

REVIZOR: A Data-Driven Approach to Automate Frequent Code Changes Based on Graph Matching

Oleg Smirnov,^{*†} Artyom Lobanov,^{*§} Yaroslav Golubev,^{*} Elena Tikhomirova,^{*} Timofey Bryksin^{*†§}

^{*}JetBrains Research, [†]Saint Petersburg State University, [§]Higher School of Economics

{oleg.smirnov, artem.lobanov, yaroslav.golubev, elena.tikhomirova, timofey.bryksin}@jetbrains.com

Abstract—Many code changes that developers make in their projects are repeated and constitute recurrent change patterns. It is of interest to collect such patterns from the version history of open-source repositories and suggest the most useful of them as quick fixes. In this paper, we present REVIZOR—a tool aimed to build custom plugins for PyCharm, a popular Python IDE. A REVIZOR-based plugin can take change patterns and highlight potential places for their application in the developer’s code editor. If the developer accepts the quick fix, the plugin automatically performs the edit. Our approach uses a graph-based representation of code changes, which allows it to support complex distributed code patterns. Experienced developers have also rated the usability and the performance of such REVIZOR-based plugin positively.

The source code of the tool and test plugin prototype are available on GitHub: <https://github.com/JetBrains-Research/revizor>. A demonstration video with a short tool description can be found on YouTube: <https://youtu.be/5eLs14nco7E>.

I. INTRODUCTION

In recent decades, Integrated Development Environments (IDEs) have been evolving to provide tools that help developers write and edit code ever more efficiently [1]. Writing code involves making a lot of incremental changes and small fixes. Some of such scrupulous high-concentration tasks can be automated to boost developers’ performance. To this end, IDEs incorporate *static analysis tools* that find and highlight known problems in code, as well as suggest certain *quick fixes* as possible solutions to these problems in real time. A lot of works focus on automated program repair (APR) [2], which includes localizing bugs and vulnerabilities in code and fixing them. We aim to show that approaches which are similar to APR can be applied not only to bug fixes, but to almost any code change that is of interest to developers—from stylistic code enhancement to the migration of APIs to a different language version.

A good place to search for possible improvements of code is the history of it being changed. Prior research suggests that code changes are repetitive: from time to time different developers not only face the same kind of problems, but try to solve them in a similar way [3], [4]. Such code changes constitute recurrent change patterns that can be mined from the histories of existing software projects.

Semantic change patterns often differ in structure significantly. While some of them are one-liners or even touch only a single statement [5], others can be quite complex and *distributed*, *i.e.*, involving isolated tokens from different lines of code or even different scopes connected by data or control

dependencies (as in Figure 3). From that point of view, code analysis approaches can benefit greatly from utilizing the fact that code is more than just plain text and work with graph-based code representations that make it possible to keep track and make use of the code’s structure and make decisions about its semantics [4], [6].

In this paper, we investigate whether it might be possible to use graph-based representations of recurrent change patterns for code improvement suggestions within an IDE. For that purpose, we have developed REVIZOR—a tool that allows to build code enhancement plugins for PyCharm [7], a popular IDE for Python developed by JetBrains. We also propose our prototype plugin built with nine pre-approved code change patterns. For any given change pattern, the plugin uses a fine-grained program dependence graph (*fgPDG* introduced by Nguyen et al. [4] for mining patterns in Java) of the code fragment before the change to localize its occurrences in the user’s code via building subgraph isomorphisms. If such an occurrence is found, the plugin highlights the code fragment for the user, indicating that a quick fix can be applied. If the user chooses to apply the fix, the plugin can do it by performing a sequence of edit actions, which are generated for each pattern from the versions of code before and after the change.

To evaluate the proposed plugin, we conducted a survey of nine experienced developers. The participants positively rated the plugin’s usability and performance and mostly approved the idea of using recurrent code changes to suggest relevant quick fixes in the IDE.

II. BACKGROUND

In this section, we describe the approaches that we adopted to build our pipeline. In Sections II-A and II-B, we characterize the solutions that we considered for the plugin to localize patterns in source code and to apply the respective changes (see the two final stages in Figure 1), and we also contrast the adopted solutions with similar ones. In Section II-C, we relate a prospective graph-based approach to code change patterns mining. Integrating it into a code enhancement pipeline is unprecedented and seems promising.

A. Code Pattern Localization

There exists a number of relevant approaches that aim to mine fix patterns and use them to localize potential code flaws.

