DeepMutation: A Neural Mutation Tool

Michele Tufano
Microsoft
Redmond, WA, USA

Jason Kimko, Shiya Wang
William & Mary
Williamsburg, VA, USA

Cody Watson
Washington and Lee University
Lexington, VA, USA

Gabriele Bavota
Università della Svizzera italiana (USI)
Lugano, Switzerland

Massimiliano Di Penta
University of Sannio
Benevento, Italy

Denys Poshyvanyk
William & Mary
Williamsburg, VA, USA

ABSTRACT

Mutation testing can be used to assess the fault-detection capabilities of a given test suite. To this aim, two characteristics of mutation testing frameworks are of paramount importance: (i) they should generate mutants that are representative of real faults; and (ii) they should provide a complete tool chain able to automatically generate, inject, and test the mutants. To address the first point, we recently proposed an approach using a Recurrent Neural Network Encoder-Decoder architecture to learn mutants from ~787k faults mined from real programs. The empirical evaluation of this approach confirmed its ability to generate mutants representative of real faults. In this paper, we address the second point, presenting DeepMutation, a tool wrapping our deep learning model into a fully automated tool chain able to generate, inject, and test mutants learned from real faults. **Video:** https://sites.google.com/view/learning-mutation/deepmutation

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Neural networks**.

KEYWORDS

software testing, mutation testing, neural networks

ACM Reference Format:
Michele Tufano, Jason Kimko, Shiya Wang, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, and Denys Poshyvanyk. 2020. DeepMutation: A Neural Mutation Tool. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382146

1 INTRODUCTION

The goal of mutation testing is to inject artificial faults into a program [8, 12] to simulate defects. Mutation testing has multiple applications, including guiding developers to write a test suite (which discovers as many artificial faults as possible), or driving automatic test data generation [11]. Also, mutants can be used to assess the effectiveness of a test suite or the extent to which a testing strategy discovers additional faults with respect to another [4].

A crucial issue in mutation testing is the representativeness of mutation operators — *i.e.,*types of artificial faults to be seeded — with respect to real faults. Empirical research in this area highlights advantages of mutation testing (carefully-selected mutants can be as effective as real faults [2, 7]), but also disadvantages (mutants might underestimate a test suite effectiveness [2]). As a result, literature suggests to carefully devise mutant taxonomies [16], especially to reflect domain-specific characteristics of the software. While past research has proposed mutants for specific domains [9, 13, 14, 18, 19, 26, 27], such a task has been recognized to be effort-intensive.

We have proposed a Neural Machine Translation (NMT) approach to automatically learn mutants from a large corpus of existing bug fixes mined from software repositories [24]. After changes related to bug-fixes have been extracted using an AST-based differencing tool [10], they are grouped into clusters, where each cluster represents a common, recurring set of changes. After that, a Recurrent Neural Network (RNN) Encoder-Decoder architecture [6, 17, 23] first learns a model from the corpora of bug-fixing changes, and then applies it to the unseen code. Key advantages of the proposed approach — differently from previous work by Brown *et al.*, which extracts all possible mutations from fixing changes — are its capability to determine where and how to mutate source code, as well as its capability to introduce new (unseen) literals and identifiers in the mutated code.

This paper describes DeepMutation, the tool that implements the NMT-based mutation approach we have previously proposed [24]. Overall, the major contributions of the paper are:

- The design and release of DeepMutation, the first mutation tool based on an NMT model;
- The tool is available as open source on GitHub [1];
- The summary of the evaluation we performed for the mutation approach behind DeepMutation[24] and a study showing the applicability of the tool on real systems.

2 ARCHITECTURE

Figure 1 provides an overview of the architecture of DeepMutation. Given as input a configuration file and a software system to mutate, DeepMutation begins by invoking an *Extractor*, which builds a meta-model of the project and extracts the methods for which mutants will be generated. Each method is then transformed by an *Abstractor*, which creates an abstracted representation of the method. The abstracted method is fed to the *Mutator* which, using a trained NMT model, generates *n* (user-specified) different abstract mutations. Subsequently, the abstract mutant is translated into

