APIFIX: Output-Oriented Program Synthesis for Combating Breaking Changes in Libraries

XIANG GAO, National University of Singapore, Singapore
ARJUN RADHAKRISHNA, Microsoft, USA
GUSTAVO SOARES, Microsoft, USA
RIDWAN SHARIFFDEEN, National University of Singapore, Singapore
SUMIT GULWANI, Microsoft, USA
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Use of third-party libraries is extremely common in application software. The libraries evolve to accommodate new features or mitigate security vulnerabilities, thereby breaking the Application Programming Interface (API) used by the software. Such breaking changes in the libraries may discourage client code from using the new library versions thereby keeping the application vulnerable and not up-to-date. We propose a novel *output-oriented program synthesis* algorithm to automate API usage adaptations via program transformation. Our aim is not only to rely on the few example human adaptations of the clients from the old library version to the new library version, since this can lead to over-fitting transformation rules. Instead, we also rely on example usages of the new updated library in clients, which provide valuable context for synthesizing and applying the transformation rules. Our tool APIFIX provides an automated mechanism to transform application code using the old library versions to code using the new library versions - thereby achieving automated API usage adaptation to fix the effect of breaking changes. Our evaluation shows that the transformation rules inferred by APIFIX achieve 98.7% precision and 91.5% recall. By comparing our approach to state-of-the-art program synthesis approaches, we show that our approach significantly reduces over-fitting while synthesizing transformation rules for API usage adaptations.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; **Software maintenance tools**; • **Computing methodologies** → **Artificial intelligence**.

Additional Key Words and Phrases: API usage adaptation, Breaking changes, Program transformation, Program synthesis, Programming by example

ACM Reference Format:
Xiang Gao, Arjun Radhakrishna, Gustavo Soares, Ridwan Shariffdeen, Sumit Gulwani, and Abhik Roychoudhury. 2021. APIFIX: Output-Oriented Program Synthesis for Combating Breaking Changes in Libraries. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 161 (October 2021), 27 pages. https://doi.org/10.1145/3485538

1 INTRODUCTION

In the process of software development, developers usually rely on third-party libraries to implement certain functionalities. To enable developers to use different components, these libraries usually provide a set of public Application Programming Interfaces (APIs), which define the contract of

Authors' addresses: Xiang Gao, National University of Singapore, Singapore, gaoxiang@comp.nus.edu.sg; Arjun Radhakrishna, Microsoft, USA, arradha@microsoft.com; Gustavo Soares, Microsoft, USA, gustavo.soares@microsoft.com; Ridwan Shariffdeen, National University of Singapore, Singapore, ridwan@comp.nus.edu.sg; Sumit Gulwani, Microsoft, USA, sumitg@microsoft.com; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@comp.nus.edu.sg.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART161
https://doi.org/10.1145/3485538

using the libraries, such as the kinds of calls that can be invoked, the ways to invoke them, the right arguments that should be passed, etc. The client applications that rely on a certain library must use the API correctly and respect the contract built by the APIs. However, when a library evolves to accommodate new features or fix security vulnerabilities, it may change the contract defined via APIs and cause its existing client applications to break. The changes that can fail client applications are called *breaking changes*, which makes around 15% of API modifications [Xavier et al. 2017].

Fixing API usage errors caused by breaking changes is a time-consuming and error-prone task. In order to use up-to-date libraries, the developers/maintainers of clients have to keep track of the library update, analyze the changed code, and manually fix the API usage errors. Due to the complexity, developers are not willing to update their dependencies. Indeed, inertia dictates that 82% of developers continue to use the same version as they have previously used [Kula et al. 2018]. The practice that uses outdated vulnerable libraries will expose the clients to the risk of malicious attacks. This becomes more serious with financial applications (e.g., bank clients) which could cause a bigger impact. This indicates the necessity and importance of automatically updating clients' dependencies in an efficient manner.

In recent years, we have seen an emerging trend of tools and techniques that synthesize abstract transformation rules using examples of human code edits and apply the synthesized rules to automate program transformations [Bader et al. 2019; Meng et al. 2011, 2013; Miltner et al. 2019; Rolim et al. 2017]. Existing program transformation techniques have been studied to automatically update clients' dependencies [Dagenais and Robillard 2011; Fazzini et al. 2019; Haryono et al. 2020; Henkel and Diwan 2005; Nguyen et al. 2010; Xu et al. 2019]. Those techniques first infer transformation rules from the before- and after-adaptation examples from human-adapted clients, and then apply the inferred rules to adapt the clients that are relying on outdated libraries.

Although existing approaches can adapt many clients of widely-used libraries, e.g., Android SDK [Fazzini et al. 2019; Xu et al. 2019], since the given human adaptation examples represent incomplete adaptation specifications, the synthesized rules are prone to *overfit* to the given examples. That is, the synthesized rule works well on the given examples, but does not reflect developers' intent and hence may incorrectly adapt unseen code outside the given examples. Synthesizing transformation rules is a process of generalizing concrete examples with the objective that the synthesized rules can be applied to any target code. An overfitted transformed rule can be either over-generalized or under-generalized. An under-generalized rule can lead to false negatives where it fails to transform some target codes that should be transformed. While an over-generalized rule can lead to false positives (transform code that should not be transformed or transform code in an incorrect way). Balancing false positives with false negatives is one of the main challenges in synthesizing transformation rules.

Existing approaches have studied how to handle the generalization problem in different ways. For instance, Meditor [Xu et al. 2019] and AppEvolve [Fazzini et al. 2019] simply generate the most general rule by abstracting all the project-specific details (e.g., variable identifiers), which may lead to over-generalized transformation rules. Many approaches, e.g., LASE [Meng et al. 2013] and ReFazer [Rolim et al. 2017], synthesize the most specific rule over the given examples. To synthesize a properly generalized rule, these approaches require multiple examples. However, multiple human adaptation examples are not always available in reality because client developers are not willing to upgrade their dependencies. Further, for a library that is updated recently, there could be very few clients that have been adapted to the new library. Even though CocciEvolve [Haryono et al. 2020] learns from a single adaptation example, it can only adapt Android deprecated-API usages by introducing an if-condition.

Semi-supervised synthesis [Gao et al. 2020] proposes to learn transformation rules based on input-output examples and additional inputs. The additional inputs, which are a set of inputs

without available correct outputs, can help disambiguate how to generalize the transformation rule by providing more examples of input ASTs. Apart from additional inputs, we observe that there are a large number of available additional outputs. The additional outputs are the after-transformation codes without before-transformation codes being available. This leads to our research question: *can additional outputs also be used to synthesize transformation rules?* In our setting, the additional outputs are the usages of the new library version. The additional outputs are embedded with the human intelligence which demonstrates the structures of the after-transformation code, i.e., the pattern of using the new library version. The additional outputs can be helpful in synthesizing transformations in the following two aspects: (1) help us disambiguate how to generalize the transformation rule; (2) help synthesize more transformation rules by providing after-transformation AST patterns.

In this paper, we propose *output-oriented program synthesis* to generate program transformation rules according to (1) a set of input-output edit examples, and (2) a set of additional outputs. The synthesis goal is to produce transformation rules that are consistent with the given input-output edit examples, and the synthesized transformation rules should be able to produce the additional outputs on some "unknown" inputs. Compared with existing techniques that synthesize rules according to input-output examples, considering additional outputs can help synthesize more properly generalized individual rules, and generate more transformation rules. Our technique synthesizes transformation rules using the following workflow:

- Since there could be a large number of additional outputs, our technique first determines the additional outputs that are useful for the synthesis task. An additional output will be regarded as useful if it can help improve the transformation rules. Basically, the given input-output edit examples represent some code structures before- and after-transformation. If one additional output reflects after-transformation code structures that are not reflected in the input-output examples, our technique regards it as a useful additional output since it could be helpful for synthesizing more substantial transformation rules.
- For each useful additional output, our technique then infers its corresponding input by analyzing its relationship with the given examples. Linking the inferred input to the additional output constructs an additional example, which can be then used as a normal input-output example by any existing synthesizer.
- Since the generated additional examples represent new code structures, they may not be unified with existing input-output examples to produce a single transformation rule. Therefore, our technique groups the examples, including the user-provided and inferred examples, into clusters, and then synthesizes a transformation rule for each cluster. Determining which synthesized transformation rule should be applied to a given input depends on the context of the code that should be transformed.

We then use output-oriented program synthesis to automate API usage adaptations. Although the number of available human adaptations is limited, the new clients usually use the updated library directly, which gives us an opportunity to mine usages for the new version(s) of the libraries. In this setting, output-oriented program synthesis takes human adaptations as input-output examples and **the usages of the updated library** as additional outputs to synthesize transformation rules. Relying on output-oriented program synthesis to synthesize transformation rules has two main advantages. First, with the help of additional output, our technique does not require a large number of available examples to synthesize a proper transformation rule. Second, mining usage of new libraries is much more efficient than mining adaptation examples since we just need to search for usages in the latest version of the client instead of going through all the commits.

We realized our approach in a tool called APIFix, which is built on top of ReFazer. We evaluated APIFix on a benchmark with seven well-known C# libraries and 138,206 clients that depend on those libraries. Totally, we collect 218 human adaptations (concrete examples) and 2973 new usages (additional outputs) and evaluate our approach in three experiments. First, we measure the effectiveness of output-oriented program synthesis via cross-validation. Evaluation results show that output-oriented program synthesis achieves 91% accuracy in correctly transforming programs. Second, we applied APIFix on 2154 API usages of outdated libraries, achieving 98.7% precision and 91.5% recall. Last, we compared our output-oriented program synthesis with existing program synthesis tools ReFazer and semi-supervised program synthesis. Evaluation results show that our approach improves both precision and recall over ReFazer. Compared with semi-supervised program synthesis, our technique significantly improves the precision, while not affecting the recall significantly. We summarize our contributions as follows:

- We propose a novel output-oriented program synthesis that synthesizes transformation rules based on both input-output examples and additional output. This helps reduce the *over-fitting* problem (to given input-output examples) in program synthesis.
- We apply the output-oriented program synthesis to automatically fix API usage errors caused by API evolution by learning from human adaptation examples and usage of updated libraries. Compared to existing techniques, the proposed approach requires fewer adaptation examples and can generate proper transformation rules.
- We implemented our technique in a tool called APIFix, which includes the full pipeline: collecting human adaptations from open-source repositories, collecting additional API usages, inferring transformation rules, and generating patch suggestions for the API usage errors caused by breaking changes.
- We evaluated our tool on a benchmark with seven C# libraries and 138,206 clients. Our evaluation results show that APIFix achieves 98.7% precision and 91.5% recall. Our tool is open-source available at https://github.com/gaoxiang9430/APIFix.