Meng et al. presented an approach called LASE [8], where the authors initially built a *context-aware edit script* from two or more code change examples and then identified appropriate locations for code transformation with a *generalized tree-based edit context*. To localize such a context in the AST of target code, the authors used the Maximum Common Embedded Subtree Extraction algorithm. A similar approach to localization with a tree-matching algorithm was also used by Bader et al. in their tool called GETAFIX [9], in which bug fix patterns mined in Java are automated. However, these approaches cannot be used to localize *distributed* code patterns, which may include many subtrees of an AST.

The authors of DEVREPLAY [10] collected code change patterns via AST comparison, converted the source code of the patterns into regular expressions, and then tried to match any of them with the user’s code to localize possible problems in it. Such method is able to handle multi-line patterns, but still cannot automatically manage control or data flow dependencies between elements of the pattern as graph-based approaches do.

B. Edit Template Application

After a pattern is localized in code and the developer confirms applying the fix, the exposed code fragment is changed in accordance with the appropriate edit script. This is usually done via AST transformations similarly to how it was described by Meng et al. [8], [11]. For that purpose, the authors used edit actions of four types—*insert*, *delete*, *update* and *move*—generated by a modified version of CHANGEDISTILLER [12], a source code differencing tool for extracting fine-grained edit scripts from two versions of an AST: before and after the change.

Bader et al. [9] used a similar tool called GUMTREE, and, according to Falleri et al. [13], GUMTREE represents edit actions in a more accurate and concise way compared to other source code differencing tools.

Nowadays, researchers also widely exploit the ideas of Neural Machine Translation (NMT) that view the task of applying changes as a translation problem from a defective code fragment into a correct one [14], [15]. These approaches are hardly interpretable and require collecting a large number of similar patches to train the model, which is a difficult task. On the contrary, heuristic approaches like the ones that employ AST edit actions need far less input and computational resources to perform.

C. Collection of Recurrent Code Changes Using Graphs

To keep track of semantic features, as well as data and control dependencies between the elements of source code, more complex data structures such as graphs can be used. Nguyen et al. proposed an approach called CPATMINER [4] for mining graph-based change patterns in Java code. The representation of code that they used is called *Fine-Grained Program Dependence Graph (fgPDG)* and is based on the AST of the source code. Such a graph includes three types of nodes: data nodes (for variables, literals, etc.), operation nodes (for

arithmetic expressions, assignments, function calls, etc.), and control nodes (for control statements like *if*, *for*, *while*, etc.). These nodes are linked with additional data and control dependency edges.

To represent code changes, Nguyen et al. introduced the concept of a *change graph*. A change graph is built using two fgPDGs of the code before and after a given change; corresponding unchanged graph nodes are connected with mapping edges. The authors also suggested a way to use this data structure to build a pattern-mining algorithm. The main idea behind it is to recursively extend each already mined change graph to the most frequently encountered adjacent vertex and then match isomorphic graphs using a hash-based heuristic [16] to put them into one particular pattern.

In our prior work [17], we re-implemented this approach for Python, collected and analyzed fgPDG-based code change patterns from 120 popular GitHub repositories. This allowed us to collect recurrent in-the-wild code changes: code enhancements, bug fixes, refactorings, etc. In this work, our goal was to implement quick-fixing actions so that these graph-based changes could be applied automatically in the developer’s code in the IDE.

III. IMPLEMENTATION

A. Pipeline Overview

We have implemented an approach to enhancing IDEs with valuable up-to-date code improvement suggestions in a data-driven way. Our contribution is REVIZOR, a tool that allows to create custom plugins with pre-approved quick fixes for Python code. We also built a prototype plugin for PyCharm, a popular Python IDE based on the IntelliJ Platform.¹ The plugin is created using code change patterns and respective code samples mined with PythonChangeMiner [17], which analyzes the graph-based representation of code changes in Git repositories and detects change patterns without any prior specification of what is worth changing in Python code, thus sparing the necessity to devise and manually write code enhancement rules. The proposed approach allows plugins to locate *distributed* code patterns, *i.e.*, the patterns involving isolated tokens that are located on multiple lines of code and connected by some data or control dependencies. Using fgPDGs enables greater matching flexibility and structural awareness compared to general regex patterns.

