FixJS: A Dataset of Bug-fixing JavaScript Commits

Viktor Csuvik
csuvikv@inf.u-szeged.hu
Department of Software Engineering
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary

László Vidács
lac@inf.u-szeged.hu
Department of Software Engineering
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary

ABSTRACT

The field of Automated Program Repair (APR) has received increasing attention in recent years both from the academic world and from leading IT companies. Its main goal is to repair software bugs automatically, thus reducing the cost of development and maintenance significantly. Recent works use state-of-the-art deep learning models to predict correct patches, for these teaching on a large amount of data is inevitable almost in every scenarios. Despite this, readily accessible data on the field is very scarce. To contribute to related research, we present *FixJS*, a dataset containing bug-fixing information of ~2 million commits. The commits were gathered from GitHub and processed locally to have both the buggy (before bug fixing commit) and fixed (after fix) version of the same program. We focused on JavaScript functions, as it is one of the most popular programming language globally and functions are first class objects there. The data includes more than 300,000 samples of such functions, including commit information, before/after states and 3 source code representations.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; *Software testing and debugging*; *Maintaining software.*

KEYWORDS

Automated Program Repair, Software engineering, Bug-fixing commits

ACM Reference Format:

Viktor Csuvik and László Vidács. 2022. FixJS: A Dataset of Bug-fixing JavaScript Commits. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528480

1 INTRODUCTION

Data-driven repair approaches [21] recently reached promising results [2, 4, 14, 20, 27]. Such tools usually require a large dataset and split it in train-test-validate subsets to be able to train and evaluate. The most commonly used datasets for APR research (Defects4J [15], QuixBugs [26], ManyBugs [17]) are not constructed for such kind of approaches: they include too few programs (in terms of deep learning) and the data is not preprocessed. We hypothesize that this may be the reason why data-driven approaches prefer to use their own datasets, which is often not publicly available. Unfortunately, thus the evaluation and comparison of such repair tools is almost impossible.

The paper of Tufano *et al.* [24] introduced a large dataset from mined GitHub commits and trained a Neural-Machine Translation model on it. Their seminal work has been encased in the CodeXGLUE benchmark [19], which includes diverse programming language tasks covering code-code, text-code, code-text and text-text scenarios. The dataset is highly successful among researchers, although it have not become a de-facto evaluation platforms as Defects4J has become. As the existing public benchmarks are designed for the Java programming language, we ought to gain ground for JavaScript (JS) in the APR field as well.

For the eighth year in a row, JavaScript is the most commonly used programming language [23]. It is the de-facto web programming language globally and the most adopted language on GitHub [8]. JavaScript is massively used in the client-side of web applications to achieve high responsiveness and user friendliness. In recent years, due to its flexibility and effectiveness, it has been increasingly adopted also for server-side development, leading to full-stack web applications [13].

Lacking sufficient commit information, the evaluation of the proposed APR methods is always difficult. Our aim was to ease this burden by providing a dataset that can contribute to the efforts of the community. This paper introduces the *FixJS* dataset, describes its properties and structure. The proposed dataset is available on GitHub [1] and have a DOI to make it easily citable [2]. It contains roughly two million bug-fixing commits from GitHub. From these commits, the modified functions were extracted (the state before and after the bug fix happened). These functions are then tokenized and abstracted, resulting in three different source code representations. After that, the samples are divided in three categories based on the number of tokens presented in them. The original commits are also preserved, along with the files that are included in the commits. The resulted dataset contains overall ~300k samples. Compared to data used in related research, this dataset is a big step forward in detailedness, in volume and in supplying the context to the fixes. FixJS can be used to train and evaluate a deep learning model that predicts correct fixes without any further processing steps.

[1] https://github.com/RGAI-USZ/FixJS
[2] 10.5281/zenodo.6340207

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528480

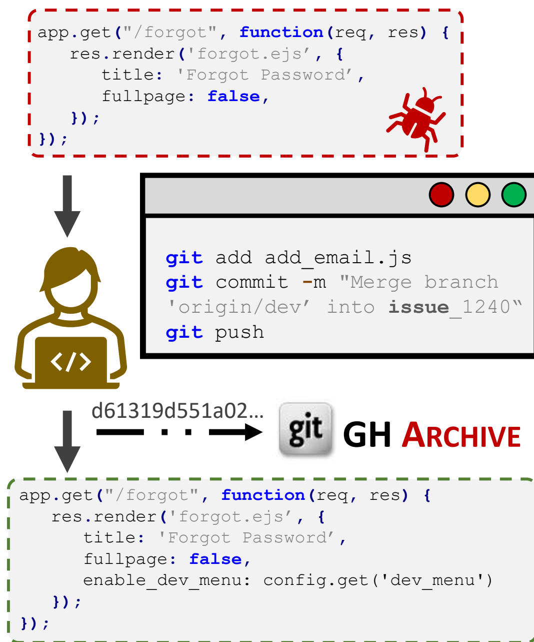

Figure 1: Relevant aspects of software development.

