# Automating Repetitive Code Changes using Examples

Reudismam Rolim*
Federal University of Campina Grande, Brazil
reudismam@copin.ufcg.edu.br

## ABSTRACT

While adding features, fixing bugs, or refactoring the code, developers may perform repetitive code edits. Although Integrated Development Environments (IDEs) automate some transformations such as renaming, many repetitive edits are performed manually, which is error-prone and time-consuming. To help developers to apply these edits, we propose a technique to perform repetitive edits using examples. The technique receives as input the source code before and after the developer edits some target locations of the change and produces as output the top-ranked program transformation that can be applied to edit the remaining target locations in the codebase. The technique uses a state-of-the-art program synthesis methodology and has three main components: **a)** a DSL for describing program transformations; **b)** synthesis algorithms to learn program transformations in this DSL; **c)** ranking algorithms to select the program transformation with the higher probability of performing the desired repetitive edit. In our preliminary evaluation, in a dataset of 59 repetitive edit cases taken from real C# source code repositories, the technique performed, in 83% of the cases, the intended transformation using only 2.8 examples.

## CCS Concepts

•**Software and its engineering** → **Domain specific languages; Programming by example;**

## Keywords

Program synthesis, Programming by examples, Software evolution

## 1. PROBLEM AND MOTIVATION

During software evolution, developers often perform repetitive `edits` [5, 12, 13, 16, 20], i.e., similar concrete edits in the codebase. For instance, to apply an API change, the developer locates the references to the old API and replaces it by the new API [10, 11, 19, 21]. Modern IDEs provide automated `transformations` such as general-purpose refactoring suites to automate repetitive `edits`. A `transformation` generalizes an `edit` allowing it to be applied in different locations of the codebase. However, more complex `transformations` often do not fit into the IDEs refactoring suites and have to be performed manually [6, 17]. A task that is error-prone and time-consuming.

```
- while (receiver.CSharpKind() == SyntaxKind.ParenthesizedExpression)
+ while (receiver.IsKind(SyntaxKind.ParenthesizedExpression))

- foreach (var m in modifiers) { if (m.CSharpKind() == modifier) return true; };
+ foreach (var m in modifiers) { if (m.IsKind(modifier)) return true; };
```

**Figure 1:** Repetitive `edit` applied to Roslyn.

To motivate the problem, we present a repetitive `edit` applied to Roslyn[1] (Figure 1). Minus lines show the original code and plus lines show the edited code. Developers replaced the `==` operator to compare the value returned by `CSharpKind` with a specific syntax kind by the invocation of `IsKind`, passing the right-hand side expression as the argument. Commit *8c14644*[2] shows these developers edited 26 locations. However, developers had missed 689 target locations of the repetitive `edit`. The lack of tool support, effort to manually apply it, and unawareness of all locations to edit may contribute to this high number of missed locations. We performed this repetitive `edit` with our technique and opened a pull request in Roslyn's repository. Developers confirmed the `edit` and merged the pull request with the codebase.

## 2. BACKGROUND AND RELATED WORK

The interdisciplinary field of automation of repetitive operations using inductive programming (IP) studies the automatic synthesis of programs from input-output examples [4], [2]. It has been popularized in the industry by FlashFill [3], a feature of Microsoft Excel 2013 that synthesizes string `transformation` macros. FlashFill raised a novel framework of methods for IP, the PROSE framework [15], applied in many industrial applications. An example is FlashExtract, a tool for data extraction from semi-structured text files, deployed as a cmdlet in Microsoft PowerShell for Windows 10 [7]. FlashFill and FlashExtract perform string `transformations`, we focus on learning program `transformations` to edit ASTs.

