BatFix: Repairing Language Model-based Transpilation

DANIEL RAMOS, Carnegie Mellon University, Pittsburgh, United States, INESC-ID, Lisbon, Portugal, and Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal
INÊS LYNCE, INESC-ID, Lisboa, Portugal and Instituto Superior Técnico - Universidade de Lisboa, Lisboa, Portugal
VASCO MANQUINHO, INESC-ID, Lisboa, Portugal and Instituto Superior Técnico Universidade de Lisboa, Lisboa, Portugal
RUBEN MARTINS, Carnegie Mellon University, Pittsburgh, United States
CLAIRE LE GOUES, Carnegie Mellon University, Pittsburgh, United States

To keep up with changes in requirements, frameworks, and coding practices, software organizations might need to migrate code from one language to another. Source-to-source migration, or transpilation, is often a complex, manual process. Transpilation requires expertise both in the source and target language, making it highly laborious and costly. Languages models for code generation and transpilation are becoming increasingly popular. However, despite capturing code-structure well, code generated by language models is often spurious and contains subtle problems. We propose BatFix, a novel approach that augments language models for transpilation by leveraging program repair and synthesis to fix the code generated by these models. BatFix takes as input both the original program, the target program generated by the machine translation model, and a set of test cases and outputs a repaired program that passes all test cases. Experimental results show that our approach is agnostic to language models and programming languages. BatFix can locate bugs spawning multiple lines and synthesize patches for syntax and semantic bugs for programs migrated from Java to C++ and Python to C++ from multiple language models, including, OpenAI's Codex.

CCS Concepts: • **Software and its engineering → Software maintenance tools;**

Additional Key Words and Phrases: Program analysis, automated refactoring, machine learning, transpilation

ACM Reference Format:
Daniel Ramos, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2024. BatFix: Repairing Language Model-based Transpilation. *ACM Trans. Softw. Eng. Methodol.* 33, 6, Article 161 (June 2024), 29 pages. https://doi.org/10.1145/3658668

This work was partially supported under National Science Foundation Grant Nos. CCF-1910067, CCF-1750116, and CCF-1762363, and by Portuguese national funds through FCT, Fundação para a Ciência e a Tecnologia, under PhD grant SFRH/BD/150688/2020 and projects UIDB/50021/2020, PTDC/CCI-COM/2156/2021, 2022.03537.PTDC, and project ANI 045917 funded by FEDER and FCT.
Authors' Contact Information: Daniel Ramos, Carnegie Mellon University, Pittsburgh, PA, United States, INESC-ID, Lisbon, Portugal, and Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal; e-mail: danielrr@cmu.ed; Inês Lynce, INESC-ID, Lisbon, Portugal and Instituto Superior Técnico - Universidade de Lisboa, Lisbon, Portugal; e-mail: ines.lynce@inesc-id.pt; Vasco Manquinho, INESC-ID, Lisbon, Portugal and Instituto Superior Técni Universidade de Lisboa, Lisbon, Portugal; e-mail: vasco.manquinho@inesc-id.pt; Ruben Martins, Carnegie Mellon University, Pittsburgh, PA, United States; e-mail: rubenm@cs.cmu.edu; Claire Le Goues, Carnegie Mellon University, Pittsburgh, PA, United States; e-mail: clegoues@cs.cmu.edu.

ACM 1049-331X/2024/06-ART161
https://doi.org/10.1145/3658668

1 INTRODUCTION

The right programming language can make or break a software project. Source-to-source translation or *transpilation* refers to the task of (usually at least partially automatically) migrating software from one language to another. For languages designed accordingly (like Typescript, which transpiles to JavaScript), this is a tricky, but generally tractable problem.

However, transpilation across arbitrary high-level programming languages is often complex, requiring significant manual labor and expertise on top of partial tool support [47, 51]. As a result, the process is often highly laborious and costly. According to media reports [25], the Commonwealth Bank of Australia spent over $750 million to replace its core banking infrastructure built in COBOL with Java. Similarly, Swedish bank Nordea has reportedly invested over €1.3 billion in overhauling its entire technology infrastructure. Traditional approaches to transpilation are costly because they are rule-based. Transpilers require programmers to manually look at the code and establish correspondences between the source and target language APIs.

By contrast with rule-based transpilers, there is increasing recent effort on training source code-oriented **language models (LMs)** for transpilation [5, 61, 62]. The idea is to use language models to aid developers transpiling small code snippets [8], thus speeding up and easing the transpilation process. Language models for source code translation are analogous to those used for natural language translation, a rich and established area of research [3, 69, 76]. However, key differences between the two tasks intrinsically limit how well LM-based source-to-source transpilers can be expected to perform. First, natural language translation works best when models can be trained on pairs of translated texts [36]. Unfortunately, there are vanishingly few bilingual pairs of software projects [22]. Instead, LM-based transpilation typically relies on unsupervised methods [61, 62], effectively using syntactic similarity to learn code transformations. General purpose language models like Codex [4] offer promising opportunities to learn transpilation models using few- or one-shot techniques [4, 7], from a very small number of contextual examples. Nonetheless, language models for transpilation have already been evaluated and shown to outperform even commercial alternatives based on traditional transpiler technology [61].

Second, natural language is blessedly tolerant of minor errors—the human brain naturally corrects any number of grammatical or spelling errors [6]. In source code, however, small syntactic mistakes like misplaced parentheses will prevent code from running altogether. Despite capturing code-structure reasonably well, the code generated by machine learning models (whether for transpilation or other tasks [8, 26]) can be spurious, ranging from general syntax errors, like referencing undefined variables, to semantic imprecisions, like calling the wrong API. Developers dislike (and distrust) tools that make unpredictable or "silly" mistakes [29]. For LM-based transpilation to reach mature practical utility, its output must be more robust and accurate [75].

We argue that stochastic methods like language models can and should collaborate with more traditional program analyses to provide stronger assurances and enhance the usability of the produced code. This collaboration can take many forms depending on the task at hand, e.g., using compiler errors to improve a code generation model [35] or embedding hard constraints into language generation [49].

In this article, we propose to post-process transpiled code generated by machine learning models to enhance the quality of the model's output before returning it to users. We frame the post-processing step as a **program repair (APR)** problem [38]. Several key ideas underlie our contribution. First, the original program being translated provides a rich source of information. It effectively provides a reference solution for the problem, albeit in a different programming language. This *provides an oracle for desired behavior that can be used to both inform and check syntactic and semantic fixes for the transpiled program.*

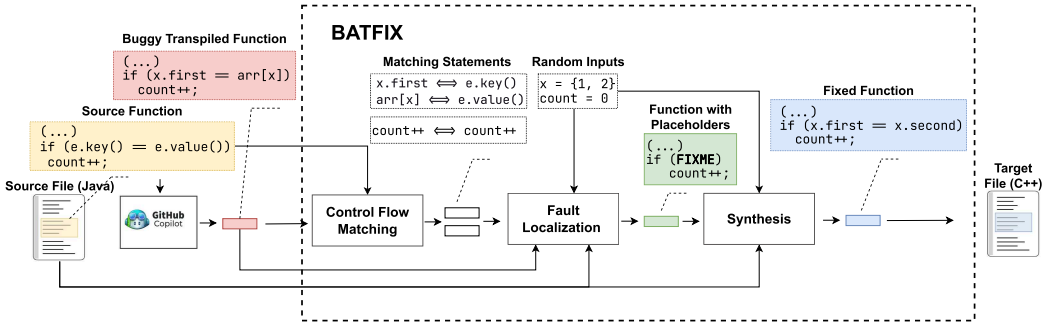


Fig. 1. Overview of BatFix architecture. BatFix takes as input a source file in the source language (e.g., Java) and a machine translation to a target language (e.g., C++) obtained from a language model. For each function pair in the file, BatFix computes a control flow matching of statements in that function. Using the original source file, the buggy transpiled function, the matching statements, and a set of random inputs, BatFix computes statements that are likely to be buggy in the translation and replaces them with placeholders. Finally, the synthesis component searches for statements to fill the placeholders until it finds a combination such that the source and target functions yield the same input on the random inputs.

Second, although languages vary in their particulars, we observe that central commonalities exist between them, especially within broad paradigms (like across imperative languages). These commonalities can be exploited for analysis, transformation, and repair [68]. Our intuition is that *two programs, one transpiled from the other, typically display similar control flow.* Beyond syntactic hints from a compiler, divergences between program traces of the reference and transpiled programs can pinpoint good repair sites.

Third, given that control flow should be similar, *it is possible to establish mappings between the statements of the two programs and use those mappings to inform repair synthesis.* Statement mappings provide important clues to a synthesizer to help replace incorrect statements with corrected versions. The idea is to enumerate and test potential replacements that are semantically or syntactically close to the statements of the original program, accounting for the language shift. Previous APR techniques demonstrate that syntactic similarity to the original buggy program can improve success [45], an insight we adapt to transpilation.

We instantiate these insights in BatFix **(Bugs After Transpilation Fix)**.[1] BatFix is a novel approach that can augment LM-based transpilation by leveraging program repair and synthesis to fix the source code generated by these models. BatFix (Figure 1) takes as input the original program, the target program generated by the machine translation model, and a set of corresponding sample inputs for both the source and target programs. At a high level, BatFix uses compiler error messages and divergences in the control flow on equivalent inputs to replace syntactically and potentially semantically incorrect statements with placeholders. In contrast to conventional Automated Program Repair techniques [38], BatFix introduces a novel approach to fault localization using execution traces and a new synthesis approach for fixing buggy statements. Both approaches are guided by source-level control flow mappings. In particular, BatFix uses mappings between *source-level control flow statements and diverging inputs to pinpoint where and when during the execution the transpiled program starts to behave erroneously.* Moreover, the control flow matching allows us to *use approximate mapping between the statements of the two programs to inform an enumerative program synthesis [23]* that constructs replacement code for those placeholders.

[1]While bats use echolocation to determine the location of objects BatFix combines novel takes on fault localization and program synthesis to locate and repair code produced by LM-based transpilation.

BatFix focuses on fixing programs transpiled across broadly imperative programming languages, a wide-ranging use case in practice. Our evaluation focuses on fixing programs transpiled from `Java` and `Python` to `C++`. Although much of the machinery is language-agnostic, elements of the frontend and code generation, in particular, are by necessity language-specific, and so we make this choice to ease the implementation burden for our prototype. We expect that the approach should generalize across other pairs of broadly imperative-style programming languages. We present results on a set of benchmark programs from three state-of-the-art machine translation models: Transcoder [61], Transcoder-ST [62], and OpenAI's Codex [4]. More powerful recent models do improve on their predecessors (i.e., Codex produces fewer buggy transpiled programs than Transcoder). Interestingly, BatFix ends up fixing a higher *percentage* of the buggy program's output by the better models, because better models still cannot contend with the types of semantic issues more traditional analyses are well-suited for. Indeed, our results show that BatFix can fix as many as 50% of the buggy programs generated by OpenAI's Codex on a dataset of 56 programs transpiled from `Java` to `C++`, with syntax and semantic errors. Preliminary results also show that we can fix 22% of the bugs from `Python` to `C++` on a dataset of 109 buggy programs.

