# Adopting Program Synthesis for Test Amplification

Mehrdad Abdi, Henrique Rocha
Universiteit Antwerpen
België

Serge Demeyer
Universiteit Antwerpen and Flanders Make
België

## Abstract

Program synthesis is the task of enabling a computer system to automatically write program code based on user intent. Test amplification on the other hand is an emerging research area, where the goal is to generate new test cases from manually written ones. From this perspective, test amplification is awfully similar to program synthesis. Therefore, in this short paper, we explore the benefits of using program synthesis for test amplification.

***Index terms***— test amplification, program synthesis, test generation, big code

## 1 Introduction

Software is used in most aspects of modern human life and the importance of reliability in software is undeniable. Software testing is the de facto technique for reducing the risk of defects, thus making software systems more reliable.

Test cases can be generated automatically by tools or can be written manually by developers. Auto-generated test cases provide reasonable test coverage but in most cases are hard to maintain because generators fail to generate human understandable test cases. On the other hand, manually written test cases cover most of the main functionality of the program but may suffer from low coverage. *Test amplification* serves as a kind of middle ground [1]. Test amplifiers exploit the knowledge of the manually written tests in order to enhance existing test cases to increase coverage.



In: D. Di Nucci, C. De Roover (eds.): Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop, Brussels, Belgium, 28-11-2019, published at http://ceur-ws.org

In an earlier paper we demonstrated the feasibility of test amplification for Pharo Smalltalk, a dynamically typed programming language [2]. Nevertheless, we observed some areas for improvement, especially with regard to the readability of the amplified test cases. Therefore, in this paper, we explore the benefits of using *program synthesis* for test amplification.

The remainder of this paper is organized as follows. In section 2, we describe the Small-Amp tool prototype, discussing some of its shortcomings. In section 3 we describe promising ideas from program synthesis as a potential way for addressing these shortcomings. In section 4 we suggest possible benefits of program synthesis for test amplification. Finally in 5 we conclude this paper.

## 2 Test Amplification: Small-Amp

DSpot is an example of a test amplification tool created for Java [3]. We created Small-Amp as a replication of DSpot for the Pharo Smalltalk ecosystem [2] Small-Amp generates new test cases using an evolutionary algorithm, to increase mutation coverage. In this evolutionary algorithm, many versions of the existing test methods are created by making small changes on the test body (Input amplification). Then, each test method is executed and the state of the object under test is captured during the execution of the test body. After that, the collected data obtained from observations is used to insert new observation statements in the test body (Assertion amplification). Finally, the generated test methods are compared to their original one (parent). If the generated method improves the coverage, it will be promoted to the next generation instead of its parent, else the generated test method is ignored.

With our Small-Amp tool prototype, we demonstrated the feasibility of test amplification, even for a dynamically types programming language [2]. Nevertheless, we observed some areas for improvement. We list them below.

- *Type Information* Dynamic languages lack type information in the source code. So, during static analysis, the type information is not directly accessible. We need type information to add new method calls during input amplification. In Small-Amp we used a dynamic profiling step to discover type information of variables in existing test methods. But still, some methods are not covered during the profiling step.
- *State Information* In order to generate valid oracles, Small-Amp observes objects at run-time and generate corresponding assert statements. For capturing the state of the object under test, the tool saves the values returned from getter methods (methods in accessing protocol in the context of Pharo). Unfortunately, there is no clear distinction of which function serves as just a getter and it doesn't update the state of the object. Furthermore, the tool generates the same assert statements every time regardless of the previous update statements.
- *Readbility.* The automatically generated test cases are not easy to understand for humans. This is especially worrisome because incomprehensible test cases will not incorporated by test engineers.

# 3 Program Synthesis

Program Synthesis is the task of automatically creating programs from the underlying programming language that satisfy user intent [4]. This user intent is typically expressed in some form of constraints like input-output examples, demonstrations, natural language, partial programs, and assertions.

In this section, we enumerate a few recent advances in the program synthesis area which may benefit test amplification.