2 MOTIVATING EXAMPLE

In this section, we give a high-level overview of the output-oriented program synthesis in automating API usage adaptations by presenting an example from DbUp. DbUp is a .NET library that helps developers to deploy changes to SQL Server databases. It supports most of the widely-used databases, such as MySQL, SQLite, SQLServer, Oracle, etc. DbUp has 10.7M total downloads according to the Nuget Statistics and more than 1800 open-source dependents according to the dependency graph of Github[1]. At the time of this paper's writing, DbUp has officially released 40 versions ranging from v1.0.8 to v4.5.0. Each release, especially the major releases, may change some public APIs and hence introduce a number of breaking changes to the old versions. For instance, when DbUp was updated from v3.3.5 to v4.0 [2], the constructor of the widely used class `SqlScriptExecutor` was changed as follows:

```
- public SqlScriptExecutor(Func<IConnectionManager>, Func<IUpgradeLog>,
-     string, Func<bool>, IEnumerable<IScriptPreprocessor>) ...
+ public SqlScriptExecutor(Func<IConnectionManager>, Func<IUpgradeLog>,
+     string, Func<bool>, IEnumerable<IScriptPreprocessor>, Func<IJournal>)...
```

When client applications upgrade their dependency DbUp from v3.3.5 or older versions to v4.0, they may receive compilation error "*'SqlScriptExecutor' does not contain a constructor that takes 5 arguments*". Even though the change of this constructor is simply inserting an additional parameter,

[1] https://github.com/DbUp/DbUp/network/dependents

[2] https://github.com/DbUp/DbUp/compare/3.3.5...4.0.0-beta0003#diff-bfbdbc9a

```
E1: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),() => new
        ConsoleUpgradeLog(), null, () => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),() => new
        ConsoleUpgradeLog(), null, () => true, null, () => Substitute.For<IJournal>())

E2: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),() => new
        ConsoleUpgradeLog(), "foo",() => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),() => new
        ConsoleUpgradeLog(), "foo",() => true, null,() => Substitute.For<IJournal>())

E3: - new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        {IsScriptOutputLogged = true}, () => new ConsoleUpgradeLog(),
        "foo", () => true, null)
    + new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        {IsScriptOutputLogged = true}, () => new ConsoleUpgradeLog(),
        "foo", () => true, null,() => Substitute.For<IJournal>())
```

Fig. 1. History edits on *SqlScriptExecutor* that adapt clients from DbUp v3.3.5 or older version to v4.0.

```
I1: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
    => new ConsoleUpgradeLog(), null,() => false, null)
    ▹ new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),() => new
        ConsoleUpgradeLog(), null,() => false, null, () => Substitute.For<IJournal>())

I2: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true) {
    IsScriptOutputLogged = true },() => new ConsoleUpgradeLog(), "foo",() => true
    , null)
    ▹ new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true)
        IsScriptOutputLogged = true ,() => new ConsoleUpgradeLog(), "foo",() => true, null, ()
        => Substitute.For<IJournal>())

I3: new SqlScriptExecutor(()=> c.ConnectionManager,() => c.Log, schema,() => c.
    VariablesEnabled, c.ScriptPreprocessors)
    ▹ new SqlScriptExecutor(()=> c.ConnectionManager,() => c.Log, schema,() => c.VariablesEnabled,
        c.ScriptPreprocessors, () => c.Journal)
```

Fig. 2. Code from clients that still use DbUp v3.3.5 or older versions

it is not easy for client developers to figure out what new argument should be passed, how the new argument is relevant to the other arguments, and how the surrounding context affects the creation of the new argument. In order to use the latest version of the library (i.e., DbUp 4.0), the client developers have to read documents of the library, understand the change, and manually fix the usage errors, which is an error-prone and time-consuming task.

Fortunately, the DbUp developers have provided several examples on how to perform the adaptation within the DbUp codebase itself. For instance, the test cases of the *SqlScriptExecutor*'s constructor are also updated along with this breaking change. Figure 1 shows three example edits that are relevant to the `SqlScriptExecutor` constructor. Basically, developers modified the `SqlScriptExecutor` object creations by inserting an additional argument (`() => Substitute.For<IJournal>()`) to match the new constructor signature in DbUp v4.0. Meanwhile, we observe that there are still 411 clients relying on DbUp v3.3.5 or even older versions. Out of which, 84 clients use the `SqlScriptExecutor` constructor which require the similar adaptations. Figure 2 shows three `SqlScriptExecutor` object creation examples relying on the DbUp v3.3.5 that require adaptations.

Several approaches have been proposed to help developers to update their API usages by learning a transformation rule from the human edits (Figure 1) [Dagenais and Robillard 2011; Fazzini et al.

```
O1: new SqlScriptExecutor(() => new TestConnectionManager(dbConnection, true),()
    => logger, null,() => true, null,() => Substitute.For<IJournal>())

O2: new SqlScriptExecutor(() => Substitute.For<IConnectionManager>(),() => null,
    null,() => false, null,() => Substitute.For<IJournal>())

O3: new SqlScriptExecutor(() => c.ConnectionManager,() => c.Log, schema,() => c.
    VariablesEnabled, c.ScriptPreprocessors, () => c.Journal)

O4: new SqlScriptExecutor(() => connectionManager,() => Substitute.For<
    IUpgradeLog>(), null,() => true, null,() => versionTracker)
```

Fig. 3. Code from clients that use DbUp v4.0 or newer versions

2019; Henkel and Diwan 2005; Nguyen et al. 2010; Xu et al. 2019], and automatically transform the code requiring adaptations (Figure 2). For example, ReFazer [Rolim et al. 2017] learns a transformation rule R by looking at the edit history, and represents the learned rule using a domain-specific language (DSL). The DSL will be explained in Section 3.3. For simplicity, we show the rule R for this example as follows:

$$
\begin{aligned}
&\textit{new SqlScriptExecutor}(X_1, X_2, X_3, X_4, X_5) \mapsto \\
&\textit{new SqlScriptExecutor}(X_1, X_2, X_3, X_4, X_5, ()\text{=>}\textit{Substitute.For<IJournal>}()) \\
&\quad \textbf{where } X_1.\texttt{Type} = \text{“}\texttt{Func}\text{”} \\
&\qquad X_2.\texttt{Type} = \text{“}\texttt{Func}\text{”} \land X_2.\texttt{Text} = \text{“}() \text{ => } \textit{newConsoleUpgradeLog}()\text{”} \\
&\qquad X_3.\texttt{Type} = \text{“}\texttt{String}\text{”} \\
&\qquad X_4.\texttt{Type} = \text{“}\texttt{Func}\text{”} \land X_4.\texttt{Text} = \text{“}() \text{ => } \textit{true}\text{”} \\
&\qquad X_5.\texttt{Type} = \text{“}\texttt{Func}\text{”} \land X_5.\texttt{Text} = \text{“}\textit{null}\text{”}
\end{aligned}
$$

Terms X_1, X_2, X_3, X_4, X_5 represent the least general generalization of the five arguments in the three examples, respectively. Each term is guarded by predicates in terms of `Type` and `Text`. Term X_1 and X_3 do not have a `Text` predicate because the text of the first and third arguments in the given examples are different. The existing inductive program synthesis techniques, e.g., LASE [Meng et al. 2013], prefer to synthesize the least general generalization across the examples to avoid false positives. Unfortunately, R is not applicable to I_1 and I_3 because the text of X_2 is not "`() => new ConsoleUpgradeLog()`", X_4 is not "`() => true`" or X_5 is not "`null`". Therefore, R cannot transform I_1 and I_3, hence produce false negatives. In contrast, if we simply generalize R by ignoring all predicates on `Text` (i.e., delete predicate on X_2.`Text`, X_4.`Text` and X_5.`Text`), the generalized R will be applicable to all the usages in Figure 2. However, it may produce false positives, i.e., transform some code in an incorrect way. For instance, the inferred transformation rule from existing examples transforms I_3 by inserting `() =>Substitute.For<IJournal>()`, which is a false positive according to our observation from other clients (explained in the following). The correct transformation of I_3 (shown in ▹ red in Figure 2) should be inserting `() => c.Journal` as the last argument (it is not clear what the correct output is since the ground truth is not available, and the correct transformation is manually inferred by observing O_3 in Figure 3). How to balance false negatives and false positives is one of the main challenges in the transformation rule inference.

Our solution: To address the above challenges, we propose to learn transformation rules not only from edit examples but also from the usages of new library versions. The main insight behind our idea is that the clients created after the release of DbUp v4.0 likely use the latest version.

We find many examples from those clients that use DbUp v4.0, which are also embedded with human intelligence on how to use the updated *SqlScriptExecutor* constructor. From these usages of DbUp 4.0, we can learn the API usage patterns and infer transformation rules. These examples can improve our transformation rule to support different types of code structures as explained above. Our solution mines usages of the new version of the library DbUp 4.0, by automatically crawling through Github repositories based on the dependency graph of the library. In total, we find 36 usages of `SqlScriptExecutor` relying on DbUp v4.0 in existing clients, and Figure 3 shows four of them. By referring to the first usage O_1, we would learn that the inferred transformation rule R is also applicable even if X_2.`Text` (*() => logger* in O_1) is not exactly the same as the given edit examples (*() => new ConsoleUpgradeLog()*). Similarly, by looking at O_2, we would know that R can be applied even if X_4.`Text` is *() => false*. This knowledge helps us to improve the transformation rule R learned from the edit examples E_1, E_2, and E_3 by generalizing the context where R can be applied. Furthermore, using O_3 and O_4, we can learn that the inserted last argument is not necessary to be `()=>Substitute.For<IJournal>()`. According to the information embedded in O_3, we can infer a new transformation rule R_1. The simplified representation of R_1 is as follows:

$$
\begin{aligned}
&new\ SqlScriptExecutor(X_1, X_2, X_3, X_4, X_5) \mapsto \\
&new\ SqlScriptExecutor(X_1, X_2, X_3, X_4, X_5, () \text{ => } X_6.Journal) \\
&\quad \textbf{where } X_1.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_2.\texttt{Text} = \text{“}() \text{ => } c.ConnectionManager\text{”} \\
&\qquad X_2.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_2.\texttt{Text} = \text{“}() \text{ => } c.Log\text{”} \\
&\qquad X_3.\texttt{Type} = \text{“}\texttt{String}\text{”} \wedge X_2.\texttt{Text} = \text{“}schema\text{”} \\
&\qquad X_4.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_4.\texttt{Text} = \text{“}() \text{ => } c.VariablesEnabled\text{”} \\
&\qquad X_5.\texttt{Type} = \text{“}\texttt{Func}\text{”} \wedge X_5.\texttt{Text} = \text{“}c.ScriptPreprocessors\text{”} \\
&\qquad X_6.\texttt{Type} = \text{“}\texttt{UpgradeConfiguration}\text{”} \wedge X_6.\texttt{Text} = \text{“}c\text{”}
\end{aligned}
$$

Similarly, we will also infer another transformation rule R_2 from O_4. Rules R, R_1 and R_2 form a complete disjunctive transformation rule. Which rule R, R_1 or R_2 should be applied is determined according to the context, i.e., the values of $X_1 \dots X_5$ in this case. This additional knowledge helps us infer substantial transformation rules. With the additional outputs, we can be more confident about how to generalize the rule or create new rules, and hence be more confident on the transformed codes by the synthesized rules. Combining the knowledge learned from human edits from Figure 1 and usages of new library from Figure 3, our technique can automatically adapt all the client codes shown in Figure 2 to DbUp v4.0. After transforming the usages in Figure 2, we manually verified that all the transformed codes can be successfully compiled.

