OrdinalFix: Fixing Compilation Errors via Shortest-Path CFL Reachability

Wenjie Zhang[†], Guancheng Wang[†], Junjie Chen[‡], Yingfei Xiong[†], Yong Liu[§], Lu Zhang[†]*
[†] Key Laboratory of High Confidence Software Technologies (Peking University), MoE
School of Computer Science, Peking University, P. R. China
[‡]College of Intelligence and Computing, Tianjin University, P. R. China
[§]College of Information Science and Technology, Beijing University of Chemical Technology, P. R. China
zhang_wen_jie@pku.edu.cn, guancheng.wang@pku.edu.cn, junjiechen@tju.edu.cn,
xiongyf@pku.edu.cn, lyong@mail.buct.edu.cn, zhanglucs@pku.edu.cn

***Abstract*—The development of correct and efficient software can be hindered by compilation errors, which must be fixed to ensure the code's syntactic correctness and program language constraints. Neural network-based approaches have been used to tackle this problem, but they lack guarantees of output correctness and can require an unlimited number of modifications. Fixing compilation errors within a given number of modifications is a challenging task. We demonstrate that finding the minimum number of modifications to fix a compilation error is NP-hard. To address compilation error fixing problem, we propose OrdinalFix, a complete algorithm based on shortest-path CFL (context-free language) reachability with attribute checking that is guaranteed to output a program with the minimum number of modifications required. Specifically, OrdinalFix searches possible fixes from the smallest to the largest number of modifications. By incorporating merged attribute checking to enhance efficiency, the time complexity of OrdinalFix is acceptable for application. We evaluate OrdinalFix on two datasets and demonstrate its ability to fix compilation errors within reasonable time limit. Comparing with existing approaches, OrdinalFix achieves a success rate of 83.5%, surpassing all existing approaches (71.7%).**

***Index Terms*—Compilation Error, CFL Reachability**

I. Introduction

Errors are inevitable in software development. Manual error fixing is a time-consuming and error-prone process, especially when software becomes larger and more complex. Therefore, automated error fixing has attracted extensive attention in recent years [1]. In the field of automated error fixing, most research focuses on fixing logic errors in compilable programs and ignores a large class of errors occurring at the stage of compiling, i.e., compilation errors. Compilation errors are frequent in practice and typically due to inexperience of programmers or lack of attention to details, such as missing scope delimiters [2]. According to an existing study at Google [3], both novices and experienced developers make such errors. Also, more and more tools are designed for processing code snippets from the web, where compilation errors are common. Therefore, automated compilation-error fixing is a critical task.

For a typical programming language, compilable programs need to satisfy two types of constraints: syntactic constraints and semantic constraints. The syntactic constraints ensure that the given program conforms to some context-free grammar (CFG). The semantic constraints require that the program passes some checks for non-context-free properties, such as definition-and-use relations of variables and type matching relations in expressions. We should mention that, in this paper, our use of semantic constraints is focused on representing compiler constraints, rather than ensuring the program's functional correctness.

Given a non-compilable program, which violates some syntactic constraints or semantic constraints, or both, the task of fixing compilation errors in the program is to modify the program to change it into a compilable form. Typically, it is not expected that we massively change the program since such a way of modifying the program would have the changed program deviate largely from the original program. In an extreme case, if we delete all the tokens in the non-compilable program, what we obtain is a compilable but useless program. Therefore, in this paper, we define the problem of fixing compilation errors in a program as the problem of finding a minimum number of changes to make the program compilable.

Over the years, several approaches have been proposed for helping solve this problem. Parsing-stage-based approaches (see e.g., [4], [5]) provide fixing recommendations by analyzing the surrounding code of the error reported in the error message. Some of these approaches have been integrated into mature IDEs such as Eclipse. However, the effectiveness of these approaches is limited due to inaccurate error messages. Machine-learning-based approaches (see e.g., [6], [7]) predict fixes for compilation errors via learning from known fixes using some machine learning algorithms. However, the capability of these approaches highly depends on the sufficiency of the training dataset and thus the effectiveness is also limited. Furthermore, these approaches cannot guarantee fixing the errors with minimum changes. In the field of algorithms, given a context-free grammar (denoted as CF) and a string (denoted as p) such that $p \notin L(CF)$, Aho and Peterson [8] proposed an algorithm for calculating a modified string (denoted as p') with minimum edge distance, and demonstrated that the asymptotic execution time of the algorithm is $O(n^3)$, where n is the length of p (i.e., $|p|$). In principle, when we are facing only violations of syntactic constraints, this algorithm is applicable

*Corresponding author.

and guarantees to find fixes with minimum changes. However, this algorithm is not applicable when there are violations of semantic constraints. Furthermore, even if there are only syntactic errors, this algorithm is not able to guarantee that the resulting program satisfies all required semantic constraints.

In fact, fixing semantic errors is more complex than fixing syntactic errors. Fixing compilation errors for common programming languages, such as Java, is NP-hard, as shown in Section III. That is to say, to fix compilation errors, there is no complete algorithm that guarantees to produce minimum modifications in polynomial time unless P=NP. A naïve algorithm for compilation error fixing is to enumerate all syntactically correct fixes from fewer modifications to more modifications with a syntactic error fixing algorithm (e.g., [8]) and then check the compilability with a compiler. However, an exponential number of syntactically correct but semantically incorrect programs would be generated during fixing by the algorithm.

In real-world scenarios, addressing compilation errors through human intervention commonly takes a few seconds to several minutes [3]. Consequently, any algorithm designed to assist developers in compile error fixing should operate within a comparably brief time frame. Unfortunately, the previously mentioned naïve algorithm does not fulfill this requirement due to the rapid increase in potentially valid but semantically erroneous solutions, which leads to an exponential growth.

In order to reduce the time consumption of compilation error fixing, we propose OrdinalFix[1], which tries to perform a more fine-grained semantic checking with merging and branching reduction, so that the algorithm can complete in an acceptable period of time.

For syntactic analysis, modern compilers typically rely on LL-based or LR-based parsers to efficiently analyze program structures. These parsers can accurately process correct programs. However, for erroneous programs, they only contain a single parse state when encountering errors, which may lead to ignoring possible modifications that could fix the error. Although there are approaches to recovering from errors during parsing, they do not consider all possible fixes. To explore all possible fixes, we utilize a modified shortest-path CFL reachability algorithm. Although this algorithm has a higher time complexity than LL-based or LR-based parsers, it can capture all possible fixes.

For semantic constraints, we utilize the attribute grammar approach commonly used in compiler construction to perform attribute checking in CFL reachability. We encode semantic constraints into attributes and perform merged attribute checking on top of each reduction relationship obtained via CFL reachability on the modification graph. This approach allows us to effectively handle semantic constraints and ensure that the modified program satisfies all constraints.

We evaluate OrdinalFix on fixing compilation errors with two datasets, consisting of a synthesized dataset in an object-oriented language (i.e., Middleweight Java) and a real world dataset in a procedural language (i.e., C). The empirical results show that, for the Middleweight Java programs, OrdinalFix is able to fix all the programs within 600 seconds. For C programs the median time for fixing a function body is only 0.6 seconds, demonstrating the efficiency of OrdinalFix. Furthermore, we compare the fixing rate of OrdinalFix with that of existing approaches on the same C dataset, where OrdinalFix achieves a success rate of 83.5% (5,757 out of 6,978) for programs with compilation errors within 600 seconds, surpassing all existing approaches.

To sum up, we make the following contributions in this paper:

- We demonstrate the NP-hardness of the problem of fixing compilation errors with the minimum number of modifications.
- We propose a fine-grained algorithm (named OrdinalFix) for fixing compilation errors based on attribute checking over the CFL reachability graph, where we rely on merging and pruning to reduce its time consumption.
- We implement OrdinalFix for compilation error fixing and evaluate OrdinalFix on generated programs written in Middleweight Java and real-world C programs with compilation errors, demonstrating the effectiveness and efficiency of OrdinalFix.

II. Problem Formulation

When fixing compilation errors, it is typical that we want to modify the erroneous program as little as possible. That is, given a programming language, which is defined by some context-free grammar and some semantic constraints in form of typing rules and a program p that does not satisfy the syntactic constraints and/or the semantic constraints, the goal is to find a program p' that is closest to p to make $p' \in L(CF)$ established and p' satisfies the semantic constraints. It means that we are able to get p' from p by performing some minimum modifications on p.