## 3.1 Predicting Program Properties

JSNice uses a probabilistic model that is learned from existing data to predict the properties of new programs[5]. The program properties can be classic semantic properties like type annotation or syntactic program elements like identifiers.

DeepTyper is a similar tool, uses a deep learning model that understands which types are most probable used in certain contexts[6]. It can be applied to code in Javascript and Python. To solve the problem of type inference with machine learning, the authors got inspiration from natural language processing, such as part-of-speech tagging and named entity recognition.

## 3.2 Code Completion

Allamanis at al. used gated graph neural networks [7] to represent a snippet of a program code [8]. This graph includes relations between tokens like *Child*, *NextToken* edges as well as *LastUse*, *LastRead* and *ComputedFrom* to model program code as a graph. Using such a deep learning model, they solved *VarNaming* and *VarMisuse* problems. In *VarMisuse* problem, a variable usage is masked, and the synthesizer is asked to guess which variable is most suitable to use in the hole.

Raychev et al. reduced the problem of code completion: given a program with holes, complete the holes with the most likely sequence of method calls[9]. They incorporated a natural language technique, predicting probabilities of sentences. They construct a statistical language model by extract a large number of histories of API method calls from code snippets obtained from Github and use regularities found in sequences of method invocations to predict and synthesize a similar method invocation sequence. This idea is implemented in a tool called SLANG.

## 3.3 Program Sketching

In sketching, the programmer provides her/his 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 [10].

## 3.4 Learning to write code

DeepCoder is an approach to write programs for solving competition-style problems from input-output examples [11]. In this work, a simple Domain Specific Language (DSL) is defined and DeepCoder solves the problem by finding a program among all possible programs that can be written using this language and can produce desired outputs processing the inputs. For minimizing the search space, they use a machine learning model to predict which statements are most probable to be used in the resulting program.

## 3.5 Big Code

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 [12].

## 4 Program Synthesis For Test Amplification

In this section we suggest some possible direction in using program synthesis advances in test generations.

### 4.1 Amplification in Dynamic Languages

The type of objects in a dynamic language like Javascript, Python and Pharo Smalltalk is not directly accessible. Type information is helpful in the Input amplification phase when the tool adds new method calls to the test body to manipulate the state of the object under test. Recent advances in type inference [6] and predicting program properties [5] are promising to benefit type system information indirectly.

### 4.2 Intelligent Assert Amplification

Automatically generating test oracles, compared to other aspects of software testing, has received less attention [13].

Using methods in [8], [9] and [11] promises to find patterns between updating methods sequence and the asserted method considering the internal elements of the object under test. Using these relations, the amplifier can generate more relevant asserts regarding to the updates on the object under test.

### 4.3 Measuring the Readability of the Tests

But there is a question: what is a readable test case? A possible answer is a test case that is similar to the test that is written by humans is readble.

Benefiting from big code, we can mine existing test case codes from open source project (Similar to [14]) to find some patterns in tests made by humans. Then, we can rate generated test suites to find out how much these tests are readable.

## 5 Conclusions

In this paper, we explore the benefits of using *program synthesis* for test amplification. We discussed the opportunity of using machine learning and big code to improve the readability of amplified tests.

We hope to hear from the community some related ideas including introducing potential challenges, other possible use cases, or any previous experiences on using machine learning in generating code or software testing.

## Acknowledgments

This work is supported by (a) the Fonds de la Recherche Scientifique-FNRS and the Fonds Wetenschappelijk Onderzoek - Vlaanderen (FWO) under EOS Project 30446992 SECO-ASSIST (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

## References

[1] 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.

[2] Mehrdad Abdi, Henrique Rocha, and Serge Demeyer. Test amplification in the pharo smalltalk ecosystem. In *International Workshop on Smalltalk Technologies (IWST)*, 2019.

[3] 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, Apr 2019.

[4] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and TrendsÂ® in Programming Languages*, 4(1-2):1–119, 2017.

[5] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[6] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 152–162. ACM, 2018.

[7] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[8] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[9] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[10] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5-6):475–495, 2013.

[11] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *5th International Conference on Learning Representations (ICLR)*, 2017.

[12] 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.

[13] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[14] 2019.