2 BACKGROUND

In Figure 1 we highlighted the relevant aspects of the bug-fixing process. As a bug is discovered, the developer creates a patch for it and writes a description (i.e. commit message) about the applied changes. As can be seen on the figure, this process is recorded by GH Archive [7]. There are other source code hosts as well, but GitHub is the largest, having more than 28 million public repositories [10]. This vast amount of source code available online is a good starting point for data mining.

We designed our process with great care to identify bug-fixing commits on JavaScript files and fetch their before-and after states. The approach of this paper technically consists of two main parts: (1) bug-fix mining and (2) abstraction. On Figure 2 we illustrated the high-level steps which were carried out during the work. The thick red boxes represent the main phases, the black (right-directed) arrows indicate data flows, the dashed (upward pointing) arrows denote dependencies on external tools, while the blue (faced down) arrows show the by-product of each step. In this section we describe each phase in details, starting from the bug-fix mining process.

2.1 Bug-fix mining

We used two external tools in this phase: GH Archive [7] to retrieve commits from a specific time range and GitHub REST API [9] to get detailed information about a commit. To start off, we fetched every *push event* from GH Archive from ranging between 01.01.2012 and 30.12.2012. The reason we have chosen this time interval is quite simple: GH Archive stores required information from 2012, and because of time and resource constraints we could only fetch one year of data. Since GH Archive stores the commit hash and the commit message as well, we could filter on bug-fixing commits in this step. All commit messages containing one of the following keywords are identified as a bug-fix: `["fix", "solve", "bug", "issue", "problem", "error"]`. The same patterns are used in the work of Tufano *et al.* [24] and a similar approach in [6]. We were able to retrieve 2,129,715 bug-fixing commits. These commits are saved in a csv file contating the date, event type, commit hash (sha), message and url in a monthly breakdown.

Next, files are being fetched that are affected by the commit. Using GitHub API [9], we were able to filter out non-JavaScript files (files with not ".js" extension) and download the before- (i.e. buggy program) and after (i.e. fixed program) state of it. During the process some of the commits were ignored because their repository were renamed or deleted. At the end of this phase we identified 103,115 commits containing 201,198 files overall. These commits are saved in a folder named by their sha, containing three files: *before.js* (the JS file before the bug-fix) *after.js* (the JS file after the fix) and *diff* (the git diff of these files).

2.2 Abstraction

The goal of this phase is to (1) identify the modified functions and (2) create a representation of it that can be fed into an AI model. Functions are extracted without their names. Function expressions and arrow functions do not have names by definition in JavaScript, while function statement and member functions do have (see Listing 1). The overwhelming majority of bug-fixes take place at the body of a function, thus ignoring names does not result with significant data loss.

```
1  function foo() { }        // function statement
2  var foo = function() { } // function expression
3  var foo = () => { }       // arrow function
4  var o = { f: function() { } } // member function
```

Listing 1: Function declaration differences in JavaScript

We created three different representation of each raw function. To retrieve the Abstract Syntax Tree (AST) of the observed function, we relied on the Esprima library [5]. During this phase syntactical errors are filtered out. Part of the following representations are adapted from [24], while others are introduced here. We depict a small example on the right part Figure 2. Note that, in an AI-model point of view the main difference is in vocabulary size. While in the tokenized function every kind of identifier and literal can occur (resulting in a vocabulary of arbitrary size), in the other two representations the vocabulary size is fixed. We created three datasets with different token lengths: *small* (#tokens <50), *medium* (50 <= #tokens <100) and *large* (100 <= #tokens).

2.2.1 Tokenization. In this representation the function is split into tokens without any further modifications. Each token is separated by a space in the dataset. The vocabulary size is arbitrary.

