts the traversed paths to test methods, installs them in the system, and verifies that they are runnable. At the current proof-of-concept, we do not generate assertion statements, and we will use the assertion-amplification component from another project (SMALL-AMP) in the host project after transplantation.

The main lessons we have learned so far from this proof-of-concept are:

- Tracer needs to manipulate the classes in advance. However, it is difficult to find a list of classes to be manipulated. We use all defined classes in the project as default.
- Manipulating the system classes like `Array`, `Stream` and `Dictoriary` is challenging. It is why we skip manipulating these classes and use some predefined representa-

[2] https://github.com/dionisiydk/Mocketry
[3] https://github.com/mabdi/small-mince

tion for them. However, the number of system classes is considerable, and language-specific knowledge bases are required beforehand.

— When all methods in a project are proxied, the program's execution gets dramatically slow. This shows that it is important to keep the instructions in the method proxies as minimized as possible.

8.5 RELATED WORK

The works by Artzi et al. [158], and Zhang et al. [152] use a similar graph representation to extract a model from the class-under-test and guide a random-based test generator. However, we use this graph to reconstruct a call sequence as a test slice. Staff et al. [151] use a capture and replay technique to replace the environment part of the program with mocks and create unit tests from slower system tests.

GENTHAT is a unit test extraction tool for R language [153]. It analyses execution traces from running in example code and reverse-dependency projects and synthesizes new unit tests. However, they only work with primitive data types.

Messaoudi et al. introduce DS3, an approach to slice system level test by a static analysis enhanced with log analysis [159]. In their work, it is considered that the system test is huge, and the dynamic slicing is not possible, so they use the software-under-test as a black-box and do not analyze it. They only consider the code in the test method and the log produced in the execution. Therefore, they only generate smaller tests with the same statements from the original test method.

Generating differential unit tests by carving [160, 161] represents related work that captures the state of the program before and after of execution of the unit. When the unit evolves, the recorded (carved) pre-state is loaded to memory and the unit is executed, and the state after is compared to the carved post-state. Tiwari et al. [162] also introduce PANKTI which observes the program execution in production and generates a set of differential tests that expands the test coverage. It manipulates the methods under test and serializes the state of receiving object, arguments, and the return value. Then it generates a test method by deserializing the states from trace files and reconstructing the execution state. We see our proposed approach as a complement to these two works, whereas we can use objects' history to create a sequence of method calls to reconstruct the same state instead of deserializing the states from files.

8.6 CONCLUSION

This chapter addressed the problem of test transplantation from a dependent project to the imported libraries. We choose a test in the dependent project that amplifies the cov-

erage in the library, then we slice it, isolate it if necessary, and move it to the library's test suite. We reduced the test slicing problem to a subgraph finding and backward traversing problem. We illustrated the proposed traversing algorithm using an example and also mentioned our learned lessons from the implementation of a proof-of-concept.

In the future, we will evaluate the algorithm for the test transplantation problem on real projects. In addition, we will explore different use-cases for test slicing: (1) Slicing amplified tests can improve their readability by removing unnecessary statements. (2) Amplifying sliced tests can increase the input amplification surface and, consequently, the test amplification performance. (3) By slicing tests in code clones, similarity-based test transplantation can be possible where we can cover untested clones.

Part IV

Conclusion

Chapter 9

Conclusion

Modern software projects do not usually live in isolation, interacting with each other and forming a larger socio-technical unit called *software ecosystems*. While software ecosystems are gaining more importance gradually, we attempted to address the problem of strengthening software tests in the context of software ecosystems. In this dissertation, we pursued three main goals:

Expanding the state-of-the-art in test amplification. In *test amplification*, the valuable source of knowledge in the existing test suite is exploited to enhance testing based on an engineering goal. We adapted DSpot, the state-of-the-art test amplifier in Java, to dynamically typed languages of Pharo and Python by exploiting *dynamic type profiling*. In this exploration, we also addressed the problems of unnecessary oracles, test input explosion, and slow mutation testing.

Making steps toward zero-touch test amplification. A test amplifier should be autonomous and act as a virtual developer in the team. Developers' involvement should be reduced as possible, and their attention is needed only for approving the result. We proposed the zero-touch test amplification proof-of-concept solution by integrating Small-Amp with GitHub-Actions. In this proof-of-concept, we introduced a test method prioritizing heuristic, sharding, and crash recovery mechanisms. We also introduced zero-touch mutation testing in the Pharo ecosystem and also introduced a test amplification DevBot which submits pull requests derived from the output of the test amplification.