In summary, we make the following contributions:

- BatFix, a novel approach to repairing the output of machine translation models that is language-agnostic.
- An approach using **Maximum Satisfiability (MaxSAT)** to map control flow nodes between the original and transpiled programs.
- A new, iterative fault localization approach based on execution traces and control flow matching.
- An evaluation on three datasets of programs with errors transpiled from `Python` and `Java` to `C++`, using three state-of-the-art machine translation models.

2 MOTIVATION AND OVERVIEW

Assume a developer, Alex, needs to transpile an existing codebase from Java and Python to C++ to improve performance. Unfortunately, Alex is unfamiliar with C++ and decides to use GitHub Copilot [20] (an AI-based code tool) to help with the task. Alex approaches this transpilation task as a refactoring problem, manually transpiling each source file on a function-by-function basis. To do this, Alex right-clicks each function in the source file to invoke Copilot, to attempt to translate it to `C++`. As an example, consider the source code shown in Listing 3, which Copilot translates to `C++` as shown in Listing 4. Alex inspects and manually tunes the translation before accepting it.

Alex demonstrates the use case we envision for language-model-based transpilers. In practice, developers treat transpilation as a refactoring task, combining manual effort with refactoring tools [77], rather than using end-to-end transpilers. AI tools for code transpilation have been recently proposed and drawn attention, but their impact on developers' workflow and productivity is unclear [75]. Resistance towards AI tools for code generation spawns from two significant challenges. First, often it is harder for programmers to understand than to write the code. Studies [67, 75] have shown that code suggestions from AI are often not useful because they are hard to understand. Second, code generated by AI tools can be unreliable and contain bugs, ranging from easily spottable syntax errors to subtle semantic bugs that cause developers' distrust [16, 34]. Indeed, one of the biggest barriers to the adoption of AI tools for code generation is the lack of tools to help users edit, debug, and repair code generated by these models [67]. The second problem (the reliability issue) is the problem we aim to address.

In this article, we propose a post-processing step for LM-based tools for code translation to enhance their reliability and prevent them from returning hard-to-debug code. The use case for tools

Listing 1. Java source code

```
1  int f(int start, int end, int
       arr[]){
2    var mp = new HashMap<(...)>();
3    for (int i=start ; i<=end ; i++)
4      mp.put(arr[i], ...) ;
5    int count = 0;
6    for (var e: mp.entrySet())
7      if (e.getKey() == e.getValue())
8        count++;
9    return count;
10 }
```

Listing 2. Buggy C++ transpilation

```
1  int f(int start, int end, int arr[]){
2    map<int, int> mp;
3    for (int i=start; i<=end; i++)
4      mp[arr[i]] = ...;
5    int count = 0;
6    for (auto it : mp)
7      if (it.first == arr[i])      // bug
8        count++;
9    return count;
10 }
```

Fig. 2. Example of a Java source code and the buggy transpilation generated by Transcoder. There are two bugs in this transpilation: (1) variable i in line 7 is undefined, and (2) even if i was defined, the if condition of line 7 would still be wrong.

like Copilot remains the same (no extra information is necessary from the user). It is important to note that model-based transpilation differs significantly from traditional transpilers. Traditional transpilers, such as `Java2C#` [27], operate with a predetermined set of API mappings and execute program transformations deterministically. Language models, however, do not rely on a fixed mapping but generate translations based on patterns learned from data. This introduces a level of unpredictability and the potential for novel errors in the transpiled code. However, they have the benefit of being able to help translate small snippets between arbitrary programming languages without relying on any compiler infrastructure. Moreover, machine-learning based transpilers have been shown to outperform commercial alternatives based on traditional transpiler technology [61].

Thus, our idea is to add a consistency check to code generated by language models and automatically repair it when necessary. Before detailing our approach, we first illustrate the two major categories of mistakes language models make in transpilation tasks: syntax and semantic errors. Recall the code example from earlier as shown in Listing 1. Transpiling the code in Listing 1 using Transcoder (a language model explicitly trained for transpilation) yields the C++ program shown in Listing 2. This translation is almost correct but contains two problems. First, the variable `i` is undefined in line 7. Second, the condition is incorrect (even if `i` were defined, it should read `(it.first == it.second)`). In this case, the bug is easy to spot, as the code does not parse; however, mistakes can sometimes be even more subtle. For example, consider the following code snippet from a source file in Python as shown in Listing 3: Transpiling this using Copilot yields the C++ program shown in Listing 4. Although the `C++` code is syntactically correct, it is not semantically equivalent to the Python code. Notice that the variable `table` is uninitialized but used in line 8. This results in undefined behavior, since the variable's value can be arbitrary.

Our tool, BatFix, can fix both syntax and semantic errors that language models make. BatFix takes as input the source program, its proposed translation to the target language, and a set of valid inputs (obtained, for instance, from differential testing techniques). Next, we provide an overview of the steps BatFix takes to repair the translation.

Fault Localization. The first step in fixing transpiled output is to identify the buggy lines of code. In our use case, this task is facilitated by access to a correct program (despite being in a different language). Our insight is that we can significantly outperform existing fault localization [1] approaches by leveraging the code in the source program to guide the search for faulty statements in the corresponding translation. BatFix identifies (potentially) spurious statements in two ways:

Listing 3. Python source code

```python
def f_gold (m, n, x):
  tab = [[0] * (x + 1) for i in
      range(n + 1)]
  for j in range(1, min(m+1, x+1)):
    tab[1][j] = 1
  for i in range (2, n+1):
    for j in range(1, x+1) :
      for k in range(1, min(m+1,
          j)):
        tab[i][j] +=
            tab [i-1][j-k]
  return tab[-1][-1]
```

Listing 4. C++ translation of Listing 3

```cpp
int f_filled(int m,int n,int x) {
  int tab[n+1][x+1]; // fault of omission
  for(int j=1; j < min(m+1,x+1); j++)
    tab[1][j] = 1;
  for(int i=2; i<n+1 i++)
    for(int j=1; j<x+1; j++)
      for(int k=1; k < min(m+1,j);
          k++)
        tab[i][j] += tab[i-1][j-k];
  return tab[n][x]; }
```

Fig. 3. Example of a Python source code and the buggy transpilation generated by Copilot. The problem with this code is that the `table` variable is uninitialized.

(1) **Syntax-directed fault localization.** If the translated program does not compile, then we start by using compiler error messages to identify potentially non-compliant statements. For instance, Clang [65] reports the following message when attempting to compile the code in Listing 2: "`use of undeclared identifier i`". BatFix works at the source-line level, and thus marks the entire `if` condition as potentially spurious.

(2) **Trace-driven fault localization.** If the translated program compiles, then BatFix computes a set of test-cases by running random inputs on the original source program. If the transpiled program yields a different output on at least one of the inputs (i.e., one test case fails), then we use control flow divergences to localize implicated statements. BatFix computes execution traces consisting of a set of program states (i.e., source code line and variable content) for both programs on the failing test-cases. We use an approximate matching of these traces to find where execution diverges.

This second approach benefits from additional illustration. For example, consider the execution of the Java program in Listing 1 on a failing test case:

```
Source Line: 2
State: start = 0; end = 31; arr = {1, ...}
Source Line: 3
State: start = 0; end = 31; arr = {1, ...}
Source Line: 4
State: i = 0; start = 0; end = 31; ...
```

Each step in each trace naturally maps to a corresponding node in the associated program's **control-flow graph (CFG)** by looking at the **Program Counter (PC)**. The key technical detail instead lies in computing a mapping between the control-flow graphs of the source and translated programs. The idea is to walk both control-flow graphs simultaneously according to the execution traces, while asserting that every time we move one step, the executions do not diverge in either control flow or local variable values. If the execution diverges because the nodes diverge, then we assume that the faulty statement is the conditional statement that most closely preceded the current step. For example, consider the Java trace for the program in Listing 1 takes the `if` branch from line 7 to 8, but the corresponding trace in C++ buggy program jumps from line 7 to 6. In this case, we say that the bug is in the conditional in line 7. If the execution diverges because two values diverge, then we mark the last executed statement as buggy.

BatFix replaces each identified buggy statement with a code placeholder. BatFix attempts to repair as many identified suspicious locations at once as it can identify using these fault localization procedures.

Synthesis. The goal of the synthesis component is to assign concrete statements to each placeholder such that all test cases pass. We again leverage the mapping between the CFGs of the source and translated programs to accomplish this, because the mapping suggests what the correct statement might "look like" for each placeholder. In our running example, the faulty statement is `(it.first == arr[i])`, and from the control flow matching, we get that its corresponding statement in Java is `(e.getKey()== e.getValue())`. This suggests that the placeholder should contain an equality comparison. BatFix leverages this information to guide the synthesis/repair process. In particular, it generates an *abstract sketch*, like `foo1 == foo2`, where `foo1` and `foo2` are abstract (and should be concretized). The synthesis process uses this sketch and hints from the local context in the translated code to explore concrete statements satisfying this sketch, seeking an instantiation that causes (more) tests to pass. BatFix can do this for multiple faulty statements at once.

We provide details on each component in subsequent sections. Because both the fault localization and synthesis steps rely on the approximate control-flow matching between source and translated programs, we describe that step first (Section 3). We then discuss the fault localization (Section 4) and synthesis (Section 5) components in turn, before turning to evaluation (Section 6).

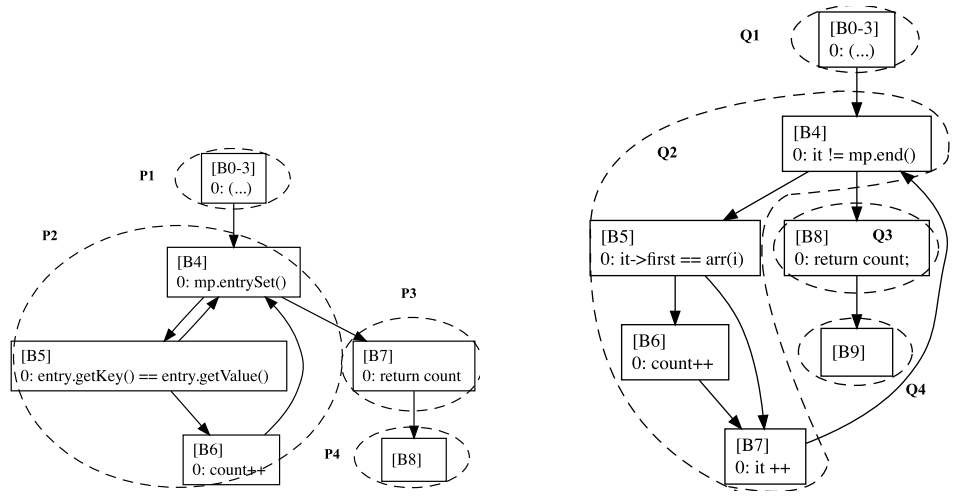
3 CONTROL-FLOW MATCHING

An essential step in BatFix is to match the *source-level* **control-flow graphs (CFGs)** of the original program with the translated program. This mapping informs both fault localization (Section 4) and synthesis (Section 5). Because the programs are in different languages, we do not expect these CFGs to match one-to-one; instead, single blocks or statements in one program may map to many blocks or statements in the other and vice versa. In this section, we propose constructing these mappings using Maximum Satisfiability [39] to match the control flow between the two programs.

