一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

LLMを活用したクラスファイルの局所的な編集による代替コンパイラの提案

渡邉　凌雅†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{ryg-wtnb,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　Java プロジェクトの動作検証や性能評価のために，開発に関与していないユーザーが Java プロジェクトを実行することがある．実行時にプロジェクトのソースファイルやリソースファイルに編集を加える場合，ビルドによる JAR ファイルの更新が必要である．しかし，多くの Java プロジェクトではユーザーの実行環境が Java プロジェクトの想定する実行環境と一致しない．そのため，ビルドのために依存関係の解決や実行環境の再構築が必要で時間と労力を要する．一方，編集がビルド環境に影響しない場合，JAR ファイルに含まれるクラスファイルを局所的に編集するだけで十分である．そこで本研究では，大規模言語モデルを活用してクラスファイルを局所的に編集する手法を提案する．提案手法の評価実験では，LLM として gpt-3.5-turbo を使用し，491 件の Java クラスのソースコードの 1 行削除に対応するようにクラスファイルを編集した．その結果，正解のバイトテキスト差分と完全に一致したのは 286 件で，実行可能なクラスファイルは 261 件であった．

キーワード　Java，ビルド，クラスファイル，コンパイラ，代替モデル，大規模言語モデル（LLM）

1. はじめに

GitHub 等で一般公開されている Java プロジェクトの動作検証や性能評価などを行うために，Java プロジェクトの開発に関与していないユーザーが，ビルド済みの JAR ファイルを使用し，ユーザーのローカル環境で Java プロジェクトを実行する場合がある．しかし，実行の目的によっては，Java プロジェクト内の Java ソースファイルやリソースファイルの一部を編集することがある．その場合，編集後の Java プロジェクトをユーザーのローカル環境でビルドし，JAR ファイルを更新する必要がある．このとき，ビルドを成功させるには，Java プロジェクトのビルド環境の整合性が保持されている必要がある．つまり，Java ソースファイルやリソースファイルに誤りがなく，ビルドの実行環境が Java プロジェクトの想定する実行環境と一致している必要がある．しかし，ユーザーのローカル環境が Java プロジェクトが想定する環境と一致しないことが多い．実際，ダウンロードしビルドコマンドを実行するだけでビルドに成功する Java プロジェクトは少ない [1] [2]．そのため，Java ソースファイルやリソースファイルを書き換えてビルドする場合，依存関係の解決や実行環境の再構築が必要であり，多大な時間と労力を要する．

一方，Java ソースファイルに加えた編集がビルド環境の整合性に影響しない場合，ビルド済み JAR ファイルとユーザーのローカル環境でのビルドによって新たに生成された JAR ファイルで生じる差分は，編集を加えた Java ソースファイルに対応するクラスファイルのみである．そのため，ビルド環境の整合性が原因でビルドに失敗する Java プロジェクトに対しても，JAR ファイルに含まれるクラスファイルを何らかの方法で局所的に編集することで，従来のビルドを実行せずに JAR ファイルを更新できる．

そのため，従来のビルドプロセスに依存せずクラスファイルを生成する代替コンパイラを実現するための研究が行われている．橋本ら [3] は，代替コンパイラの実現に向け，Transformer [4] ベースの機械翻訳モデルを用いて Java ソースファイルからクラスファイルへの変換を試みた．Transformer ベースのモデルでは，バイナリ表現を直接扱うことができない．そこで橋本らは，バイトコードの各命令に対応するテキスト表現（バイトテキスト）と，バイトコードとバイトテキストの相互変換プログラムを作成した．そして，ソースファイルの変換先をバイトテキストにすることで，バイナリ表現の課題を解決した．しかし，評価実験では，ReCa [5] から収集した 18,290 件の Java ソースファイルのうち正しく動作するクラスファイルを生成できたのはわずか 152 件であった．ソースコードの変換をファイル単位で実行したため，変換対象が広範囲にわたり誤りが生じやすかったことが原因の一つと考えられる．