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382146

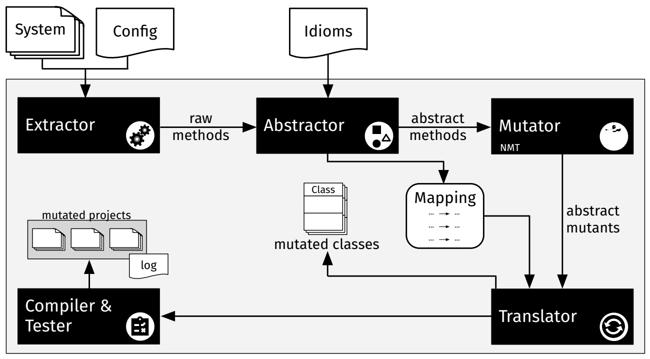

Figure 1: Overview of DeepMutation

source code by the *Translator*. Finally, the *Tester* compiles and tests the generated mutants.

2.1 Extractor

DeepMutation starts by reading the user-specified settings from a configuration file, storing the paths to the software under test, output folders, and other mutation preferences (details on how to use our tool, starting with the setting of the configuration file, are available in our GitHub repository [1]).

The first component to execute is the *Extractor*, which starts by building a meta-model of the software project using Spoon [22]. Spoon is an open-source library capable of building a well-designed Abstract Syntax Tree (AST), on top of which it provides powerful analysis and transformation APIs. In our case, we primarily use this model to query program elements (*i.e.,*specifically methods, but we plan, as future work, to allow DeepMutation to work on different granularities) from the software project to analyze, filter, and extract their source code.

Specifically, the *Extractor* queries all the concrete methods in the software system, disregarding abstract or interface methods. It can optionally discard getters/setters methods. Additionally, the *Extractor* can select only the methods of interest for the user, by matching the signature of the methods with those listed in a user-provided file. The *Extractor* outputs a list of methods M from the system which pass the filtering criteria.

2.2 Abstractor

The main goal of the *Abstractor* is to transform the source code of the methods selected by the *Extractor* into an abstract representation more suitable for our NMT-based *Mutator*. This abstract form is a high-level representation, which reduces the number of tokens used to represent the method (*i.e.,*vocabulary size) yet retains all the method's syntactic and semantic information. It contains the following types of tokens:

Java keywords and separators: the *Abstractor* keeps all the keywords and separators in order to guarantee syntactic correctness.

Idioms: frequent identifiers and literals (*e.g.,*`size`, `index`, `0`, `1`, *etc.*), which we refer to as *idioms*, are retained in the abstract representation, since they provide meaningful semantic information.

Typified IDs: the *Abstractor* replaces any other tokens (not idioms nor keywords/separators) with IDs, which represents the role and type of the identifier/literal replaced.

Specifically, for each method $m \in M$ the *Abstractor* starts from a canonical form of its source code, where any comments or annotations are removed and types are expressed in a fully qualified fashion. Next, a Java Lexer (based on ANTLR [20, 21]) reads the source code tokenizing it into a stream of tokens. The tokenized stream is then fed into a Java Parser [25], which discerns the role of each identifier (*i.e.,*whether it represents a variable, method, or type name) and the type of literals (*e.g.,*string, char, int, *etc.*).

The *Abstractor* generates and substitutes a unique ID for each identifier/literal within the tokenized stream. If an identifier or literal appears multiple times in the stream, it will be replaced with the same ID. IDs are represented using the following format: `<TYPE>_#`, where `<TYPE>` refers to the role of the identifier (*i.e.,*`VAR`, `TYPE`, `METHOD`) or type of literal (*i.e.,*`STRING`, `CHAR`, `INT`, `FLOAT`), and `#` is a numerical ID. These Typified IDs are assigned to identifiers and literals in a sequential and positional fashion. Thus, the first string literal found will be assigned the ID of `STRING_0`, likewise the second string literal will receive the ID of `STRING_1`.

This process continues for all identifiers and literals in the given method m, except for those recognized as idioms, that will be kept in the abstract version m_a with their original token. The list of idioms can be specified in a text file by the user, while we provide a default list of 300 idioms mined from thousands of GitHub repositories.