Maximum Satisfiability (MaxSAT) is the optimization version of the well-known Propositional **Satisfiability (SAT)** problem, with applications in software analysis [63]. Given a conjunction of propositional clauses, the goal of MaxSAT is to find an assignment to the problem variables that satisfies the maximum number of clauses. In this article, we assume a more general MaxSAT formulation where clauses are classified as *hard* or *soft*. Let ϕ_h and ϕ_s denote the hard and soft clauses in a MaxSAT formula, respectively, where each soft clause $c_i \in \phi_s$ is associated with a positive integer weight w_i that represents the cost of not satisfying it. Here, the goal is to find an assignment that satisfies *all* hard clauses and minimizes the total weight of unsatisfied soft clauses.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two directed graphs that represent the control flow of two programs to match. Figure 4 shows the CFGs from Listings 1 and 2. To match program structure across control flow, we build the graph of the **Strongly Connected Components (SCC)** of each graph [11]. Let $G_1^{SCC} = (V_1^{SCC}, E_1^{SCC})$ and $G_2^{SCC} = (V_2^{SCC}, E_2^{SCC})$ be the directed graphs of the SCCs of G_1 and G_2. Clearly, both G_1^{SCC} and G_2^{SCC} form a tree. Our idea is to encode the SCCs of the CFGs in a MaxSAT formula to match CFG statements. For our running example, Figure 5 shows a partial mapping for the SCCs for the CFGs. Nodes with just one statement in the matching generated by the model (for example, `P3` to `Q3`) are immediately matched with each other. SCCs with more than one statement (e.g., `P2` to `Q2`) are unrolled and matched in the subsequent iteration.

To formalize the graph matching problem in a MaxSAT formula, we further define the predecessors and successors of a node in the graph as follows: For every edge (u, v), u is a predecessor of v and v is a successor of u. Let $p(u)$ denote the set of nodes in any path from the root to node u, and let $s(u)$ denote the set of nodes in any path from node u to a leaf. Next, we define the set of



(a) Java cfg of Listing 1 (b) C++ cfg of Listing 2

Fig. 4. Simplified version of the control-flow graphs from the example of Figure 2, with a partial SCC graph matching. To complete matching, we need to solve a new MaxSAT problem for blocks P_2 and Q_2.

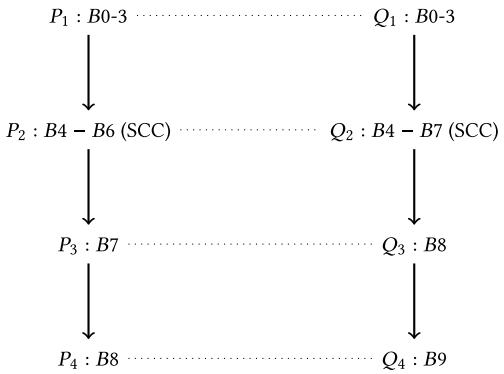


Fig. 5. Initial SCC graph matching for the control-flow graphs in Figure 4.

possible matchings between the nodes in the two trees. For each pair of nodes u and v such that $u \in V_1^{SCC}$ and $v \in V_2^{SCC}$, we define a Boolean variable x_{uv} denoting whether u and v are matched ($x_{uv} = 1$) or not ($x_{uv} = 0$). After defining the possible node matches, a MaxSAT formula can be built such that each solution corresponds to a valid matching between the two graphs.

Figure 6 presents the MaxSAT model. The hard clauses (1) and (2) ensure that all nodes in each graph match at least one node from the other graph. Clauses (3) and (4) ensure that *if two nodes u and v match*, then *no predecessor of node u can match a successor of node v* and vice versa. Clauses (5) and (6) assert that *successors of two matching nodes must either match with each other or the parent node*. Formally, clauses (5) ensure that if there is a match between nodes u and v, and u does not match any of the successors of v, then the successors of u can only match either with node v or any of its immediate successors. Clauses (6) are analogous in the other direction.

Finally, to find an optimal matching, the soft constraints are defined as trying to minimize the number of matches between nodes of the two trees. This is achieved using soft clauses in (7), where

Hard Clauses:

$$\forall u \in V_1^{SCC} \qquad \bigvee_{v \in V_2^{SCC}} x_{uv} \tag{1}$$

$$\forall v \in V_2^{SCC} \qquad \bigvee_{u \in V_1^{SCC}} x_{uv} \tag{2}$$

$$\forall u \in V_1^{SCC}, v \in V_2^{SCC}, u' \in p(u), v' \in s(v) \qquad (\neg x_{uv} \vee \neg x_{u'v'}) \tag{3}$$

$$\forall u \in V_1^{SCC}, v \in V_2^{SCC}, u' \in s(u), v' \in p(v) \qquad (\neg x_{uv} \vee \neg x_{u'v'}) \tag{4}$$

$$\forall u \in V_1^{SCC}, v \in V_2^{SCC}, (u,u') \in E_1^{SCC} \qquad \left(\neg x_{uv} \bigvee_{(v,v') \in E_2^{SCC}} x_{uv'} \vee x_{u'v} \bigvee_{(v,v') \in E_2^{SCC}} x_{u'v'} \right) \tag{5}$$

$$\forall u \in V_1^{SCC}, v \in V_2^{SCC}, (v,v') \in E_2^{SCC} \qquad \left(\neg x_{uv} \bigvee_{(u,u') \in E_2^{SCC}} x_{u'v} \vee x_{uv'} \bigvee_{(u,u') \in E_1^{SCC}} x_{u'v'} \right) \tag{6}$$

Soft Clauses:

$$\forall u \in V_1^{SCC}, v \in V_2^{SCC} \qquad ((\neg x_{uv}), w(u,v)) \tag{7}$$

Fig. 6. MaxSAT graph matching model.

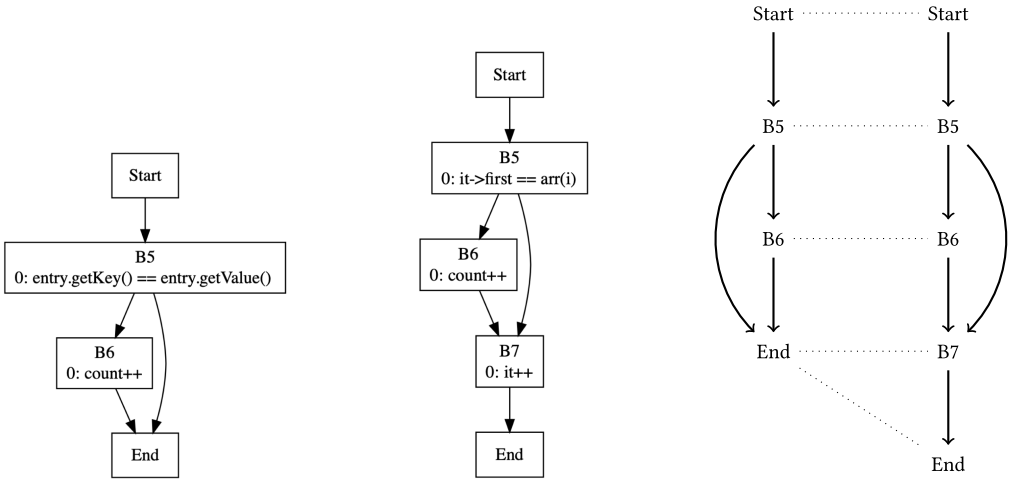
$w(u,v)$ denotes the cost of matching nodes u and v. In BatFix, we define the cost of matching nodes u and v as the edit distance between the code of the two nodes.

In Figure 5, the nodes in the SCC P_2 are matched with nodes in the SCC Q_2. Hence, the match of the nodes inside each SCC still needs to be determined. However, each SCC only has one node with incoming edges from previous nodes in the tree in our context. Let this node be the *root* of the SCC. Necessarily, if there is a match between two SCCs, then the roots must match. Thus, *B4:* `mp.entrySet()` and *B4:* `it != mp.end()` are matched with each other.

To match the remaining SCC nodes, the roots of both SCCs are removed (breaking the SCC), and the same matching procedure is applied. To preserve the structure of the subgraphs, we substitute the root of the SCCs with two separate nodes: `Start` and `End`. In Figure 7, we show the resulting subgraphs after replacing the root nodes with the `Start` and `End` nodes. Figure 7(c) shows the matching between the nodes in the subgraphs. Notice that nodes B_5 and B_6 in Figure 7(a) are matched against their counterparts in Figure 7(b) due to their syntactic similarity, whereas `End` from P_2 matches both `it++` and `End` from Q_2 (i.e., `it++` does not have a real matching statement, as expected).

4 FAULT LOCALIZATION

The task of identifying bugs in code is known as fault localization [55, 73]. Traditional approaches to fault localization, like **Spectrum-based Fault Localization (SBFL)** [1, 13, 31] rely on metrics such as the number of times a line of code is executed in failure vs. passing test cases to pinpoint which statements are the most likely to be faulty. However, our task differs from the traditional SBFL setting. First, we aim to fix programs that may contain syntax errors. Fault localization methods tend to focus on code that compiles and runs. Second, and more importantly, our problem statement means BatFix has access to significantly more information than in the general fault localization formulation, by way of the reference program. Therefore, our methods for fault localization differ substantially from traditional methods (and outperform them in our setting, as we show in Section 6).



(a) Java subgraph corresponding to the SCC P_2 from Figure 4 (a).

(b) C++ subgraph corresponding to the SCC Q_2 from Figure 4 (b).

(c) Second iteration of the MaxSAT matching for P_2 and Q_2.

Fig. 7. Subgraphs of the control-flow graphs from Figure 4. The subgraphs correspond to the nodes P_2 and Q_2 in the original SCC graph. The entry point to the SCCs is broken down into two nodes, `Start` and `End`.

Table 1. Types of Syntax Errors that BatFix Can Address

Type	#	Description	Example Error Message
Rule-based	1	Small typos the compiler can automatically recover from	"no template named 'Vector'; did you mean 'vector'?"
	2	References to unknown/undeclared types	"unknown type name 'Integer"'
	3	Type declaration does not match assignment	"cannot initialize a variable of type 'int' with ..."
	4	Random keyword (public, static)	"expected unqualified-id (public)"
Placeh.	5	Use of undeclared identifiers (function and variable names)	"use of undeclared identifier 'x"'
	6	Other errors that the compiler can bound (start-end)	"cannot convert 'std::pair⟨const int, int⟩' to 'int"'

4.1 Syntax Errors

LM-based code generation often produces code that does not compile due to syntax errors of all stripes [8]. For many transpiled programs, BatFix's first task is thus to modify it so it compiles, while minimizing the chances of introducing new mistakes. Compiling is necessary to run the program (such as on its tests cases) as well as to construct control-flow graphs and other structures necessary for semantic localization and synthesis. Recovering from syntax errors without human input is a notoriously ambiguous and difficult task [66].