3 OUTPUT-ORIENTED PROGRAM SYNTHESIS

In this section, we first provide background on program transformations and program synthesis, and then introduce the output-oriented program synthesis problem and present the technical details of our solution to the problem.

3.1 Synthesizing Program Transformation Rules

Typed Abstract Syntax Trees. An abstract syntax tree (AST), denoted as T, is a tree representation of the abstract syntactic structure of program source code. Each node of the tree denotes a construct occurring in the source code. Each node is associated with a set of attributes including the node kind (e.g., Identifier, Expression, etc), the text value (source code fragment corresponding to the node), and a list of child nodes. A typed AST also associates each node with the node type (e.g., Integer,

Boolean, etc). For a node that does not have a type, we leave its type empty. We use `SubTree(T)` to denote the set of sub-trees of T. We use $\mathbb{T}$ to represent the set of all ASTs.

Program transformation rule. A program transformation rule: $R : \mathbb{T} \not\mapsto \mathbb{T}$ is a partial function that transforms one AST to another AST.

Program synthesis. For a given input domain $\mathbb{I}$ and an output domain $\mathbb{O}$, program synthesis techniques (more specifically, programming-by-example techniques) take a set of input-output examples $\{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$ and synthesizes a program $P : \mathbb{I} \mapsto \mathbb{O}$ such that $P(i_k) = o_k$ for $k \in 0 \ldots n$. Program synthesis can be used to automatically synthesize program transformation rules. To synthesize transformation rules, both $\mathbb{I}$ and $\mathbb{O}$ are set as $\mathbb{T}$.

Semi-supervised program synthesis. In Section 2 we highlighted the challenge of finding the correct level of generalization which can balance the false positive and false negative for the unseen inputs. Semi-supervised program synthesis [Gao et al. 2020] is designed to synthesize transformation rules by relying both on input-output examples and a set of additional inputs. Specifically, given a set of input-output examples $\{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$ and a set of additional inputs $\{ai_0, \ldots, ai_m\}$, it produces a transformation rule R, such that $R(i_k) = o_k$ for $k \in 0 \ldots n$ and $R(ai_j) \neq \perp$ for $j \in 0 \ldots m$. Intuitively, semi-supervised synthesis can improve the generalization for the given concrete input-output examples by specifying a set of additional inputs that should be incorporated into the input domain of the synthesized transformation rule R. Semi-supervised synthesis assumes the availability of additional inputs, however, finding additional inputs is an error-prone task. If provided with additional inputs that should not be manipulated (i.e., invalid additional inputs), semi-supervised synthesis will generate over-generalized transformation rules.

Example 3.1. Let us revisit the example shown in Section 2. Suppose we take the human edits in Figure 1 as input-output examples E, and the old usages from Figure 2 as additional inputs AI, semi-supervised synthesis will synthesize a transformation rule that is applicable to all the inputs from E and all the additional inputs I_1, I_2 and I_3 from Figure 2. Therefore, it will over-generalize the predicates on $X_1 \ldots X_5$ (the arguments of `SqlScriptExecutor`'s constructor), and hence transform I_3 by incorrectly inserting `() => Substitute.For<IJournal>()`. □

3.2 Problem Statement

Instead of using additional inputs for the synthesis, we use additional outputs in the synthesis process. Compared to additional inputs, considering additional outputs has two main advantages. First, determining whether an additional input is valid for the synthesis task requires human feedback. Although Gao. et al. [Gao et al. 2020] proposed semi-automated and fully automated feedback to generate additional inputs, they are implemented based on heuristic hence may not always work. Synthesizing with invalid additional inputs can lead to over-generalized or over-specified transformation rules. In contrast, the additional outputs are the after-transformation codes written by developers, which are guaranteed to be valid additional outputs for the to-be-synthesized transformation rule. Compared with additional inputs, using additional outputs does not require the involvement of human in the synthesis process. Second, the additional outputs have been embedded with human intelligence, e.g., the structure of the after-transformation codes from which we could learn new transformation rules that are not reflected in the given input-output examples.

Formally, the output-oriented program synthesis takes a set of input-output examples $\{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$ and a set of additional outputs $\{ao_0, \ldots, ao_m\}$. The synthesis goal is to produce transformation rule R such that $R(i_k) = o_k$ for $k \in 0 \ldots n$ and $R(ai_j) = ao_j$ for $j \in 0 \ldots m$, where ai_j represents some input.

```
rules           := rule | Disjunction(rules, rule)
rule            := (guard, trans)
guard           := pred | Conjunction(pred, guard)
pred            := IsNthChild(node, n)
                   | IsKind(node, kind)
                   | IsType(node, type)
                   | Attribute(node, attr) = value
                   | Not(pred)

trans           := select | construct
construct       := Tree(kind, attrs, childrenlist)
childrenlist    := EmptyChildren | select | construct
                   | Cons(construct, childrenlist)
                   | Cons(select, childrenlist)
select          := Match(guard, node)
node            := ...
```

Fig. 4. Domain-specific language for program transformation rule

3.3 Domain-Specific Language

State-of-the-art program synthesis tools, like ReFazer [Rolim et al. 2017], search for a transformation rule that satisfies the provided examples over a predefined Domain-Specific Language (DSL). The output-oriented program synthesis inherits the DSL of ReFazer, and extends it to the language $\mathcal{L}$ shown in Figure 4 (the differences are highlighted in grey).

In this DSL, program transformation rules are formulated as a function pair (`guard`, `trans`).

- The guard `guard` : $\mathbb{T} \rightarrow$ `Boolean` defines the context where the transformation rule is applied. The `guard` is composed of a set of conjunctive predicates on the attributes (e.g., Type, Kind, TextValue, etc) of the AST and its sub-trees. The `guard` evaluates where an AST node satisfy its predicate and return a `Boolean` value accordingly.
- The transformation `trans` : $\mathbb{T} \rightarrow \mathbb{T}$ defines how to transform the input AST to an output AST. In our setting, `trans` recursively constructs the output AST using two operators: (1) `select`, which returns a sub-tree of the input AST and (2) `construct` that returns a AST constructed from scratch.

Essentially, the *guard* determines which AST should be transformed, while the `trans` specifies how the AST should be transformed. Formally, we have that $R(T)$ = `trans`(T) if `guard`(T) = `true`, and $R(T) = \bot$ otherwise. In general, given a set of input-output examples $\{i_0 \mapsto o_0, \ldots, i_n \mapsto o_n\}$, synthesizing a transformation rule (`guard`, `trans`) is a generalization process of the concrete transformation examples, such that `guard`(i_k) = `true` and `trans`$(i_k) = o_k$ for all $k \in 0 \ldots n$.

Example 3.2. Consider input-output examples E_1: `handler. Handle(request)` $\mapsto$ `handler .Handle(request, token)` and E_2: `TestSubject.Handle (request)` $\mapsto$ `TestSubject.Handle (request, token)`. Given this example, Refazer [Rolim et al. 2017] will synthesize a rule given by (`guard`, `trans`). The `guard` of the transformation rule is given by a conjunction of predicates

$$\texttt{IsKind}(node, \text{“}\texttt{InvokeExpr}\text{”}) \land \texttt{IsKind}(node.children[0], \text{“}\texttt{MemberAccess}\text{”}) \\ \land \texttt{IsKind}(node.children[1], \text{“}\texttt{ArgumentList}\text{”}) \land \ldots$$

which means that this transformation can be applied to *node* only if its kind is “`InvokeExpr`”, its first child’s kind is “`MemberAccess`”, its second child’s kind is “`ArgumentList`” and etc. While the

`trans` of the transformation rule is:

$$\texttt{Tree}(\text{“}\texttt{InvokeExpr}\text{”}, [], [\mathit{select}_1, \texttt{Tree}(\text{“}\texttt{ArgumentList}\text{”}, [], [\mathit{select}_2, \mathit{construct}_1])$$

The `trans` describes how an AST should be transformed. Specifically, it transforms a input node by creating a new tree of type `InvokeExpr` with two children. The first child is a node selected from the input AST (`handler.Handle` for E_1 and `TestSubject.Handle` for E_2) using *select₁*. The sub-rule *select₁* extracts a node from input that satisfies:

$$\texttt{IsKind}(\mathit{node}, \text{“}\texttt{MemberAccess}\text{”}) \land \texttt{IsKind}(\mathit{node.children}[0], \text{“}\texttt{Identifier}\text{”})$$
$$\land\ \texttt{Attribute}(\mathit{node.children}[1], \text{“}\texttt{Text}\text{”}) = \text{“}\mathit{Handle}\text{”}) \land \texttt{IsKind}(\mathit{node.children}[1], \text{“}\texttt{Identifier}\text{”})$$

and the second child is an argument list, which is a newly created tree of type “`ArgumentList`” with two children select_2 and $\mathit{construct}_1$. Sub-rule select_2 extracts a node from input AST that satisfies:

$$\texttt{IsKind}(\mathit{node}, \text{“}\texttt{Identifier}\text{”}) \land \texttt{Attribute}(\mathit{node}, \text{“}\texttt{Text}\text{”}) = \text{“}\mathit{request}\text{”})$$

The sub-rule $\mathit{construct}_1$ returns the constant AST node `Tree`(“`Identifier`”, [“`token`”], []), which is an “`Identifier`” named as “`token`”. The detailed syntax of the transformation rule is described in Figure 4. □

Additional outputs are embedded with human intelligence that can help to synthesize new transformation rules, as discussed in Section 3.2. The transformations reflected by additional outputs and input-output examples may not be able to be unified to a single transformation rule.