本研究では，代替コンパイラの実現に向け，大規模言語モデルを活用したクラスファイルの局所的な編集手法を提案する．提案手法では，クラスファイルの編集範囲を Java ソースファイルの編集差分に対応する局所的なバイトコード命令列に限定し，編集差分に正確に対応するようにクラスファイルを局所的に編集する．クラスファイルの編集では，橋本らの先行研究と同様にクラスファイルをバイトテキストとして扱う．そして，大規模言語モデル（LLM）を使用して，バイトコード命令列の差分に対応するバイトテキスト差分を生成する．さらに，

ソースコード差分と対応するバイトテキスト差分（差分ペア）をもとにLLMをファインチューニングし，バイトテキスト差分生成の精度を向上させる．

本稿では，以下の設定のもと提案手法を実装し，評価実験を実施した．

- LLMとしてgpt-3.5-turboを使用
- ソースコードの編集差分の単位を1行削除に限定

評価実験では，491件の検証データを使用して，提案手法によるクラスファイル編集能力を評価した．実験の結果，gpt-3.5-turboが正解のバイトテキスト差分と完全に一致する差分を予測した件数は491件中286件であった．また，提案手法によって得られたクラスファイルのうち，実行可能な件数は491件中261件であった．

2. 準　　備

2.1 コンパイルとビルド

コンパイルとは，プログラミング言語のソースコードを実行可能形式に変換する処理であり，この処理を実行するプログラムをコンパイラという．Javaプログラムの開発では，Javaのコンパイラが Javaソースファイルをクラスファイルと呼ばれる実行可能形式に変換する．

大規模なJavaプロジェクトでは，Javaソースファイルの他にリソースファイルや外部ライブラリへの依存関係も含まれる．そのため，コンパイルだけでは実行可能なファイルを生成できない．このようなJavaプロジェクトでは，代わりにビルドと呼ばれるプロセスが実行される．ビルドとは，プロジェクトのソースコードやリソースファイルを入力し，ソースコードのコンパイルと依存関係の解決を経て，実行可能ファイルを生成するまでの一連のプロセスを指す．Javaでは，実行可能ファイルは一般的にJARファイルとして生成される．JARファイルは，プログラムを構成するクラスファイルや設定ファイルを圧縮したJavaの実行可能ファイル形式の1つである．

また，ビルド実行の効率化を目的として，ビルド自動化ツールが開発されている．Java用の代表的なビルド自動化ツールとして，Maven (注1)，Gradle (注2)，Ant (注3)がある．

2.2 大規模言語モデル（LLM）

大規模言語モデル（LLM）は，多数のニューラルパラメータを持ち，大規模なコーパスで事前学習された言語モデルである．最先端の性能を持つモデルの多くがTransformerと呼ばれるアーキテクチャをベースとする．これまでに，BERT [6]，GPT [7]，T5 [8] といった数多くのTransformerベースのLLMが提案され，機械翻訳や感情分析といった自然言語処理のタスクにおいて高い性能を達成している．特にGPTは，その後継モデルであるGPT-3.5やGPT-4を利用したChatGPT (注4)のリリース以降，その高い文章生成能力から多くの分野で急速に利用が拡大している．

自然言語処理分野におけるLLMの成功を受けて，コード関連タスクにおけるLLMの応用が注目されている．LLMは，コードクローン検出，コード検索といったコード理解系タスク，およびコード翻訳，コード要約といったコード生成系タスクにおいても高い性能を達成している [9]．

LLMは汎用的なモデルであり，特定のタスクに最適化されていない．特定のタスクを実行する際は，LLMをそのタスクに合わせてファインチューニングするか，プロンプトエンジニアリングによってLLMへの入力を工夫するのが一般的である．

3. 先行研究とその課題

本研究は，橋本らの先行研究 [3] に基づく．橋本らは，Transformerベースの機械翻訳モデルを用いてJavaソースファイルからクラスファイルへの変換を試みた．

3.1 先行研究におけるバイナリ表現の扱い

TransformerベースのモデルでJavaソースファイルをJavaクラスファイルに変換する際，Javaクラスファイルがバイナリ表現であることが課題となる．Transformerベースのモデルでは，バイナリ表現を直接扱えないためである．そこで橋本らは，バイトコードの各命令に対応するテキスト表現（バイトテキスト）と，バイトコードとバイトテキストの相互変換プログラムを作成した．これにより，ソースファイルの変換先をバイトテキストにすることで，バイナリ表現を直接扱えない課題を解決した．