Table 1 lists two families of syntax errors we attempt to fix:

— **Rule-based Syntax Corrections**. These are simple errors that can be fixed with a set of rules. Error types #1–#4 belong to this category. The compiler can automatically recover small typos, (e.g., type #1) by using the suggested fixes. For type #2 errors, we use a set of static rules and type inference. For example, if the function argument are undefined, then we use types from the test cases to infer the corresponding type. Type #3 errors are fixed using the keyword "auto" to match type declarations. Type #4 errors are fixed by removing `public` and `static` keywords.

— **Placeholder-driven Syntax Corrections.** For syntax errors that cannot be fixed via straightforward rules, BatFix replaces the buggy statements with placeholders to be

ALGORITHM 1: FaultLocalization($\mathcal{S}, \mathcal{T}, \mathcal{M}, C$)

Input: $\mathcal{S}$: source program, $\mathcal{T}$: buggy transpiled program, $\mathcal{M}$: control-flow matching model, C: test cases
Output: c: code block at which the execution diverged

```
1: F_S, F_B := GetTrace(S, C), GetTrace(T, C)
2: M := GetMatchingBlocks(M, F_S, F_T)
3: for each (M_S ↔ M_T) ∈ M do
4:     if Diverges(M_S, M_T) then
5:         return M_T
6:     end if
7: end for
```

replaced via synthesis (Section 5). For instance, for error type #5, we declare an uninitialized new variable "x", leaving initialization to the synthesis phase. Type #6 errors are other types of placeholder-driven errors that the compiler can pinpoint and tell where they start and end (e.g., if the condition does not parse). Since the compiler can bound the faulty code, we can replace it with a placeholder by, e.g., synthesizing a new condition.

There are other types syntax errors that we do not attempt to tackle, such as imbalanced brackets, multiple undefined user objects, or syntax errors that generate ambiguity. In these cases, the compiler may not return enough information for BatFix to succeed. Insights or techniques from prior work in automatic repair of compilation errors [24, 48] could likely supplement our approach. Although not explored in this article, large language models like `GPT-4` could also be used to fix such syntax errors.

4.2 Semantic Errors

Beyond syntax errors, LM-transpiled programs can also contain semantic errors, like the faulty `if` condition in Listing 2 of Figure 2. BatFix assumes the original program is correct and relies on test inputs to identify behavioral divergences (that is, the output of the original program is assumed correct; a transpiled program that gives a different output for the same input is assumed buggy). We assume these test inputs are given and leave integration with techniques that can, e.g., automatically generate differential tests to future work. BatFix uses these test inputs to construct execution traces and then uses the CFG matching model (Section 3) to identify points in the target code where execution diverges from the source program.

Algorithm 1 shows the pseudocode of our fault localization procedure. Our key insight is to consider a failing test case of the transpiled code and to find the code block for which the execution of the program diverges from the original source program. There are two possible ways in which the program diverges:

(1) the control flow diverges (e.g., if condition evaluates to true in the source program and to false in the buggy program); (2) matching variables have different values after the execution step[2]

To determine when a program diverges, we traverse the CFG of the transpiled code progressively and compare the current (mapped) CFG node and variable values between the transpiled and the original program until we find one of these two divergence scenarios.

Our fault localization procedure takes as input the source code of the original program $\mathcal{S}$, the buggy transpiled program $\mathcal{T}$, the control-flow matching model $\mathcal{M}$, and a set of test cases C generated from user provided sample inputs. We execute both programs and store their execution

[2]We assume that when a program diverges, it will no longer converge.

trace for a failing test of the transpiled program (line 1). An execution trace T of a program P is a sequence of states $s_1, s_2, \ldots, s_n$, where each state s_i contains information on the source code line and the values of each variable for each code block. Using the control-flow matching model, we compute the matching blocks between the execution traces (line 2). Matching blocks are analyzed following the topological order of the CFG to ensure that we find the first block where a divergence occurs. For each matching block, the DIVERGES function (line 4) checks if the control flow diverges, i.e., if both programs are following the same execution path such as in if-else statements. It also checks if the equivalent variable values diverge in the same code block. If either of these cases occurs, then the fault location procedure returns the code block of the transpiled code where the divergence occurred (line 5).

Note that, in general, block matching will not necessarily be one-to-one and may even be many-to-many. For example, consider the following toy program in C++:

```
int dp[n];
for (int i = 0; i < n; i++) dp[i] = i;
```

And its corresponding translated Python:

```
dp = [i for i in range(n)]
```

The CFG matching model maps all CFG nodes of the C++ program to the only node of the Python program, and the variable `dp` can only be compared between the programs after the `for` loop completes, which the block matching procedure ensures. In practice, state-of-the-art transpilation models maintain variable names in translation, and our matching assumes this for state comparison. If this were not the case, then our fault localization would require mapping between variables.

5 PROGRAM SYNTHESIS

Given a set of test cases and a mapping between the statements of the original source code $\mathcal{S}$ and the buggy transpiled code $\mathcal{T}$, our synthesis engine automatically generates code to replace all the placeholder statements in $\mathcal{T}$ created during fault localization (Section 4). The output of the synthesis component is a correct version of the transpiled code $\mathcal{T}'$ (i.e., passes all test cases).

BATFIX uses program sketching [64] to synthesize replacement code for placeholders. Program sketching takes as input a specification and an incomplete program with placeholders (or "hole") and automatically finds a replacement for each hole in the program such that the program satisfies the specification. Here, the specification is a set of test cases that are used to check the correctness of the transpiled code. Note that the buggy code $\mathcal{T}$ may have multiple placeholders.

We present our synthesis engine in Algorithm 2. BATFIX starts by identifying spurious code blocks (Line 1) as described in Section 4. Next, we follow the standard approach [18, 59] to program sketching: (1) sketch generation and (2) program enumeration.

5.1 Sketch Generation

Each spurious statement is replaced with a generic placeholder (Line 2). Consider that FAULTLOCATIZATION returns the code block corresponding to an if condition such as the one presented in the motivating example in Listing 2 of Figure 4. The buggy condition is replaced by a placeholder condition `if(function#0)`. To improve the structure of our sketch, we refine the sketch using information from the CFG matching as follows: For each spurious placeholder, we get its corresponding correct statement in the source program. The corresponding statement is then abstracted to generate a refined sketch for the generic placeholder that matches the structure of the original code (Line 4). The refined sketch constrains the number of possible statements that the synthesizer will need to enumerate. For example, consider the source and transpiled programs in Listing 2 of

Figure 4. In this case, the only spurious statement is `it.first == arr[i]`, and its corresponding statement in the source program is `e.getKey()== e.getValue()`. We abstract this statement by replacing both `e.getKey()` and `e.getValue()` with placeholders, resulting in the refined sketch `if(function#1 == function#2)`, which replaces the placeholder `if(function#0)` in our initial sketch.

ALGORITHM 2: SYNTHESIZE($\mathcal{S}, \mathcal{T}, \mathcal{M}, C$)

Input: $\mathcal{S}$: source program, $\mathcal{T}$: buggy transpiled program, $\mathcal{M}$: control-flow matching model, C: test cases
Output: $\mathcal{T}'$: fixed transpiled code

```
1: s⃗ : = FaultLocalization(S, T, M, C)
2: P : = GenerateSketch(T, s⃗)
3: for each s ∈ s⃗ do
4:     P := RefineSketch(P, M[s])
5: end for
6: while SearchSpaceNotExhausted(P) do
7:     T′ : = P
8:     for each s ∈ s⃗ do
9:         (v⃗, f⃗) := GetVarsFunsInScope(T′, s)
10:        T′ := FillSketch(T′, s, v⃗, f⃗)
11:    end for
12:    if PassAllTests(T′, C) then
13:        return T′
14:    else if PassMoreTests(T′, C) then
15:        return Synthesize(S, T′, M, C)
16:    end if
17: end while
```

5.2 Program Enumeration

After generating and refining the sketch, BATFIX needs to fill in the placeholders in the sketch to synthesize concrete programs. BATFIX starts by collecting the variables and functions available in the scope of each statement that needs to be replaced (Line 9). These variables and functions will be used to generate new statements. For example, for the program in Listing 2 of Figure 4 and the placeholder `if(function#1 == function#2)` in Line 7, the available variables are `start, end, arr, mp, count, e`, and the available functions are the fields and methods associated with objects defined in the scope (e.g., `mp.insert, e.first, e.second, etc`) and a set of functions from the `stdlib` (e.g., `sort, max, min, etc`). These variables and functions are used to fill the placeholder for that statement (Line 10). We repeat this process for all statements that need to be replaced to generate a concrete program.

We use an exhaustive enumeration to fill all placeholders in the sketch (Line 10). There are only two kinds of placeholders: variable placeholders and function placeholders. Variable placeholders are replaced with concrete variables that are available in the statement's scope. For function placeholders, we need to constrain (1) which functions to consider and (2) how to fill the arguments. To reduce the search space, we only consider functions whose arity does not differ more than one from the corresponding function call. To fill function arguments, we search for arguments that type-check and are compliant with the function type signature. To limit the search space, we

Table 2. Number of Programs BatFix Fixes from a Total of 698 Programs Transpiled from **Java** to C++

Model	Java Programs				
	w/ Syntax Errs.			w/ Sem. Errs	
	Fixed		Total	Fixed	Total
	Stx only	Stx + Sem			
Transcoder	26 (28.0%)	8 (8.6%)	93	20 (52.6%)	38
Transcoder-ST	5 (11.6%)	6 (14.0%)	43	23 (59.0%)	39
Codex	4 (16.7%)	3 (12.5%)	24	21 (65.6%)	32

only consider the variables or methods associated with the variables when filling in each function argument. This avoids arbitrary applications that could lead to unbounded recursive calls.

Once all the holes in the sketch are filled, we test the concrete program against the test cases (Line 12). If the program passes all test cases, then we return the repaired program to the user. If the concrete program $\mathcal{T}'$ passes more tests than before but not all of them, then it is possible it requires more changes to be fully repaired. In this case, we call synthesize again (Line 15) using $\mathcal{T}'$ as our starting point. This will call the fault localization procedure (Line 1) with a new failure test, which may lead to a new divergence point. This process is repeated until either the space of all possible programs is exhausted or a concrete correct program is found.

6 EVALUATION

We evaluated our approach on three datasets of buggy programs obtained by transpiling Java and Python to C++ using three different machine translation models, Transcoder [61], Transcoder-ST [62], and Open AI's Codex [8]. Our evaluation aims to answer the following research questions:

RQ1. How successful is BatFix in fixing transpilations between syntactically close languages?
RQ2. Does BatFix generalize to different transpilation models?
RQ3. Does BatFix generalize to different languages?
RQ4. How does BatFix's fault localization compare to spectrum-based fault localization (SBFL)?
RQ5. What kind of bugs can BatFix fix and what are its limitations in bug fixing?
RQ6. How does BatFix's compare with state-of-the-art large language models for bug fixing?