Demonstrating test transplantation. Test amplification based on DSpot is not the only solution for getting advantages of the knowledge source in the software ecosystems. We

focused on the dependency relationship between projects and introduced the notion of *dependency-based test transplantation* and a method for *dynamically slicing test methods*.

These three goals are derived from our thesis: there is a symbiotic relationship between test amplifiers and software ecosystems, and we investigated this relationship based on two feedback loops: (1) The test amplifier is fed by knowledge extracted *out* of the ecosystem, like the source code and interproject dependencies. (2) The test amplifier provides improvements *in* the available tests to reduce the impact of software defects.

Our observations during these studies show the presence of great opportunities in the synergy between software evolution in the context of ecosystems and software testing. The existing code in projects and their relations are two knowledge sources we exploited in our studies. However, more knowledge sources can be exploited in developing intelligent tools in future works. Zero-touch test amplifiers can have a significant role in the team. However, we observed that there is still room for improvement. One of the missing aspects is defining a model for a good test. Current engineering goals, such as code coverage and mutation analysis, identify bad tests and show where they need to improve. However, when a tool suggests a new test, it should meet different requirements like: *Is the code readable and maintainable? Are the chosen names for the test understandable? Is a human-understandable comment provided when it is needed? Does it conform to the testing style of the team? Should this test be merged into an existing test or be a new test? Should this test be split into smaller tests?* and so forth. A model for an acceptable test still needs to be defined by considering these requirements. In Chapter 2, we manually translated a generated test based on mutation coverage into developer-understandable tests.

9.1 FINAL WORDS

Around 80 years ago, the first programmable general-purpose computers were created. The early computers were limited in speed and memory, and machine code was used to program them. However, within a few years, software engineers realized that coding in low-level machine language is cumbersome and needs a lot of effort, and eventually, higher-level programming languages were created. Programmers now write programs in a high level code which gets compiled into machine-understandable code. Yet, we are in an era where the chips on $15 smart lamps have enough computing power to load and run a classic PC game[1]. However, in essence we are still stuck with the same development paradigm: loading instructions in some high-level language into our computers to tell them exactly what we want them to do.

Fortunately, humans are taking another step. We have realized that current program-

[1] https://uk.pcmag.com/games/133930/you-can-run-doom-on-a-chip-from-a-15-ikea-smart-lamp

ming tasks like writing `if` and `for` statements, creating methods and classes, or writing unit tests, are too low-level. Similar to the cover photo —which is generated by Artificial Intelligence from a textual description— people should just sketch what they need, ask an intelligent agent to materialize their intentions. People should only respond with what is good and what is bad, and computers should satisfy these intentions. Although we are still far from that vision, the software engineering community made some promising steps in that direction. Human developers have developed a vast number of projects in open source software ecosystems, and their knowledge is accessible through the existing `big code` repositories. These repositories are mined continuously to extract software development models fed into intelligent agents which are the basis for a "zero-touch" approach towards software development.

The future of software engineering looks bright. With this dissertation, we have made s small yet crucial step towards that future.

Bibliography

[1] Konstantinos Manikas and Klaus Hansen. Software ecosystems – a systematic literature review. *Journal of Systems and Software*, 86:1294–1306, 05 2013. doi: 10.1016/j.jss.2012.12.026. (Cited on pages 1 and 144).

[2] Tom Mens, Maálick Claes, Philippe Grosjean, and Alexander Serebrenik. *Studying Evolving Software Ecosystems based on Ecological Models*, pages 297–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-45398-4. doi: 10.1007/978-3-642-45398-4_10. URL https://doi.org/10.1007/978-3-642-45398-4_10. (Cited on page 2).

[3] Oscar Franco-Bedoya, David Ameller, Dolors Costal, and Xavier Franch. Open source software ecosystems: A systematic mapping. *Information and Software Technology*, 91:160–185, 2017. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.07.007. URL https://www.sciencedirect.com/science/article/pii/S0950584917304512. (Cited on page 2).

[4] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings ASE 2009* (*International Conference on Automated Software Engineering*), pages 433–444. IEEE CS, 2009. doi: 10.1109/ASE.2009.17. (Cited on pages 2 and 14).

[5] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4), 2006. doi: 10.1109/MS.2006.117. (Cited on pages 2 and 14).

[6] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *International Journal on Empirical Software Engineering*, 16(3):325–364, 2011. doi: 10.1007/s10664-010-9143-7. (Cited on pages 2 and 14).

[7] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40 (11):1100–1125, 2014. doi: 10.1109/TSE.2014.2342227. (Cited on pages 2 and 14).