先行研究では，バイトコードとバイトテキストの相互変換にASM (注5)を使用している．ASMとは，Javaバイトコードを操作・分析するためのAPIを提供するJavaライブラリである．ASMは，クラスファイルを読み込みイベントのシーケンスに変換する`ClassReader`抽象クラスと，イベントのシーケンスからクラスファイルを生成する`ClassWriter`抽象クラスを提供する．先行研究では，ASMで定義されるイベントに対し一対一対応するバイトテキストの構文を定義し，それをもとに`ClassReader`と`ClassWriter`を継承した独自クラスを実装してバイトコードとバイトテキストの相互変換を実現している．図1に，`"Hello, World!"`と出力するJavaクラスのソースコードと，コンパイルして得られたクラスファイルをバイトテキストに変換した結果を示す．ソースコードに含まれるクラス宣言やメソッド宣言，メソッド呼び出しに対応したバイトコード命令の列がテキスト表現として出力される．

3.2 先行研究の課題と解決策

先行研究における評価実験で，ReCa [5] から収集した18,290件のJavaソースファイルのうち，正しく動作するクラスファイルを生成できたのはわずか152件であった．この原因の1つとして，ソースコードの変換をファイル単位で実行したため，変換対象が広範囲にわたり，バイトテキストに誤りが生じやすかったことが考えられる．

そこで本研究では，Javaソースファイルを局所的に編集した

(注1)：https://maven.apache.org/
(注2)：https://gradle.org/
(注3)：https://ant.apache.org/
(注4)：https://chatgpt.com/
(注5)：https://asm.ow2.io/

```java
public class Sample {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

(a) "Hello, World!"と出力する Java クラスのソースコード