6.1 Benchmarks and Implementation

6.1.1 Benchmarks and Models. We evaluate BatFix on buggy transpiled programs produced by three different machine learning models: Transcoder [61], Transcoder-ST [62], and Codex [8]. Transcoder and Transcoder-ST are machine translation models trained for transpilation between between Java, C++, and Python with 311M parameters. Codex is OpenAI's large language model for general source code tasks, with 12B parameters. We use the benchmark program set collected by Rozière et al. [61] to evaluate Transcoder and Transcoder-ST. This benchmark is a parallel corpus of functions in Java, C++, and Python gathered from GeeksForGeeks [19] of implementations of well-known algorithms with between 2–45 lines of code; the benchmark also provides a testing framework for these programs. For Java, we use each model to attempt to transpile each of the 698 Java programs to C++. We run each transpiled program on the default test cases the benchmark provides for that program and determine whether it succeeds, or alternately fails syntactically or runs but fails one or more of the tests.

Tables 2 and 3 summarize the number of Java and Python programs that the three models fail to transpile, either due to semantic or syntactic errors. Given the number of programs involved, we

Table 3. Number of Programs BatFix Fixes from a Total of 465 Programs Transpiled from **Python** to C++

Model	Python Programs				
	w/ Syntax Errs.			w/ Sem. Errs	
	Fixed		Total	Fixed	Total
	Stx only	Stx + Sem			
Transcoder	15 (7.7%)	0 (0.0%)	194	17 (11.8%)	144
Transcoder-ST	1 (2.5%)	2 (1.7%)	118	30 (13.7%)	218
Codex	6 (12.0%)	1 (2.0%)	50	17 (28.8%)	59

do not manually inspect whether programs that fail syntactically would also fail semantically if the syntax errors were corrected. Codex performs best at this task; Transcoder-ST outperforms Transcoder for all tasks.

For RQ1, RQ2, and RQ4, we focus on the Java to C++ results, as they are more comprehensive; we discuss both transpilations tasks (i.e., Java to C++ and Python to C++) in RQ3, RQ5, and RQ6, to address language generalizability, BatFix's limitations and strengths, and to compare against large language models for APR.

6.1.2 Implementation and Execution. BatFix is implemented in C++ and integrates multiple existing tools. LibTooling [65], the C++ interface to Clang's API, is used to (1) parse and fix syntax errors; (2) extract contextual information in specific scopes (e.g., which variables are available in a for loop, what functions); (3) rewrite the code generated by the synthesis engine. We also use LibTooling to ensure that most of the code is syntactically correct and type checks. We use LLDB's C++ API [40] to extract program traces (i.e., execute programs step-by-step to do fault localization). Similarly, we use JDB [28] and PDB [54] to extract program traces from Java and Python programs, respectively. To extract CFGs, we use LibClang, Spoon [53], and staticfg [10] for C++, Python, and Java programs, respectively. To solve CFG matching formulas, we use the Open-WBO [42] MaxSAT solver. All results were obtained on a laptop running macOS Monterey with an Intel Core i9-9880H, 32 GB of RAM, and a time limit of 5 minutes per program.

6.2 Results

6.2.1 RQ1-2: Effectiveness & Model Generalizability. To evaluate effectiveness, we used BatFix to attempt to fix the buggy programs produced by the three models for the transpilation task from Java to C++. BatFix was originally developed using insights from code generated by Transcoder; we validate that it generalizes to other models by evaluating it on the output of two more recent state-of-the-art models (Transcoder-ST and Codex). We consider a program "fixed" if both reference and patched programs yielded the same output on all inputs provided by the benchmark.

Table 2 shows the number of programs BatFix could fix in the Java dataset. We split the programs into three categories: (1) programs with only syntax errors, (2) programs with both syntax and semantic errors, and (3) programs with only semantic errors.[3] BatFix can fix 41.2% (54 out of 131) of the buggy programs generated by Transcoder, 41.5% (34 out of 82) of the buggy programs generated by Transcoder-ST, and 50.0% (28 out of 56) of the buggy programs generated by Codex. Despite each model outperforming the previous one, BatFix fixes a larger *percentage* of the buggy programs that the better models produce: BatFix fixes 52.6%, 59.0%, and 65.6% of the buggy programs with only semantic errors generated by Transcoder, Transcoder-ST, and

[3]Recall that programs with syntax errors that BatFix *cannot* fix may also contain semantic errors.

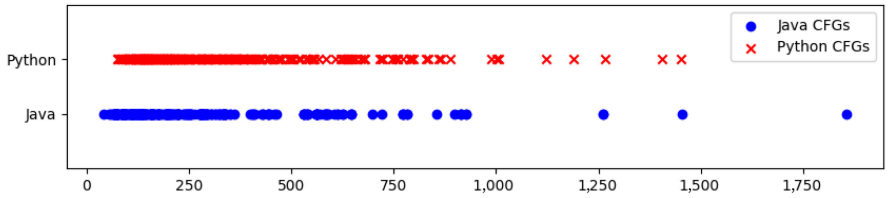


Fig. 8. Values obtained from the MaxSAT optimal solutions for CFG matching. Lower values indicate a closer match, measured using the sum of the Levenshtein distance between matching statements. The median Java solution is 199, whereas the Python median is 266.

Codex, respectively. Language models are becoming more powerful: Transcoder-ST and Codex are better at generating syntactically correct code than Transcoder. However, the mistakes they *still* make are exactly those that BatFix's more traditional analyses are well-equipped to address.

6.2.2 RQ3: Language Generalizability. Although BatFix's conceptual approach is language-agnostic, there is an implementation burden in creating new front/backends to support additional languages. To validate BatFix's generalizability to other languages, we conducted a preliminary evaluation on a Python to C++ task. Since Python and C++ are not syntactically similar, we expected that BatFix would not be able to fix as many programs as it did on the Java to C++ task. We translated a subset of 465 programs out of the 698 programs for which we had test cases for Python using Transcoder, Transcoder-ST, and Codex, resulting in one dataset of buggy programs for each model. Then, we tasked BatFix with fixing the programs.

In this task, we see that BatFix 9.5% (32 out of 338) of buggy programs generated by Transcoder, and 9.8% (33 out of 336) of buggy programs generated by Transcoder-ST. For both models, we note that the repair rate is much higher for programs with semantic errors, likely because the programs with syntax errors are of very low quality on the Python to C++ task. However, BatFix can fix 22.0% (24 out 109) of buggy programs generated by Codex. In this set, our success rate is higher (28.8%) when the buggy programs only have semantic errors.

The overall percentage of fixed programs for the Python task is lower (though still *within observed bounds of repairability for general-purpose APR*). We believe this happens for a number of reasons. First, the quality of the translations generated by Transcoder and Transcoder-ST is substantially lower on the Python to C++ tasks. This evidence is supported by the fact that BatFix performs significantly better at fixing Codex translated programs. Second, we hypothesize that BatFix works best when the source and target languages are similar and the control-flow graphs are better aligned. To substantiate this hypothesis, we assessed the quality of the MaxSAT optimal solutions, which evaluate control flow similarity by measuring the edit distance between corresponding statements. Lower values signify a greater similarity of matching statements, while higher values suggest considerable differences, either due to distinct language structures or because of poor statement alignment. As depicted in Figure 8, Java data points predominantly cluster towards the left, suggesting a higher matching quality and greater language similarity, while Python solutions are noticeably skewed to the right, reflecting less optimal matching.

6.2.3 RQ4: Fault Localization. We compare BatFix fault localization procedure with traditional **Spectrum Based Fault Localization (SBFL)**, a family of techniques that uses test suites to rank source code lines by a computed "suspiciousness" score. We compared to five well-studied SBFL techniques [55], as listed in Table 4.

Table 4. Accuracy of Each Fault Localization Method on the Semantics Dataset from the Java to C++ Task

	% Programs where the bug is localized		
Method	TRANSCODER	TRANSCODER-ST	CODEX
Tarantula [30]	15.8%	15.4%	18.8%
Ochiai [1]	28.9%	25.6%	34.4%
Op2 [50]	23.7%	28.1%	31.2%
Barinel [2]	15.8%	15.4%	18.8%
DStar[1] [72]	21.1%	23.1%	31.2%
BATFIX	**55.2%**	**58.97%**	**65.6%**

We manually annotated all the buggy statements for each program in the three datasets of semantically incorrect programs for the Java to C++ task.[4] Although this task is time-consuming, the programs are small, and a direct manual comparison to the reference solutions allows a straightforward pinpoint of buggy lines. We associate faults of omission (i.e., lack of variable initialization) with the variable declaration in question. We consider localization successful if the top-1 suspicious line (from SBFL) or the single first identified line (for BATFIX) is one of the buggy lines (of possibly many). Note that BATFIX typically localizes faults iteratively after repair; however, here, we only run our own localization procedure once. Although conservative, this provides consistent, comparable settings between SBFL and our technique.

Table 4 summarizes results, showing that BATFIX outperforms the SBFL methods. BATFIX can locate faults due to control flow divergence, like SBFL can, but also problems due to variable initializations, or buggy statements due to wrong method calls (e.g., sorting an array in increasing order instead of decreasing), by inspecting runtime values (where SBFL only focuses on control flow). We also note that in all cases but one (which would require a four-line repair) whenever BATFIX successfully localizes a bug, it succeeds at repair. We believe this is because both fault localization and the synthesis depend on control-flow matching accuracy.

6.2.4 RQ5: Bugs BATFIX *Can and Cannot Fix.* First, BATFIX can produce patches that modify multiple lines of code. This is a challenging long-standing problem in general program repair. For instance, there were six programs transpiled with CODEX from Java to C++ that BATFIX repaired by patching two lines of code. When using TRANSCODER-ST to transpile Java to C++, there were four programs that BATFIX repaired by patching two lines of code and two programs that required fixing three lines of code. Similarly, when using TRANSCODER, five programs need two-lines repair and two programs that require three-lines repair. Table 5 shows statistics regarding the number of times each rule is triggered during the repair of syntax errors for the Java to C++ tasks.

In terms of bug types, Listing 5 shows an example of a program with incorrect variable initializations that BATFIX fixes with changes to two lines of code. In this case, the program compiles, and our fault localization procedure based on program trace pinpoints two variables that had been incorrectly initialized: `profit` (Line 2) and `prevDiff` (Line 6). Using the values of the corresponding variables from the trace of the original program (i.e., the reference solution), BATFIX correctly initializes the variables.

Listing 6 shows another example of a bug and patch BATFIX generated for a transpiled program with both syntax and semantic errors. Compiling the original program using Clang (with the red lines that start with "-") generated the following error message at Line 3: "**error: cannot initialize a variable of type 'char *' with an rvalue of type 'const char *'**". This bug corresponds to one

[4]We did not consider the programs with syntactic errors, because SBFL requires programs to be executable and testable.