2.2.2 Full-mapping. We call mapping the process in which identifiers and literals are mapped to generic IDs. Every ID follows the pattern `TYPE_INDEX`, where TYPE is the corresponding token type, while index ensures that each ID is unique in a before/after function pair. The indexing is sequential: when the parser finds e.g. an identifier, it will assign the ID `METHOD_0` to it, the second method will have the ID `METHOD_1`, and so on. The used types are the following: `[STRING, NUMERIC, BOOLEAN, REGULAREXPRESSION]` for literals and `[VAR, METHOD]` for identifiers. Vocabulary size <130 + I (where

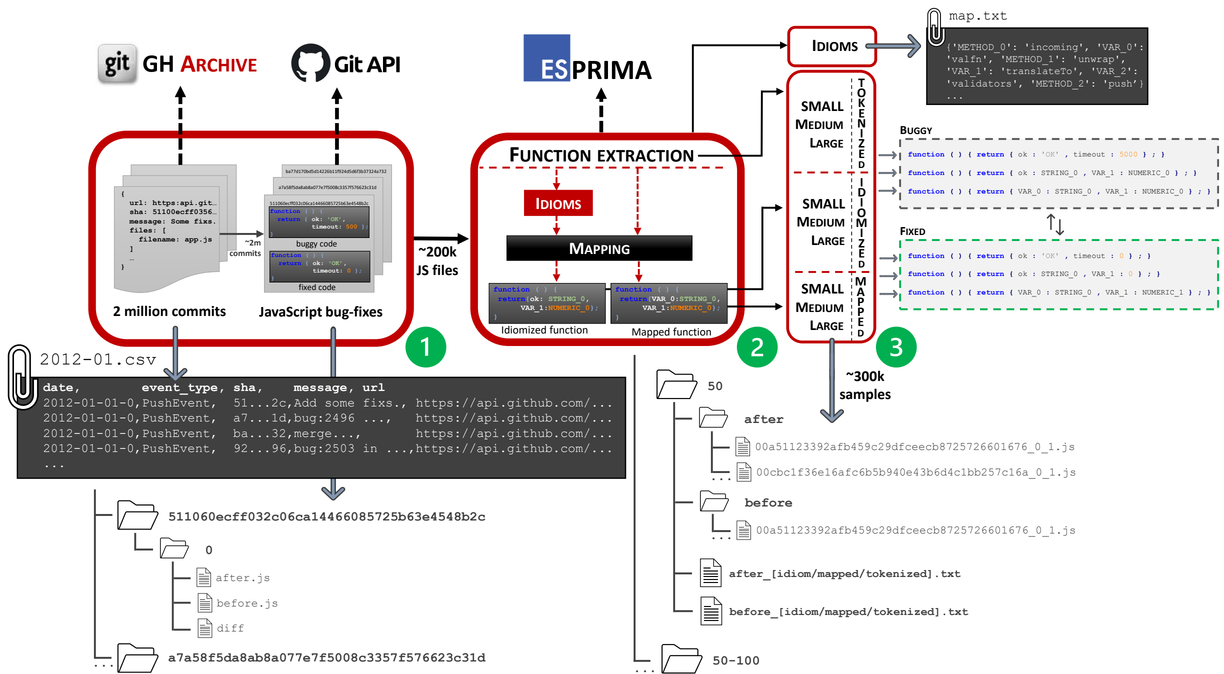

Figure 2: A high level overview of the dataset creation approach

I is the sum of the largest index in each of the mapped keywords, 130 = 6 defined types + 63 JS keywords + ~60 special characters).

2.2.3 Idiomization. Idiomization is the generalization of the full-mapping representation. Frequent identifiers and literals are often referred to as *idioms* [1]. In some cases they appear so often in the code that, they can almost be treated as keywords of the language (e.g. `i, j, 0, -1`). To retrieve these common keywords we counted the frequency of every token present in the fetched commits. From this then we picked the TOP-N idioms (N is arbitrary), let us call it the idiom-set. When parsing, and a token occurs, the parser first examines whether it is present in the idiom-set. If yes, the token value is being used, otherwise the same mapping process is executed as in full-mapping. Vocabulary size <N + I + 130 (where N is the number of idioms, other same as before).

3 DATASET

3.1 Structure

FixJS distinguishes three source code representations in three different sized setting. We separated the functions based on token numbers and organized the files in different directories. Each of the folders contain 7 text files: a mapping file, 3 before and 3 after files. In the latter mentioned files each line corresponds to a function of a specific representation. For example the 6th line in `after_idiom.txt` is the 6th bug-fix in 2012 that affects a JavaScript file and is preprocessed as described in Section 2.2.3, it's corresponding buggy function can be found in `before_idiom.txt` in the 6th line. The same applies to the tokenized and mapped representations as well, the before/after state of the functions can be connected using their index in the files.