[8] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398, 2019. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2019.110398. URL `http://www.sciencedirect.com/science/article/pii/S0164121219301736`. (Cited on pages 2, 14, 15, 16, 70, 72, 74, 106, 151, and 155).

[9] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering, Springer Verlag*, 2019. (Cited on pages 2, 7, 14, 15, 23, 34, 35, 37, 54, 58, 63, and 126).

[10] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 257–269, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2771796. URL `https://doi.org/10.1145/2771783.2771796`. (Cited on pages 2 and 154).

[11] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM transactions on autonomous and adaptive systems (TAAS)*, 4(2):1–42, 2009. (Cited on page 3).

[12] Sigrid Eldh. Test automation improvement model - taim 2.0. In *ICSTW-NEXTA 2020 (IEEE International Conference on Software Testing, Verification and Validation Workshops – NEXTA)*, pages 334–337, 2020. doi: 10.1109/ICSTW50294.2020.00060. (Cited on pages 4 and 105).

[13] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by example*. Lulu. com, 2010. (Cited on pages 4, 15, and 17).

[14] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013. ISBN 978-3-9523341-6-4. URL `http://books.pharo.org/deep-into-pharo/`. (Cited on pages 4, 15, 17, and 126).

[15] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live programming and software evolution: Questions during a programming change task. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 30–41, 2019. (Cited on pages 4 and 107).

[16] Steven L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-6265-8. URL http://dl.acm.org/citation.cfm?id=2662726.2662735. (Cited on pages 4 and 107).

[17] Christopher Parnin. A History of Live Programming, 01 2013. URL http://liveprogramming.github.io/liveblog/2013/01/a-history-of-live-programming/. (Cited on pages 4 and 107).

[18] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and TrendsÂ® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010. (Cited on pages 5, 6, and 136).

[19] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013. (Cited on page 5).

[20] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):1–37, Jul 2018. ISSN 0360-0300. doi: 10.1145/3212695. URL http://dx.doi.org/10.1145/3212695. (Cited on page 5).

[21] Anita D Carleton, Erin Harper, John E Robert, Mark H Klein, Dionisio De Niz, Edward Desautels, John B Goodenough, Charles Holland, Ipek Ozkaya, Douglas Schmidt, et al. Architecting the future of software engineering: A national agenda for software engineering research and development. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA, 2021. (Cited on page 6).

[22] Richard C. Waters. The programmer's apprentice: A session with kbemacs. *IEEE Transactions on Software Engineering*, (11):1296–1320, 1985. (Cited on pages 6 and 136).

[23] Charles Rich and Richard C. Waters. The programmer's apprentice: A research overview. *Computer*, 21(11):10–25, 1988. (Cited on pages 6 and 136).

[24] Geert Heyman, Rafael Huysegems, Pascal Justen, and Tom Van Cutsem. Natural language-guided programming. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2021, page 39–55, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391108. doi: 10.1145/3486607.3486749. URL https://doi.org/10.1145/3486607.3486749. (Cited on pages 6 and 136).

[25] Benjamin Danglot. *Automatic unit test amplification for DevOps*. PhD thesis, Université de Lille, 2019. (Cited on page 7).

[26] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. Dspot: Test amplification for automatic assessment of computational diversity. *CoRR: a computing research repository*, abs/1503.05807, 2015. URL `http://arxiv.org/abs/1503.05807`. (Cited on pages 7, 14, 23, and 63).

[27] K. Beck. *Test-driven Development: By Example*. Addison-Wesley signature series. Addison-Wesley, 2003. ISBN 9780321146533. URL `https://books.google.be/books?id=gFgnde_vwMAC`. (Cited on page 14).

[28] Andrey Agibalov. What is a normal "functional lines of code" to "test lines of code" ratio?, 2015. [on line] https://softwareengineering.stackexchange.com/questions/156883/ — last accessed In April 2021. (Cited on page 14).

[29] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2016. doi: 10.1109/TSE.2016.2597136. (Cited on page 14).

[30] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130, 2012. doi: 10.1109/ICST.2012.92. (Cited on page 14).

[31] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016. doi: https://doi.org/10.1002/stvr.1601. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1601`. (Cited on pages 14, 63, and 72).

[32] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012. doi: https://doi.org/10.1002/stvr.435. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.435`. (Cited on pages 14, 62, 63, and 72).

[33] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA, 1983. ISBN 0201113716. (Cited on page 15).

[34] Steven Costiou, Vincent Aranega, and Marcus Denker. Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming*, 4(3), 2020. (Cited on pages 15, 30, and 74).