Example 3.3. Consider the input-output examples shown in Example 3.2 and one additional output `_handler.Handle(request, new CancellationToken())`. The second argument in the additional output is a constructor `new CancellationToken()`, instead of a variable `token` as in the input-output examples. By looking at the additional output, the inferred $\mathit{construct}_1$ in `trans` is

$$\texttt{Tree}(\text{“}\texttt{Constructor}\text{”}, [], [\texttt{Tree}(\text{“}\texttt{Identifier}\text{”}, [\text{“}\texttt{new}\text{”}], [])$$
$$\texttt{Tree}(\text{“}\texttt{Identifier}\text{”}, [\text{“}\texttt{CancellationToken}\text{”}], [])])$$

The transformation inferred from the additional output cannot be unified with the transformation rule synthesized from the given examples, because their `construct` are different and cannot be unified into a single sub-rule. □

Different from traditional synthesis techniques that produce a single transformation rule in the form of (`guard`, `trans`), the output-oriented program synthesis can generate multiple transformation rules $\{(\texttt{guard}_0, \texttt{trans}_0), \ldots, (\texttt{guard}_n, \texttt{trans}_n)\}$. The above example will produce two transformation rules with different $\mathit{construct}_1$. The *transformation rules* are defined by a set of disjunctive transformation rules, where each of $\texttt{trans}_i$ applies to a different interval in the domain defined by $\texttt{guard}_i$. Hence, applying the synthesized transformation rules to a given AST node *node*, we have:

$$\texttt{if}\ (\texttt{guard}_0(\mathit{node}))\ \{\ \texttt{return}\ \texttt{trans}_0(\mathit{node})\ \}\ \texttt{if}\ \ldots$$
$$\texttt{if}\ (\texttt{guard}_n(\mathit{node}))\ \{\ \texttt{return}\ \texttt{trans}_n(\mathit{node})\ \}\ \texttt{return}\ \bot$$

Note that, the `guard` of transformation rules can overlap, i.e., for a AST node *node*, there may exist multiple `guard`s such that `guard`(*node*) = `true`. In this situation, *node* can be transformed in multiple ways.

Algorithm 1: Output-oriented program synthesis

Input: Input-output examples: $E = \{i_0 \mapsto o_0, \dots, i_n \mapsto o_n\}$,
additional output: $AO = \{ao_0, \dots, ao_m\}$

Output: re-write rule: R

1 $(\tau, <\sigma_0, \dots, \sigma_n>) := \bowtie \{o_0, \dots, o_n\}$;
2 $selectedAO := \{ao \mid \neg \texttt{IsInstance}(ao, \tau) \land ao \in AO\}$;
3 $AE := \texttt{InferExample}(selectedAO, E)$;
4 $EditClusters := \texttt{ClusterEdit}(E \cup AE)$;
5 $R = \{\ \}$;
6 **for** $editCluster \in EditClusters$ **do**
7 $\quad R_{guard} := \text{ReFazer}_{\texttt{guard}}(\{i \mid (i \mapsto o) \in editCluster\})$;
8 $\quad R_{trans} := \text{ReFazer}_{\texttt{trans}}(editCluster)$;
9 $\quad R := R \cup (R_{guard}, R_{trans})$;
10 **end**
11 **return** R;

12 **Function** $\texttt{InferExample}(AO, E)$:
13 $\quad AE := \{\}$;
14 $\quad \pi := \texttt{Provenance}(i_0 \mapsto o_0)$;
15 $\quad$ **for** $ao \in AO$ **do**
16 $\quad\quad (\tau, <\sigma_0, \sigma_1>) := o_0 \bowtie_\pi ao$;
17 $\quad\quad AE := AE \cup \{\ \sigma_1(\sigma_0^{-1}(i_0)) \mapsto ao\ \}$;
18 $\quad$ **end**
19 $\quad$ **return** AE;

3.4 Output-Oriented Program Synthesis

In this section, we present the technical details of output-oriented program synthesis. The procedure is depicted in Algorithm 1 and works as follows:

- Given a set of input-output examples E and additional outputs AO, we first determine which additional outputs are useful for improving the synthesized transformation rule (lines 1 - 2);
- For each useful additional output, we infer a candidate input, and hence create a set of additional examples AE (lines 12 - 19).
- We then categorize the given input-output examples E and inferred additional examples AE into clusters, and synthesize transformation rules via $\text{ReFazer}_{\texttt{guard}}$ and $\text{ReFazer}_{\texttt{trans}}$ (lines 4 - 9). $\text{ReFazer}_{\texttt{guard}}$ takes all the inputs from input-output examples, and produces a `guard` that is true on all of them, while $\text{ReFazer}_{\texttt{trans}}$ takes a set of examples and produces a `trans` consistent with them.

Filtering additional outputs. Our objective is to find the patterns reflected in the additional outputs AO but not reflected in the given input-output examples E. Program synthesis is a generalization process of the given concrete input-output examples. Informally, program synthesis infers a generalized transformation rule $\tau_i \mapsto \tau_o$ such that i_k is a instance of τ_i and o_k is an instance of τ_o for $k \in \{0, \dots, n\}$.

Definition 3.1 (usefulness)**.** We define that an additional output *ao* is `useful` for improving the transformation rule in the synthesis process, if *ao* is not an instance of τ_o (i.e., *ao* is a counter-example). In other words, a useful additional output is a concrete output that cannot be generated by the transformation rule synthesized using E. Considering the useful additional outputs will help refine the transformation rule in the synthesis process.

To find useful additional outputs, the output-oriented program synthesis first generates a common pattern for all the outputs in E via anti-unification technique [Plotkin 1970] (line 1). Given a set of ASTs $\{o_0, \dots, o_n\}$, anti-unification process (denoted by $\bowtie \{o_0, \dots, o_n\}$) produces a pair $(\tau, <\sigma_0, \dots, \sigma_n>)$, where τ is a generalized AST with labelled holes $\{h_0, \dots, h_l\}$, and $\sigma_0, \dots, \sigma_n : \{h_0, \dots, h_l\} \mapsto$ `AST` are a set of substitutions, such that $\sigma_0(\tau) = o_0 \dots$ and $\sigma_n(\tau) = o_n$. The anti-unification process produces the most specific generalization τ of the given ASTs. For each additional output *ao*, output-oriented program synthesis then checks whether *ao* is an instance of τ (line 2) by searching for a substitution $\sigma = \{h_0 \mapsto subtree_0, \dots, h_l \mapsto subtree_l\}$, where $subtree_j \in$ `SubTrees`(ao) for $j \in 0 \dots l$, such that $\sigma(\tau) = ao$. If a substitution σ exists, *ao* is an instance of τ. Otherwise, *ao* is not an instance of τ, and it will be regarded as a useful additional output that will be utilized in the following synthesis steps.

Example 3.4. Let us revisit the two input-output examples shown in Example 3.2. Suppose we have two additional outputs AO_1: `this.inner.Handle (request, token)` and AO_2: `_handler.Handle (request, new CancellationToken())`. Anti-unifying the outputs of the two input-output examples `handler.Handle(request, token)` $\bowtie$ `TestSubject .Handle(request, token)` will generate (h_0`.Handle(request, token)`, <{$h_0 \mapsto$ `handler`}, {$h_0 \mapsto$ `TestSubject`}>). Because we can find a substitution σ={$h_0 \mapsto$ `this.inner`}, such that σ(h_0`.Handle(request, token)`)=AO_1, we will not regard AO_1 as an useful additional output. However, AO_2 is a useful additional output since we cannot find such a substitution. □

Additional input inference. To utilize the additional outputs (AO) in the synthesis process, our key idea is to infer a set of additional input-output examples using the provided additional outputs by analyzing its relation with E. Specifically, for each *ao* in AO, we infer a corresponding input *ai* that can be potentially transformed to the additional output *ao*. Transformation $ai \mapsto ao$ forms an additional example (line 17). To obtain this additional example, *ao* is first anti-unified with an output from E, (e.g., o_0) using *anti-unification modulo provenance* [Gao et al. 2020]. Provenance analysis calculates which fragments of the outputs are the same as which fragments of the inputs. Anti-unification modulo provenance of o_0 and *ao*, denoted as $o_0 \bowtie_\pi ao$, produces an generalization $(\tau, <\sigma_0, \sigma_1>)$ by just anti-unifying the provenance nodes. We do not go into the details of computing anti-unification modulo provenance, instead, we show it using an example.

Example 3.5. Consider input-output example `handler. Handle(request)` $\mapsto$ `handler.Handle (request, token)`, the provenance analysis would produce four nodes {`handler.Handle`, `handler`, `Handle`, `request`}, since these nodes in the output can be constructed using the nodes from input. Anti-unification modulo provenance of `handler.Handle(request, token)` and `_handler.Handle (request, new CancellationToken())` just unifies `handler` with `_handler`, since `handler` is a provenance node. However, `token` and `new CancellationToken()` will not be unified since `token` is not a provenance node. □

Anti-unification modulo provenance produces σ_0 and σ_1, representing the substitutions that are applied to o_0 and *ao*, respectively. Typically, $\sigma_0(\tau) = o_0$ and $\sigma_1(\tau) = ao$, for generating additional input, we apply the same substitution to the origin input $\sigma_1(\sigma_0^{-1}(i_0))$ (line 17). These additional examples can be used to expand the given input-output examples for the synthesis process. This allows our approach to be integrated with any existing program synthesis technique.

Example 3.6. Following Example 3.5, the produced substitutions are $\sigma_0 = \{h_0 \mapsto \texttt{handler}\}$ and $\sigma_1 = \{h_0 \mapsto \texttt{_handler}\}$. By applying $\sigma_1\sigma_0^{-1}$ to `handler.Handle(request)`, it would generate `_handler.Handle(request)`, which is the inferred input for the additional output. □

However, the correctness of the inferred additional inputs cannot be guaranteed. If the inference fails or the inferred additional input is incorrect, it would lead to incorrect transformation rules. To alleviate this problem, we give higher priority to human-provided examples than inferred additional examples. When observing conflict, e.g., two examples transform the same input in a different way, we drop the conflicted examples with lower priority.

Synthesis procedure. Given the input-output examples E and the inferred additional examples AE, output-oriented program synthesis then synthesizes a set of transformation rules. In the synthesis procedure, AE can be helpful in the following two aspects: 1) *guard generalization*: determine the proper context where the transformation rule should be applied; and 2) *transformation rule enhancement*: enhance the transformation rules by encoding more substantial transformation operations. The transformation operations from both E and AE may not be able to be unified in a single transformation rule as discussed in Section 3.3. Therefore, examples in $E \cup AE$ are first classified into clusters according to their transformation operations (line 4 ClusterEdit). Specifically, for each input-output example $i \mapsto o$, we calculate a set of edit operations $\{op_0, \ldots, op_n\}$ that can transform input i to output o using GumTree [Falleri et al. 2014]. Just as the edit script in GumTree, the edit operations include `Insertion`, `Deletion` and `Update`. The input-output examples with the same edit operations are grouped into the same cluster. The main intuition of the clustering is that the transformations from the same cluster can be represented using one single transformation rule. Our technique generates a transformation rule (R_{guard} and R_{trans}) for each cluster (according to all the edits in this cluster) using $\text{ReFazer}_{\texttt{guard}}$ and $\text{ReFazer}_{\texttt{trans}}$ (lines 7 - 8). The transformation rules of each cluster are combined together to form the complete set of transformation rules (line 9)).