Definition 1 (compilation error fixing)**.** *Given a programming language and a program p such that p does not compile, the problem of fixing compilation errors in p is to find program p' such that p' compiles and the edit distance between p and p' is minimum, i.e., for all p'' that compiles, $d(p, p') \leq d(p, p'')$, where $d(p_1, p_2)$ is the edit distance between p_1 and p_2.*

Here, the edit distance between two programs p_1 and p_2 accounts for the number of modifications required to transform p_1 into p_2. These modifications encompass token insertion, deletion, or replacement operations.

For the ease of presentation, we assume that the empty program always compiles. This assumption is reasonable because the empty program means a program doing nothing. Under the assumption, for any program p that does not compile, we can always find a fix with $|p|$ removal operations, where $|p|$ is the number of tokens in p. Usually, the empty program is not the fix we want, because it means to remove the whole program. However, the empty program gives an upper bound

[1]The source code and data of OrdinalFix is available at https://github.com/myxxxsquared/OrdinalFix

```
class InMIS {
    void addEdge(OutMIS obj);
}
class OutMIS extends InMIS {
    void addEdge(InMIS obj);
}
```

(a) External declarations

```
InMIS v1;
InMIS v2;
InMIS v3;
...
v2.addEdge(v3);
v2.addEdge(v3);
...
```

(b) Program

Fig. 1. Program to fix for MIS.

of compilation error fixing. Any program p can be fixed within $|p|$ modifications.

We also assume that the context-free grammar of a programming language is unambiguous, thus a program p will have at most one parse tree.

III. NP Hardness

According to an earlier study, finding fixes that conform to context-free grammars with the minimum number of modifications can be done within $\mathrm{O}(n^3)$ [8]; however, it would be much more complex to find fixes that are compilable. Below, we show that for a simple object-oriented language that allows local variable declarations, finding fixes that satisfy the typing semantic constraints with the minimum number of modifications is NP-hard. Specifically, we reduce the maximum independent set problem to the compilation error fixing problem.

The maximum independent set (MIS) is defined as follows. Given an undirected graph $G = (V, E)$, the problem is to find the largest subset of vertices $V' \subset V$ such that any two vertices in V' are not adjacent in the graph G. MIS is proven to be NP-complete.

Given a MIS problem, let us denote the graph in the MIS problem as $G = (V, E)$. We reduce the MIS problem to a problem of fixing compilation errors in a Java-like language. Fig. 1a depicts the external declaration of two classes (i.e., *InMIS* and *OutMIS*). This declaration ensures that *a.addEdge(b)* is legal based on the typing semantic constraints only if either *a* or *b* is of type *OutMIS*.

Based on this declaration, we construct a program snippet p with $(n + mn)$ lines (depicted in Fig. 1b), where n is the number of vertices in G, and m is the number of edges in G. Specifically, program snippet p consists of two parts, which have n and mn lines, respectively. For the first n lines, the content of the i-th line is "*InMIS v*$\{i\}$*;*", representing the i-th vertex in G. Here, $\{i\}$ means replacing with the number of i; for example, the third line is "*InMIS v3;*". The following mn lines range from line $(n + 1)$ to line $(n + mn)$. Given $1 \leq k \leq m$, let the k-th edge in G be in the form of (v_x, v_y), from line $(kn + 1)$ to line $(kn + n)$, the content is the same, i.e., "*v*$\{x\}$*.addEdge(v*$\{y\}$*);*". For example, if the first edge in the graph is (v_2, v_3), from line $(n+1)$ to line $2n$, the content of each line is "*v2.addEdge(v3);*".

In p, the first n lines define n variables for the n vertex in G. The remaining mn lines represent the edge constraints. It is obvious that p does not compile as long as the edge set of the graph is not empty. Each line after the first n lines has a typing error, because *InMIS.addEdge(InMIS)* is not defined in the external declaration. If we modify some of the first n lines by changing *InMIS* to *OutMIS*, the program may compile, because *InMIS.addEdge(OutMIS)*, *OutMIS.addEdge(InMIS)*, and *OutMIS.addEdge(OutMIS)* are all defined in the declaration.

Based on the preceding analysis, if the set of variables changed by a fix of p with a minimum number of modifications is V', $V - V'$ must be a maximum independent set; otherwise, a larger independent set would lead to a fix of p containing fewer modifications. Similarly, a maximum independent set in G also corresponds to a fix with a minimum number of modifications. Therefore, the maximum independent set problem is reducible to the compilation error fixing problem. That is to say, the problem of finding a minimum fix that compiles is NP-hard.

If we have a closer look at the program reduced from the maximum independent set problem, we know that the NP-hardness comes from the fact that there are up to an exponential number of different ways of variable declaration by modifying at most a polynomial number of places. In principle, type compliance should be checked against each way of variable declaration, incurring an exponential time complexity if $P \neq NP$.

IV. Preliminaries

In this section, we review two existing techniques that OrdinalFix is built on: CFL reachability and attribute grammar.

CFL reachability is a parsing algorithm that captures all possible fixes for a given erroneous program. It has a higher time complexity than LL-based or LR-based parsers, but it can account for all potential modifications needed to fix the error.

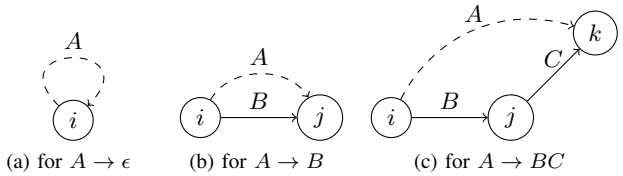
Attribute grammar is a way to encode semantic constraints into attributes and perform attribute checking on top of each reduction relationship obtained via a parsing algorithm. This technique can help ensure that any modifications made during error fixing maintain the program's semantic correctness.

A. CFL Reachability

The CFL reachability algorithm is used to determine whether there exists a path between two given nodes in a directed graph, such that the symbol string of the path conforms to a context-free grammar. This algorithm was first proposed for interprocedural program analysis [9].

To apply the CFL reachability algorithm, the grammar is first normalized into a normal form where the right-hand side of each production rule contains at most two symbols. This means that all production rules have one of the following three forms: $A \rightarrow \epsilon$, $A \rightarrow B$, or $A \rightarrow BC$, where A represents a non-terminal symbol and B or C represents a terminal or a non-terminal symbol.

Next, the algorithm iteratively adds new edges labeled with non-terminal symbols in the context-free grammar to the



(a) for $A \to \epsilon$ (b) for $A \to B$ (c) for $A \to BC$

Fig. 2. Edges caused by productions in CFL reachability.

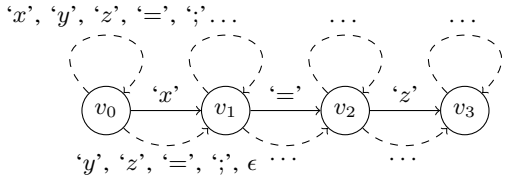


Fig. 3. Modification graph for $x = z$. Multiple symbols on the one edge indicate multiple edges.

directed graph G, with three principles. 1) If a production rule is of the form $A \to \epsilon$, then self-loop edges labeled with A are added to each vertex in the initialization stage, as shown in Fig. 2a. 2) If the production rule is of the form $A \to B$, a new edge labeled with A is added from vertex v_i to vertex v_j, where the original edge between v_i and v_j is labeled with B, as shown in Fig. 2b. 3) If the production rule is of the form $A \to BC$, a new edge labeled with A is added from vertex v_i to vertex v_k, where the original edges between v_i and v_j and between v_j and v_k are labeled with B and C, respectively, as shown in Fig. 2c.

The algorithm maintains a work list to record changes and determine where to add edges caused by productions $A \to B$ and $A \to BC$, and it terminates when no more edges can be added.

Finally, the CFL reachability algorithm checks if there exists an edge labeled with the start symbol after the loop exits.

B. Shortest-Path CFL Reachability.

Given a weighted graph, the shortest-path CFL reachability algorithm is to find the shortest-path that conforms to the context-free grammar. The shortest-path CFL reachability was first proposed to infer missing specifications in Android libraries [10].

The shortest-path CFL reachability algorithm is an extension to the basic CFL reachability algorithm. Instead of the work list, the shortest-path CFL reachability algorithm maintains a priority queue H, in which an element with a higher weight is served after an element with a lower weight, and introduces a map I to store the shortest path. In the initialization stage, H stores the edges with their weights in G labeled with terminal symbols, and I is empty. Then, additional edges are added to G according to the productions until no more edges can be added as in CFL reachability.