[35] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Test amplification in the pharo smalltalk ecosystem. In *Proceedings of the 14th Edition of the International Workshop on Smalltalk Technologies, IWST*, volume 19, pages 1–7, 2019. (Cited on pages 15, 70, and 104).

[36] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 112:275–378, 2019. ISSN 0065-2458. doi: 10.1016/bs.adcom.2018.03.015. URL `http://www.sciencedirect.com/science/article/pii/S0065245818300305`. (Cited on pages 16, 70, and 104).

[37] Stéphane Ducasse. *Pharo with Style*. Square Bracket Associates, 2019. (Cited on page 19).

[38] Stefan Fischer, Evelyn Nicole Haslinger, Markus Zimmermann, and Hannes Thaller. An empirical evaluation for object initialization of member variables in unit testing. In *Proceedings VST 2020* (*IEEE Workshop on Validation, Analysis and Evolution of Software Tests*), pages 8–11, 2020. doi: 10.1109/VST50071.2020.9051634. (Cited on page 21).

[39] Paolo Tonella. Evolutionary testing of classes. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*, 2004. doi: 10.1145/1007512.1007528. URL `http://dx.doi.org/10.1145/1007512.1007528`. (Cited on pages 25 and 73).

[40] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. *Lecture Notes in Computer Science*, pages 380–403, 2006. ISSN 1611-3349. doi: 10.1007/11785477_23. URL `http://dx.doi.org/10.1007/11785477_23`. (Cited on pages 26, 28, 63, and 155).

[41] F. Palomba. Flaky tests: Problems, solutions, and challenges. In *BENEVOL*, 2019. (Cited on page 27).

[42] Stéphane Ducasse. Evaluating message passing control techniques in smalltalk. *Journal of Object Oriented Programming*, 12:39–50, 1999. (Cited on page 30).

[43] Mariano Martinez Peck, Noury Bouraqadi, Luc Fabresse, Marcus Denker, and Camille Teruel. Ghost: A uniform and general-purpose proxy implementation. *Science of Computer Programming*, 98:339–359, 2015. (Cited on page 30).

[44] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. *Journal of Object Technology*, 6:275–295, 10 2007. doi: 10.5381/jot.2007.6.9.a14. URL `http://www.jot.fm/contents/issue_2007_10/paper14.html`. (Cited on page 30).

[45] Hernán Wilkinson, Nicolás Chillo, and Gabriel Brunstein. Mutation testing, Sep 2009. European Smalltalk User Group (ESUG 09). Brest, France. http://www.esug.org/data/ESUG2009/Friday/Mutation_Testing.pdf. (Cited on pages 34, 107, and 127).

[46] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P Black, and Anne Etien. Rotten green tests. In *41th International Conference on Software Engineering*, ICSE '19, pages 500–511. IEEE, May 2019. ISBN 978-1-7281-0869-8. URL https://hal.inria.fr/hal-02002346. (Cited on page 35).

[47] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000. ISBN 978-0792386827. doi: 10.1007/978-3-642-29044-2. (Cited on page 61).

[48] Suresh Thummalapenta, Madhuri R Marri, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. Retrofitting unit tests for parameterized unit testing. In *International Conference on Fundamental Approaches to Software Engineering*, pages 294–309. Springer, 2011. (Cited on pages 62 and 63).

[49] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2):73–96, 2005. (Cited on pages 62 and 63).

[50] Matthew Patrick and Yue Jia. Kd-art: Should we intensify or diversify tests to kill mutants? *Information and Software Technology*, 81:36–51, 2017. (Cited on page 62).

[51] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78, 2014. (Cited on page 62).

[52] Mauro Pezze, Konstantin Rubinov, and Jochen Wuttke. Generating effective integration test cases from unit ones. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 11–20. IEEE, 2013. (Cited on page 62).

[53] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374, 2011. (Cited on page 63).

[54] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 international symposium on software testing and analysis*, pages 309–319, 2012. (Cited on pages 63, 115, and 123).

[55] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 910–913, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2803206. URL `https://doi.org/10.1145/2786805.2803206`. (Cited on pages 63, 72, 115, and 123).

[56] Franck Chauvel, Brice Morin, and Enrique Garcia-Ceja. Amplifying integration tests with camp. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 283–291, 2019. doi: 10.1109/ISSRE.2019.00036. (Cited on page 63).

[57] Zhihong Xu, Myra B Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1365–1372, 2010. (Cited on page 63).

[58] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 406–413. IEEE, 2009. (Cited on page 63).

[59] Hiroaki Yoshida, Susumu Tokumoto, Mukul R Prasad, Indradeep Ghosh, and Tadahiro Uehara. Fsx: fine-grained incremental unit test generation for c/c++ programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 106–117, 2016. (Cited on page 63).