Here we provide an example of how the *Abstractor* would transform the following method:

```java
public Integer getMinElement(List myList) {
    if(myList.size() >= 1) {
        return ListManager.min(myList);
    }
    return null;
}
```

into the following abstract stream of tokens, where we highlighted java keywords in blue, idioms in green, and typified IDs in red:

```
public TYPE_1 METHOD_1 ( List VAR_1 ) { if (
VAR_1 . size ( ) >= 1 ) { return TYPE_2 . min (
VAR_1 ) ; } return null ; }
```

In conclusion, for each method $m \in M$ the *Abstractor* generates an abstracted version m_a and a mapping file *Map* where the mapping of identifiers/literals with their corresponding IDs is stored.

2.3 Mutator

The *Mutator* takes as input an abstracted method m_a and generates k different mutants $m'_{a1} \ldots m'_{ak}$, where k is a user-specified parameter defining the number of mutants to generate for each method. The *Mutator* relies on a NMT model that was trained on real bug-fixes mined from thousands of GitHub repositories. Given a bug-fix $bf = (m_b, m_f)$ which modifies a buggy method m_b into a fixed method m_f, and their abstracted representation $bfa = (m_{ab}, m_{af})$, the NMT model is trained to learn the translation $m_{af} \rightarrow m_{ab}$, that is, learning to generate a buggy method given a method without a bug. In the next paragraphs, we provide technical details behind the NMT model and the approach used to generate multiple mutants.

2.3.1 NMT Model.

Our model is based on an RNN Encoder-Decoder architecture, commonly adopted in Machine Translation [6, 17, 23]. This model comprises two major components: an RNN Encoder, which *encodes* a sequence of terms $\boldsymbol{x}$ into a vector representation, and a RNN Decoder, which *decodes* the representation into another sequence of terms $\boldsymbol{y}$. The model learns a conditional distribution over a (output) sequence conditioned on another (input) sequence of terms: $P(y_1, .., y_m | x_1, .., x_n)$, where n and m may differ. In our case, given an input sequence $\mathbf{x} = m_{af} = (x_1, .., x_n)$ and a target sequence $\mathbf{y} = m_{ab} = (y_1, .., y_m)$, the model is trained to learn the conditional distribution: $P(m_{ab}|m_{af}) = P(y_1, .., y_m | x_1, .., x_n)$, where x_i and y_j are abstracted source tokens: Java keywords, separators, IDs, and idioms. The Encoder takes as input a sequence $\mathbf{x} = (x_1, .., x_n)$ and produces a sequence of states $\mathbf{h} = (h_1, .., h_n)$. We rely on a bi-directional RNN Encoder [3] which is formed by a backward and forward RNNs, which are able to create representations taking into account both past and future inputs [5]. That is, each state h_i represents the concatenation of the states produced by the two RNNs reading the sequence in a forward and backward fashion: $h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}]$. The RNN Decoder predicts the probability of a target sequence $\mathbf{y} = (y_1, .., y_m)$ given $\mathbf{h}$. Specifically, the probability of each output term y_i is computed based on: (i) the recurrent state s_i in the Decoder; (ii) the previous $i-1$ terms $(y_1, .., y_{i-1})$; and (iii) a context vector c_i. The latter constitutes the attention mechanism. The vector c_i is computed as a weighted average of the states in $\mathbf{h}$, as follows: $c_i = \sum_{t=1}^{n} a_{it} h_t$ where the weights a_{it} allow the model to pay more *attention* to different parts of the input sequence. Specifically, the weight a_{it} defines how much the term x_i should be taken into account when predicting the target term y_t. The entire model is trained end-to-end (Encoder and Decoder jointly) by minimizing the negative log likelihood of the target terms, using stochastic gradient descent.

2.3.2 Generating Multiple Mutants via Beam Search. The main intuition behind Beam Search decoding is that rather than predicting at each time step the token with the best probability, the decoding process keeps track of k hypotheses (with k being the beam size or width). Formally, let $\mathcal{H}_t$ be the set of k hypotheses decoded till time step t: $\mathcal{H}_t = \{(\tilde{y}_1^1, \dots, \tilde{y}_t^1), (\tilde{y}_1^2, \dots, \tilde{y}_t^2), \dots, (\tilde{y}_1^k, \dots, \tilde{y}_t^k)\}$

At the next time step $t+1$, for each hypothesis there will be $|V|$ possible y_{t+1} terms (V being the vocabulary), for a total of $k \cdot |V|$ possible hypotheses: $C_{t+1} = \bigcup_{i=1}^{k} \{(\tilde{y}_1^i, \dots, \tilde{y}_t^i, v_1), \dots, (\tilde{y}_1^i, \dots, \tilde{y}_t^i, v_{|V|})\}$

From these candidate sets, the decoding process keeps the k sequences with the highest probability. The process continues until each hypothesis reaches the special token representing the end of a sequence. We consider these k final sentences as candidate mutants for the given method m. When $k = 1$, Beam Search decoding coincides with the greedy strategy.