```
cl visit 61 33 Sample null java/lang/Object
cl Method 1 <init> ()V null null
me Code
me VarInsn 25 0
me MethodInsn 183 java/lang/Object <init> ()V false
me Insn 177
me Maxs 1 1
me End
cl Method 9 main ([Ljava/lang/String;)V null null
me Code
me FieldInsn 178 java/lang/System out Ljava/io/PrintStream;
me LdcInsn class java.lang.String Hello, World!
me MethodInsn 182 java/io/PrintStream println (Ljava/lang/String;)V false
me Insn 177
me Maxs 2 1
me End
cl End
```

(b) (a) に対応するバイトテキスト

図 1: Java クラスのソースコードと対応するバイトテキスト

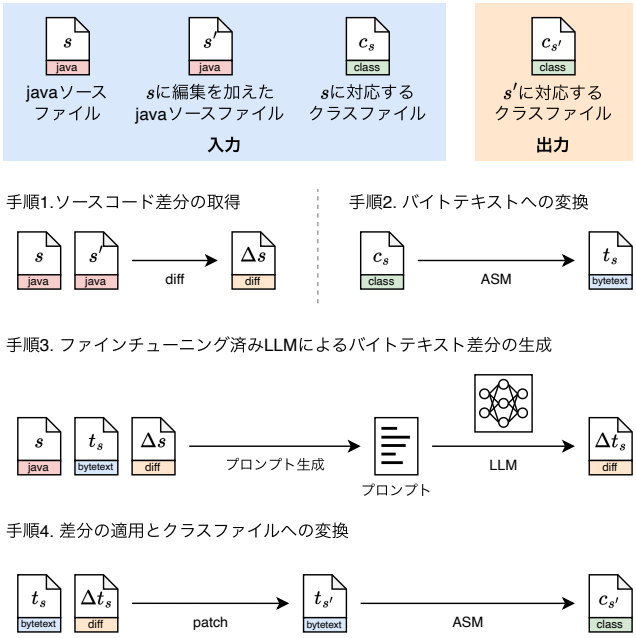

図 2: 提案手法の流れ

場合に，ビルドによってクラスファイルに生じるバイトコード命令列の差分も局所的である点に着目する．そして，クラスファイルの編集範囲を Java ソースファイルの編集差分に対応する局所的なバイトコード命令列に限定し，編集差分に正確に対応するようにクラスファイルを局所的に編集する．さらに，ソースコードの編集差分とそれに対応するバイトテキスト差分をもとに LLM をファインチューニングし，バイトテキスト差分生成の精度を向上させる．

4. 提 案 手 法

本研究では，代替コンパイラの実現に向け，大規模言語モデルを活用したクラスファイルの局所的な編集手法を提案する．この手法は既存の Java コンパイラに依存しないため，ビルド環境の整合性が原因でビルドに失敗する Java プロジェクトにも適用できる．提案手法は様々な LLM に適用できる汎用的な手法である．提案手法の評価実験（6. 章）では gpt-3.5-turbo を使用している．

図 2 に提案手法の流れを示す．提案手法は，Java ソースファイル s とそれに対応するクラスファイル c_s，および s に局所的な編集を加えた Java ソースファイル s' を入力として受け取り，s' に対応するクラスファイル $c_{s'}$ を出力する．提案手法は以下の 4 つの手順から構成される．

手順 1. ソースコード差分の取得
手順 2. バイトテキストへの変換
手順 3. LLM によるバイトテキスト差分の生成
手順 4. 差分の適用とクラスファイルへの変換

手順 1. ソースコード差分の取得

`diff` コマンドを使用して s と s' の差分 Δs を取得する．テキストファイルの差分を表現する方法には様々な種類があるが，本研究では Unified 形式を使用する．Unified 形式は，`diff` コマンドで定義される差分の表現方式の 1 つであり，可読性が高く，変更箇所が一目でわかる点が特徴である．また，`patch` コマンドを使えば，差分を機械的に適用できる．

手順 2. バイトテキストへの変換

s をコンパイルして生成されたクラスファイル c_s のバイトコードをバイトテキスト t_s に変換する．本研究では，3. 1 節で述べた橋本ら [3] のバイトテキスト定義と相互変換プログラムを使用する．

手順 3. LLM によるバイトテキスト差分の生成

s に Δs の編集を加えたときに，バイトテキスト t_s に生じる差分 Δt_s を LLM に予測させる．LLM のバイトテキスト差分生成の正確性を向上させるため，あらかじめソースコード差分と対応するバイトテキスト差分から作成されたデータセットを用いて LLM をファインチューニングする（5. 章）．

まず, LLM に入力するプロンプトを生成する．提案手法の評価実験では LLM として gpt-3.5-turbo を使用したため，この節では gpt-3.5-turbo を使用する際のプロンプト形式を解説する．

プロンプトは，システムプロンプトと指示プロンプトの 2 つで構成される．システムプロンプトとは，LLM の役割を記述するプロンプトである． 図 3 に本研究で使用するシステムプロンプトを示す．本研究では，「Java 開発に特化した AI アシスタントであり，与えられたコンテキスト情報に基づき，Java ソースコードの変更に基づいてバイトコードのテキスト表現の差分を Unified 形式で生成する」旨の英文を使用する．

指示プロンプトは，実行したいタスクを具体的に指示するプロンプトであり，システムプロンプトの次に与えられる． 図 4 に本研究で使用する指示プロンプトの例を示す．本研究で作成する指示プロンプトは，実行したいタスクの指示文と，タスクの実行に必要なコンテキスト情報で構成される．タスクの

```
You are an AI assistant specialized in Java development.
Your task is to generate the diff of the text representation
of byte code in unified diff format, based on changes made
to the Java source code. You will be provided with three
inputs: the original Java source code, its corresponding
original text representation of byte code, and the diff of
the Java source code. Using these inputs, your objective is
to produce an accurate diff of the text representation of
byte code in unified diff format that reflects the
modifications made to the Java source code.
```

図 3: システムプロンプトの例

```
Using the following information, generate the diff of the
byte code text in unified diff format.
If there is no diff, output "No diff found".

Original Source Code:

import java.util.*;

public class Target00000 {
  public byte a2b(byte c){
    if ('0' <= c && c <= '9')   return (byte)(c - '0');
    if ('a' <= c && c <= 'z')   return (byte)(c - 'a' + 10);
    return (byte)(c - 'A' + 10);
  }
}

Original Byte Text:

cl  visit  61  33  Target00000  null  java/lang/Object
cl  Source  Target00000.java  null
cl  Method  1  <init>  ()V  null  null
me  Code
me  Label  l1
... (省略) ...