C. Attribute Grammar

For compiler construction, attribute grammars are used in addition to parsing to represent semantic information [11]. In attribute grammars, each symbol is associated with inherited and synthesized attributes. The inherited attributes are computed in a top-down manner, while the synthesized attributes are computed in a bottom-up manner. Rules are defined to verify the compatibility of the given attributes and then compute the inherited/synthesized attributes. If the attributes are incompatible, the compiler raises an error to indicate that the input program violates semantic constraints.

V. Approach

A. Overview

In this subsection, we provide a brief illustration of OrdinalFix using a toy example. The details of components of OrdinalFix will be described in the next subsections.

We consider a simple programming language that contains only one assignment statement, described by the following context-free grammar:

$$\mathcal{S} \to \texttt{identifier = identifier ;}$$

The statement contains a semicolon indicating the end of the statement.

Let us consider three identifiers in this language: x, y, and z. Identifiers x and y are of the same type denoted as type A, while z is of a different type denoted as type B. Supposing that only variables of the same type can be assigned to each other, x and y can be assigned to each other, but neither z can be assigned to x or y, nor x or y can be assigned to z.

We focus on fixing the following code snippet:

$$x = z \tag{1}$$

Code snippet (1) is not compilable due to two errors. First, there is no semicolon at the end of the statement, resulting in a syntactic error that violates the context-free grammar. Second, x and z are of different types and cannot be assigned to each other, resulting in a semantic error that violates the typing semantic constraints.

To fix compilation errors in the program, our approach, OrdinalFix, initially constructs a weighted modification graph as depicted in Fig. 3. The graph comprises original edges and candidate modification edges. Original edges correspond to the current tokens in the code snippet and are assigned a weight of 0 (solid edges in Fig. 3). Candidate modification edges denote possible modifications to the code snippet, including insertion (self-loops), deletion (edges from v_i to v_{i+1} with the empty symbol ϵ), and update (edges from v_i to v_{i+1} with a non-empty symbol). Candidate modification edges are assigned a weight of 1 (dashed edges in Fig. 3). After constructing the weighted modification graph, a path from the first vertex (v_0) to the last vertex (v_3) in the modification graph indicates a modified program, and the total weight of the path indicates the number of modifications required to transform the original code snippet into the modified program.

Subsequently, our approach attempts to find a compilation error fix by checking from a small number of modifications

to a large number of modifications. Specifically, OrdinalFix utilizes an adapted shortest-path CFL reachability algorithm on the graph to identify a group of programs that conform to syntactic constraints with the minimum number of modifications. Thereafter, OrdinalFix checks the semantic constraints of these programs through merged attribute checking. If any of these programs satisfies semantic constraints, OrdinalFix constructs the fix and returns the result. If not, OrdinalFix employs the adapted CFL reachability algorithm again to identify another group of programs with more modifications and verifies these programs with merged attribute checking. This process continues until a fix is found.

For the CFL reachability part, we made a slight modification to the shortest-path CFL reachability algorithm by storing not only the shortest path but also other paths. This modification enables our algorithm to find fixes with more modifications. For example, in the above illustration, the CFL reachability initially finds a program with one modification, $x = z;$ (inserting a semicolon at the end of the program). In the subsequent round, CFL reachability identifies several programs with two modifications. A comprehensive description of the weighted modification graph and the CFL reachability utilized in our approach will be provided in Section V-B.

For merged attribute checking, the process of merged attribute checking involves converting semantic constraints into attributes and subsequently verifying their correctness. To apply this technique to the aforementioned code snippet, each identifier is assigned a synthesized attribute indicating its type based on the corresponding edge in the CFL reachability graph (i.e., 'x' edges and 'y' edges with type value A, and 'z' edges with type value B). Then, OrdinalFix checks whether the types of the corresponding identifiers in statements match by checking their respective attributes. Further details regarding the process of merged attribute checking will be presented in Section V-C.

Attribute checking can combine branches from different locations, resulting in exponential time complexity relative to the weight of the program. To address this issue, we employ pruning and optimization techniques to scale type checking. It is important to note that since compilation error fixing is NP-hard, even with optimizations, the complete algorithm necessarily leads to exponential time complexity in the worst case, unless P = NP. Nevertheless, our experiments demonstrate that OrdinalFix can successfully identify fixes in a reasonable time. We will provide further details on the algorithm in the next subsections.

For the above example, OrdinalFix will find a fix with two modifications.

$$v_0 \xrightarrow{x} v_1 \xrightarrow{=} v_2 \xrightarrow{y} v_3 \xrightarrow{;} v_3$$

B. Dealing with Syntactic Constraints

To find syntactically correct fixes in compilation error fixing, i.e., finding programs that conform to the context-free grammar, we define a modification graph and then reduce the problem to shortest-path CFL reachability.

For a program p with n tokens, we construct a weighted modification graph. The weighted modification graph is a directed graph with each edge associated with a length (weight) l and a grammar symbol[2] t. The weighted modification graph consists of $n+1$ vertices (i.e., $v_0, v_1, \cdots, v_n$). Each edge of the weighted modification graph is represented by $\langle v_i, v_j, t, l\rangle$, where v_i and v_j is the original and the target vertices, t is the associated symbol and l is the weight of the edge.

The edges of weighted modification graph consist of four kinds of edges, one of them representing original tokens in the original program and the three others representing three kinds of modification operations. The weights of original edges are zero while the weights of other edges representing modification operations are one, so that the weight of each edge represents the number of modifications. The four kinds of edges are defined as below.

1) *original edge*: An original edge represents a token in the original program. For the i-th token t_i in original program, we have $\langle v_{i-1}, v_i, t_i, 0\rangle$ in the graph (denoted as E_M).
2) *insertion edge*: An insertion edge is represented by a self-loop edge labeled with the inserted symbol. For each vertex $v_i \in V$, edges in the form of $e_{ins} = \langle v_i, v_i, t_{ins}, 1\rangle$ is added to E_M, where t_{ins} is the inserted symbol before t_i.
3) *update edge*: An update edge is represented by an edge from vertex v_i to vertex v_{i+1} labeled with an update symbol. For each pair of vertices $v_i, v_{i+1} \in V$, edges in the form of $e_{upd} = \langle v_i, v_{i+1}, t_{upd}, 1\rangle$ is added to E_M, where t_{upd} is a symbol used to update t_i ($t_{upd} \neq t_i$).
4) *deletion edge*: A deletion edge is represented by an edge from vertex v_i to vertex v_{i+1} labeled ϵ. For each pair of vertices $v_i, v_{i+1} \in V$, edge $e_{del} = \langle v_i, v_{i+1}, \epsilon, 1\rangle$ is added to E_M.

According to the above construction, given a path from v_0 to v_n in G (denoted as P), if the weight of the path is k, P corresponds to k modifications to p.

By using shortest-path CFL reachability on the modification graph, we can generate CFL reachability graphs that contain the syntactic structures of both the original and modified programs. On the CFL reachability graph, an edge in the form of $e_s = \langle v_0, v_n, \mathcal{S}, k\rangle$, where v_0 is the first vertex, v_n is the last vertex and $\mathcal{S}$ is the start symbol, represents one or many syntactically correct programs with k modifications. We can construct syntactically correct programs from each e_s by recursively expanding productions of non-terminals. So, we refer to these e_s edges as root edges in the following discussion.

Using the shortest-path CFL reachability algorithm, we can enumerate root edges from shorter to longer and find syntactic fixes in order of increasing length. In the next subsection, we will further apply syntactic constraints to these fixes.

[2]A grammar symbol is a terminal symbol or a non-terminal symbol in the context-free grammar.

C. Dealing with Semantic Constraints

After obtaining root edges, in this subsection, we further utilize attribute checking on the reachability graph to identify semantic error fixes. This is achieved by encoding semantic constraints into attributes on the symbols, allowing for a more precise and effective error correction process.