Table 5. Number of Times Each Rule Was Triggered when Fixing Syntax Errors for the Java to C++ Datasets

Error Type	# Fixed
#1: Small typos the compiler can fix	16
#2: References to unknown/undeclared types	14
#3: Type declaration does not match assignment	10
#4: Random keyword (public, static)	14
#5: Use of undeclared identifiers	75
#6: Other errors that the compiler can bound	81

Listing 5. Transpiled program with initialization bugs

```
1  int prices (int price[], int n, int k)
       {
2    int profit[k+1][n+1];
3  + memset(profit, 0, sizeof(profit));
4    (...)
5    for (int i = 1; i <= k; i++) {
6  -   int prevDiff = INT_MAX;
7  +   int prevDiff = -2147483648;
8      for (int j = 1; j < n; j++) {
9        prevDiff = max(prevDiff, ...);
10 (...)
```

Listing 6. Transpiled program with syntax errors

```
1    string replace(string s, char c1,
         char c2) {
2      int l = s.length();
3    - char* arr = s.c_str();
4    + auto arr = s.c_str();
5      for (int i = 0; i < l; i++) {
6        if (arr[i] == c1) {
7    -     arr[i] = c2;
8    +     s[i] = c2;
9        } else if (arr[i] == c2) {
10   -     arr[i] = c1;
11   +     s[i] = c1;
12   (...)
```

Fig. 9. Two source code examples of programs that BatFix successfully repairs from the Java to C++ tasks transpiled by Transcoder. The programs are partially omitted due to their length.

of the syntax error rules we can fix using rule #3 from Table 1. Therefore, BatFix replaces the type declaration with the "auto" keyword. However, attempting to compile this program again with the new code generates two other error messages at Line 7 and 10: "**error: read-only variable is not assignable**." These are type #6 errors from Table 1. Therefore, BatFix replaces both lines with placeholders, generating a program with semantic errors. Subsequently, BatFix synthesizes appropriate replacements that comply with the test cases (Line 8 and 11).

There are instances where BatFix falls short, primarily due to poor-quality translations or inadequate control-flow alignments. Figure 9 demonstrates this limitation with a Python program (Listing 7) and its poor Transcoder-ST translation to C++ (Listing 8), where the model produces an excessively long and irrelevant translation. BatFix cannot fix such bugs, where the translation quality is degraded, and the repair task exceeds the threshold of typical bug fixing, leading to situations where substantial portions of the code would need to be rewritten or deleted to provide accurate functionality. In essence, BatFix's approach to repair is affected by both limitations of control flow matching but also by translation quality.

6.2.5 RQ6: Comparison with State-of-the-art Large Language Models. More recently, **large general-purpose language models (LLMs)**, specifically in the GPT family, have been shown very effective on code tasks. They therefore serve as an interesting baseline for BatFix's task, that is, repairing the results of transpilation. As a caveat, this experiment should be interpreted

Listing 7. Source program from the Python benchmark set.

```python
def check(arr, n):
  if n == 1: return True
  arr.sort()
  d = arr[1] - arr[0]
  for i in range(2, n):
    if arr[i] - arr[i - 1] != d:
      return False
  return True
```

Listing 8. Transcoder-ST translation to C++ of the source program in Listing 7.

```cpp
bool check(int* arr, int n) {
  if ((n == 1) || (n == 2) ||
      (n == 3) || (n == 4) ||
      (n == 5) || (n == 6) ||
      (n == 7) || (n == 8) ||
      (n == 9) || (n == 10) ||
      (n == 11) || (n == 12) ||
      (n == 13) || (n == 13)  (...)
```

Fig. 10. Python source code and corresponding C++ translation that BatFix cannot repair. In this low-quality translation, the model repetitively generates a never-ending if condition unrelated to the task at hand.

Chat API Prompt Template

System Message: You are a helpful assistant skilled in debugging and correcting C++ code.

User Prompt: I will give you a buggy program in C++. Your task is to find and fix the bugs. Buggy program:

{buggy code}

For your reference, the program was transpiled from *{language}* from this code snippet:

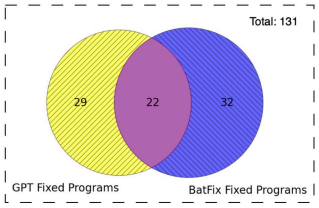
{correct code}

Please enclose the answer using ‘cpp ’."

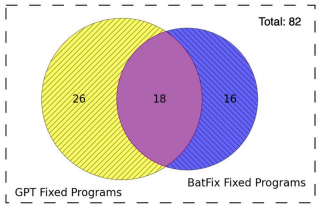
Fig. 11. Prompt template used for bug fixing in our experiments. The template incorporates dynamic placeholders (in italics) for runtime substitution with specific values.

cautiously: There is a significant risk of data leakage, given the training dates of both GPT-4 and GPT-3.5 with respect to the development of our dataset (which we discuss further in Section 7). We chose GPT-3.5 as a baseline, as the risk of memorization is somewhat lower (though still existent) compared to GPT-4. *We used the same bug datasets as our previous experiments (originating from Transcoder, Transcoder-ST, Codex)* and tasked GPT-3.5 with the same setup as given to BatFix (i.e., given the original program, repair the buggy transpilation). We crafted a prompt using best practices from existing research and documentation. The prompt template is as shown in Figure 11.

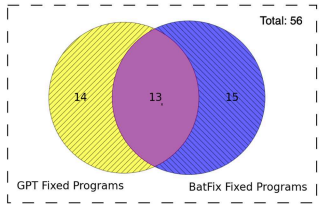
Figure 12 shows the effectiveness of GPT-3.5 and BatFix in correcting programs that were initially translated from Java to C++ using different transpilation models. We can see that GPT-3.5 fixed 51 out of 131 programs from Transcoder, 44 out of 82 for Transcoder-ST, and 27 out of 56 for Codex (total numbers of buggy programs are from Table 2). We make several observations. First, GPT-3.5 fixes a comparable number of programs for all three models in the Java to C++ task. Second, the types of bugs that GPT-3.5 fixes differ from those that BatFix can address (i.e., a large portion of the errors BatFix fixes are not addressed by GPT-3.5 and vice versa). This suggests that



(a) Buggy programs from TRANSCODER translations

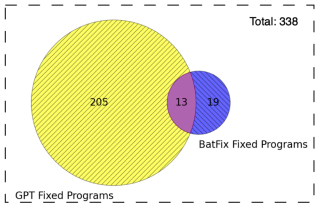


(b) Buggy programs from TRANSCODER-ST translations

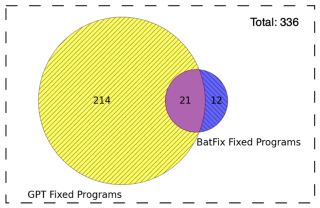


(c) Buggy programs from CODEX translations

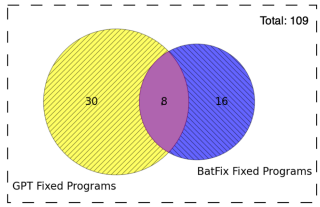
Fig. 12. Venn diagrams showing the number of programs fixed using BATFIX and GPT3.5 for the programs translated from **Java** to C++. The buggy programs are obtained from each model, as explained in Section 7.1.



(a) Buggy programs from TRANSCODER translations



(b) Buggy programs from TRANSCODER-ST translations



(c) Buggy programs from CODEX translations

Fig. 13. Venn diagrams showing the number of programs fixed using BATFIX and GPT3.5 for the programs translated from **Python** to C++. The buggy programs are obtained from each model, as explained in Section 7.1.

BATFIX is at worst complementary to GPT-3.5: Their performance is comparable, but they repair different mistakes. Additionally, GPT-3.5's performance may be influenced by memorization (see Section 7), and BATFIX requires significantly less computational power than GPT-3.5 (and is free).

Conversely, Figure 13 shows the effectiveness of GPT-3.5 and BATFIX in correcting programs from the Python to C++ task. We can see that GPT-3.5 fixed 218 out of 338 programs from TRANSCODER, 235 out of 336 for TRANSCODER-ST, and 38 out of 109 for CODEX (the total number of buggy programs comes from Table 3). Here, we observe a significant discrepancy between the performances of GPT-3.5 and BATFIX when fixing the bugs produced by TRANSCODER and TRANSCODER-ST. This is expected, as GPT-3.5 is a more powerful model compared to the models considered in previous experiments. GPT-3.5 is unlikely to have made the same mistakes as TRANSCODER and TRANSCODER-ST in the first place. Moreover, since it has access to the original code (as seen in the prompt in Figure 11), GPT-3.5 may simply be transpiling from scratch, instead of trying to correct mistakes. BATFIX, however, as a formal approach, is tied to the translation quality of the original models (designed to make small changes and fix minor bugs in transpilation) and, therefore, is not able to fix as many bugs. This is further reinforced by the fact that for CODEX, the total number of bugs fixed by BATFIX and GPT-3.5 is similar (36 and 23, respectively), albeit fewer in number. However, it is important to note that BATFIX is able to fix different kinds of bugs.

To better understand the types of bugs that GPT-3.5 was unable to resolve but BATFIX successfully addressed, we conducted a manual analysis of the subset of bugs exclusively fixed by BATFIX. Specifically, we focused on the bugs uniquely addressed by BATFIX in the Java to C++ task, as shown in Figure 12, corresponding to 32 bugs from the TRANSCODER dataset, 16 from the TRANSCODER-ST dataset, and 15 from CODEX.

We discovered that 37 programs (58.7%) that GPT-3.5 could not fix, but BatFix did, were related to the proper initialization of variables within the programs. Our analysis identified four distinct patterns in the model's behavior: (1) In some instances, GPT-3.5 either did not recognize or overlooked the need to initialize a variable or incorrectly believed the faulty code was elsewhere. For example, arrays declared on the stack in C++ are not automatically initialized to zero as in Java, thus they need to be initialized. (2) In other cases, GPT-3.5 partially initialized variables (e.g., only the first element of an array to 0), leading to corrupted data when using, for example, the += operator on uninitialized values). (3) The model occasionally attempted fixes using illegal statements (e.g., initializing variable-length arrays with `int arr[n] = {0}`, which is not permissible in C++). (4) GPT-3.5 also initialized variables with incorrect values. In contrast, BatFix successfully detected and corrected the bugs. This is expected, as these are the kind of subtle bugs that BatFix was designed to fix. Moreover, statically detecting that a variable is uninitialized, partially initialized, or improperly initialized is something traditional approaches like BatFix are well-suited for. In particular, BatFix uses mappings between the control-flow, variables, and executions traces of the original and transpiled programs to automatically detect discrepancies in the frame of the programs for particular inputs.

We identified 20 programs (31.7%) with syntax errors that GPT-3.5 was unable to fix. Although we did not identify a particular trend, we noticed that often the bugs corresponded to illegal statements that would be flagged by the type checker (type #3 and #6 errors from Table 1), either due to incorrect initializations or due to type mismatches in a computation. In these cases, GPT-3.5 would fail to correct the mistake or introduce a new one. We also noticed that GPT-3.5 was not removing incorrect keywords (e.g., static and public) from the function definitions, causing compilation to fail. Inspecting the patches produced by GPT-3.5, we also noticed that in five instances GPT-3.5 did not follow instructions properly when attempting to fix syntax bugs. However, BatFix fixed these issues using compiler hints or replacing the faulty lines with placeholders and subsequently finding a correct replacement for the placeholder, as detailed in Section 4.1.