Diff of Source Code:

@@ -1,3 +1,3 @@",
-import java.util.*;
+

public class Target00000 {

Diff of Byte Text:
```

図 4: 指示プロンプトの例

指示文として，「コンテキスト情報を使用し，ソースコードの編集差分に対応するバイトテキストの差分を Unified 形式で生成せよ．差分がない場合は “No diff found” と出力せよ」旨の英文を使用する．コンテキストとして，ソースコード s，s に対応するバイトテキスト t_s，s に加えた差分 Δs の 3 つを含める．

次に，LLM にこれらのプロンプトを入力し，Δs に対応するバイトテキスト差分 Δt_s を生成させる．

手順 4. 差分の適用とクラスファイルへの変換

`patch` コマンドを使用してバイトテキスト t_s に手順 3. で得られた差分 Δt_s を適用し，バイトテキスト $t_{s'}$ を得る．手順 3. の生成結果が “No diff found” である場合は t_s をそのまま $t_{s'}$ として扱う．次に，橋本らによる相互変換プログラムを使用して，$t_{s'}$ を s' に対応するクラスファイル $c_{s'}$ に変換する．

5. LLM のファインチューニング

LLM によるバイトテキスト差分生成の精度向上を目的として，ソースコード差分と対応するバイトテキスト差分（差分ペア）から作成されたデータセットを用いて LLM をファインチューニングする．

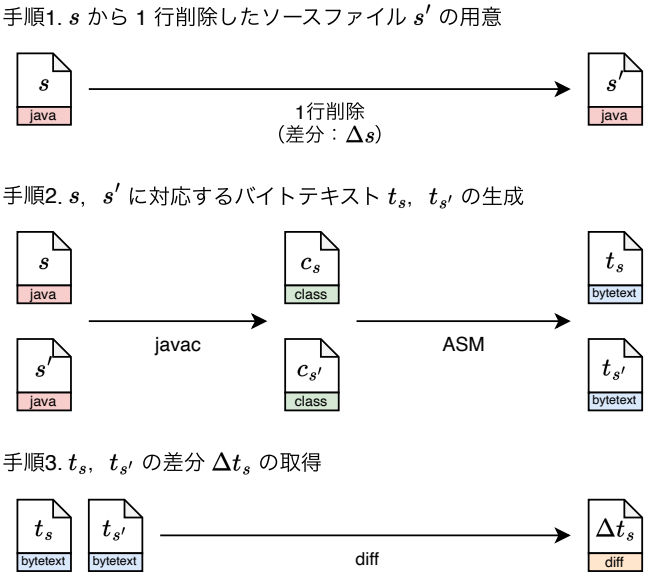

図 5: 差分ペアの収集

5.1 差分ペアの収集

複数の Java ソースファイルから差分ペアを収集する．LLM にバイトテキストの局所的な差分を生成させるため，差分ペアも局所的である必要がある．そこで本稿では，ソースコードに対する編集の単位を 1 行削除とした．

収集対象

差分ペアの収集対象として，FEMP Dataset [10] を使用した．FEMP Dataset は，機能的に等価な Java メソッドのペアを集めたデータセットである．FEMP Dataset に含まれるすべての Java メソッドは，Java の標準パッケージ（java.lang および java.util）のみに依存し，メソッド単体で処理が完結する特徴を持つ．

FEMP Dataset から，コンパイル可能で命令文の数が 10 未満の Java メソッドを 1,000 個抽出した．これらの Java メソッドを 1 つずつクラス定義でラップし，1,000 個の Java ソースファイルで構成される集合 S を作成した．S は合計 11,432 行のソースコードを含む．

手順

S に含まれるすべての Java ソースファイル s について，s に含まれるすべてのソースコード行 l を対象に，l を削除した際のソースコード差分 Δs と対応するバイトテキスト差分 Δt_s のペア $(\Delta s, \Delta t_s)$ を差分ペアとして収集する． 図 5 に，s から 1 行を削除した際の差分ペアを得る手順を示す．収集方法は以下の 3 つの手順で構成される．

手順 1. s から 1 行削除したソースファイル s' の用意
手順 2. s，s' に対応するバイトテキスト t_s，$t_{s'}$ の生成
手順 3. t_s，$t_{s'}$ の差分 Δt_s の取得

手順 1 では，ソースコード s から 1 行削除したソースコード s' を用意する．手順 2 では，s，s' それぞれを javac でコンパイルし，対応する Java クラスファイル c_s，$c_{s'}$ を生成する．次に，橋本らによる相互変換プログラムを使用して c_s，$c_{s'}$ を

バイトテキスト t_s，$t_{s'}$ に変換する．削除した行によって構文に誤りが生じ，s' のコンパイルに失敗する場合は収集対象から除外する．手順 3 では，T_s と $T_{s'}$ を比較し，バイトテキスト差分 ΔT_s を得る．差分は Unified 形式で生成する．

結　　果

S に対してバイトテキスト差分を収集した結果，2,453 個の差分ペアを得た．

5.2 ファインチューニング用データセットの作成

5.1 節で収集した差分ペアをもとに，gpt-3.5-turbo をファインチューニングするためのデータセットを作成する．このデータセットは，システムプロンプト，指示プロンプト，正解文の 3 つ組で構成される．システムプロンプトと指示プロンプトについては 4. 章で説明した通りである．正解文には ΔT_s をそのまま使用する．