1) Encoding Semantic Constraints into Attributes: We encode semantic constraints into attributes, which will be checked in the reachability graph to find shortest-path fixes. While global identifier tables are typically used during parsing for a single program, when searching for semantically correct programs on a CFL reachability graph, multiple potential fixes can exist, and the global identifier tables for different reachability paths may vary. Additionally, for the attribute checking algorithm to work effectively in merging and pruning, the attributes must be immutable, hashable, and comparable. To overcome these challenges, we develop a new method for encoding semantic constraints into attributes without relying on global identifier tables.

a) Declaration-Use Constraints: Declaration-use constraints are a fundamental aspect of programming languages, where the compiler detects an error when an undeclared identifier is used. In our approach, we encode declaration-use constraints in the reachability graph by introducing an inherited attribute to store declaration information, along with identifier selectors to determine whether an identifier can be used. Specifically, for each grammar symbol (i.e., a terminal or a non-terminal in the context-free grammar), we introduce an inherited attribute $idtab$ to store the identifiers defined before this symbol. For an identifier terminal x, we introduce another inherited attribute sel to contain possible selections for the identifier. Both $idtab$ and sel utilize an immutable map data structure to store the mapping from identifier names to their types and an immutable list data structure to store possible identifier names, respectively.

b) Expression Type Constraints: In most programming languages, expressions are associated with types, and operations between expressions must satisfy specific type constraints. These are known as expression type constraints in programming languages. To handle these constraints, we use a synthesized attribute $type$ to store the type of the expression. Computation of a new synthesized attribute checks whether the operation between sub-expressions is valid, and if so returns the type of the result. For example, for sum of two primaries $p = x + y$, the computation checks whether the type of x and y are compatible in an addition operation and returns the type of $a + b$ if a and b are compatible. The computation of this attribute varies depending on the programming language, so we use specific data structures for different languages. For function invocations, we introduce another inherited attribute to store an immutable list of types of arguments, which will help check the expression types of parameters.

c) Other Constraints: In certain programming languages, there may be additional types of constraints beyond declaration-use and expression type constraints. For example, in C, the keywords "break" and "continue" can only be used within loop or switch statements, which can be checked by introducing a new inherited attribute to store information about the enclosing loop or switch statement. Similar to the constraints mentioned earlier, these other constraints can also be encoded into attributes similarly in compiler construction.

2) Merged Attribute Checking on Reachability Graph: Above, we have encoded semantic constraints into attributes. In this subsection, we further describe how to check these attributes on the CFL reachability graph.

Before checking attributes, we should first transform the calculation of attributes into a normalized grammar form, because, in CFL reachability, the context-free grammar used are in a normalized form, where the right-hand side of each production rule contains at most two symbols. This transformation is easy to implement, by expanding the synthesized attributes into a list. After normalization, the synthesized attributes of each symbol are converted into an array of synthesized attributes in new rules and then computed according to the original grammars. If inherited attributes exist in the new rules, they are copied and computed based on the original inherited attributes.

Next, we check computation of attributes for a normalized grammar in the CFL reachability graph. To explain how OrdinalFix checks attributes on the reachability graph, we first describe the process of computing and checking attributes in recursive descent compilers. Consider a production rule $A \rightarrow BC$. During parsing of attribute grammars, the inherited attribute of symbol X is denoted by I_X, and the synthesized attribute of symbol X is denoted by S_X. The attributes of A, B, and C are computed in the following sequence:

1: Compute I_B from I_A
2: Recursively descent to B and compute S_B from I_B
3: Compute I_C from I_A and S_B
4: Recursively descent to C and compute S_C from I_C
5: Compute S_A from I_A, S_B, and S_C

To check inherited and synthesized attributes on the reachability graph, a similar recursive descent approach can be used as in compilers. However, there is a key difference: in a compiler, there is only one parse tree and each symbol has only one value for its synthesized attribute, while in the CFL reachability of a weighted modification graph, there can be multiple parse trees, and a non-terminal symbol on an edge may represent multiple paths of reduction. Therefore, it is necessary to maintain a set of inherited and synthesized attribute values for an edge on the reachability graph, and for each path of reduction, OrdinalFix computes attributes and inserts them into the appropriate sets. Specifically, for the production rule $A \rightarrow BC$ in the previous example, OrdinalFix uses a three-level loop as shown in Fig. 4. This three-level loop iterates over all possible reductions of $A \rightarrow BC$, all possible synthesized attribute values of B, and all possible synthesized attribute values of C, and computes the synthesized attribute value of A for each possible reduction.

The algorithm described above has the potential to produce

```
1: for each reduction of A → BC do
2:     Compute I_B from I_A
3:     Recursively descent to B and compute the set of
       S_B from I_B
4:     for each element of S_B do
5:         Compute I_C from I_A and S_B
6:         Recursively descent to C and compute the set of
           S_C from I_C
7:         for each element of S_C do
8:             Compute S_C from I_A, S_B, and S_C
9:             Add S_A into the synthesized attribute set.
10:        end for
11:    end for
12: end for
```

Fig. 4. Three-level loop for attribute checking.

```
Require: p, a program to be fixed
1: G = ConstructModificationGraph(p)
2: while e_s = CFLReachability(G) not empty do
3:     if CheckAttr(e_s, I_s) is not empty then
4:         return ConstructResult(e_s, I_s)
5:     end if
6: end while
```

Fig. 5. Algorithm for compilation error fixing.

an exponential number of attribute values when combining attribute values on different paths. To mitigate this issue, we have implemented several optimizations in our approach focusing on proper memoization, merging, and early pruning during the checking process. These optimizations make our algorithm applicable for real-world applications. We will further discuss these optimizations in the next subsection.

D. Algorithm for Fixing Compilation Errors

In Fig. 5, we present the algorithm for OrdinalFix[3]. The algorithm begins by constructing a weighted modification graph. Next, it uses the `CFLReachability` procedure to find the shortest root edge and then checks the semantic correctness using the `CheckAttr` procedure. If the program is semantically correct, it is constructed from the graph and returned. If the program is not semantically correct, the algorithm uses the `CFLReachability` procedure again to find the next longer root edge and repeats the process of checking the root edge using `CheckAttr`. This process is repeated until a syntactically and semantically correct program is found.

To construct the CFL reachability graph in OrdinalFix, we use an adapted shortest-path CFL reachability algorithm that collects all possible paths in the weighted modification graph. Unlike the original algorithm that only stores one shortest edge, our approach stores all CFL reachability edges since different edges may lead to different attributes during subsequent attribute checking. Similar to the original CFL reachability algorithm, we maintain a priority queue as a working queue to save all edges that need further processing. In the queue, edges with higher weight are processed after edges with lower weight, which is the same as the original shortest-path CFL reachability. However, our algorithm differs from the original by prioritizing edges with the start symbol later than other edges when they have the same weight. This ensures that all children of an edge labeled with the start symbol are obtained before attribute checking, thereby preserving the invariance of sub-problems in the dynamic programming of attribute checking, as discussed in the following paragraphs.

Next, we discuss the design of `CheckAttr`. In the following discussion, we assume that there is only one inherited attribute and only one synthesized attribute with each symbol, which can be generalized to situations where multiple inherited or synthesized attributes exist. Inspired by the top-down approach of dynamic programming with memoization used in recursive descent parsers in compiler construction [12], we design the `CheckAttr` algorithm as follows. Given an edge in the CFL reachability graph and an inherited attribute value, `CheckAttr` returns a set of all possible synthesized attribute values. In contrast to recursive descent parsers that construct one parse tree during parsing, OrdinalFix computes attribute values for one or more parse trees that have already been constructed during CFL reachability and returns a set of possible values for each synthesized attribute.

The following optimizations have been implemented in `CheckAttr`.

Pruning. To improve efficiency, we prune edges as early as possible during attribute checking when any attribute constraint is not satisfied.

Merging. We have observed that different program fragments can have the same semantic meaning. For instance, two different expressions can be of the same type, indicating that programs with either expression will either both compile or both not compile. Therefore, after checking two expressions, we merge them before checking the remaining program. To accomplish this, we merge attributes with identical values during the three-level loop of attribute computation.

Memoization. Since only a few modifications are made in the program, most remaining program fragments are the same. As a result, a reachable edge in the graph will be checked multiple times, and during different rounds of attribute checking, most remaining program fragments remain the same. To prevent duplicate computation, we use memoization. We maintain a global map to store the function's output, which ensures that the same input always returns the same output. This eliminates redundant computations, thereby enhancing the algorithm's efficiency. To maintain the invariance of the synthesized attribute value set, a priority queue is used in `CFLReachability`. The queue serves root edges e_s with weight p later than all other edges shorter than or equal to p weight. As a result, when `CheckAttr` is invoked, all

[3]Due to spatial constraints, we have included the algorithms for `CFLReachability` and `CheckAttr` in the appendices.

descendants of e_s are already constructed, and no new children edges of e_A are added after $\texttt{CheckAttr}(e_A, I_A)$ is executed.

VI. Experiments

We conducted an experimental study to evaluate the effectiveness of OrdinalFix in fixing compilation errors. Our implementation adopts a two-layer architecture: the lower layer includes an algorithm for CFL reachability with attribute checking, while the upper layer contains a language-specific fixer. By mapping the context-free grammar and attribute checking rules to the underlying algorithm, we can easily build a bug fixer for a new language.