The `map.txt` is the mapping files which contains the real world identifiers that are replaced in the tokenized and idiomized representations. Each line contains a dictionary where the keys are the IDs from the parsed function, while the values are the real world namings. So if one would like to map *e.g.* the 4th line in `after_mapped.txt` to get back the real function, the only thing needs to be done is to map the IDs from the 4th line in `map.txt` to the ones present in the sample. We extracted 3 before and after functions from the idiomized representation into Listing 2.

```
//before_idiom.txt:
function ( ) { \$ ( STRING_0 , VAR_0 ) . removeClass ( STRING_1 ) ; }
function ( ) { VAR_0 . METHOD_0 ( STRING_0 ) ; }
function ( ) { METHOD_0 ( - 1 ) }
//after_idiom.txt:
function ( ) { \$ ( STRING_0 ) . removeClass ( STRING_1 ) ; }
function ( ) { METHOD_1 ( VAR_1 ) ; VAR_0 . METHOD_0 ( STRING_0 ) ; }
function ( ) { METHOD_0 ( - 1 ) ; self . METHOD_1 ( VAR_1 ) ; }
```

Listing 2: An extract from the idiomized representation.

3.2 Data

The dataset contains information on every publicly available bug-fixing commit that affected JavaScript files in the first half of 2012. We started from scratch, and created 300k samples in three settings including different source code representations. In Table 1 we summarize the assembled datasets. Here we can see that the *Large* subset contains the majority of the samples, this and the sheer size of the functions implies that its size in megabytes is also the greatest.

FixJS contains both single- and multi-line bugs. Before- and after state of the mined functions are differentiated using their Abstract Syntax Tree, meaning that if only comments have changed the samples are filtered out.

The dataset is available on GitHub, containing a detailed README of the featured files. To use the resulting dataset one should carry out the following steps:

(1) Clone the repository and pick a dataset size (50, 100 or 100+)
(2) Load the before_*rep*.txt and after_*rep*.txt (where *rep* can be [idiom, mapped, tokenized])
(3) Split the dataset (*e.g.* 80-10-10) and train the model
(4) Evaluate the model on the test set

Table 1: Summary of the constructed datasets.

	# Tokens	# Samples	Size (mb)
Small	#tokens <50	67,070	78
Medium	50 <= #tokens <100	70,816	180
Large	100 <= #tokens	186,021	5,350
Overall		323,907	5,608

The token number (#tokens) determined using the Esprima [5] standard parser. Note that in #tokens each literal counts as one token. This can be confusing especially for string literals if they are not mapped (since they typically consist of multiple syllables).

3.3 Possible Uses

The possible uses of FixJS is quite generic and similar to existing datasets like Defects4J or BugsJS. However, these databases are small in size to teach a deep learning model, and their preparation requires a serious development effort. On the other hand, the bug-fixes in FixJS are already extracted and organized in quite large quantities. Real world bug-fixing commit information facilitates automatical software refactoring and may improve software evolution. It can enable seamless integration between continuous code changes and serve as a ground to better understand the software development cycle. The proposed dataset serves these goals, it can provide a common ground in evaluating data-driven repair solutions, potentially contributing to a better understanding of the strengths and weaknesses of different source code extraction methods and lead to their best combination. The dataset is mainly intended for Automated Program Repair research evaluation purposes. It provides more detailed data than the currently available alternatives and can also be used in different representation evaluations.

3.4 Limitations

FixJS features 3 dataset sizes, but *Large* can only be used with advanced models where the sequence length is not restricted. However, Defects4J or BugsJS is excellent for test-based patch generation approaches FixJS cannot be used for this cause, since it lacks testing features. During the mining process we did not filter on GitHub repositories (*e.g.* number of stars, forks, etc.), thus low-quality projects are also included. We know that not all the changes in a bug-fixing commit are related to bug-fixing and tangled changes might also occur [12], in future releases we plan to address these limitations by manual verification and careful data collection.

4 RELATED WORK

We based our study on the seminal work of Tufano *et al.* [24], where they created a dataset for Java program repair and evaluated a NMT (Neural Machine Translation) model on it, although we worked with JavaScript. Their small dataset contains 58,350 samples, while their medium dataset 65,465 samples. Our small dataset consists of 67,070, while the medium 70,816 buggy-fixed function pairs, thus the size of the two datasets are of similar magnitude. Additionally we created a large dataset that contains functions with more than 100 tokens, in this we included 186,021 samples. The dataset presented above is included in the CodeXGLUE benchmark [19] which is a collection of code-related tasks and a platform for model evaluation.

Other than [24], several datasets have been introduced for the APR field. Defects4J [15] is a popular dataset consisting 395 Java bugs. The ManyBugs [17] dataset contains bugs in C, it were used to evaluate many well-known APR tools (Genprog [25], Prophet [18], etc.). Bugs.jar [22] is comprised of 1,158 Java bugs and their patches, while BugsJS [11] contains reproducible JavaScript bugs from 10 open-source Github projects. These datasets contains both single- and multi-line bugs. All of these datasets can be used for a wide range of tasks, though their relatively small size makes them inappropriate for deep learning.