2.4 Translator

The *Translator* takes as input an abstract mutant m'_a generated by the *Mutator*, translates it in real source code to generate an actual mutant m', which is then injected in the subject system by replacing the existing method m. The *Translator* relies on the mapping *Map* to replace back all the original identifiers/literals in place of the typified IDs introduced by the *Abstractor* and modified by the *Mutator*. While the *Mutator* has been trained to re-use only IDs available in the input method or idioms in the vocabulary, it is still possible that the *Mutator* might introduce an ID which is not present in the original mapping *Map* (*i.e.,*`<TYPE>_#` $\notin$ *Map*).

These cases are detected by the *Translator*, which will perform a series of best-effort attempts to generate real source code for the mutant. In particular, if the missing IDs refer to literals (*i.e.,*`STRING`, `CHAR`, `INT`, `FLOAT`), it will generate a new literal (of the specified type) making sure that is not already defined in the original method m (since it would have been referred already with an ID). Conversely, if the missing IDs refer to variables or method calls, the *Translator* will discard the current mutant and select the next one.

Once all the IDs in m'_a are replaced, we obtain the concrete code of the mutant m'. Next, the code is automatically formatted and indented. Finally, DeepMutation replaces the original method m with the newly generated and translated mutant m'.

2.5 Compiler & Tester

DeepMutation allows the user to specify custom commands for compilation and testing of the mutants in the configuration file. This enables DeepMutation to compile and test every mutant generated and output a log reporting the results of these steps.

3 EVALUATION

The *goal* of this evaluation is to assess the performance of the end-to-end mutation infrastructure offered by DeepMutation. A more comprehensive evaluation of the quality of the mutants generated by the NMT model can be found in [24].

We execute the entire pipeline of DeepMutation on four projects from the Defects4j [15] dataset. Specifically, we select the first (fixed) revision of the following projects: Chart, Lang, Math, Time.

3.1 Settings

As DeepMutation core NMT model we rely on a pre-trained model from our previous work [24]. This model was trained on a large, diverse set of bug-fixes mined from thousands of GitHub repositories. For this evaluation, we do not fine-tune nor re-train the NMT model on the Defects4J codebase. This allows us to demonstrate the capabilities of the model on a novel, previously unseen codebase and provide insights on the transfer learning opportunities.

We set the beam size equal to one, thus allowing the model to generate only a single mutant for each method extracted from the project. Note that DeepMutation supports the generation of many different mutants for each method.

The maximum method size is set to 50 tokens for the purpose of this evaluation. Thus, we disregard methods larger than 50 tokens after the extraction process and before the abstraction. DeepMutation supports the mutation of methods larger than 50 tokens.

3.2 Mutation Performances

Table 1 shows the results of our preliminary evaluation in terms of the number of methods extracted and mutated, throughout the different phases of DeepMutation. The *Extractor* identified between 3k-7k methods in each project. After this stage, we automatically discard getters/setters as well as methods longer than a predefined threshold. Between 1.1k-2.8k methods are abstracted and mutated for each project. Next, the *Translator* was able to correctly translate abstract mutants generated by the *Mutator* in concrete source

Table 1: Preliminary Evaluation Results

Project	Extractor	Abstractor/Mutator	Translator	Compiler
Chart.1	3,383	1,134	1,103 (97.27%)	741 (67.18%)
Lang.1	4,350	1,607	1,322 (82.27%)	585 (44.25%)
Math.1	8,988	2,810	2,416 (85.98%)	1,397 (57.82%)
Time.1	7,047	2,028	1,941 (95.71%)	687 (35.39%)

Table 2: Timing Performances

Project	Thread Load	Compiler Time	Total Mutation Time
Chart.1	34/35	4min	9min
Lang.1	41/42	5min	12min
Math.1	75/76	15min	27min
Time.1	60/61	16min	24min

code in 82%-97% of the cases. Finally, DeepMutation creates a mutated version of the project for each mutant, by injecting the mutant in a copy of the project and attempting to compile the entire mutated project. DeepMutationwas able to successfully compile 35%-67% of the mutated projects. We claim this is a satisfactory level of compilability, although we aim to improve it, in future work, by dynamically replacing non-compilable mutants with the next probable mutant obtained via beam search.