[60] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2018. doi: 10.1109/TEVC.2017.2693219. (Cited on page 63).

[61] Carolin Brandt and Andy Zaidman. Developer-centric test amplification. *Empirical Software Engineering*, 27(4):96, May 2022. ISSN 1573-7616. doi: 10.1007/s10664-021-10094-2. URL `https://doi.org/10.1007/s10664-021-10094-2`. (Cited on pages 63, 72, 104, and 138).

[62] Nienke Nijkamp, Carolin Brandt, and Andy Zaidman. Naming amplified tests based on improved coverage. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 237–241. IEEE, 2021. (Cited on pages 63, 94, and 123).

[63] Wessel Oosterbroek, Carolin Brandt, and Andy Zaidman. Removing redundant statements in amplified test cases. In *2021 IEEE 21st International Working Confer-*

ence on Source Code Analysis and Manipulation (SCAM), pages 242–246. IEEE, 2021. (Cited on pages 63 and 123).

[64] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, 2007. doi: 10.1109/ICSE.2007.37. (Cited on pages 63 and 104).

[65] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013. doi: 10.1109/TSE.2012.14. (Cited on page 63).

[66] A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018. doi: 10.1109/TSE.2017.2663435. (Cited on page 63).

[67] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. Automated unit test generation for python. In Aldeida Aleti and Annibale Panichella, editors, *Search-Based Software Engineering*, pages 9–24, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59762-7. (Cited on pages 63, 71, and 76).

[68] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Jseft: Automated javascript unit test generation. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102595. (Cited on page 63).

[69] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012. doi: 10.1109/TSE.2011.93. (Cited on page 63).

[70] J. T. P. Wibowo, B. Hendradjaya, and Y. Widyani. Unit test code generator for lua programming language. In *2015 International Conference on Data and Software Engineering (ICoDSE)*, pages 241–245, 2015. doi: 10.1109/ICODSE.2015.7437005. (Cited on page 63).

[71] Stefan Mairhofer, Robert Feldt, and Richard Torkar. Search-based software testing and test data generation for a dynamic programming language. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 1859–1866, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305570. doi: 10.1145/2001576.2001826. URL `https://doi.org/10.1145/2001576.2001826`. (Cited on page 63).

[72] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and Rene Just. Practical mutation testing at scale: A view from google. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi: 10.1109/TSE.2021.3107634. (Cited on pages 64 and 66).

[73] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 163–171, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356596. doi: 10.1145/3183519.3183521. URL `https://doi.org/10.1145/3183519.3183521`. (Cited on pages 64, 66, and 136).

[74] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 268–277, 2021. doi: 10.1109/ICSE-SEIP52600.2021.00036. (Cited on pages 64, 66, and 136).

[75] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005. (Cited on page 65).

[76] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Adopting program synthesis for test amplification. In *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop, Brussels, Belgium*. published at http://ceur-ws.org, 2019. URL `http://ceur-ws.org/Vol-2605/11.pdf`. (Cited on page 66).

[77] Zhong Xi Lu, Sten Vercammen, and Serge Demeyer. Semi-automatic test case expansion for mutation testing. In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 1–7. IEEE, 2020. (Cited on pages 66 and 70).

[78] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. Suggestions on test suite improvements with automatic infection and propagation analysis. *arXiv preprint arXiv:1909.04770*, 2019. (Cited on pages 66, 104, 126, and 131).

[79] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. Descartes: A pitest engine to detect pseudo-tested methods: Tool demonstration. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 908–911, 2018. doi: 10.1145/3238147.3240474. (Cited on page 66).

[80] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. Fast type reconstruction for dynamically typed programming languages. In James Noble, editor, *Proceedings of the 5th Symposium on Dynamic Languages, DLS 2009, October 26, 2010, Orlando, Florida, USA*, pages 69–78. ACM, 2009. doi: 10.1145/1640134.1640145. URL `https://doi.org/10.1145/1640134.1640145`. (Cited on page 66).

[81] Hernán Wilkinson. Vm support for live typing: Automatic type annotation for dynamically typed languages. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362573. doi: 10.1145/3328433.3328443. URL `https://doi.org/10.1145/3328433.3328443`. (Cited on page 66).

[82] G. Grano, C. De Iaco, F. Palomba, and H. C. Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 336–347, 2020. doi: 10.1109/ICSME46990.2020.00040. (Cited on page 66).

[83] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997. (Cited on page 70).

[84] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011. (Cited on pages 70, 76, and 104).