A few datasets of larger-scale is also available publicly. CodRep [3] aims at being a common playground on which the machine learning and the software engineering research communities can interact. It contains 58,069 one-liner commits. A more recent work of Karampatsis *et al.* [16] introduce a dataset of similar size consisting of 153,652 single-statement bugs mined from open-source Java projects. The authors of [16] focused on estimating the frequency of SStuB templates rather than creating a general-purpose dataset. In contrast FixJS is generic and contains information of JavaScript bugs. Lastly, unlike previous datasets, we provide three representations of the mined bug-fixes and additional commit information.

5 CONCLUSIONS AND FUTURE WORK

APR tools create software patches without human intervention. State-of-the-art approaches are typically evaluated on their own datasets which are often not publicly available, which hampers the comparative evaluation of novel methods. In this paper, we described the FixJS dataset, which includes ~300k samples containing separately the buggy and fixed codes. During the process we examined ~2 million commits and mined ~200k JavaScript files. From this massive amount of data we created three datasets of different size: small, medium and large. In each of these datasets three source code representations with different abstraction levels are present.

We plan to extend the dataset in future work. It is of similar size like in [24], but note it only contains commit info for a limited time interval. Although the two datasets operate on different programming languages, tools evaluated on them are might be comparable. Constructing the database is a very time consuming task, but we still plan to include all commits to date.

ACKNOWLEDGMENTS

The research presented in this paper was supported in part by the ÚNKP-21-3-SZTE and ÚNKP-21-5-SZTE New National Excellence Programs and by the Artificial Intelligence National Laboratory Programme of the Ministry of Innovation and the National Research, Development and Innovation Office, financed under the TKP2021-NVA funding scheme. László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences.

REFERENCES

[1] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. 2013. Automatic recovery from runtime failures. *Proceedings - International Conference on Software Engineering* (2013), 782–791. https://doi.org/10.1109/ICSE.2013.6606624

[2] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 01 (sep 2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179 arXiv:1901.01808

[3] Zimin Chen and Martin Monperrus. 2018. The CodRep Machine Learning on Source Code Competition. (2018). arXiv:1807.03200

[4] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. 2020. *Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs*. Technical Report. 1–17 pages.

[5] Esprima 2020. Esprima Official Website. https://esprima.org.

[6] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. *IEEE International Conference on Software Maintenance, ICSM* (2003), 23–32. https://doi.org/10.1109/ICSM.2003.1235403

[7] GHArchive 2021. GH Archive Official Website. https://www.gharchive.org.

[8] GitHub 2021. The 2020 State of the Octoverse. https://octoverse.github.com.

[9] GitHub REST API 2021. GitHub REST API Official Website. https://docs.github.com/en/rest.

[10] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub Data on Demand. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. ACM, New York, NY, USA, 384–387. https://doi.org/10.1145/2597073.2597126

[11] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: A benchmark of javascript bugs. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*. 90–101. https://doi.org/10.1109/ICST.2019.00019

[12] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristof Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matus Sulir, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2020. A Fine-grained Data Set and Analysis of Tangling in Bug Fixing Commits. (nov 2020). arXiv:2011.06244

[13] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/2338965.2336758

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. (may 2021), 1161–1173. arXiv:2103.00073

[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 *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*. Association for Computing Machinery, Inc, 437–440.

[16] Rafael Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020* (may 2020), 573–577. https://doi.org/10.1145/3379597.3387491 arXiv:1905.13334

[17] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (dec 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[18] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016* (2016), 298–312. https://doi.org/10.1145/2837614.2837617

[19] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined* (2021). arXiv:2102.04664

[20] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* 20 (2020), 101–114.

[21] Martin Monperrus. 2020. *The Living Review on Automated Program Repair*. Technical Report.

[22] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. *Proceedings - International Conference on Software Engineering* (2018), 10–13. https://doi.org/10.1145/3196398.3196473

[23] Stack Overflow 2021. Stack Overflow Developer Survey Results 2021. https://insights.stackoverflow.com/survey/2021.

[24] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[25] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[26] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. *IBF 2019 - 2019 IEEE 1st International Workshop on Intelligent Bug Fixing* (may 2019), 1–10. https://doi.org/10.1109/IBF.2019.8665475 arXiv:1805.03454

[27] Li Yi, Shaohua Wang, and Tien N. Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering*. IEEE Computer Society, 602–614. https://doi.org/10.1145/3377811.3380345