作成した 2,453 件のデータセットを訓練データと検証データで 4 対 1 に分割した．訓練データ（1,962 件）はファインチューニングの学習データとして使用し，検証データ（491 件）は提案手法の評価実験の実験対象として使用する．

5.3 ファインチューニングの実行

5.2 節で作成したデータセットを使用して，gpt-3.5-turbo をファインチューニングする．実行には Open AI が提供するファインチューニング API を使用する．ファインチューニング実行時の設定は 表 1 に示す通りである．

6. 評 価 実 験

本稿では，以下の設定のもと提案手法を実装した．

- LLM として gpt-3.5-turbo を使用
- ソースコードの編集差分の単位を 1 行削除に限定

そして，5.2 節で作成した 491 件の検証データを使用して提案手法によるクラスファイル編集能力を評価する．評価実験にあたり，3 つの Research Questions（RQ）を設定した．

RQ1: 予測差分は正解差分と一致するか

491 件の検証データに対し，gpt-3.5-turbo でバイトテキスト差分を予測する．その後，予測されたバイトテキスト差分（予測差分）と正解のバイトテキスト差分（正解差分）を比較し，文字列として完全一致した件数とその割合を計算した．

実験の結果を 表 2 に示す．予測差分と正解差分が一致したのは 491 件中 285 件であった．そのうち，バイトテキストに差分が生じたケースについては，約 34.73% で差分が一致した．一方，バイトテキストに差分が生じなかったケースについては 2 件を除く全てのケースで差分が一致した．

表 1: ファインチューニング実行時の設定

項目	値
ベースモデル	gpt-3.5-turbo-0125
シード値	8058399
エポック数	3
バッチサイズ	2
LR マルチプライヤ	2

RQ2: 予測差分の適用とクラスファイルへの変換は可能か

`patch` コマンドを実行し，RQ1 の検証時に生成した予測差分をバイトテキストに適用可能か検証する．`patch` コマンドの実行が正常終了した場合，予測差分の形式に問題がなく，差分が適用可能であると判定する．その後，橋本らによる相互変換プログラムを実行し，予測差分を適用できたバイトテキストをクラスファイルに変換可能か検証する．相互変換プログラムの実行が正常終了した場合，差分適用後のバイトテキストに文法的な誤りが含まれておらず，クラスファイルへ変換可能であると判定する．

実験結果を 表 3 に示す．検証データ 491 件のうち，予測差分をバイトテキストに適用ができたケースは 391 件，そのうちクラスファイルに変換できたケースは 343 件であった．

RQ3: 提案手法で得られたクラスファイルは実行可能か

RQ2 において変換に成功したクラスファイルに対して，そのクラスをテスト対象とする単体テストを実行し，クラスファイルが実行可能かどうかを検証する．Java では，クラスファイルのロード時にフレームのオペランドスタックのサイズが正しいかどうかなど，クラスファイルの形式に矛盾がないか検証される．矛盾がある場合，`VerifyError` が発生し，単体テストの実行に失敗する．そのため，単体テストの実行時に `VerifyError` が発生しなければ，そのクラスファイルは実行可能であると判定する．クラスファイルの実行には，FEMP Dataset に含まれる単体テストのテストスイートを使用した．

実験の結果，RQ2 で変換に成功した 343 件のクラスファイルのうち，約 76.09% の 261 件が実行可能であった．これは，検証データ 491 件の約 53.16% にあたる．

7. お わ り に

本研究では，大規模言語モデルを活用したクラスファイルの局所的な編集による代替コンパイラの実現手法を提案した．クラスファイルの編集範囲を Java ソースファイルの編集差分に対応する局所的なバイトコード命令列に限定し，編集差分に正確に対応するようにクラスファイルを局所的に編集する．クラスファイルの編集では，橋本らの先行研究と同様にクラスファイルをバイトテキストとして扱う．そして，大規模言語モデル（LLM）を使用して，バイトコード命令列の差分に対応するバイトテキスト差分を生成する．

表 2: RQ1 の結果

	個数	正答数	正答率
全体	491	286	約 58.25%
バイトテキスト差分あり	311	108	約 34.73%
バイトテキスト差分なし	180	178	約 98.89%

表 3: RQ2 の結果

	成功した数	割合
予測差分の適用	391	約 79.63%
クラスファイルへの変換	343	約 69.86%