[85] Benoit Baudry, Simon Allier, Marcelino Rodriguez-Cancio, and Martin Monperrus. Dspot: Test amplification for automatic assessment of computational diversity. *arXiv preprint arXiv:1503.05807*, 2015. (Cited on pages 70 and 104).

[86] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, 24(4):2603–2635, 2019. (Cited on pages 70 and 72).

[87] Mehrdad Abdi, Henrique Rocha, Serge Demeyer, and Alexandre Bergel. Small-amp: Test amplification in a dynamically typed language. *Empirical Software Engineering*, 27(6):1–55, 2022. (Cited on pages 70, 72, 104, 108, and 126).

[88] Chris Laffra. Auger: Automated unittest generation for python. `https://github.com/laffra/auger`, 2016. URL `https://github.com/laffra/auger`. [Online; accessed 14-September-2021]. (Cited on pages 71 and 76).

[89] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017. (Cited on page 71).

[90] Guido Van Rossum et al. Python programming language. In *USENIX annual technical conference*, volume 41, page 36, 2007. (Cited on page 71).

[91] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Reproducible crashes: Fuzzing pharo by mutating the test methods. In *International Workshop on Smalltalk Technologies, IWST*, 2020. (Cited on pages 72, 104, 105, 108, 115, 123, and 129).

[92] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empirical Software Engineering*, 25(4):2379–2415, Jul 2020. ISSN 1573-7616. doi: 10.1007/s10664-019-09794-7. URL `https://doi.org/10.1007/s10664-019-09794-7`. (Cited on page 72).

[93] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. Directed test suite augmentation: an empirical investigation. *Software Testing, Verification and Reliability*, 25(2):77–114, 2015. doi: https://doi.org/10.1002/stvr.1562. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1562`. (Cited on page 72).

[94] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, 23(4), sep 2014. ISSN 1049-331X. doi: 10.1145/2652483. URL `https://doi.org/10.1145/2652483`. (Cited on page 72).

[95] Franck Chauvel, Brice Morin, and Enrique Garcia-Ceja. Amplifying integration tests with camp. In *2019 IEEE 30th International Symposium on Software Reliability Engineering* (*ISSRE*), pages 283–291, 2019. doi: 10.1109/ISSRE.2019.00036. (Cited on page 72).

[96] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 85–96, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588230. doi: 10.1145/1831708.1831719. URL `https://doi.org/10.1145/1831708.1831719`. (Cited on page 72).

[97] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, pages 380–403. Springer, 2006. (Cited on pages 72 and 73).

[98] inspect — inspect live objects — python 3.9.7 documentation. `https://docs.python.org/3/library/inspect.html`, 2021. (Accessed on 09/13/2021). (Cited on page 75).

[99] Jeff Knupp. *Writing Idiomatic Python 3*. Jeff Knupp; 1st edition (November 30, 2013), 2013. (Cited on pages 75 and 94).

[100] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007. (Cited on page 76).

[101] Python Software Foundation. typing - support for type hints - python 3.9.7 documentation. `https://docs.python.org/3/library/typing.html`, 2021. URL `https://docs.python.org/3/library/typing.html`. [Online; accessed 14-September-2021]. (Cited on page 82).

[102] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990. (Cited on pages 86, 126, and 131).

[103] Richard A DeMillo, A Jefferson Offutt, et al. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991. (Cited on pages 86, 126, and 131).

[104] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on software Engineering*, 18(8):717, 1992. (Cited on pages 86, 126, and 131).

[105] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering*, 14(2):131–164, 2009. (Cited on page 95).

[106] Robert K. Yin. *Case Study Research: Design and Methods, 3 edition*. Sage Publications, —, 2002. (Cited on page 95).

[107] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer, 2008. (Cited on page 104).

[108] Benjamin Danglot, Oscar Luis Vera-Pérez, Benoit Baudry, and Martin Monperrus. Automatic test improvement with dspot: a study with ten mature open-source projects. *Empirical Software Engineering*, 24(4):2603–2635, Aug 2019. doi: 10.1007/s10664-019-09692-y. URL `https://doi.org/10.1007/s10664-019-09692-y`. (Cited on pages 104 and 106).

[109] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empirical Software Engineering*, 25(4):2379–2415, 2020. (Cited on pages 104 and 124).

[110] Ebert Schoofs, Mehrdad Abdi, and Serge Demeyer. Ampyfier: Test amplification in python. *Journal of Software: Evolution and Process*, n/a(n/a):e2490, 2022. doi: 10.1002/smr.2490. (Cited on page 104).