The full pipeline behind REVIZOR is shown in Figure 1. The steps to build the plugin are as follows:

- 1) *Collecting*: Collect graph-based patterns of code changes in Python and choose the ones that should be automated.
- 2) *Preprocessing*: Build the REVIZOR plugin using any type of sources from step 1. If in future any new change patterns are added, re-build the plugin.

The steps are described in more detail in the next subsections. For more information on how to build a plugin with REVIZOR, see its README on GitHub [18].

¹<https://plugins.jetbrains.com/docs/intellij/intellij-platform.html>

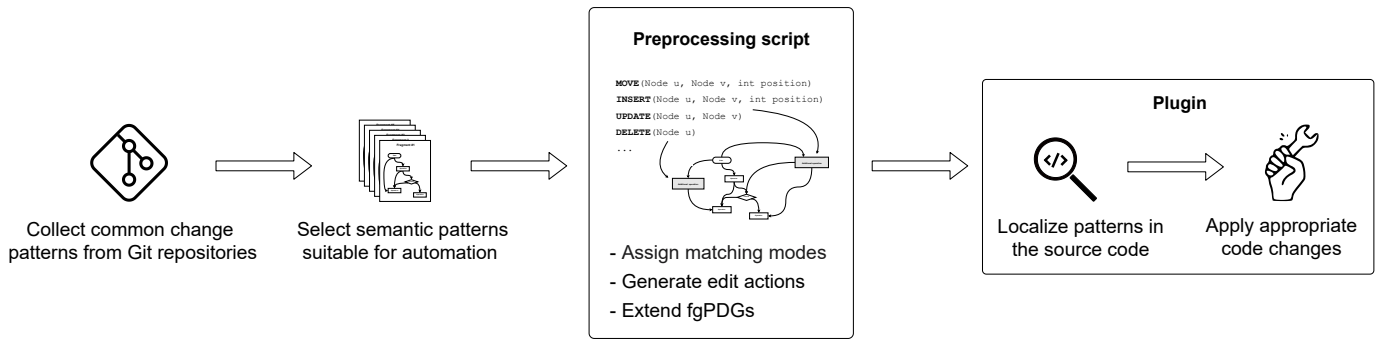


Fig. 1. An overview of the proposed approach.

After the plugin is installed in the IDE, it tracks developer’s actions when a `.py` file is opened or changed in the code editor. Namely, the plugin:

- i Builds an fgPDG for each function in the developer’s code using its PSI tree (an enriched form of a concrete syntax tree used in the IntelliJ Platform).² It is performed on-the-fly using IntelliJ’s mechanism called *code inspections*.³
- ii Checks such graphs for possible subgraph isomorphisms with the *before* version of each available change pattern stored in the plugin’s resources.
- iii Highlights the corresponding code tokens in the editor if such an isomorphism is detected and suggests the respective improvement.
- iv Performs the improvements confirmed by the developer using a sequence of *edit actions* extracted with GUMTREE during the preprocessing step.

B. Change Pattern Collecting

In our prior work [17], we had gathered a dataset of 120 popular GitHub repositories based on their domain, length of the commit history, age of the project, and number of contributors. Finally, we had discovered a total of 7,481 code change patterns. To understand their semantics better, we manually evaluated and categorized 803 patterns that appeared in at least two projects. From this pool, we selected nine patterns presented in Table 2 for the evaluation of our test plugin’s prototype according to the following criteria: the patterns had different structure and semantics and constituted good examples of what software engineers we consulted thought worth automating. The selected changes are related to developers’ best practices, which evolve rapidly with any language and are rarely documented promptly. Also, fixes for such changes as *Enumerate* are not easy to implement because the pattern could be distributed. REVIZOR-based plugins not only detect such unique change patterns but fix them as shown in Figure 3. Finding change patterns can be automated to a large extent and it is a promising direction of future work.