On the program `transformation` domain, Meng et al. [9, 11] proposed Lase, a technique to perform repetitive `edits`

*Advisor: Gustavo Soares
Email: gsoares@dsc.ufcg.edu.br



*FSE'16*, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2983944

[1] https://github.com/dotnet/roslyn
[2] https://github.com/dotnet/roslyn/commit/8c14644

using examples. Given two or more input-output methods as examples, it creates context-aware `transformations` to infer edits common to all examples. However, Lase only abstracts type names, variables, and methods. For instance, it could not apply the repetitive `edit` in Figure 1 because the expressions mismatch. Additionally, Lase performs only an edit per method, and in the method level. For example, the repetitive `edit` in Figure 1 could not be performed because it occurs many times in the method. Nguyen et al. [14] presented LibSync, a technique that migrates APIs. Given the current client API and a set of programs that already migrate the API, this technique suggests the instances and the edit operations to perform the repetitive `edit`. However, the technique scope limits to API `transformations` and requires migrated clients. Boshernitsan et al. [1] present a technique that allows developers to edit the code through a visual interface, and Robbes and Lanza [18] present a technique for program `transformations` guided by examples. Unlike these techniques that are semi-automatic, we learn the program `transformations` automatically.

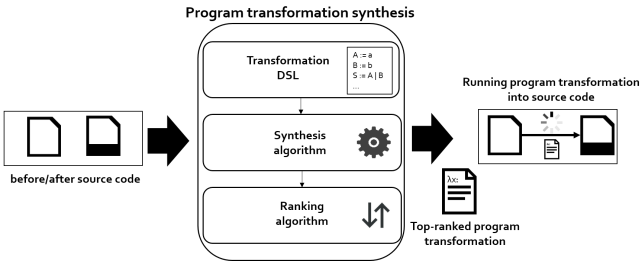


**Figure 2:** Technique for performing repetitive `edits`.

## 3. SOLUTION

In this section, we describe our technique. Based on examples of the repetitive `edits`, it learns a program `transformation` using inductive programming. The technique receives as input the program before and after the developer's edits. As output, the technique produces the top-ranked program `transformation` that can be applied to `edit` the codebase as can be seen in Figure 2. The technique is based on PROSE and contains three main components: **a)** a DSL for describing program `transformations`; **b)** algorithms for learning program `transformations` in this DSL using examples; **c)** ranking algorithms to select a program `transformation` with the higher probability to do the desired `edits` to other locations. To illustrate our DSL, Figure 3 shows a program `transformation` for the repetitive `edit` in Figure 1. A `Transformation` defines a list of `Rewrite Rules`. Each `Rewrite Rule` is a `map` function composed by an `Edit Operation` and a `Filter`. The `Edit Operation` encodes an `edit` to locations in the codebase such as `Insert`, `Update`, `Delete`, and the `Filter` locates the `edit` target locations. The program `transformation` in Figure 3 contains a `Filter`, which selects all nodes that follow the pattern `<exp>.CharpKind() == <exp>`. The `Update` edit operation replaces the `Equals` expression by an invocation to the method `IsKind`. Two `Reference` operators select the first child of the access expression and the second child of the equals expression, respectively.

**Program Synthesis**: The synthesis algorithm learns a program `transformation` consistent with developers examples. The examples work as a specification of the `transformation`.

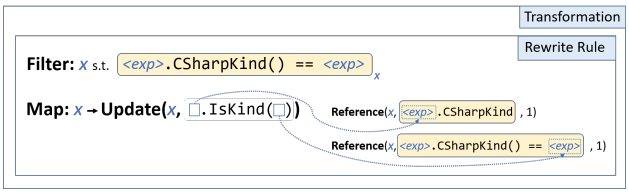


**Figure 3:** Code snippet to insert an invocation node.

The algorithm recursively decomposes the problem of learning a program `transformation` based on the examples into subproblems. For instance, from the example in Figure 1, the problem of learning a `Rewrite Rule` from the example is decomposed in two subproblems. The first subproblem of learning the `Filter`, and the second subproblem of learning the `Edit Operation`. In the decomposition, the algorithm creates new example specifications for the subproblems to be solved. Because example specification is ambiguous, the number of program `transformations` that can be generated can be huge ($> 10^{20}$). Therefore, we must an efficient way represent all these program `transformations`, and we must select the program `transformation` that has the higher probability of performing the `edit` on other locations of the codebase. We use a version space algebra that allows to efficiently storing these program `transformations` and ranking them.