To evaluate the effectiveness of OrdinalFix, we conducted experiments on programming languages with static typing. We implemented error fixers for two languages: Middleweight Java, an object-oriented language, and C, a procedural language. Type checking in Middleweight Java and C focuses on different relationships: Middleweight Java type checking examines function call relationships between user-defined classes, while C type checking concentrates on arithmetic and conversion relationships between basic data types. We used Middleweight Java to demonstrate the effectiveness of OrdinalFix in capturing external class definition relations and used C to demonstrate the effectiveness of OrdinalFix in capturing internal correctness of function bodies.

Since Middleweight Java is not widely used in practice, we generated a set of random compilable programs in this language and introduced compilation errors using a series of mutation operators. For C, we used the popular DeepFix dataset [2], which contains real-world student C programs with compilation errors, for evaluation. Our implementation focuses on fixing errors in function bodies for a given language. Therefore, our experiments focus on evaluating the effectiveness of OrdinalFix for fixing function body errors.

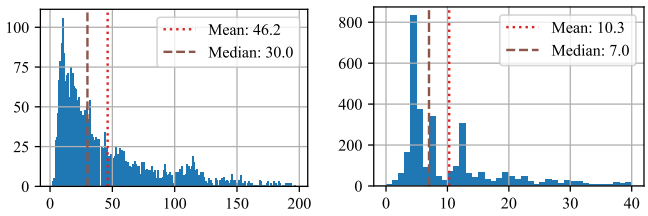
A. Middleweight Java

Middleweight Java is a lightweight subset of the Java programming language [13] that includes important features such as objects, inheritance, and polymorphism. In this study, we implemented and evaluated OrdinalFix to fix syntactic and/or semantic compilation errors in method bodies written in Middleweight Java.

The original Middleweight Java grammar [13] is designed for abstract syntax, and thus, it does not consider operator precedence or parentheses. To parse a concrete program using this grammar, we adapted it to include bracket expressions and a new symbol, primary p, which handles operator precedence.

Since Middleweight Java is not widely used in practice, to evaluate OrdinalFix, we generated a set of random compilable Middleweight Java programs and introduced compilation errors through a series of mutation operators.

1) Compilable Middleweight Java Program Generation: To ensure that a generated Middleweight Java program is compilable, we generate both the method body and all the class definitions. The following steps outline our detailed generation process: (1) We generate a random number of classes with unique names. (2) We randomly assign inheritance relationships among these classes and ensure that there is no cycle in the inheritance graph and all classes are inherited from the `Object` class. (3) We generate a constructor, a random number of methods with random parameters, and a random number of fields for each class. (4) We generate a compilable method body using top-down derivation according to the context-free grammar, and ensure that semantic constraints are satisfied during the derivation.



(a) Number of tokens (b) Number of lines

Fig. 6. The length distribution of Middleweight Java method bodies.

In the last step, we use a recursive descent generation process. When performing generation for a symbol, we randomly select a production rule associated with the symbol and then call subroutines for the symbols on the right-hand side of the production rule. We also pass down the variable table and expression types to check whether the generation is feasible. If a call to a generation subroutine fails, such as when attempting to generate a local variable with an empty variable table, we trigger a backtracking operation to select a new random production rule in some outer subroutines.

The generation of 1,000 Middleweight Java programs is completed within a few seconds. In the following, we present statistical information about the generated programs. Fig. 6 shows the length distribution of method bodies in terms of both the number of tokens and the number of lines, represented as a histogram. The average number of tokens is 45.2, and the average number of lines is 10.3. These statistics indicate that the generated programs are of reasonable size for method bodies.

2) Middleweight Java Program Mutation: To evaluate the effectiveness of our approach to compilation error fixing, we need a set of programs with compilation errors to use as test cases. To create such programs, we applied mutations to generate programs with syntactic or semantic errors.

In Middleweight Java, identifiers and other tokens behave differently. For example, replacing punctuation marks or keywords is more likely to lead to syntactic errors, while replacing identifiers is more likely to cause semantic errors.

We designed eight mutation operators (see Table I) that include three basic operations (insertion, deletion, and replacement) and a special case of insertion, i.e., duplication. The mutation operators were applied to our generated compilable programs to produce new programs with compilation errors. We used a combination of syntactic and semantic error mutations to create a diverse set of test cases for our experiments.

TABLE I
MUTATION OPERATORS

ID	Operator description
M.1	Inserting a punctuation mark or a keyword.
M.2	Deleting a punctuation mark or a keyword.
M.3	Duplicating a punctuation mark or a keyword.
M.4	Replacing a punctuation mark or a keyword.
M.5	Inserting an identifier.
M.6	Deleting an identifier.
M.7	Duplicating an identifier.
M.8	Replacing an identifier.

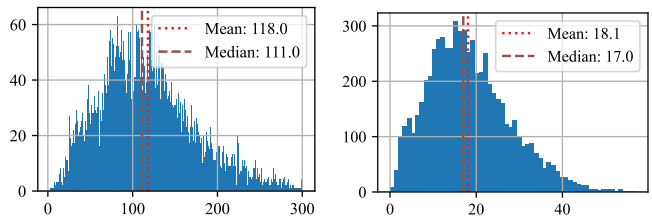
For each generated program, we produced three groups of mutants by applying the above mutation operators, i.e., a group of mutated programs with syntactic errors (denoted as $Group_{syn}$), a group of mutated programs with semantic errors (denoted as $Group_{sem}$), and a group of mutated programs mixing both syntactic and semantic errors (denoted as $Group_{mix}$). Specifically, we applied M.1∼4 to produce $Group_{syn}$ since the four mutation operators are much more likely to produce syntactic errors. We applied M.8 to produce $Group_{sem}$ since identifier replacement tends to cause semantic errors. We applied all the eight mutation operators to produce $Group_{mix}$. During the process of generating a mutated program, we first determined the mutation order as 1∼8. Then, for each mutation order i ($1 \leq i \leq 8$) we randomly selected a mutation operator from the corresponding set of mutation operators (such as M.1∼4 for $Group_{syn}$) to mutate the program (i.e., the compilable method body), and repeated this step i times[4]. In this way, we obtained 8 mutants for each group with regard to a program. In total, we obtained 2400 mutated programs, including 800 mutants in $Group_{syn}$, and 800 mutants in $Group_{sem}$, 800 mutants in $Group_{mix}$.

B. C Programs

In our evaluation on C programs, we used a dataset of 6,978 student C programs with compilation errors that was previously used to evaluate DeepFix [2]. To enable type checking, we implemented a subset of C that covers all features used by students. The context-free grammar of this simplified C contains 29 non-terminal symbols and 132 production rules, which is more complex than the grammar of Middleweight Java. Additionally, we designed type checking rules based on the C programming language to support the determination of left values and right values, pointers, expressions, and function invocations.

Since our C fixer does not recognize macros, we pre-processed the given C programs using a C pre-processor. To identify the function bodies, we used a processor that pairs the opening brackets (i.e., "{") and the closing brackets (i.e., "}"). If we encountered more opening brackets than closing brackets, we attempted to add a closing bracket at the end of the program. If a program failed the pre-processing step or had compilation errors outside the function bodies, we skipped it. We were left with 6,536 function bodies to fix. The length distribution of the function bodies is presented in Fig. 7.

[4]We introduced i errors to the method body by mutating the program i times, which means that the errors can be fixed (i.e., find a compilable method body) with fewer than or equal to i modifications.



(a) Number of tokens (b) Number of lines

Fig. 7. The length distribution of C method bodies.

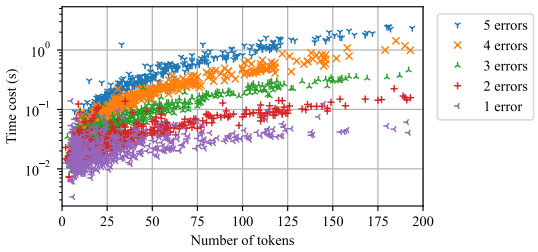


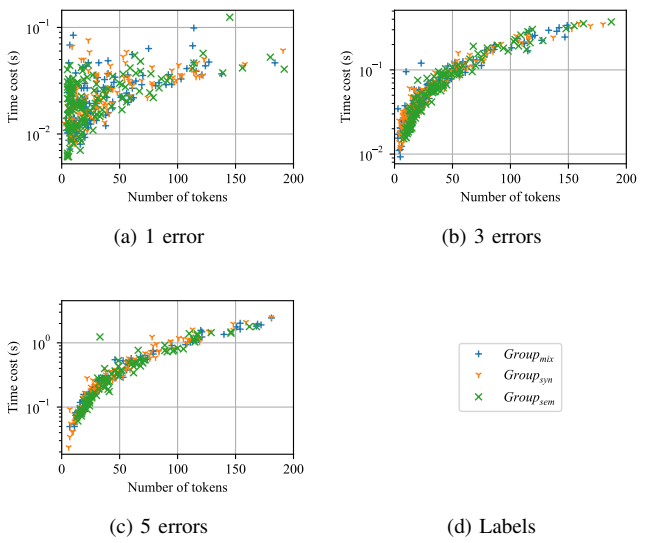
Fig. 8. Middleweight Java fixing time scatter diagram.