C. Pattern preprocessing

Before the mined change patterns can be used in the plugin, they should be preprocessed with our tool. The process is fully

Name	Before	After
<i>IsFile</i>	<code>os.path.exists(path)</code>	<code>os.path.isfile(path)</code>
<i>IsDirectory</i>	<code>os.path.exists(path)</code>	<code>os.path.isdir(path)</code>
<i>Enumerate</i>	<code>for i in range(len(lst): var = lst[i]</code>	<code>for i, var in enumerate(lst):</code>
<i>Zeros</i>	<code>np.array([0, ...])</code>	<code>np.zeros(...)</code>
<i>Keys</i>	<code>sorted(vocab.keys())</code>	<code>sorted(vocab)</code>
<i>Range</i>	<code>np.array(range(n))</code>	<code>np.arange(n)</code>
<i>Assert</i>	<code>self.assertIs(x, True)</code>	<code>self.assertTrue(x)</code>
<i>Callable</i>	<code>f = getattr(self, 'func') isinstance(f, collections.Callable)</code>	<code>f = getattr(self, 'func') callable(f)</code>
<i>Log1p</i>	<code>np.log(1 + abs(x))</code>	<code>np.log1p(abs(x))</code>

Fig. 2. The examples of the chosen nine change patterns for REVIZOR-based plugin prototype evaluation.

automated except for the specification of tooltip annotations for each pattern that will appear in the IDE.

1) *Assignment of matching modes*: During preprocessing of the supplied graphs, REVIZOR automatically specifies how *data* vertices from the pattern should be matched with the ones from developer’s code when a REVIZOR-based plugin looks for “familiar” patterns in code. Such rules are called *matching modes* and take into account the vertices’ labels, positions, and neighbours in the fgPDG.

Examples. Some *user-defined* variable names that refer to the same data element (e.g., a list may be named `lst`, `data` or `items`) do not need to be exactly matched during subgraph isomorphism search, and therefore we assigned such vertices with the `match_any_label` matching mode. Other variable names *should* be considered as having a partial match with a common suffix, e.g., `dict.keys` and `vocab.keys` (the `match_longest_common_suffix` mode). Some should always be matched exactly as they are built-in or external library Python functions and attributes, e.g., `collections.Callable` or `np.log` (the `match_original_labels` mode).

2) *Generating edit actions*: We use GUMTREE [13] to extract sequences of edit actions from PSI trees of the before and after code fragments related to the change (they are stored by the miner together with the respective graphs). Edit actions

²<https://plugins.jetbrains.com/docs/intellij/psi.html>

³<https://plugins.jetbrains.com/docs/intellij/code-inspections.html>

```

36 def iterate_and_print_even_items():
37     data = [1, 2, 3]
38     print('Start working...')
39     for i in range(len(data)):
40         print(f'Counter: {i}')
41         current_item = data[i]
42         if current_item % 2 == 0:
43             print(f'Item: {current_item}')

36 def iterate_and_print_even_items():
37     data = [1, 2, 3]
38     print('Start working...')
39     for i, current_item in enumerate(data):
40         print(f'Counter: {i}')
41         if current_item % 2 == 0:
42             print(f'Item: {current_item}')

```

Fig. 3. An example of applying a graph-based change pattern in our plugin: replacing the `range` function call in the `for` loop condition with `enumerate`. The highlighted code tokens are placed in different lines of code and include all the vertices of the detected fgPDG with respect to the data flow dependencies, e.g., `data` declaration in line 37.

keep references to the corresponding vertices of the PSI tree before the change, making it possible to apply these actions to code merely one by one.

While GUMTREE extracts *all* edits from such code changes (most of them potentially unrelated to our pattern), we need to get only *necessary* actions. We do so by calculating a Longest Common Edit Operation Subsequence with generalized identifiers by iteratively comparing the extracted edit actions sequences pairwise. A similar process was described in detail in the work about LASE [8], but instead of keeping an edit context, we use the isomorphic mappings between fgPDGs for all encountered fragments of the pattern.

3) *Extending fgPDGs*: When an appropriate subsequence of edit actions is extracted and saved, we extend the originally mined fgPDG of the pattern with additional vertices that appeared in these edits. This is done because some of them may contain PSI nodes that did not even exist in the original fgPDG of the pattern, such as parents of the *moved* or *inserted* vertices in the PSI tree.

Finally, the preprocessing script automatically saves the assigned matching modes and edit actions, as well as the extended graph and the manually provided description of each pattern. After loading all of them as resources, the plugin is ready to go.

D. Possible Application

A REVIZOR-based plugin may meet the following purposes of a code standardization linter:

- Self-education (individual level): Use our prototype with a selection of promising code changes from popular GitHub repositories.
- Enforcement of style guidelines (team/company level): Build a plugin around mined or manually created patterns of interest.

- Introduction of fresh high-quality inspirations from other developers (individual/team/company/education level): Build a plugin around mined and sifted patterns from relevant code repositories.