LLM に gpt-3.5-turbo を使用し，ソースコードの編集差分の単位を 1 行削除に限定した上で提案手法を実装し，491 件の検証データに対して提案手法によるクラスファイル編集能力を評価した．評価実験の結果，gpt-3.5-turbo が正解のバイトテキスト差分と完全に一致する差分を予測した件数は 491 件中 286 件であった．また，提案手法によって得られたクラスファイルのうち，実行可能な件数は 491 件中 261 件であった．

提案手法の実装ではソースコード編集差分の単位を 1 行削除とした．しかし，1 行に文や式が多く含まれる場合，バイトテキストの差分が大きくなり，LLM をファインチューニングする際の局所的な差分の対応関係の学習が困難となる．そこで，AST を利用した編集単位の縮小を検討している．AST のノード単位で削除を実行することで，より細かい単位での編集が可能となり，LLM によるバイトテキスト差分生成の精度が向上する可能性がある．将来的には，AST ノードの削除に加えて新たなノードの追加や変更も編集差分に含め，幅広い編集に対応する予定である．

評価実験では 3 つの RQ を設定し，提案手法のクラスファイル編集能力を評価した．しかし，それぞれの RQ の結果に対する考察は実施していない．したがって，それぞれの RQ における成功例や失敗例についての詳細なケーススタディを実施する予定である．

謝辞　本研究は JSPS 科研費（JP24H00692，JP21K18302，JP21H04877, JP23K24823, JP22K11985）の助成を得て行われた．

文　　献

[1] M. Sulír, M. Bačíková, M. Madeja, S. Chodarev, and J. Juhár, “Large-scale dataset of local java software build results,” Data, vol.5, no.3, pp.1–11, 2020.

[2] 小池　耀，眞鍋雄貴，松下　誠，井上克郎，“オープンソース Android アプリケーションのビルド可能性に関する調査，” Technical report，大阪大学，July 2022.

[3] 橋本　周，“深層学習を用いたコンパイラレスコンパイルの試み，” Master’s thesis，大阪大学，Feb. 2023.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L.u. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in Neural Information Processing Systems, pp.5998–6008, 2017.

[5] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, “Deep learning based program generation from requirements text: Are we there yet?,” IEEE Transactions on Software Engineering, vol.48, no.4, pp.1268–1289, 2022.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp.4171–4186, 2019.

[7] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, et al., “Improving language understanding by generative pre-training,” 2018.

[8] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P.J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” The Journal of Machine Learning Research, vol.21, pp.5485–5551, Jan. 2020.

[9] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, “An empirical comparison of pre-trained models of source code,” Proceedings of the 45th International Conference on Software Engineering, p.2136 − 2148, 2023.

[10] 肥後芳樹，“自動テスト生成技術を利用した機能等価メソッドデータセットの構築,” ソフトウェアエンジニアリングシンポジウム 2023，pp.30–38，2023.