C. Results

Based on these mutated Middleweight Java programs and the student C programs with compilation errors, we applied OrdinalFix to fix them in order to evaluate the performance of OrdinalFix. Our study was conducted on a workstation with Intel(R) Xeon(R) E5-2680 v4 CPUs. We set the time limit to 600 seconds and the memory limit to 15 GB for fixing a mutated program. It is important to note that we only ran one thread, even though our platform has multicore CPUs, so our evaluation results show single CPU times.

1) Middleweight Java: We observe that OrdinalFix is able to successfully fix all the compilation errors (including both syntactic and semantic errors). The results demonstrate the effectiveness of OrdinalFix.

We further analyze the factors affecting the efficiency of OrdinalFix, whose results are shown in Fig. 8. In this figure, the x-axis represents the length of a program in terms of the number of tokens, the y-axis represents the time cost spent on fixing a program in the logarithmic scale, and different marks represent different numbers of compilation errors in a program. From this figure, regardless of the number of compilation errors in a program, the growth of the time costs becomes slower in the logarithmic scale with the number of tokens increasing, indicating that the growth of the time costs with the program length is less than exponential growth. Regarding the same length of programs, the growth of the time costs is stable in the logarithmic scale with the number of errors increasing, indicating that the growth of the time costs with the number of errors is almost exponential. That is, the time



(a) 1 error (b) 3 errors

(c) 5 errors (d) Labels

Fig. 9. Middleweight Java fixing time scatter diagram by type.

TABLE II
C FUNCTION BODY FIXING RESULTS

Outcome	Count	Percentage
Success	5,888	90.0%
Memory Limit Exceeded	627	9.6%
Time Limit Exceeded	10	0.2%
Other	11	0.2%
Total	6,536	100.0%

cost spent on fixing a program is more correlated with the number of errors in the program than the program length in terms of the number of tokens.

Fig. 9 further analyzes the relationship between the time costs spent on fixing a program and different mutant groups, where different marks represent different groups of mutated programs. We find that there are very similar trends for the three groups regardless of the number of errors in a program, indicating that the efficiency of OrdinalFix is not affected by different types of compilation errors. Also, we find that OrdinalFix spends less than 10 seconds on fixing a program around 200 tokens with 5 errors. The results demonstrate the efficiency of OrdinalFix.

2) C: Table II presents the fixing outcomes of OrdinalFix on the C function bodies. In addressing compilation errors, OrdinalFix achieves a commendable success rate of 90.0% (5,888 out of 6,536) within time and memory limit. The majority of failures (627 instances) are attributed to memory limit exceedance, which is mainly from the memorization mechanism utilized in our algorithm. A subset results from time limit exceedance (10 instances), attributable to the time complexity of the compilation error resolution process. A few other failures (11 instances) arise from internal macros employed by compilers, which our implementation does not support. For instance, in GCC, the macro "`isprint`" is defined in a complex manner, leading to inconsistency between the attribute grammar and the compiler's behavior.

Fig. 10 demonstrates the time consumption of token numbers for varying numbers of modifications. OrdinalFix can fix a function body within 10 seconds with up to 3 modifications, which is acceptable in application. Similar to Middleweight Java, the increase in time costs remains steady on a logarithmic scale as the number of errors grows, suggesting that the time costs rise almost exponentially with the number of errors.

The distribution of fixing times for all fixed function bodies is depicted in Fig. 11. Our analysis reveals that the median time for fixing a function body is only 0.6 seconds, indicating that the majority of fixes are completed quickly. Specifically, 62.4% of programs can be fixed within 1 second, and 84.2% of programs can be fixed within 10 seconds. These results demonstrate that the time cost associated with fixing compilation errors is acceptable for real-world applications.

To compare OrdinalFix with existing methods, we also computed the fixing rate on the original dataset. OrdinalFix considers a program to be fixed only if three conditions are met: 1) our processor[5] successfully detects function bodies in the program, 2) the program compiles after the function bodies are removed, and 3) OrdinalFix successfully resolves all errors in each function body. We note that if our processor fails or the program is incorrect outside the function bodies, we consider that OrdinalFix fails to fix the program. Table III displays the results of OrdinalFix and several other neural network-based approaches. The fixing rates of the existing approaches are reported as per their respective papers, and we did not impose any additional time or memory constraints on them. OrdinalFix achieves a success rate of 83.5% (5,757 out of 6,978)[6] for code with compilation errors, surpassing all existing approaches. In the previous paragraph, we examined the running time of OrdinalFix. Based on the distribution of our results, it is evident that, for most instances, OrdinalFix is on par with the running time of the neural network approaches, which typically takes only a few fractions of a second or a few seconds to complete. This indicates that OrdinalFix is an efficient and effective approach for fixing compilation errors.

We conduct a more detailed analysis of the instances in which OrdinalFix encounters failures in DeepFix dataset. Our examination reveals that 507 (7.3%) instances are attributed to processor failures, while 64 (0.1%) instances result from programs that do not compile even after the removal of function bodies. These failures stem from erroneous structures outside function bodies, including improperly formed function declarations, malformed global variable declarations, and incorrect usage of brackets, among other issues. The remaining 650 (9.3%) instances are associated with situations where OrdinalFix is unable to fix the function bodies, which has been analyzed in Table II and preceding paragraphs.

[5]See Section VI-B

[6]A C program contains one or more function bodies, so the number of fixed programs is less than the number of fixed function bodies.

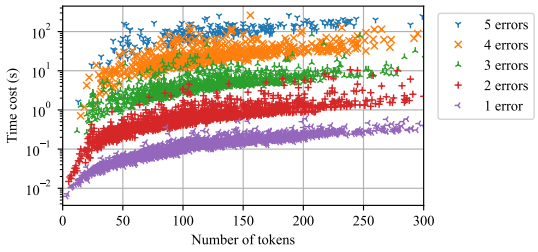


Fig. 10. C fixing time scatter diagram.

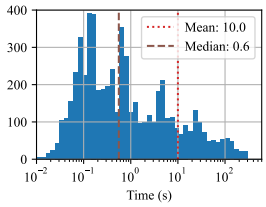


Fig. 11. C fixing time distribution.

VII. Discussion

The inherent NP-hardness of the compile error fixing problem highlights a significant challenge in the scalability of OrdinalFix. In response, our algorithm has been crafted to mitigate time-related concerns. As discussed further in the preceding sections and supported by the experimental findings, the time complexity is predominantly exponential in relation to the count of modifications required to rectify a given program. Real-world scenarios reveal that humans typically spend several seconds to minutes addressing compilation errors [3]. Comparatively, the time invested by OrdinalFix in error resolution for most function bodies in languages such as Java or student-written C programs is strikingly similar to the time expended by human counterparts.

For more complex programming languages or more complex programs, we need to design more efficient algorithms to reduce the time consumption of OrdinalFix. Additionally, OrdinalFix finds a compilation error fix with minimum number of modifications. In some cases, there are several possible fixes with the same number of modifications. It is interesting to further use some heuristics or statistical models to guide the searching process, improve the efficiency of the algorithm and find the fix that is the most similar to the human fix. This is our future work.

VIII. Related Work

Automatic Program Repair. Automated program repair (APR) is a task to fix unexpected behavior in programs [1], [17], [18], i.e., the programs are compilable, but some functionality is wrong. Recently, many search-based [19], [20], semantic-based [21], [22], pattern-based [23], [24], and deep-learning-based [25]–[27] APR techniques have been proposed. These methods are developed to fix functionality defects in software and usually require compilable programs.