The output-oriented program synthesis improves transformation rules by (1) relaxing constraints, and (2) generating more rules. First, our technique generates a more properly generalized guard by relaxing its constraints, since it takes into account the inputs of both E and AE. Second, it synthesizes more substantial transformation rules (one rule for each cluster) by considering the different transformations (AE) inferred from additional outputs.

Example 3.7. Consider the input-output example E: `handler.Handle(request)` $\mapsto$ `handler.Handle(request, token)` and additional example AE: `_handler.Handle(request)` $\mapsto$ `_handler.Handle(request, new CancellationToken())`. The edit operation of E is { INSERT("`token`") }, while the edit operation of AE is { INSERT("`new CancellationToken()`") }. Since the edit operations of E and AE are different, they will be categorized into different clusters. □

4 APIFIX: AUTOMATED API USAGE ADAPTATION

In this section, we present how output-oriented program synthesis is used to automate API usage adaptations. To achieve this, output-oriented program synthesis first synthesizes transformation rules using human-adapted examples (input-output examples) and the usages of the updated library (additional outputs), and then applies the synthesized transformation rules to all codes that require a transformation. Considering usages of the updated library in the synthesis process enables us to learn substantial API usage patterns of the new library.

Figure 5 depicts the architecture of output-oriented program synthesis for automating API usage adaptations. Given a library and its clients, APIFix first determines the breaking changes caused by library update. For each broken API, the *Miner* of APIFix mines relevant *human adaptations*, *new usages* and *old usages* from both library itself and client codes. Specifically, the human adaptations

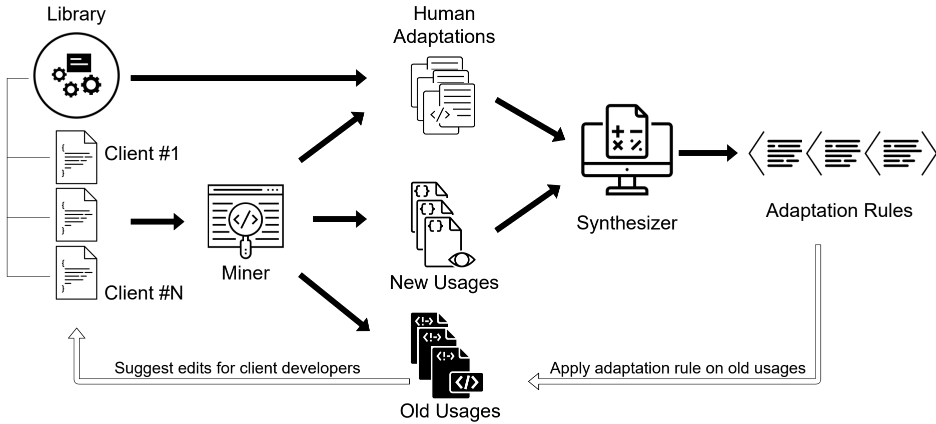

Fig. 5. APIFix: output-oriented program synthesis for automating API usage adaptations

are extracted from the library itself and the clients that have already been adapted to the new library version by client developers. Note that mining human adaptations from clients is time-consuming, it is an optional step in APIFix. The *new usages* and *old usages* are mined from clients, which represent the usages of the old and new library version, respectively. The output-oriented program synthesis takes as inputs the human adaptations and new usages, and synthesizes a set of transformation rules. APIFix then applies the synthesized transformation rules to transform the old usages and applies the transformed code back to clients for verifying syntactic correctness. The syntactically correct code edits are then sent to developers as edit suggestions. We will present the technical details of each step in the following.

4.1 Mining Human API Usage Adaptations and Library Usages

Given a library, the first step to automate API usage adaptations is to determine which public APIs have been modified in a library update that leads to *breaking changes*, mine the human adaptations of such API usages, and mine the usages of the new library version.

Change summary. Given a library X with two versions v_o and v_n, representing the old and new library version, we first construct a *change summary* by comparing v_o and v_n. Basically, change summary models the set of modified APIs introduced by the X's update from v_o to v_n, including the modified identifier names, modified modifiers, modified function/constructor arguments, inserted/deleted public classes/methods, and so on. The workflow to generate change summary is as follows:

- For each file f_n in X's version v_n, we use the Git version control system to determine the corresponding file f_o in X's version v_o;
- If f_n is different from f_o, then, for each API api_n in f_n, we use clone detection to determine its matched api_o in f_o that is most similar to api_n;
- If the signatures of matched $<api_n, api_o>$ are different, we will save the signatures of $<api_n, api_o>$ into the change summary.

Human adaptations. Given the change summary of library X's update $v_o \rightarrow v_n$, we then mine the human adaptations that adapt codes relying on v_o to the new library version v_n. The human adaptations can be mined from two sources: the library X itself and the clients that depend on library X. When the developer of X introduces a breaking change on an API, he or she also needs

to adapt the usages of the relevant API within X (e.g. tests of API). The human adaptations within X itself are great sources to learn transformation rules because (1) those adaptations are available immediately after the new library version is released; and (2) the developers of library X know best about the breaking changes.

Besides X itself, human adaptations can also be mined from clients of X that have already upgraded X from v_o to v_n. To do that, we first search for the commit that upgraded X, among the commit history. Searching for the exact commit that performed the migration is not efficient since it requires traversing through all the commit history. Second, we extract human adaptations by comparing the client codes before and after the X-upgrading commit. Specifically, for a modified API $<api_n, api_o>$, we mine its corresponding human adaptations by searching for the usages of api_o in the client code before upgrading X, and the corresponding usages of api_n in the client code after upgrading X. The usages of api_n and api_o form human adaptations. Mining human adaptations from clients is less efficient, because it requires traversing through all the commit history of all the client projects. Existing approaches, e.g., AppEvolve [Haryono et al. 2020], propose to collect human adaptations from clients overnight, but this can significantly affect the efficiency.

API usages. Similar to finding human adaptations for the APIs, the usages of the API are also mined from its open-source clients. Instead of traversing through all the commits, mining API usages just requires checking the latest version of clients. For a clients that relying on v_n, we mine the usages of api_n to form *new usages*, while for a clients that relying on v_o, we create *old usages* by mining the usages of api_o.

Mining useful human adaptations and API usages is based on the assumption that the developer-provided codes are valid. However, in practice, the correctness and quality of developer-provided codes cannot be guaranteed. Bad human adaptations and additional outputs can be misleading and hence lead to incorrect transformation rules. To alleviate this problem, we just consider the well-maintained and well-tested client projects, e.g., Github repositories with more stars and more dependents. In general, we can assume a competent developer made changes now merged to the stable/release branches.

4.2 Clustering Algorithm

Given a set of human adaptations E, a set of new usages NU and old usages OU of a certain API, Algorithm 2 depicts the workflow of automated API usage adaptations. First, since there could be multiple adaptation strategies for a certain broken API, APIFIX categorizes the human adaptations into different clusters based on the adaptation operations (line 1). The motivation behind this categorization is that the edits in the same cluster should have the same adaptation strategies which can be represented by a single transformation rule. We use the same clustering algorithm (i.e. `ClusterEdit`) used in Section 3.4. For each cluster, APIFIX then determines the new usages that are `relevant` to the edits in this cluster according to their similarity (line 6).

Definition 4.1 (Relevance)**.** We define a new usage as relevant to a cluster if the new usage is similar, in terms of code structure, to the output of the edits in this cluster. The code structure includes API name, number of arguments, type of arguments, return type, and etc.

We determine the relevant new usages for each cluster by calculating the AST tree distance between the new usage and the output of the edits (lines 15-16). However, the relevance is defined in terms of high-level code structure, so the code details (e.g., identifier name) may significantly affect the distance. Therefore, instead of calculating the tree distance of the two concrete ASTs, APIFIX first abstracts the code details because directly comparing the concrete ASTs will result in false negatives. Specifically, APIFIX abstracts the ASTs using the anti-unification modulo provenance [Gao et al.

Algorithm 2: APIFix: Automated API Usage Adaptation

```
   Input: Human adaptation: E;  Old Usages: OU;  New Usages: NU;
          Similarity Thresholds: T1, T2
   Output: Transformed Old Usages: TOU
 1 EditClusters := ClusterEdit(E) ;
 2 TOU := { };
 3 for editCluster ∈ EditClusters do
 4 |   i0 ↦ o0 := GetFirst(editCluster) ;
 5 |   π := Provenance(i0 ↦ o0);
 6 |   relevantNU := {nu | nu ∈ NU ∧ Distance(o0, nu, π) < T1};
 7 |   relevantOU := {ou | ou ∈ OU ∧ Distance(i0, ou, π) < T2} ;
 8 |   rules := synthesiser(editClsuter, relevantNU);
 9 |   for ou ∈ relevantOU do
10 |   |   TOU := TOU ∪ {ou ↦ t | t ∈ rules(ou)};
11 |   end
12 end
13 return TOU;

14 Function Distance(t1, t2, π):
15 |   (τ, <σ1, σ2>) := t1 ⋈π t2;
16 |   return TreeDistance(σ1^-1(t1), σ2^-1(t2));
```

2020] and then calculates the distance between the abstracted ASTs. Here is an example of how to calculate AST distance with anti-unification modulo provenance.

Example 4.1. Consider the following example that adds a new argument `token` to the function call, `handler.Handle(request)` ↦ `handler.Handle(request, token)` and a relevant new usage `handler.Handle(new Request{Value = pValue}, token)`, the tree distance between the new usage with output of the given example is large because `request` and `new Request{Value = pValue}` are quite different. However, the anti-unification module provenance tells us that `request` is a relevant part of input since it also appears in the output. Therefore, `request` and `new Request{Value = pValue}` can be abstracted because we just care about the high-level API usage patterns. By comparing the abstracted nodes, APIFix will determine this new usage is relevant to this cluster. □

Similarly, we also determine the relevant old usages according to their similarity with the inputs of the edits in this cluster. These old usages are the codes that should be transformed by the transformation rules learned from this cluster.