Finally, in six programs (9.6%) the bugs corresponded to an incorrect computation. In these instances, the model attempted to patch the bug but failed to either accurately locate the faulty statement or correct it appropriately. In these cases, BatFix managed to fix the programs by locating the bug using the trace-based fault localization approach and replacing the faulty statements with a sketch generated from the source program.

7 LIMITATIONS AND DISCUSSION

We present the main limitations of our approach as well as avenues for future work.

7.1 Benchmarks and Models

Evaluating transpilation is difficult due to the limited availability of equivalent program pairs written in different languages. Consequently, we rely on well-documented datasets that include evaluation test cases specifically designed for this type of research. For this work, we use the dataset developed by the Transcoder [61] authors and are grateful to them for creating and providing such a valuable resource.

Since the programs in our benchmark set were previously used to evaluate state-of-the-art transpilation models [7, 62], the mistakes the models made are likely indicative of how the models generally behave. Moreover, we evaluated BatFix on the output of three different machine translation models of different sizes, and Codex is one of the most widespread language models for source code (e.g., it powers GitHub Copilot [20]). Therefore, we believe that the models chosen to generate the dataset of bugs are representative of what language models, in general, would

produce. However, recall we limited BatFix's to a 5-minute timeout, suggesting larger programs may be feasible in still-reasonable timescales.

7.2 Comparison against OpenAI GPT Family

New general-purpose large language models, including OpenAI's `GPT-3.5` and `GPT-4`, have shown promising results in various code tasks, including in APR. However, evaluating these models is challenging. They are often trained on vast quantities of data, including datasets such as CommonCrawl [12] and The Stack [33], making them susceptible to data contamination during evaluation.

One particular challenge is their likely exposure to all publicly available datasets used for transpilation and APR, making objective evaluation difficult. This issue is more pronounced in closed-source models like the OpenAI GPT family, which are opaque in many aspects (i.e., the model architecture and training data sources are mostly unknown). Researchers have shown that OpenAI GPT models excel in tasks such as Codeforces and HackerRank problems up to a specific date (September 2021), but their performance significantly deteriorates afterward [60]. Moreover, large language models' performance has been shown to heavily correlate with their compression ability [14]. Thus, model capability can often be mistaken for its memorization ability. To mitigate this problem, researchers [58] have attempted to evaluate LLMs on more recent datasets, but this is hampered by continuous model updates (and the challenges of developing new datasets for transpilation, in our case). For example, as of the time of publishing this article (April 2024), the training data for `GPT-4` extends up to December 2023. In contrast, the data for `GPT-3.5` only goes up to September 2021. Our dataset [19], a prominent one used for transpilation evaluation (from 2019), is part of The Stack [33], and thus it is also affected by data leakage issues.

We therefore interpret the results of our `GPT-3.5` comparison cautiously and emphasize the importance of experiments that rely on open-source models. Nonetheless, we noticed that a significant portion of the bugs BatFix was able to fix, but `GPT-3.5` could not (even with the data leakage risk), were related to initialization issues. While it might be argued that `GPT-3.5` could address these problems with a more tailored prompt or other techniques like chain of thought [71], BatFix is not only more cost-effective but also specifically designed to identify and correct such errors, making it a preferable choice over more complex approaches. Thus, we believe BatFix is at worst complementary to `GPT-3.5`.

7.3 Control Flow Matching in Varied Transpilation Contexts

While BatFix and its fault localization approach perform well in our benchmark evaluations (particularly for the `Java` to `C++` task), its foundational hypothesis—that similar functionalities imply similar control flows— has limitations in complex transpilation scenarios. For example, transpiling between languages with different threading models (like `Python`'s Global Interpreter Lock vs. `C++`'s threading libraries) is challenging due to lack of direct equivalence in control-flow. Moreover, when transpiling between syntactically distinct languages, finding equivalent language constructs (e.g., transpiling `Python`'s lambdas and list comprehensions to `C++`) is also challenging.

A potential way to mitigate some of these issues would be to first transform the code using a set of code transformation rules to make the code more idiomatic before attempting to match statement. We do this to a limited extent for `Python` to `C++`, such as to transform `Python` idiomatic for loop structures (e.g., `for i in range(n)`) to classic `C++` loops (e.g., `for(int i = 0; i < n; i++)`). Recent work [15] has shown that `Python` code can be refactored to be more similar to `Java` before performing code analysis. A similar approach could improve our synthesis algorithm for `Python`.

Although BatFix's basic premise may not be universally applicable, our results demonstrate it is effective for closely related languages, such as `Java` and `C++`.

7.4 Executing and Comparing Transpiled Code

BatFix requires sample test inputs to decide whether the transpilation is equivalent to the source code. The inputs used in our evaluation were automatically generated, and we used the original program's output as the test oracle. We believe it is reasonable to assume that for code considered sufficiently mature to justify transpilation, a user will have an idea of expected inputs, and the original program provides an oracle. This can be supplemented by differential testing techniques [43].

BatFix compares program traces across programming languages to find divergence points where states do not match. This was not an issue for our benchmark set, because most programs only use primitive types or standard library objects. If and when transpilation technology matures to produce user defined objects, then we expect it will be possible to compare them across languages as compositions of primitive types.

We also assume that it is possible to run transpiled code independently of rest of the codebase. We believe this is a reasonable assumption, as it allows for a gradual migration process, starting with the code that has no dependencies and moving on to more complex ones. Migrations involving circular dependencies (e.g., class A depends on class B and vice versa) would not be supported.

7.5 Fault Localization and Synthesis

If the control flow matching model is incorrect, then fault localization can fail. BatFix currently only asks the MaxSAT solver for the best matching model. However, asking the solver to enumerate multiple matching models is possible. This would resemble traditional SBFL, returning a list of potentially faulty locations, instead of just one.

Code generation similarly relies on control flow matching. In addition to the quality of the matching model, BatFix also relies on the similarity of statements across languages. BatFix's current implementation only works if the languages are syntactically similar, since we use corresponding statements structure as sketches; this is reasonable in this context where, e.g., the Transcoder family is an unsupervised method that uses "anchor" tokens to translate across languages.

7.6 Generalizability

We have shown BatFix effectiveness on two transpilation tasks (`Java` and `Python` to `C++`) and three language models. Although BatFix performs better when the source and target programming languages are more similar, it still performs reasonably well on `Python` to `C++`. We partially attribute the lower repair rate for Transcoder and Transcoder-ST to the low quality of the initial translations; BatFix's is to some degree constrained by the quality of the starting transpiled program, and it performs rather better on Codex accordingly.

That said, BatFix can consistently repair transpilation tasks for different language models and appears particularly effective at repairing semantic errors that are more challenging for language model–based transpilers. Moreover, our experiments show that the types of bugs BatFix can fix differ from those large language models can tackle.

8 RELATED WORK

In this section, we present a brief overview of relevant prior work related to our approach.

8.1 Automated Program Repair

Automated Program Repair [38] describes techniques that aim to automatically fix bugs in programs. Similarly to BatFix, APR tools such as GenProg [21] and ACS [74] also follow a generate-and-validate approach, dividing the repair task into fault localization, patch generation, and testing. However, traditionally, APR tools do not take as input a reference solution. The closest work to

ours are References [44, 70], which take a reference solution as input, albeit in the same programming language (whereas our input is in a different one). Other approaches to APR, such as S3 [37] and Angelix [46], are semantic-based and extract constraints from test cases to substitute faulty statements. Techniques such as DeepFix [24] and DeepDelta [48], or LLM-based approaches [17], aim to correct syntax errors specifically; fine-tuning these kinds of neural models to the types of mistakes made by MT-transpilers may enable BatFix to more effectively address syntax errors.

8.2 Repairing Language Model Code

Language models for source code often treat code as regular natural language text [4, 8]. This approach is problematic because it overlooks the structured nature of code and its precise syntax. One common problem with language models is that they cannot guarantee the correctness of the generated code. Thus, recent efforts have been made to improve the code generated by these models. Like BatFix, Jigsaw [26] also aims to fix code generated by machine learning models, specifically for Python code that uses the pandas API. However, our context is different, being agnostic to libraries or APIs used.

Other approaches augment the training data of the language model with static analysis data and compiler error messages [35]. Mukherjee et al. [49] train an NSG using weak-supervision by leveraging information from a static analyzer to augment training data. Despite training a model orders of magnitude smaller compared to models like Codex, NSG outperforms large language models in terms of the number of syntax and semantic errors generated. Synchromesh [56] takes a different approach by constraining the language models to only valid or syntactically correct programs. Along related conceptual lines, NGST2 [41] is a framework that combines program synthesis and machine learning for transpilation between imperative and functional languages.

Much of this work focuses more directly on reducing syntax errors; we show the value of coupling this type of code generation with techniques that can also reason about and correct semantic mistakes.

8.3 Fault Localization

Localizing bugs is essential to program repair. **Spectrum-based fault localization (SBFL)** [1, 13, 31] is a common family of approaches to this problem that uses a test suite. BatFix goes beyond these assumptions to leverage the existence of an input/reference solution, and we show that this additional information benefits the fault localization task. BugAssist [32] also uses MaxSAT to localize bugs by reasoning about program traces in passing and failing test cases but does not leverage an alternative oracle program.

8.4 Transpilation

Using machine learning models for source-to-source transformations is becoming increasingly popular, which supports the need for tools such as BatFix. Transcoder [61, 62] is one of the most recent examples. Similarly, Reference [9] proposes to use neural networks to transpile between Coffescript and JavaScript. NGST2 [41] is a framework that combines program synthesis and machine learning for transpilation between imperative and functional languages.

Previous work on mining API mappings across libraries could also enhance the BatFix repair strategy. By extracting API mappings, BatFix could refine or prioritize the search space of potential alternative APIs to replace erroneous calls. Specifically, DeepAM [22] learns to map between APIs in different languages, while SOAR [52] leverages API documentation to learn API mappings. Integrating these techniques could bolster BatFix's repair capabilities.

9 CONCLUSION

Transpilation is often a laborious, manual task, and automating transpilation using machine learning for languages not specifically designed for this purpose (like Java, and C++) has been growing in popularity. However, LM-generated code can contain subtle problems. We argue that machine learning methods should collaborate with classical analyses to enhance usability. Indeed, while LMs for transpilation are improving, the types of bugs that they still cannot avoid are particularly well-suited for our paradigm. We proposed BatFix—a novel tool designed to fix common errors generated by LMs. BatFix uses a new fault localization approach that leverages the fact that we have access to a reference solution. Our synthesis method also uses the reference solution to search for replacements for buggy code. Our evaluation shows that BatFix fixes 50% of buggy programs generated from Codex state-of-the-art transpilation model on a Java to C++ dataset and 22% on a Python to C++ dataset. Futhermore, we have shown BatFix works with different language models and transpilation tasks and can repair programs that require patching multiple lines of code.

DATA AVAILABILITY

We provide the source code and logs of our experiments [57].