TABLE III
Comparison of Fixing Rate

Approach	Fixing Rate
DeepFix [2]	33.4%
DrRepair [14]	66.1%
BIFI [15]	71.7%
TransRepair [16]	68.5%
OrdinalFix	**82.5%**

Syntactic Error Fixing and Error Recovery. In 1972, Aho et al. [8] proposed an error-correction parser to find a fix for context-free grammars with minimum modifications. Later, more work focused on syntactic error recovery. We divide them into two groups: those for LL parsing [5], [29]–[31] and those for LR parsing [4], [30], [32]–[35]. LL(*) [5] is the parsing strategy used in Antlr, which is a powerful parser generator. To produce more useful error messages, LL(*) takes into account more of the surrounding code as the context by using an increasing look-ahead scope of instead of a fixed look-ahead scope. Merr [36] enables useful syntactic error messages in LR-based compilers by mapping parse states and input tokens to error messages. Recently, Seq2Parse [37] combines symbolic error correcting parsers and neural networks to repair parse errors effectively. These methods focus on only syntactic errors for context-free grammars.

Compilation Error Fixing. Recently, some datasets in Java and C for compilation error fixing were released [6], [38]. Several techniques [2], [14]–[16], [39]–[41] addressed the problem of compilation error fixing targeting student programs. The state-of-the-art approach Break-It-Fix-It [15] designed a graph neural network model to process the source code and diagnostic feedback information. These techniques use statistical models to predict the repair results. Different from these approaches, we model the problem of compilation error fixing as the problem of finding compilable programs with minimum modifications and propose a complete algorithm to solve the problem.

IX. Conclusion

In this paper, we consider the problem of compilation error fixing with minimum modifications. We show the NP-hardness of this problem and present a complete algorithm for this problem via shortest-path CFL reachability with attribute checking. We also conducted an experimental study to evaluate OrdinalFix on a number of generated Middleweight Java programs and real-world student erroneous C programs, demonstrating its effectiveness.

Acknowledgment

The work is supported in part by the National Key Research and Development Program of China under Grant 2022YFB4501902 and the National Natural Science Foundation of China under Grant Nos. 62232003, 62232001, 61902015, and 62002256.

References

[1] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: a survey," in *ICSE*, 2018. [Online]. Available: https://doi.org/10.1145/3180155.3182526

[2] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *AAAI*, 2017. [Online]. Available: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[3] H. Seo, C. H. Sadowski, S. G. Elbaum, E. E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at Google)," in *ICSE*, 2014. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[4] R. Corchuelo, J. A. Pérez, A. R. Cortés, and M. Toro, "Repairing syntax errors in LR parsers," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 698–710, 2002. [Online]. Available: https://doi.org/10.1145/586088.586092

[5] T. J. Parr and K. Fisher, "Ll(*): the foundation of the ANTLR parser generator," in *PLDI*, 2011. [Online]. Available: https://doi.org/10.1145/1993498.1993548

[6] S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv*, vol. 1603.06129, 2016. [Online]. Available: http://arxiv.org/abs/1603.06129

[7] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. E. Aftandilian, "Deepdelta: learning to repair compilation errors," in *ESEC/FSE*, 2019. [Online]. Available: https://doi.org/10.1145/3338906.3340455

[8] A. V. Aho and T. G. Peterson, "A minimum distance error-correcting parser for context-free languages," *SIAM J. Comput.*, vol. 1, no. 4, pp. 305–312, 1972. [Online]. Available: https://doi.org/10.1137/0201022

[9] T. Reps, "Program analysis via graph reachability," in *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*. MIT Press, 1997, pp. 5–19. [Online]. Available: https://doi.org/10.1016/S0950-5849(98)00093-7

[10] O. Bastani, S. Anand, and A. Aiken, "Specification inference using context-free language reachability," in *POPL*, 2015. [Online]. Available: https://doi.org/10.1145/2676726.2676977

[11] D. E. Knuth, "The genesis of attribute grammars," in *Proceedings of the International Conference WAGA on Attribute Grammars and their Applications*, 1990. [Online]. Available: https://doi.org/10.1007/3-540-53101-7_1

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Recursive-descent parsing," in *Compilers: principles, techniques & tools Second Edition*. Pearson Addison-Wesley, 2007, pp. 219–220. [Online]. Available: https://www.pearson.com/us/higher-education/program/Aho-Compilers-Principles-Techniques-and-Tools-2nd-Edition/PGM167067.html

[13] G. M. Bierman, M. Parkinson, and A. Pitts, "Mj: An imperative core calculus for java and java with effects," University of Cambridge, Computer Laboratory, Tech. Rep., 2003. [Online]. Available: https://doi.org/10.48456/tr-563

[14] M. Yasunaga and P. Liang, "Graph-based, self-supervised program repair from diagnostic feedback," in *ICML*, 2020. [Online]. Available: http://proceedings.mlr.press/v119/yasunaga20a.html

[15] ——, "Break-It-Fix-It: Unsupervised learning for program repair," in *ICML*, 2021. [Online]. Available: http://proceedings.mlr.press/v139/yasunaga21a.html

[16] X. Li, S. Liu, R. Feng, G. Meng, X. Xie, K. Chen, and Y. Liu, "Transrepair: Context-aware program repair for compilation errors," in *ASE*, 2022. [Online]. Available: https://doi.org/10.1145/3551349.3560422

[17] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, pp. 56–65, 2019. [Online]. Available: https://doi.org/10.1145/3318162

[18] D. Cui, L. Fan, S. Chen, Y. Cai, Q. Zheng, Y. Liu, and T. Liu, "Towards characterizing bug fixes through dependency-level changes in apache java open source projects," *Sci. China Inf. Sci.*, vol. 65(7), no. 172101, 2022. [Online]. Available: https://doi.org/10.1007/s11432-020-3317-2

[19] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014. [Online]. Available: https://doi.org/10.1145/2568225.2568254

[20] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *ISSTA*, 2016. [Online]. Available: https://doi.org/10.1145/2931037.2948705

[21] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *SANER*, 2019. [Online]. Available: https://doi.org/10.1109/SANER.2019.8667970

[22] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, pp. 14:1–14:27, 2021. [Online]. Available: https://doi.org/10.1145/3418461

[23] A. Ghanbari and L. Zhang, "Prapr: Practical program repair via bytecode mutation," in *ASE*, 2019. [Online]. Available: https://doi.org/10.1109/ASE.2019.00116

[24] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019. [Online]. Available: https://doi.org/10.1145/3360585

[25] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object oriented program repair," in *ASE*, 2017. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115675

[26] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *ISSTA*, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397369

[27] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE*, 2021. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[28] Y. Xiong, Y. Tian, Y. Liu, and S.-C. Cheung, "Toward actionable testing of deep learning models," *Sci. China Inf. Sci.*, vol. 66(7), no. 176101, 2023. [Online]. Available: https://doi.org/10.1007/s11432-022-3580-5

[29] C. N. Fischer, D. R. Milton, and S. B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," *Acta Informatica*, vol. 13, pp. 141–154, 1980. [Online]. Available: https://doi.org/10.1007/BF00263990

[30] K. Hammond and V. J. Rayward-Smith, "A survey on syntactic error recovery and repair," *Comput. Lang.*, vol. 9, no. 1, pp. 51–67, 1984. [Online]. Available: https://doi.org/10.1016/0096-0551(84)90012-2

[31] C. N. Fischer and J. Mauney, "A simple, fast, and effective LL(1) error repair algorithm," *Acta Informatica*, vol. 29, no. 2, pp. 109–120, 1992. [Online]. Available: https://doi.org/10.1007/BF01178502

[32] D. T. Barnard and R. C. Holt, "Hierarchic syntax error repair for LR grammars," *Int. J. Parallel Program.*, vol. 11, no. 4, pp. 231–258, 1982. [Online]. Available: https://doi.org/10.1007/BF00999442

[33] G. V. Cormack, "An LR substring parser for noncorrecting syntax error recovery," in *SIGPLAN*, 1989. [Online]. Available: https://doi.org/10.1145/73141.74832

[34] B. J. McKenzie, C. Yeatman, and L. D. Vere, "Error repair in shift-reduce parsers," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 4, pp. 672–689, 1995. [Online]. Available: https://doi.org/10.1145/210184.210193

[35] I. Kim and K. Choe, "Error repair with validation in lr-based parsing," *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 4, pp. 451–471, 2001. [Online]. Available: https://doi.org/10.1145/504083.504084

[36] C. L. Jeffery, "Generating LR syntax error messages from examples," *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 5, pp. 631–640, 2003. [Online]. Available: https://doi.org/10.1145/937563.937566

[37] G. Sakkas, M. Endres, P. J. Guo, W. Weimer, and R. Jhala, "Seq2Parse: Neurosymbolic parse error repair," in *OOPSLA*, 2022. [Online]. Available: https://doi.org/10.1145/3563330

[38] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: a large scale repository of novice programmers' activity," in *SIGCSE*, 2014. [Online]. Available: https://doi.org/10.1145/2538862.2538924

[39] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: for the student programs, from the student programs," in *ICSE-SEET*, 2018. [Online]. Available: https://doi.org/10.1145/3183377.3183383

[40] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *SANER*, 2018. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330219

[41] R. Gupta, A. Kanade, and S. K. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," in *AAAI*, 2019. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.3301930

APPENDIX
ALGORITHMS FOR CFLREACHABILITY AND CHECKATTR

We briefly describe `CFLReachability` and `CheckAttr` in Fig. 12 and Fig. 13.