[111] STAMP project. Use cases validation report v3, 2019. [on line] https://github.com/STAMPproject/docs-forum/blob/master/docs/ — last accessed In February 2022. (Cited on pages 104 and 107).

[112] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 55–66, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. doi: 10.1145/2642937.2643002. URL https://doi.org/10.1145/2642937.2643002. (Cited on pages 104 and 124).

[113] Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. *Pharo by Example*. Square Bracket Associates, c/o Oscar Nierstrasz, 2010. (Cited on pages 106 and 126).

[114] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Lulu. com, 2013. (Cited on pages 106 and 109).

[115] André N. Meyer, Earl T. Barr, Christian Bird, and Thomas Zimmermann. Today was a good day: The daily life of software developers. *IEEE Transactions on Software Engineering*, 47(5):863–880, 2021. doi: 10.1109/TSE.2019.2904957. (Cited on page 107).

[116] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of visual studio usage in practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134, 2016. doi: 10.1109/SANER.2016.39. (Cited on page 107).

[117] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35, 2015. doi: 10.1109/ICPC.2015.12. (Cited on page 107).

[118] Stéphane Ducasse, Manuel Oriol, and Alexandre Bergel. Challenges to support automated random testing for dynamically typed languages. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 9:1–9:6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1050-5. doi: 10.1145/2166929.2166938. URL http://doi.acm.org/10.1145/2166929.2166938. (Cited on page 107).

[119] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. doi: 10.1145/

2635868.2635920. URL https://doi.org/10.1145/2635868.2635920. (Cited on page 108).

[120] August Shi, Jonathan Bell, and Darko Marinov. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 112–122, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362245. doi: 10.1145/3293882.3330568. URL https://doi.org/10.1145/3293882.3330568. (Cited on page 108).

[121] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. doi: 10.1145/2970276.2970358. URL https://doi.org/10.1145/2970276.2970358. (Cited on page 109).

[122] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous delivery practices in a large financial organization. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 519–528. IEEE, 2016. (Cited on page 109).

[123] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE international conference on software maintenance and evolution*, pages 401–405. IEEE, 2014. (Cited on page 109).

[124] GitHub. Github actions usage limits, billing, and administration, 2022. [on line] https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration — last accessed In February 2022. (Cited on pages 109 and 118).

[125] GitHub. About custom actions, 2022. [on line] https://docs.github.com/en/actions/creating-actions/about-custom-actions — last accessed In May 2022. (Cited on page 109).

[126] GitHub. Storing workflow data as artifacts, 2022. [on line] https://docs.github.com/en/actions/using-workflows/storing-workflow-data-as-artifacts — last accessed In May 2022. (Cited on page 109).

[127] GitHub. Reusing workflows, 2022. [on line] https://docs.github.com/en/actions/using-workflows/reusing-workflows — last accessed In May 2022. (Cited on page 109).

[128] Wikipedia. Fitness proportionate selection, 2020. [on line] https://en.wikipedia.org/wiki/Fitness_proportionate_selection — last accessed In February 2022. (Cited on pages 111 and 112).

[129] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application heartbeats for software performance and health. *SIGPLAN Not.*, 45(5):347–348, jan 2010. ISSN 0362-1340. doi: 10.1145/1837853.1693507. URL https://doi.org/10.1145/1837853.1693507. (Cited on page 115).

[130] Samuel Kounev, Peter Lewis, Kirstie L Bellman, Nelly Bencomo, Javier Camara, Ada Diaconescu, Lukas Esterle, Kurt Geihs, Holger Giese, Sebastian Götz, et al. The notion of self-aware computing. In *Self-Aware Computing Systems*, pages 3–16. Springer, 2017. (Cited on page 115).

[131] Jianyi Zhou, Junjie Chen, and Dan Hao. Parallel test prioritization. *ACM Trans. Softw. Eng. Methodol.*, 31(1), sep 2021. ISSN 1049-331X. doi: 10.1145/3471906. URL https://doi.org/10.1145/3471906. (Cited on page 123).

[132] G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, and B. van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 96:141–155, 2014. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2013.10.008. URL https://www.sciencedirect.com/science/article/pii/S0167642313002797. Special issue on Advances in Smalltalk based Systems. (Cited on page 123).

[133] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. Sista: Saving optimized code in snapshots for fast start-up. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, page 1–11, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353403. doi: 10.1145/3132190.3132201. URL https://doi.org/10.1145/3132190.3132201. (Cited on page 123).

[134] Martín Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. In *IWST-2013-5th International Workshop on Smalltalk Technologies*, 2013. (Cited on page 123).

[135] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to design a program repair bot? insights from the repairnator project. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 95–104, 2018. (Cited on page 123).