## 4. EVALUATION AND RESULTS

**Evaluation Methodology**: Our evaluation focuses on the following research questions:

- *RQ1: Can the technique perform real world repetitive `edits`?*
- *RQ2: How many examples are required to apply the desired program `transformation`?*

We evaluate our technique on repetitive code `edits` from software repositories. To answer RQ1, for each `edit`, we compare the results after application of the learned program `transformation` to `edits` on developer's commits. To answer RQ2, we count the number of examples required to perform the `edit`.

**Preliminary Results**: We conducted an empirical study on real source code repositories to investigate repetitive `edits` on C# systems. We found 59 repetitive `edit` scenarios out of 404 commits, where the repetitive `edits` are applied from 3 up to 60 locations. We compared the technique `edit` to developer commit `edit` and, in 83% of the cases, our technique applied the intended transformation using only 2.8 examples.

**Future Work**: The evaluation was performed on a small set of repetitive `edits`, identified manually. We aim to increase this study, identifying characteristics of repetitive `edits` in an automatic fashion and validate our technique on this set of `edits`. In addition, although some studies focused on automating repetitive `edits`, few studies evaluate models of user-interaction in programming by examples. For instance, examples are incomplete specification, which can result in incorrect program `transformations`. If two program `transformations` produce different results for the same input, we can show to the developer code `edit` in which these two program `transformations` differ and ask him which one produces the desired result [8]. We also want to investigate the time to learn and apply the desired transformation.

## REFERENCES


[1] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *Proceedings of the SIGCHI Conference on Human Factors in @Computing Systems*, CHI '07, pages 567–576, New York, USA, 2007. ACM.

[2] S. Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pages 13–24, New York, USA, 2010. ACM.

[3] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT @Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, USA, 2011. ACM.

[4] S. Gulwani, J. Hernandez-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[5] M. Kim and N. Meng. *Recommending Program Transformations to Automate Repetitive Software Changes*, pages 1 – 31. Springer Berlin Heidelberg, 2014.

[6] G. Kniesel and H. Koch. Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51, 2004.

[7] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on @Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM.

[8] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User @Interface Software & Technology*, UIST '15, pages 291–301, New York, NY, USA, 2015. ACM.

[9] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 392–402, Piscataway, USA, 2015. IEEE Press.

[10] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 329–342, New York, USA, 2011. ACM.

[11] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on @Software Engineering*, ICSE '13, pages 502–511, Piscataway, USA, 2013. IEEE Press.

[12] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 803–813, New York, NY, USA, 2014. ACM.

[13] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of 28th the IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 180–190, New York, NY, USA, 2013.

[14] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation. In *Proceedings of the ACM International Conference on Object @Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 302–321, New York, USA, 2010. ACM.

[15] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the ACM international conference on Object @oriented programming systems languages and applications*, OOPSLA '15, pages 542–553, New York, NY, USA, 2015. ACM.

[16] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann. The uniqueness of changes: Characteristics and applications. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 34–44, Piscataway, NJ, USA, 2015. IEEE Press.

[17] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev. Refactoring with synthesis. In *Proceedings of the 28th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 339–354, New York, USA, 2013. ACM.

[18] R. Robbes and M. Lanza. Example-based program transformation. In *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2008.

[19] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'08, pages 295–312, Atlanta, USA, 2008. ACM.

[20] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing*, VL-HCC '14, pages 173–180, Piscataway, USA, 2004. IEEE Press.

[21] L. Wasserman. Scalable, example-based refactorings with refaster. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, pages 25–28, New York, USA, 2013. ACM.