Fig. 12 shows the `CFLReachability` algorithm. The procedure `CFLReachability` operates similar to the shortest-path CFL reachability algorithm [10]. The procedure `CFLReachability` operates on the weighted modification graph, referred to as G. Additionally, `CFLReachability` utilizes a priority-based working queue, denoted as H, to store edges to be processed. The priority queue H is ordered by the weight of edges, where edges with lower weights are processed first. For edges with the same weight, root edges are processed last in the queue H to ensure the invariance of sub-problems in the dynamic programming. The queue H is implemented as a heap with a customized comparator.

When initially invoked, `CFLReachability` carries out a preparatory step, aiming to handle $A \rightarrow \epsilon$ production rules, as shown in Fig. 2a. Subsequently, the edges within G are enqueued into H to initialize the processing sequence.

The core of the algorithm is a looping construct that continues until the working queue H becomes empty. Within each iteration, the algorithm dequeues an edge e from H, capturing its associated label A. If A corresponds to the start symbol and the edge begins at the first vertex and ends at the last vertex, the algorithm promptly terminates, signifying the attainment of a valid path. Otherwise, `CFLReachability` systematically explores potential productions to reduce the current path. It does so by invoking three distinct subroutines: `try_production_1`, `try_production_2_left`, and `try_production_2_right`. These subroutines extend the exploration process, utilizing the edge's information and the graph G to attempt various production possibilities, as shown in Fig. 2b and Fig. 2c.

Different from original shortest-path CFL reachability algorithm, when inserting an edge with a higher weight into the graph, `CFLReachability` does not remove the edge with the same label and the same end vertex from the graph to avoid the loss of potential solutions.

Through this algorithm, `CFLReachability` endeavors to uncover viable paths that contain syntactically correct programs.

Fig. 13 shows the `CheckAttr` algorithm. The procedure `CheckAttr` performs attribute checking on the CFL reachability graph. The procedure `CheckAttr` checks an edge e_A with an inherited attribute value I_A and returns a set of synthesized attribute values $\mathbb{S}_A$.

To avoid repeated computation, `CheckAttr` utilizes a memoization map $\mathbb{M}$ to store the results of previous computations. If the result of the current computation is already in $\mathbb{M}$, `CheckAttr` returns the result directly.

The procedure `CheckAttr` is to utilize a generator-like mechanism to return the synthesized attribute values. Instead of returning all the synthesized attribute values at once, `CheckAttr` returns an iterator that yields one synthesized attribute value at a time. This allows `CheckAttr` to return the synthesized attribute values one by one, which is more efficient than returning all the synthesized attribute values at once.

For terminal edges, `CheckAttr` invokes `process_terminal` to compute the synthesized attribute value, where `process_terminal` is a user-defined function for computing the synthesized attribute value of a terminal edge. If `process_terminal` fails, `process_terminal` returns **null**, indicating the inherited attribute of the terminal does not conform to semantic constraints, so that `CheckAttr` skips the current value and continues to check the next attribute value.

For non-terminal edges, `CheckAttr` iterates over all the possible productions and invokes `process_left_inherited`, `process_right_inherited`, and `process_synthesized` to compute the inherited attribute value, the right inherited attribute value, and the synthesized attribute value, respectively. Here, `process_left_inherited`, `process_right_inherited`, and `process_synthesized` are user-defined functions for computing the inherited attribute value, the right inherited attribute value, and the synthesized attribute value of a non-terminal edge, respectively. Similarly, these functions return **null** if the attribute value does not conform to semantic constraints, so that `CheckAttr` skips the current value and continues to check the next attribute value.

The optimizations of `CheckAttr` are implemented as follows.

Pruning. `CheckAttr` only continues to check the attribute if the inherited attribute value conforms to semantic constraints. Otherwise, `CheckAttr` skips the current value and continues to check the next attribute value. This is implemented by returning **null** in `process_terminal`, `process_left_inherited`, `process_right_inherited`, and `process_synthesized` when the attribute value does not conform to semantic constraints.

Merging. `CheckAttr` stores a list of synthesized attributes in $\mathbb{S}_A$. The list $\mathbb{S}_A$ will remove duplicated synthesized attributes before returning. This is implemented by using an extra set to store the synthesized attributes in $\mathbb{S}_A$.

Memoization. `CheckAttr` stores the results of previous computations in $\mathbb{M}$. If the result of the current computation is already in $\mathbb{M}$, `CheckAttr` returns the result directly.

With the help of `CheckAttr`, OrdinalFix can perform attribute checking on the CFL reachability graph and find the shortest path that satisfies all the attribute constraints.

```
1: procedure CFLReachability(G, H)
2:     {G is the weighted modification graph}
3:     {H is the working queue with priority, which stores edges to be processed}
4:     if first time to call CFLReachability then
5:         for each rule A → ϵ, each vertex v_i do
6:             add ⟨v_i, v_i, A, 0⟩ to G
7:         end for
8:         for each edge e ∈ G do
9:             H.push(e)
10:        end for
11:    end if
12:    while H not empty do
13:        e = H.pop()
14:        A = e.label
15:        if A is the start symbol and e begins at v_0 and ends at v_n then
16:            return e
17:        end if
18:        try_production_1(e, G, H)
19:        try_production_2_left(e, G, H)
20:        try_production_2_right(e, G, H)
21:    end while
22: end procedure
```

Fig. 12. The `CFLReachability` algorithm.

```
1: procedure CheckAttr(e_A, I_A, 𝕄)
2:     {e_A is the edge to be checked}
3:     {I_A is the inherited attribute value}
4:     {𝕄 is the memoization map}
5:     {denote the symbol of e_A as A}
6:     if (e_A, I_A) is in 𝕄 then
7:         return 𝕄[(e_A, I_A)]
8:     end if
9:     construct empty list 𝕊_A to store results
10:    𝕄[(e, I)] = 𝕊_A
11:    if A is an edge with terminal symbol then
12:        S_A = process_terminal(A, I_A)
13:        if S_A is not null then
14:            𝕊_A.add(S_A)
15:            yeild S_A
16:        end if
17:    else
18:        for each edge expansion r for e do
19:            if r is in the form of e_A → ϵ then
20:                S_A = process_synthesized(A → ϵ, I_A, null, null)
21:                if S_A is not null then
22:                    𝕊_A.add(S_A)
23:                    yeild S_A
24:                end if
25:            end if
26:            if r is in the form of e_A → e_B then
27:                {denote the symbol of e_B as B}
28:                I_B = process_left_inherited(A → B, I_A)
29:                if I_B is not null then
30:                    𝕊_B =CheckAttr(e_B, I_B, 𝕄)
31:                    for each S_B ∈ 𝕊_B do
32:                        S_A = process_synthesized(A → B, I_A, S_B, null)
33:                        if S_A is not null then
34:                            𝕊_A.add(S_A)
35:                            yeild S_A
36:                        end if
37:                    end for
38:                end if
39:            end if
40:            if r is in the form of e_A → e_B e_C then
41:                {denote the symbol of e_B as B and the symbol of e_C as C}
42:                I_B = process_left_inherited(A → BC, I_A)
43:                if I_B is not null then
44:                    𝕊_B =CheckAttr(e_B, I_B, 𝕄)
45:                    for each S_B ∈ 𝕊_B do
46:                        I_C = process_right_inherited(A → BC, I_A, S_B)
47:                        if I_C is not null then
48:                            𝕊_C =CheckAttr(e_C, I_C, 𝕄)
49:                            for each S_C ∈ 𝕊_C do
50:                                S_A = process_synthesized(A → BC, I_A, S_B, S_C)
51:                                if S_A is not null then
52:                                    𝕊_A.add(S_A)
53:                                    yeild S_A
54:                                end if
55:                            end for
56:                        end if
57:                    end for
58:                end if
59:            end if
60:        end for
61:    end if
62: end procedure
```

Fig. 13. The CheckAttr algorithm.