ACKNOWLEDGMENT

All statements are those of the authors and do not necessarily reflect the views of any funding agency.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Proceedings of the Academic and Industrial Conference Practice and Research Techniques: Testing (TAIC PART-Mutation'07)*, 89–98. DOI: https://doi.org/10.1109/TAIC.PART.2007.13

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Computer Society, 88–99. DOI: https://doi.org/10.1109/ASE.2009.25

[3] Roee Aharoni, Melvin Johnson, and Orhan Firat. 2019. Massively multilingual neural machine translation. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 3874–3884. DOI: https://doi.org/10.18653/v1/n19-1388

[4] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR* abs/2108.07732 (2021).

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, Yoshua Bengio and Yann LeCun (Eds.). Retrieved from http://arxiv.org/abs/1409.0473

[6] Laura Batterink and Helen J. Neville. 2013. The human brain processes syntax in the absence of conscious awareness. *J. Neurosci.* 33, 19 (05 2013), 8528–8533. DOI: https://doi.org/10.1523/JNEUROSCI.0618-13.2013

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'20)*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). Retrieved from https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias

Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR* abs/2107.03374 (2021).

[9] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the Annual Conference on Neural Information Processing Systems*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 2552–2562. Retrieved from https://proceedings.neurips.cc/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html

[10] Aurelien Coe. 2024. *StatiCFG: Python3 Control Flow Graph Generator*. University of Geneva. Retrieved from https://github.com/coetaur0/staticfg

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1989. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company.

[12] Common Crawl. 2024. Common Crawl. Retrieved from https://commoncrawl.org/

[13] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. In *Proceedings of the European Conference on Object-Oriented Programming (Lecture Notes in Computer Science,* Vol. 3586*)*, Andrew P. Black (Ed.). Springer, 528–550.

[14] Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. 2023. Language modeling is compression. *CoRR* abs/2309.10668 (2023).

[15] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering repetitive code changes in Python ML systems. In *Proceedings of the International Conference on Software Engineering*. ACM, 736–748. DOI: https://doi.org/10.1145/3510003.3510225

[16] Mary T. Dzindolet, Scott A. Peterson, Regina A. Pomranky, Linda G. Pierce, and Hall P. Beck. 2003. The role of trust in automation reliance. *Int. J. Hum. Comput. Stud.* 58, 6 (2003), 697–718. DOI: https://doi.org/10.1016/S1071-5819(03)00038-7

[17] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*. IEEE, 1469–1481. DOI: https://doi.org/10.1109/ICSE48619.2023.00128

[18] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 422–436. DOI: https://doi.org/10.1145/3062341.3062351

[19] GeeksForGeeks. 2024. *GeeksforGeeks: A Computer Science Portal for Geeks*. Retrieved from https://www.geeksforgeeks.org

[20] GitHub. 2024. *GitHub Copilot*. Retrieved from https://github.com/features/copilot

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* 38, 1 (2012), 54–72. DOI: https://doi.org/10.1109/TSE.2011.104

[22] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Carles Sierra (Ed.). ijcai.org, 3675–3681. DOI: https://doi.org/10.24963/ijcai.2017/514

[23] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program synthesis. *Found. Trends Program. Lang.* 4, 1–2 (2017), 1–119.

[24] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351. Retrieved from http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[25] Anna Irrera. 2017. *Banks Scramble to Fix Old Systems as IT "Cowboys" Ride into Sunset*. Reuters. Retrieved from https://reut.rs/3RnBuqw

[26] Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the International Conference on Software Engineering*. ACM, 1219–1231. DOI: https://doi.org/10.1145/3510003.3510203

[27] Java2CSharp. 2024. *Java 2 CSharp Translator for Eclipse*. Retrieved from https://sourceforge.net/projects/j2cstranslator/

[28] JDB. 2024. *JDB: The Java Debugger*. Oracle. Available: Retrieved from https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

[29] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering*, David

Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 672–681. DOI: https://doi.org/10.1109/ICSE.2013.6606613

[30] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 273–282. DOI: https://doi.org/10.1145/1101908.1101949

[31] James A. Jones, Mary Jean Harrold, and John T. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*. ACM, 467–477.

[32] Manu Jose and Rupak Majumdar. 2011. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Mary W. Hall and David A. Padua (Eds.). ACM, 437–446. DOI: https://doi.org/10.1145/1993498.1993550

[33] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* abs/2211.15533 (2022).

[34] Rafal Kocielnik, Saleema Amershi, and Paul N. Bennett. 2019. Will you accept an imperfect AI?: Exploring designs for adjusting end-user expectations of AI systems. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI'19)*, Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos (Eds.). ACM, 411. DOI: https://doi.org/10.1145/3290605.3300641

[35] Tomasz Korbak, Hady Elsahar, Marc Dymetman, and Germán Kruszewski. 2021. Energy-based models for code generation under compilability constraints. *CoRR* abs/2106.04985 (2021).

[36] Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Unsupervised machine translation using monolingual corpora only. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=rkYTTf-AZ

[37] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the ACM SIGSOFT Foundations of Software Engineering Conference*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. DOI: https://doi.org/10.1145/3106237.3106309

[38] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. DOI: https://doi.org/10.1145/3318162

[39] Chu Min Li and Felip Manyà. 2009. MaxSAT, hard and soft constraints. In *Handbook of Satisfiability*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). *Frontiers in Artificial Intelligence and Applications*, Vol. 185. IOS Press, 613–631. DOI: https://doi.org/10.3233/978-1-58603-929-5-613

[40] LLDB. 2024. *LLDB: Scripting Bridge API*. Retrieved from https://lldb.llvm.org/design/sbapi.html

[41] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Isil Dillig. 2022. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. DOI: https://doi.org/10.1145/3527315

[42] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. 2014. Open-WBO: A modular MaxSAT solver. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT'14) held as part of the Vienna Summer of Logic (VSL'14) (Lecture Notes in Computer Science,* Vol. 8561*)*, Carsten Sinz and Uwe Egly (Eds.). Springer, 438–445. DOI: https://doi.org/10.1007/978-3-319-09284-3_33

[43] William M. McKeeman. 1998. Differential testing for software. *Digit. Technol. J.* 10, 1 (1998), 100–107. Retrieved from http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[44] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 129–139. DOI: https://doi.org/10.1145/3180155.3180247

[45] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for simple program repairs. In *Proceedings of the International Conference on Software Engineering*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 448–458. DOI: https://doi.org/10.1109/ICSE.2015.63

[46] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 691–701. DOI: https://doi.org/10.1145/2884781.2884807

[47] Troy Melhase. 2024. *Java2Python: Simple but Effective Tool to Translate Java Source Code into Python*. Java2Python. Retrieved from https://github.com/natural/java2python

[48] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to repair compilation errors. In *Proceedings of the ACM SIGSOFT Foundations of Software Engineering Conference (ESEC/FSE'19)*. Association for Computing Machinery, New York, NY, 925–936. DOI: https://doi.org/10.1145/3338906.3340455

[49] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. 2021. Neural program generation modulo static analysis. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'21)*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 18984–18996. Retrieved from https://proceedings.neurips.cc/paper/2021/hash/9e1a36515d6704d7eb7a30d783400e5d-Abstract.html

[50] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32. DOI: https://doi.org/10.1145/2000791.2000795

[51] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical learning approach for mining API usage mappings for code migration. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 457–468. DOI: https://doi.org/10.1145/2642937.2643010

[52] Ansong Ni, Daniel Ramos, Aidan Z. H. Yang, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2021. SOAR: A synthesis approach for data science API refactoring. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*. IEEE, 112–124. DOI: https://doi.org/10.1109/ICSE43902.2021.00023

[53] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. SPOON: A library for implementing analyses and transformations of Java source code. *Softw.: Pract. Exper.* 46 (2015), 1155–1179. DOI: https://doi.org/10.1002/spe.2346

[54] PDB. 2024. *PDB: The Python Debugger*. Python Software Foundation. Retrieved from https://docs.python.org/3/library/pdb.html

[55] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the International Conference on Software Engineering*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 609–620.

[56] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *Proceedings of the International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=KmtVD97J43e

[57] Daniel Ramos. 2024. *BatFix's source code*. Carnegie Mellon University. Retrieved from https://figshare.com/s/014bb66aa13c6ee0a0c3

[58] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco M. Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining effective lightweight transformations from pull requests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE'23)*. IEEE, 1516–1528. DOI: https://doi.org/10.1109/ASE56229.2023.00117

[59] Daniel Ramos, Jorge Pereira, Inês Lynce, Vasco M. Manquinho, and Ruben Martins. 2020. UNCHARTIT: An interactive framework for program recovery from charts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 175–186. DOI: https://doi.org/10.1145/3324884.3416613

[60] Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. 2023. Data contamination through the lens of time. *arXiv preprint arXiv:2310.10628* (2023).

[61] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the Annual Conference on Neural Information Processing Systems*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). Retrieved from https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[62] Baptiste Rozière, Jie Zhang, François Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2022. Leveraging automated unit tests for unsupervised code translation. In *Proceedings of the International Conference on Learning Representations*. OpenReview.net. Retrieved from https://openreview.net/forum?id=cmt-6KtR4c4

[63] Xujie Si, Xin Zhang, Radu Grigore, and Mayur Naik. 2017. Maximum satisfiability in software analysis: Applications and techniques. In *Proceedings of the International Conference on Computer-aided Verification (Lecture Notes in Computer Science,* Vol. 10426*)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 68–94. DOI: https://doi.org/10.1007/978-3-319-63387-9_4

[64] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495.

[65] The Clang Team. 2024. *LibTooling: Library to Support Writing Standalone Tools Based on Clang*. The Clang Team. Retrieved from https://clang.llvm.org/docs/LibTooling.html

[66] V. Javier Traver. 2010. On compiler error messages: What they say and what they mean. *Adv. Hum.-comput. Interact.* 2010, Article 3 (Jan. 2010), 26 pages. DOI: https://doi.org/10.1155/2010/602570

[67] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.). ACM, 332:1–332:7. DOI: https://doi.org/10.1145/3491101.3519665

[68] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 363–378. DOI: https://doi.org/10.1145/3314221.3314589

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Annual Conference on Neural Information Processing Systems*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[70] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 481–495. DOI: https://doi.org/10.1145/3192366.3192384

[71] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'22)*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[72] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar method for effective software fault localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308. DOI: https://doi.org/10.1109/TR.2013.2285319

[73] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Trans. Softw. Eng.* 42, 8 (2016), 707–740. DOI: https://doi.org/10.1109/TSE.2016.2521368

[74] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE/ACM, 416–426. DOI: https://doi.org/10.1109/ICSE.2017.45

[75] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE code generation from natural language: Promise and challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2 (2022), 29:1–29:47. DOI: https://doi.org/10.1145/3487569

[76] Biao Zhang, Philip Williams, Ivan Titov, and Rico Sennrich. 2020. Improving massively multilingual neural machine translation and zero-shot translation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL'20)*, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault (Eds.). Association for Computational Linguistics, 1628–1639. DOI: https://doi.org/10.18653/v1/2020.acl-main.148

[77] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. 2010. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 195–204. DOI: https://doi.org/10.1145/1806799.1806831

Received 3 February 2023; revised 29 March 2024; accepted 4 April 2024