IV. EVALUATION

As a preliminary study, we asked nine developers to install our plugin in their PyCharm IDE and test it on an example project [19], which contained manually selected code snippets from several Python projects where the chosen patterns were encountered during mining. The participants were requested to find and perform all the suggested code changes, considering the usability of the tool. All the developers had from two to five years of professional experience and confirmed that they often used intention actions in the IntelliJ-based IDEs to improve their code quality. The survey participants also agreed that the idea of using the most common code changes mined from GitHub as quick fixes in the IDE looked potentially useful.

We asked them to rate different aspects of the plugin's performance with one of the four responses: *very dissatisfied* (1 point), *not really satisfied* (2 points), *rather satisfied* (3 points) and *very satisfied* (4 points). The average score we received regarding the correctness of the plugin's edit operations was 3.66 out of 4, and the overall usability was rated at 3.77 out of 4. Developers also highly evaluated the performance of REVIZOR in terms of not affecting the overall IDE performance (3.88 out of 4). The lowest rated feature of the plugin was the pattern's visualization part (3.33 out of 4), as it turned out that the current approach to highlighting complex distributed patterns sometimes could be confusing for the developers.

V. CONCLUSION AND FUTURE WORK

In this paper, we presented an extendable approach to automated code enhancement. We proposed a data-driven tool called REVIZOR for building static code analysis plugins that use subgraph isomorphism mappings for pattern localization and GUMTREE edit actions to automate changes. The tool uses frequent Python change patterns mined from GitHub repositories. We also created a test prototype of the plugin for a popular Python IDE called PyCharm and evaluated it on several experienced developers who approved its usability.

We received a lot of feedback about improving the UI/UX of the plugin, *i.e.*, how it treats the patterns, including an idea to highlight distributed patterns in a more intelligent way: first highlight the key token and when the user clicks it, highlight the other parts of the pattern. Also, we received feature requests such as to apply the selected change across the whole project or to suppress the suggestions for particular scopes of code. We aim to implement these features in future.

Overall, the described data-driven approach is potentially extendable to any other programming languages, but in order to capture any data or control dependencies in the code, the extended approach should be tightly dependent on the language grammar. We also plan to support the unified fgPDG representations in our tool for other languages using the PSI.

REFERENCES

- [1] I. Zayour and H. Hajjdiab, “How much integrated development environments (IDEs) improve productivity?” *JSW*, vol. 8, no. 10, pp. 2425–2431, 2013.
- [2] M. Monperrus, “The living review on automated program repair,” 2020.
- [3] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, “A study of repetitiveness of code changes in software evolution,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 180–190.
- [4] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, “Graph-based mining of in-the-wild, fine-grained, semantic code change patterns,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 819–830.
- [5] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? The ManySStuBs4J dataset,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.
- [6] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” *arXiv preprint arXiv:1711.00740*, 2017.
- [7] (2021) PyCharm official website. [Online]. Available: <https://www.jetbrains.com/pycharm/>
- [8] N. Meng, M. Kim, and K. S. McKinley, “LASE: locating and applying systematic edits by learning from examples,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 502–511.
- [9] J. Bader, A. Scott, M. Pradel, and S. Chandra, “Getafix: Learning to fix bugs automatically,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.
- [10] Y. Ueda, T. Ishio, A. Ihara, and K. Matsumoto, “Devreplay: Automatic repair with editable fix pattern,” *arXiv preprint arXiv:2005.11040*, 2020.
- [11] N. Meng, M. Kim, and K. S. McKinley, “Systematic editing: Generating program transformations from an example,” *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.
- [12] B. Fluri, M. Wursch, M. Plnzer, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.
- [13] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.
- [14] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.
- [15] T. Lutellier, L. Pang, V. H. Pham, M. Wei, and L. Tan, “ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair,” *arXiv preprint arXiv:1906.08691*, 2019.
- [16] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, “Accurate and efficient structural characteristic feature extraction for clone detection,” in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 440–455.
- [17] Y. Golubev, J. Li, V. Bushev, T. Bryksin, and I. Ahmed, “Changes from the trenches: Should we automate them?” *arXiv preprint arXiv:2105.10157*, 2021.
- [18] (2021) Revizor repository. [Online]. Available: <https://github.com/JetBrains-Research/revizor/>
- [19] (2021) Example project to test the plugin. [Online]. Available: <https://doi.org/10.5281/zenodo.5179752>