Remark (Relevance vs Usefulness)**.** To find meaningful additional outputs, we need to balance relevance and usefulness (See Definition 3.1). The additional outputs must be *similar* (relevance) in terms of structure with the human adaptations. Meanwhile, they must be *not identical* (usefulness) in terms of the nodes to ensure they are helpful in refining transformation rules.

4.3 Synthesizing and Applying Transformation Rule

Given the edits in a cluster and the corresponding relevant new usages, we invoke the output-oriented program synthesis to produce transformations (line 8), i.e., *transformation rules* in the context of API usage adaptation. The synthesized transformation rules are then applied to transform the relevant old usages. Recall that, APIFix synthesizes a set of transformations (rules)

$\{(\mathtt{guard}_0, \mathtt{trans}_0), \ldots, (\mathtt{guard}_n, \mathtt{trans}_n)\}$. When applying the rules to an old usage ou, if there exist multiple guards such that $\mathtt{guard}(ou) = \mathtt{true}$, we could generate multiple $\mathtt{trans}(ou)$. In this situation, we will try to apply each $\mathtt{trans}(ou)$ to the client code and check whether it causes compilation errors. We save all transformations that do not cause a compilation error.

5 EVALUATION

In this section, we evaluate the output-oriented program synthesis and answer the following research questions:

RQ1 What is the effectiveness of output-oriented program synthesis in generating correct code transformations?
We split the mined human adaptations into training and testing sets. We evaluate the effectiveness of output-oriented program synthesis via cross-validation by measuring the syntactic and semantic equivalence between auto-transformed codes with the human adaptations.

RQ2 How does APIFIX perform in automating API usage adaptations?
APIFIX should generate effective suggestions for API usage adaptation that can be used by developers. We measure the number of false positives and false negatives generated by APIFIX.

RQ3 How does APIFIX compare with the state-of-the-art techniques?
Our main contribution is to enable the program synthesis system to utilize the additional output. We measure the output-oriented program synthesis by comparing it with the general program synthesis REFAZER and semi-supervised program synthesis.

Implementation. APIFIX is implemented in Python and C#, and it is composed of three main components: `Miner`, `Build Engine` and `Synthesis Engine`. The Miner, which is implemented using GitHub APIs, is used to mine GitHub repositories to find breaking changes, existing human adaptations, and library usages. The Build Engine is used initially to build typed ASTs and finally to validate the transformed codes. We implemented the Build Engine on top of Microsoft MSBuild [msb 2021], and used Roslyn framework [ros 2021] to parse source files and generated ASTs. The Synthesis Engine is implemented on top of an extended REFAZER [ref 2020] (it is extended to support our DSL).

Dataset. To evaluate our output-oriented program synthesis, we build our dataset by mining from GitHub repositories. Our miner searches for the "Most starred" C# libraries and selects libraries to construct our dataset using the following criteria:

- The library has at least 300 dependents reported in the statistics of the GitHub Dependency graph [dep 2021];
- The library has multiple released versions, and there is at least one breaking change;
- There are available human adaptations that adapt library/clients to the new library version;
- There are new usages and old usages of the broken APIs.

Finally, we select seven libraries with 138,206 clients. From those libraries/clients, we mined 218 human adaptations, 2973 new usages and 2154 old usages following the procedure described in Section 4.1. The detailed statistics of the selected dataset are shown in Table 1. Column "`#Clients`" presents the number of clients for each library. For each library, column "`API Name`" gives the modified APIs that are broken by the library update from "`Old version`" to "`New version`". As we mentioned in Section 4.1, the human adaptations can be mined from the library itself and its clients. Column "$\mathtt{\#Edit}_l$" and "$\mathtt{\#Edit}_c$" show the number of human adaptations mined from library itself and clients, respectively. The last two columns present the number of new usages and old usages, respectively. For simplicity, the number of new/old usages is bounded to 1,000.

Table 1. Statistics on our dataset used for evaluation

Library	#Clients	Old version	New version	API name	#Edit$_l$	#Edit$_c$	#New usages	#Old usages
Polly	8531	5.5.0	6.1.2	Execute	61	11	3	13
		6.1.2	7.0.0	WrapAsync	3	3	19	18
		6.1.2	7.0.0	WaitAndRetryAsync	3	5	30	195
MediatR	14099	5.0.1	6.0.0	Handle	1	6	721	56
		6.0.0	7.0.0	Process	1	0	18	33
DbUp	1819	3.3.5	4.0.0	StoreExecutedScript	2	0	28	147
		3.3.5	4.0.0	SqlScriptExecutor	6	0	36	84
		3.3.5	4.0.0	AdHocSqlRunner	2	0	4	28
SteamKit	332	2.0	2.1	Disconnect	2	0	1	38
AutoMapper	89151	7.0.0	8.0.0	Ignore	2	30	38	220
		7.0.0	8.0.0	ResolveUsing	16	42	1000	271
FluentValidation	20568	8.0.0	9.0.0	Validate	9	6	1000	1000
MimeKit	3716	1.22.0	2.0.0	DecodeTo	1	6	75	51
Total	138,206	-	-	-	109	109	2973	2154

Table 2. The type of breaking changes

Type	change argument type	change return type	insert/delete new argument(s)	changes containing class	rename	Total
#APIs	3	2	6	1	2	14*

* there is one breaking change that changes both argument type and return type.

In our evaluation, we just use human adaptations from the library itself (`#Edit`$_l$) to synthesize transformation rules because (1) mining human adaptations from client projects is less efficient, which requires to traverse each commit of each client; (2) the adaptations from the library become available once the new library version is released, while mining clients usually needs to wait for developers until they adapt their client projects to the new library version. The cost of existing approaches, e.g., AppEvolve [Haryono et al. 2020], that rely on adaptations from both library and clients, is dominated by the search for examples. Those approaches propose to collect examples overnight, but this can significantly affect the efficiency. In practice, just relying on `#Edit`$_l$ enables APIFix to synthesize transformation rules efficiently.

The evaluated API changes cover various change types and table 2 shows their distribution. Typically, among the 13 different APIs, three of them change argument type, two change return type, six insert/delete new argument(s), one changes their containing class, and two rename the APIs. Note that, there is one API change that modifies both argument type and return type.

In our experiment, we set the threshold T_1 and T_2 in Algorithm 2 as 0.25 and 0.15, respectively. All experiments are conducted on a Dell Precision Tower 7810 with Intel(R) Xeon(R) CPU E5-2630 processor and 32GB RAM running 64-bit Windows 10.

5.1 Exp-1: Effectiveness of Output-Oriented Program Synthesis

We first evaluate the effectiveness of our output-oriented program synthesis using a cross-validation experiment on our dataset. We use the human adaptations for each breaking change collected in

Table 3. Exp-1: Cross-Validation Results of Output-Oriented Program Synthesis

Library	API name	#Instances	Syntactic	Semantic	Accuracy
Polly	Execute	11	0	9	81%
	WrapAsync	3	1	3	100%
	WaitAndRetryAsync	5	2	3	60%
MediatR	Handle	6	0	6	100%
AutoMapper	Ignore	30	30	30	100%
	ResolveUsing	42	42	42	100%
FluentValidation	Validate	6	6	6	100%
MimeKit	DecodeTo	6	0	0	0%
Total	-	109	81	99	91%

our data-set to measure the accuracy of the automatically transformed codes by comparing them to the developer transformed codes.

Experimental Setup. For each subject, we take human adaptations from the library itself (column "$\texttt{Edit}_l$") as the training set to synthesize the transformation rules, and human adaptations from clients (column "$\texttt{Edit}_c$") as testing set to validate the correctness of the synthesized transformation rules. Specifically, output-oriented program synthesis generates transformation rules by taking the human adaptations from the library as input-output examples and the new usages (column "`#New usages`") as additional outputs. The synthesized transformation rules are then evaluated on the human adaptations from clients. We measure the correctness of automatically transformed codes by manually checking their syntactic and semantic equivalence with human-adapted codes, i.e., the ground truth. To check semantic equivalence, we first apply the transformation rule to clients. We then manually compare the transformed code with the human-adapted code by analyzing the semantics of the modified code and its surrounding context (usually the function where the transformation is applied). As observed in our evaluation, we could easily figure out the semantically equivalent codes for most of the cases. For instance, if the automatically transformed code splits one human-generated statement into two separate statements by introducing a temporary variable, we can figure out they are semantically equivalent. In case of the potential bias caused by manual analysis, two authors of this paper double-checked their semantic equivalence and reached an agreement. We use Inter-Rater Reliability (IRR) [3] to measure our agreement. Inter-rater reliability presents the level of agreement between raters or judges. Cohen's kappa statistic [4] is a widely-used measure of inter-rater reliability. It determines inter-rater reliability by considering the percent agreement and chance agreement. The Kappa statistic varies from 0 to 1, where 0 means agreement equivalent to chance and 1 presents perfect agreement.

Experimental Results. The results of our experiment are summarized in Table 3. Columns "Library" and "API name" indicate the names of the library and API name that introduce breaking changes, respectively. Column "#Instances" indicate the number of transformations for each API for which the ground truth is available. Columns "Syntactic" and "Semantic" represents the number of transformations for which the result is syntactically and semantically equivalent to the ground truth, respectively. Column "Accuracy" shows the percentage of correct (syntactically or semantically) transformations with respect to the total number of instances for each breaking change.

[3]https://www.statisticshowto.com/inter-rater-reliability
[4]https://www.statisticshowto.com/cohens-kappa-statistic/

Table 4. Performance on different types of breaking changes

Type	change argument type	change return type	insert/delete new argument(s)	changes containing class	rename
Success/Total	18/20	6/8	6/6	0/6	72/72

In total, out of 109 instances, 81 transformed codes by output-oriented program synthesis are syntactically equivalent to human-adapted codes, while 99 of them are semantically equivalent. If a transformed code is syntactically different from the developer patch, two authors of this paper manually analyze their semantic equivalence. In this case, we analyzed 28 pairs, and find 18 of them are semantically equivalent. Initially, out of 28 manually analyzed cases, one of the authors rates that 18 are semantically equivalent and 10 are not equivalent, while the other author rates that 16 are equivalent and 12 are not. Specifically, 16 cases are rated as equivalent by both authors and 10 cases are rated as not equivalent by both. The calculated Kappa statistic is 0.85, meaning that we achieved near-perfect agreement. Later, we discussed the disagreement cases, analyzed the codes again, and achieved perfect agreement eventually.