3.3 Timing Performances

Table 2 provides details on the timing performance of DeepMutation. We report the total mutation time for all the mutants for a given project, the time spent specifically by the compiler, and the thread load during these experiments. The results show that DeepMutation can mutate thousands of methods (see Table 1) in a reasonable amount of time. Specifically, if we exclude the time spent compiling the projects, DeepMutation takes between 5-12min to extract, abstract, mutate, and translate the mutants.

4 CONCLUSIONS

We presented DeepMutation, a mutation infrastructure featuring an NMT model trained to generate mutants that resemble real bugs [24]. We described the infrastructure, able to extract, abstract, and mutate methods from a given project, as well as automatically compile and test the generated mutants. We provided an evaluation of the infrastructure by generating mutants for four projects.

5 ACKNOWLEDGMENT

This work is supported in part by the NSF CCF-1927679, CCF-1815186, CNS-1815336 grants. Bavota acknowledges the financial support of the Swiss National Science Foundation for the project CCQR (SNF Project No. 175513).

REFERENCES

[1] 2019. DeepMutation - GitHub. https://github.com/micheletufano/DeepMutation

[2] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Software Eng.* 32, 8 (2006), 608–624.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2014). arXiv:1409.0473 http://arxiv.org/abs/1409.0473

[4] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche. 2004. Assessing and Improving State-Based Class Testing: A Series of Experiments. *IEEE Trans. Software Eng.* 30, 11 (2004), 770–793.

[5] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. 2017. Massive Exploration of Neural Machine Translation Architectures. *CoRR* abs/1703.03906 (2017). arXiv:1703.03906 http://arxiv.org/abs/1703.03906

[6] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR* abs/1406.1078 (2014). arXiv:1406.1078 http://arxiv.org/abs/1406.1078

[7] Muriel Daran and Pascale Thévenod-Fosse. 1996. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis, ISSTA 1996, San Diego, CA, USA, January 8-10, 1996.* 158–171.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (April 1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[9] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. 2015. Towards mutation analysis of Android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 1–10. https://doi.org/10.1109/ICSTW.2015.7107450

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. [n. d.]. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014.*

[11] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291.

[12] R. G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *TSE* 3, 4 (July 1977), 279–290. https://doi.org/10.1109/TSE.1977.231145

[13] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. DeepMutation++: a Mutation Testing Framework for Deep Learning Systems. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE '19).* ACM/IEEE.

[14] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 208–219.

[15] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014).* ACM, New York, NY, USA, 437–440.

[16] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).*

[17] Nal Kalchbrenner and Phil Blunsom. 2013. Recurrent Continuous Translation Models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, Seattle, Washington, USA, 1700–1709. http://www.aclweb.org/anthology/D13-1176

[18] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. 2019. MuSC: A Tool for Mutation Testing of Ethereum Smart Contract. In *Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering (ASE '19).* ACM/IEEE.

[19] A. Jefferson Offutt and Ronald H. Untch. 2001. Mutation Testing for the New Century. Chapter Mutation 2000: Uniting the Orthogonal.

[20] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[21] Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11).* ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

[22] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Softw. Pract. Exper.* 46, 9 (Sept. 2016), 1155–1179. https://doi.org/10.1002/spe.2346

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *CoRR* abs/1409.3215 (2014). arXiv:1409.3215 http://arxiv.org/abs/1409.3215

[24] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *Proceedings of the 35th IEEE International Conference on Software Maintenanance and Evolution (ICSME '19).* IEEE.

[25] Danny van Bruggen. 2014. JavaParser. https://javaparser.org/about.html

[26] Mario Linares Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 233–244.

[27] Kaiyuan Wang. 2015. *MuAlloy: an automated mutation system for alloy.* Ph.D. Dissertation.