[136] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Chapter six - mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275–378. Elsevier, 2019. doi: https://doi.org/10.1016/bs.adcom.2018.03.015. URL https://www.sciencedirect.com/science/article/pii/S0065245818300305. (Cited on page 125).

[137] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2017. doi: 10.1109/TSE.2016.2597136. (Cited on pages 126 and 131).

[138] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will my tests tell me if i break this code? In *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, pages 23–29. IEEE, 2016. (Cited on page 127).

[139] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3):1195–1225, Jun 2019. ISSN 1573-7616. doi: 10.1007/s10664-018-9653-2. URL https://doi.org/10.1007/s10664-018-9653-2. (Cited on page 127).

[140] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 163–171, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356596. doi: 10.1145/3183519.3183521. URL https://doi.org/10.1145/3183519.3183521. (Cited on pages 129 and 131).

[141] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52 (6):1–35, 2019. (Cited on page 129).

[142] Wei Ma, Thomas Laurent, Miloš Ojdanić, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. Commit-aware mutation testing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 394–405. IEEE, 2020. (Cited on page 131).

[143] GitHub. Github docs: About apps, 2022. [on line] https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps. (Cited on page 135).

[144] Linda Erlenhov, Francisco Gomes de Oliveira Neto, and Philipp Leitner. An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software*

Engineering, ESEC/FSE 2020, page 445–455, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409680. URL `https://doi.org/10.1145/3368089.3409680`. (Cited on page 136).

[145] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. Current and future bots in software development. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 7–11, 2019. doi: 10.1109/BotSE.2019.00009. (Cited on page 136).

[146] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. AddissonWesley, 2000. (Cited on page 144).

[147] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62. (Cited on page 144).

[148] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2 – 12, 2017. doi: 10.1109/SANER.2017.7884604. (Cited on page 144).

[149] Hongyu Zhai, Casey Casalnuovo, and Prem Devanbu. Test coverage in python programs. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 116–120, 2019. doi: 10.1109/MSR.2019.00027. (Cited on page 144).

[150] Jetbrains. The State of Developer Ecosystem 2021, 2021. URL `https://www.jetbrains.com/lp/devecosystem-2021/`. Last accessed: March 2022. (Cited on page 144).

[151] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 114–123, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139934. doi: 10.1145/1101908.1101927. URL `https://doi.org/10.1145/1101908.1101927`. (Cited on pages 150 and 162).

[152] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 353–363, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305624. doi: 10.1145/2001420.2001463. URL `https://doi.org/10.1145/2001420.2001463`. (Cited on pages 150, 155, and 162).

[153] Filip Křikava and Jan Vitek. Tests from traces: Automated unit test extraction for r. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 232–241, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213863. URL `https://doi.org/10.1145/3213846.3213863`. (Cited on pages 150, 151, and 162).

[154] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software*, 183:111097, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111097. URL `https://www.sciencedirect.com/science/article/pii/S0164121221001941`. (Cited on page 151).

[155] Mark Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7): 446–452, jul 1982. ISSN 0001-0782. doi: 10.1145/358557.358577. URL `https://doi.org/10.1145/358557.358577`. (Cited on page 155).

[156] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988. ISSN 0020-0190. doi: https://doi.org/10.1016/0020-0190(88)90054-3. URL `https://www.sciencedirect.com/science/article/pii/0020019088900543`. (Cited on page 155).

[157] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014. (Cited on page 155).

[158] Shay Artzi, Michael D Ernst, Adam Kie Zun, Carlos Pacheco Jeff, and H Perkinsmit Csail. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *In 1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*. Citeseer, 2006. (Cited on page 162).

[159] Salma Messaoudi, Donghwan Shin, Annibale Panichella, Domenico Bianculli, and Lionel C Briand. Log-based slicing for system-level test cases. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 517–528, 2021. (Cited on page 162).

[160] Alexander Kampmann and Andreas Zeller. Carving parameterized unit tests. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ICSE '19, page 248–249. IEEE Press, 2019. doi: 10.1109/ICSE-Companion.2019.00098. URL `https://doi.org/10.1109/ICSE-Companion.2019.00098`. (Cited on page 162).

[161] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Matthew Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering*, 35(1):29–45, 2009. doi: 10.1109/TSE.2008.103. (Cited on page 162).

[162] Deepika Tiwari, Long Zhang, Martin Monperrus, and Benoit Baudry. Production monitoring to improve test suites. *IEEE Transactions on Reliability*, pages 1–17, 2021. doi: 10.1109/TR.2021.3101318. (Cited on page 162).