Overall, Output-oriented program synthesis achieves 91% overall accuracy in correctly transforming old usages of the APIs. In our evaluation, we noticed that there can be multiple possible ways to transform an old usage. For instance, the human adaptation transformed the API invocation `requestHandler.Handle(request)` to two statements `var token = new CancellationToken(); requestHandler.Handle(request, token)` [5]. In contrast, our technique transforms this statement to `requestHandler.Handle(request, new CancellationToken())`, which is syntactically different, but semantically equivalent to the human adapted code. On the other hand, our technique generates 10 false negatives. The main reason is that the synthesized transformation rule is over-specialized to the given examples and additional outputs. Furthermore, we notice that APIfix performs differently on different types of API changes (Table 4). It performs well on `API rename` and `insert/delete new argument(s)` since the transformation rules for such API changes are usually simple. However, APIfix shows relatively worse performance on the API changes that modify types due to the complex language features, e.g., inheritance and polymorphism. Here is a failure example.

Example 5.1. Consider a synthesized transformation rule from given human adaptation and new usages is simplified as follows:

$$\texttt{Policy}\ X_1 = X_2.\texttt{WaitAndRetryAsync}(\ldots) \mapsto \texttt{AsyncPolicy}\ X_1 = X_2.\texttt{WaitAndRetryAsync}(\ldots)$$

This rule fails to transform `RetryPolicy retry = polly.WaitAndRetryAsync(...)`, which should be transformed to `AsyncRetryPolicy retry = polly.WaitAndRetryAsync(...)`. The reason is that the synthesized rule is not general enough to be applied to this case and hence produces a false negative. □

RQ1: In a cross-validation experiment, the output-oriented program synthesis achieves an overall **91%** accuracy, indicating its effectiveness in synthesizing correct transformation rules.

[5] https://github.com/transformania/tt-game/commit/a58e410

Table 5. Evaluation results of APIFIX with different synthesis techniques. APIFIX represents our tool with Output-Oriented Program Synthesis. APIFIX^R and APIFIX^S represent APIFIX with REFAZER and Semi-Supervised Program Synthesis as the synthesis engine, respectively. APIFIX^{O+S} uses a combined Semi-Supervised and Output-Oriented Program Synthesis as the synthesis engine.

API	APIFIX			APIFIX^R			APIFIX^S			APIFIX^{O+S}		
	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP
Execute	8	0	0	7	1	0	8	0	5	8	0	0
WrapAsync	2	2	0	4	0	0	4	0	2	2	2	0
WaitAndRetryAsync	2	2	0	2	2	0	4	0	0	4	0	0
StoreExecutedScript	120	24	3	24	123	0	123	0	24	120	24	3
SqlScriptExecutor	84	0	0	40	28	16	56	8	20	84	0	0
AdHocSqlRunner	28	0	0	14	14	0	28	0	0	28	0	0
Disconnect	38	0	0	12	0	26	12	0	26	38	0	0
Ignore	220	0	0	220	0	0	220	0	0	220	0	0
ResolveUsing	271	0	0	271	0	0	271	0	0	271	0	0
Handle	54	0	2	0	0	56	35	0	21	54	0	2
Process	14	0	0	3	11	0	3	11	19	14	0	0
Validate	7	0	6	0	13	0	13	0	8	7	0	6
DecodeTo	0	51	0	0	51	0	0	0	51	0	0	51
Total	848	79	11	597	243	98	777	19	176	850	26	62

5.2 Exp-2: Effectiveness in Automating API Usage Adaptations

We apply our output-oriented program synthesis to automatically adapt transformations and generate patches to assist client developers to upgrade their library usages from the old version to the new version. We evaluate the effectiveness of APIFIX in automating these usage adaptations and generating valid transformations which can be directly applied by the client developers.

Experimental Setup. In this experiment, we take the human adaptations from the library itself (“$\texttt{Edit}_l$”) as input-output examples and new usages as additional outputs. We only use “$\texttt{Edit}_l$” as our input-output examples because mining human adaptations from clients can be a time-consuming task in practice. Mining human adaptations from the library are much faster than from clients, which enables APIFIX to quickly synthesize transformation rules and generate code edit suggestions. The synthesized transformation rules are then used to transform the old usages. We evaluate the correctness of the transformed codes as follows:

- Upgrade the dependency version for each client which should cause compilation error(s);
- Apply the transformed code to the client;
- Check whether the transformed code fixes the compilation error(s) caused by the target broken APIs.

We only evaluate the syntactic correctness of the transformed codes because the correct code transformation (i.e., ground truth) is not available for the clients that still rely on the old version of the library. Checking the semantic correctness of the transformed codes can be left for the developers. Recall that APIFIX is designed to *assist developers* (e.g. providing code edit suggestions) instead of replacing developers.

Experimental Results. In Table 5, columns APIFIX summarize the evaluation results, where columns TP, FN and FP represent the number of true positives, false negatives and false positives,

respectively. True positive represents that APIFix correctly transforms old usage, false negative means that APIFix fails to do a transformation, while false positive means that APIFix incorrectly transforms codes or transforms codes that should not be transformed. Although we apply the synthesized transformation rule to all the old usages, not all of them need to be transformed. For instance, the breaking change of `WaitAndRetryAsync` is changing its return type from `Policy` to `AsyncPolicy`. The old usage `var policy = polly.WaitAndRetryAsync(...)` does not need to be transformed, since the implicit "type" `var` allows compiler determines its real type at compilation time. That is, `var` allows the type `polly` to be `Policy` at old version, and to be `AsyncPolicy` at the new version, hence this statement does not need to be transformed. Among the old usages requiring transformation, APIFix produces 848 true positives, 79 false negatives, and only 11 false positives, achieving 98.7% precision and 91.5% recall. Similar to the result shown in Exp-1, APIFix produces false negatives since the synthesized transformation rule is over-specialized to the given data. For instance, `DecodeTo` has 51 false negatives because this breaking change modified its containing class (`Content` → `ContentObject`). The synthesized rule can only transform the usage of this API when it is used with the simple name (`ContentObject`), but it cannot be applied to transform API usages with canonical name (`package_name.ContentObject`). On the other hand, it produces false positives mainly because it transforms some codes that should not be transformed.

To evaluate the effectiveness of APIFix in helping developers, we randomly select 20 actively maintained clients and submit the transformed codes by sending pull requests. We did not send too many pull requests in case of creating noises for client developers. Until the time of this paper's writing, we have received four confirmations and one rejection. One developer rejected our pull request since our edit failed to update the dependency code and incorrectly modifies the program behaviors.

> **RQ2:** By learning from human adaptation and new usages, APIFix automatically adapts 848 old usages with **98.7%** precision and **91.5%** recall.

5.3 Exp-3: Comparison with State-of-The-Art Techniques

We provide an empirical evaluation for APIFix by comparing it with the recently proposed program transformation techniques. Specifically, we consider three different comparable synthesis techniques: original ReFazer (APIFix^R), semi-supervised program synthesis (APIFix^S) and a combination of semi-supervised and output-oriented program synthesis (APIFix^{O+S}). We provide the same mining procedure to all the considered approaches to make a fair comparison, and we only replace the synthesis technique.

Experimental Setup. Given input-output examples E, additional input AI and additional output AO, APIFix^R synthesizes transformation rule using E, APIFix^S utilizes both E and AI, APIFix uses E and AO, and APIFix^{O+S} uses all E, AI, and AO. Specifically, APIFix^{O+S} combines both semi-supervised and output-oriented program synthesis by constructing additional examples via (1) AE_1: inferring corresponding outputs for additional inputs (SEMI-SUPERVISED SYNTHESIS), and (2) AE_2: inferring corresponding inputs for additional outputs (output-oriented program synthesis). If any inferred additional example in AE_1 and AE_2 conflict with each other, we simply drop the conflicted examples in AE_1. Consider the following example from Section 2, for a certain input, the inferred examples from AE_1 and AE_2 transform it in different ways.

```
new SqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, )↦
    AE1: newSqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, ()=>Substitute.For<IJournal>())
    AE2: newSqlScriptExecutor(()=>c.ConnectionManager, _, _, _, _, ()=>c.IJournal)
```

Table 6. The precision and recall of APIFIX, APIFIX^R, APIFIX^S and APIFIX^{O+S} in transforming old usages

Approach	APIFIX	APIFIX^R	APIFIX^S	APIFIX^{O+S}
Precision	98.7%	85.9%	81.5%	93.2%
Recall	91.5%	71.0%	97.6%	97.0%

In this scenario, we drop the example from AE_1 that causes the conflict because the outputs of AE_2 are produced by developers which should be given higher confidence. For this experiment, we use the same setting as Exp-2 described in Section 5.2. We only evaluate the syntactic correctness of the transformed codes because the correct code transformation (i.e., ground truth) is not available for the clients that still rely on the old version of the library. We also evaluate the number of true positives, false positives, and false negatives of each approach. Note that, the sum of `TP`, `FN` and *FP* might be different across those four approaches since APIFIX^R transforms some codes that should not be transformed, hence producing more false positives.

Experimental Results. The summarized results of our comparison with the state-of-the-art techniques are also shown in Table 5. Columns `TP`, `FN` and `FP` represent the number of true positives, false negatives and false positives, respectively for each tool. Table 6 presents the precision and recall of each tool. Compared with APIFIX^R, APIFIX significantly reduces both numbers in false negatives and false positives. This is because considering additional output enables APIFIX to synthesize more accurately generalized transformation rules. Compared with APIFIX^S, APIFIX performs better with much fewer false positives, while incurring a little bit more false negatives. APIFIX^S produces fewer false negatives because additional inputs enable it to synthesize a more generalized transformation rule. However, an over-generalized rule causes more false positives. In practice, we argue that false positive is more harmful since it causes developers to lose trust in the tool.

Example 5.2. Let us revisit the false negative in Example 5.1 produced by our approach again. If the input `RetryPolicy retry = polly.WaitAndRetryAsync(...)` is regarded as additional input that should be transformed, the transformation rule will be further generalized as

$$\mathtt{X}_3\ X_1 = X_2.\mathtt{WaitAndRetryAsync(...)} \mapsto \mathtt{AsyncPolicy}\ X_1 = X_2.\mathtt{WaitAndRetryAsync(...)}$$

where X_3 represents an identifier which can be matched with `RetryPolicy`. With the generalized rule, the target input is transformed to `AsyncPolicy retry = polly.WaitAndRetryAsync(...)`. Although the transformed code is syntactically different from human adaptation, since `AsyncPolicy` is the parent class of `AsyncRetryPolicy`, they are semantically equivalent. □

However, if APIFIX^S over-generalizes the synthesized rule to the inputs that should not be transformed, it will lead to false positives. Example 3.1 presents such a false positive. Balancing false positives and false negatives is challenging.

Furthermore, we propose to combine both semi-supervised and output-oriented program synthesis, i.e., APIFIX^{O+S}, to utilize both additional inputs and additional outputs. Experimental results show that combining additional input and additional output can further improve the number of true positives. However, the improvement is not significant. Our results are based on a straightforward and simple combination of semi-supervised and output-oriented program synthesis. We believe other possible combinations may further improve the results. How to better combine them to infer better transformation rules can be an interesting question in future work.

RQ3: APIFix improves both precision and recall over APIFix^R. Compared with semi-supervised program synthesis APIFix^S, our technique significantly improves the precision, while does not significantly affect the recall.

5.4 Threats to Validity

Three threats may affect the validity of our empirical evaluation. First, in our experiments Exp-2 (Section 5.2) and Exp-3 (Section 5.3), we evaluated the correctness of transformed code using the compiler. The compiler can ensure syntactic correctness, but it cannot check semantic errors. To solve this problem, we performed a cross-validation in Exp-1(Section 3). The evaluation results in Exp-1 and Exp-2 are consistent. Second, although APIFix shows its effectiveness on the evaluated benchmark, it may not perform well on other subjects. To mitigate this problem, we selected a fairly large dataset with more than 2000 cases that cover different scenarios. Last, we manually compare automatically transformed code with human adaptations to verify their correctness. In case of the potential bias caused by manual analysis, two authors of this paper independently checked the correctness of the transformed code.

Limitation. One of the limitations of APIFix is the lack of context analysis when applying the transformation rules to client codes. Lacking context analysis may result in two problems: (1) using incorrect identifiers in the transformed codes, and (2) missing updating the dependent statements of the changed API usage. First, when applying the transformation rule to code context `C`, the transformed codes may use identifiers that are not available at `C`, which can cause compilation errors. To solve this problem, we heuristically fix the incorrect identifiers by searching for a correct one at `C` according to identifiers' type. For instance, if a transformation requires parameter `foo.getSize()` and the type of `foo` is `Foo`, we replace foo by searching for a variable with type `Foo` among the live variables at `C`. This heuristic fixes many compilation errors, although the fix may be incorrect. Better adaptation requires more fine-grained context analysis. Second, when the usage of an API is updated, its dependent statements may also need to be modified accordingly. Whether an update affects its dependent statements depends on the type of the transformation. In our evaluation, we noticed that if a transformation modifies the return type of an API, its dependent statements usually need to be updated. To support such systematic changes, it is necessary to perform context analysis, which is out of the scope of this paper. Existing API usage adaptation techniques, such as LibSync [Nguyen et al. 2010], have investigated this problem. APIFix can be potentially integrated with the program dependency analysis employed by those techniques in the future.

6 RELATED WORK

In this section, we compare our approach with the relevant techniques, including general program synthesis, automated program transformation, semi-supervised program synthesis, and API usage adaptation techniques.

6.1 Program Synthesis

Program synthesis is a technique to automatically generate programs according to given input-output examples. Program synthesis has been applied in many domains, such as string manipulation [Gulwani 2011; Singh 2016], data structure transformation [Feng et al. 2017; Singh and Solar-Lezama 2011], concurrent programming [Černý et al. 2011] and so on. To synthesize programs efficiently and effectively, syntax-guided synthesis [Alur et al. 2013] (SyGuS) specifies both syntax and semantics of the desired program and constructs programs using the pre-defined syntax and

semantics with reference to a set of given input-output examples. Sketching [Solar-Lezama et al. 2008, 2005] allows user to express their desire about target implementations as a partial program with holes and uses program synthesis to fill the holes. They split the synthesis task into multiple sub-tasks and solve the programming-by-example sub-tasks separately. The general-purpose synthesis techniques have shown great performances in many domains, but they require well-defined specifications in the form of input-output examples. In contrast, output-oriented program synthesis is domain-specific, and it is designed to construct programs using a pre-defined domain-specific language. Further, compared with those techniques relying on input-output examples, output-oriented program synthesis also considers additional outputs in constructing programs, which gives us more confidence about the synthesized programs.

6.2 Program Transformation

Automated program transformation techniques infer abstracted transformation rules from human-generated transformations and apply the inferred rules to transform the codes in other codebases. Program transformation techniques have been applied in many domains, such as fixing software bug [Bader et al. 2019; Bavishi et al. 2019; Long et al. 2017], automating repetitive edits [Meng et al. 2011, 2013; Rolim et al. 2017], and intelligent code refactoring [Gao et al. 2020; Miltner et al. 2019]. Similar to our approach, these techniques also learn transformation rules from concrete human transformations. The main difference between our approach with these techniques lies in that our output-oriented program synthesis also considers the human intelligence embedded in the additional outputs. This feature reduces the dependency on human-generated transformations and enables us to learn more substantial transformation rules.

6.3 Semi-Supervised Program Synthesis

Semi-supervised synthesis techniques [Gao et al. 2020; Singh 2016] learn models from both labeled data (i.e., input-output examples) and unlabeled data (additional inputs). More specifically, semi-supervised synthesis learns program transformation rules from input-output examples and additional inputs that should be transformed; therefore, it can better understand the structure of input space by referring to additional inputs. Similar to the semi-supervised synthesis, APIfix also synthesizes transformation rules not only from input-output examples but also from the additional outputs. However, unlike these approaches, we assume that additional outputs instead of additional inputs are available; therefore, our goal is to learn the intelligence embedded in additional outputs, e.g., API usage patterns. At the same time, our approach in this paper is complementary and compatible with the techniques employed by the semi-supervised synthesis: the additional input helps to understand the structure of the input space, and additional outputs help to mine the embedded human intelligence. Our simple combination strategies do not work well, how to better combine those two approaches may be an interesting research question to explore.

6.4 API Usage Adaptation

Existing program transformation techniques have been studied to be applied to automatically update the dependencies of clients [Dagenais and Robillard 2011; Fazzini et al. 2019; Haryono et al. 2020; Nguyen et al. 2010; Xu et al. 2019]. These techniques first infer adaptation rules from the before- and after-adaptation examples from human-adapted clients, and then apply the inferred rules to adapt clients that are relying on outdated libraries. The main difference between these techniques and APIfix is that APIfix synthesized adaptation rules from both input-output examples and additional outputs. Therefore, APIfix can achieve good performance with fewer human adaptations, while these techniques require a large number of human adaptations to synthesize a proper adaptation rule. Even though CocciEvolve [Haryono et al. 2020] learns from a single adaptation example, it

can only adapt Android deprecated-API usages by introducing an if-condition. Furthermore, one of the main focuses of these techniques is to perform program dependency analysis to extract code skeletons before and after API usages, e.g., find the dependent statements of the API usage via data dependency analysis. Differently, the focus of APIFix is the output-oriented program synthesis by just considering the API usages without dependent statements. APIFix is complementary and compatible with the dependency analysis techniques employed by these existing approaches. APIFix can be potentially combined with them to synthesize more complete adaptation rules.

Apart from the inductive adaptation rule inferences, researchers have also studied approaches that are integrated into the development environment with the aim of automatically adapting API usages. For instance, CatchUp [Henkel and Diwan 2005] records the refactoring actions when developers evolve an API, and replays the recorded refactoring actions in the client codes. SemDiff [Dagenais and Robillard 2009] analyzes how a library was adapted to its own changes and then provides adaptation suggestions to client programs. These approaches require that the library and clients are developed in the same development environment. In contrast, the adaptation rule synthesized by APIFix can be applied to any development environment and automatically adapt any client requiring adaptations.

7 CONCLUSION

Modern software systems heavily depend on third-party libraries. The breaking changes of API caused by library updates can break the client applications. We presented an output-oriented program synthesis technique to automatically adapt the client applications to let them use the new version of libraries. Compared with existing program synthesis techniques, output-oriented program synthesis infers transformation rules based on human adaptations (i.e. input-output examples) as well as the usages of the new version of libraries (i.e. additional outputs). The additional outputs can be helpful in finding the correct level of generalization of transformation rules and synthesizing new transformation rules. Our evaluation shows that output-oriented program synthesis achieves 91% accuracy in correctly transforming codes in cross-validation experiments. When compared with state-of-the-art synthesis approaches, our technique improves both precision and recall over ReFazer [Rolim et al. 2017]. Furthermore, in comparison to semi-supervised synthesis [Gao et al. 2020], our technique significantly improves precision, while does not affect the recall too much.

Once the API usage adaptation is generated, the next task is to send the adaptation suggestions to developers for verification and application. The suggestions can be provided at the Integrated Development Environment (IDE) or they can be sent to developers by generating pull requests in Git, which we will investigate in future work. Moreover, how to efficiently and effectively combine the additional inputs and additional outputs in program synthesis, can be an interesting research question to explore in the future. Such a research question can help us gain further insights on reducing *over-fitting* (to given input-output examples) in program synthesis.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed comments and suggestions.

REFERENCES

2020. Refazer: Program Synthesis Tool. https://www.nuget.org/packages/Microsoft.ProgramSynthesis.

2021. Github Dependency Graph. https://docs.github.com/en/code-security/supply-chain-security/about-the-dependency-graph.

2021. Micrsoft MSBuild. https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild-api.

2021. Roslyn Framework. https://docs.microsoft.com/en-us/visualstudio/code-quality/roslyn-analyzers-overview.

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis.* IEEE.

Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 613–624.

Pavol Černỳ, Krishnendu Chatterjee, Thomas A Henzinger, Arjun Radhakrishna, and Rohit Singh. 2011. Quantitative synthesis for concurrent programs. In *International Conference on Computer Aided Verification*. Springer, 243–259.

Barthelemy Dagenais and Martin P Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 599–602.

Barthélémy Dagenais and Martin P Robillard. 2011. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 1–35.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated software engineering*. 313–324.

Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215.

Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

Xiang Gao, Shraddha Barke, Arjun Radhakrishna, Gustavo Soares, Sumit Gulwani, Alan Leung, Nachiappan Nagappan, and Ashish Tiwari. 2020. Feedback-driven semi-supervised synthesis of program transformations. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

Stefanus A Haryono, Ferdian Thung, Hong Jin Kang, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automatic Android deprecated-API usage update by learning from single updated example. In *Proceedings of the 28th International Conference on Program Comprehension*. 401–405.

Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. 274–283.

Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 329–342.

Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 502–511.

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *PACMPL* 3, OOPSLA (2019), 1–29.

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. 2010. A graph-based approach to API usage adaptation. *ACM Sigplan Notices* 45, 10 (2010), 302–321.

Gordon D Plotkin. 1970. A note on inductive generalization. *Machine intelligence* 5, 1 (1970), 153–163.

Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE)*. IEEE Press, 404–415.

Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 289–299.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 136–148.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN conference on Programming language design and implementation*. 281–294.

Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *Intl. Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 138–147.

Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of API migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.