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

ChatGPTを用いたプログラム修正における ソースコードの意味的情報の影響

堀　翔太† 　柗本　真佑† 　肥後　芳樹† 　楠本　真二† 　安田　和矢††
伊藤　信治†† 　張潘タンフエン††

† 大阪大学大学院情報科学研究科
†† 株式会社 日立製作所
E-mail: †{sho-hori,shinsuke,higo,kusumoto}@ist.osaka-u.ac.jp,
††{kazuya.yasuda.fd,shinji.itoh.wn,thithanhhuyen.phan.gw}@hitachi.com

あらまし　ソフトウェア開発におけるデバッグ支援の一つとして，大規模言語モデル（LLM）を用いた自動プログラム修正（LLM-APR）が注目されている．既存研究では LLM-APR のプログラム修正性能を実験的に確認しているが，ソースコードに付帯する情報の影響は明らかにされていない．本研究では LLM-APR の性能改善を目的として，ソースコードが持つ意味的情報の影響を調査する．意味的情報とはソースコードのコンパイル・実行には影響を与えない，ソースコード理解のための情報であり，JavaDoc に記載された仕様やメソッド名などが該当する．調査では意味的情報の有無を制御した 4 種類のプロンプトを作成し，各プロンプトのプログラム修正成功の割合を調べる．調査の結果，多くの意味的情報を含むほど LLM-APR の性能が高くなる傾向にあることを確認した．LLM-APR は開発者によるソースコード理解と同様に，仕様や変数名などから修正対象メソッドの取るべき振る舞いを類推していると考えられる．

キーワード　大規模言語モデル（LLM），自動プログラム修正（APR），意味的情報，ChatGPT

1. は じ め に

ソフトウェア開発におけるデバッグ支援の一つとして，大規模言語モデル（LLM, Large Language Model）を用いた自動プログラム修正が注目されている．LLM とは大量のテキストデータを用いて学習した言語モデルの一種であり，様々な自然言語タスクに適用可能な汎用アプリケーションである．学習データにソースコードを含めた LLM も存在しており，ソースコードを対象とした様々なタスクにも応用できる．その応用の一つとして，LLM を用いた自動プログラム修正（APR, Automated Program Repair）が挙げられる．本稿ではこの手法を LLM-APR と略す．既存の APR 手法 [1] [2] の多くはソースコードの性質を考慮したプログラム修正への特化手法であるのに対し，LLM-APR は大量のデータに基づく学習と推論を基本とする点で実現手段が決定的に異なる．LLM-APR は自然言語による自然かつ対話的なプログラム修正が可能という利点だけでなく，その修正精度も既存 APR 技術と同等かそれ以上であると報告されている [3].

社会に大きなインパクトを与えた ChatGPT の公開が 2022 年 11 月という点から，根本的に LLM-APR 研究は黎明期にあり，様々な性能改善の可能性が残されている．Sobania ら [4] は ChatGPT を用いた LLM-APR の性能を実験的に確かめており，全 40 個のバグのうち 19 個の修正に成功したと報告している．しかしながら，この研究では *"Does this program have a bug? How to fix it?"* のような最小限のタスク指示と，バグを含むソースコードのみがプロンプトとして利用されている．ChatGPT のような対話型の生成系 AI においては，プロンプトの設計がその性能に大幅な影響を与える [5] ことが知られている．よって，LLM-APR に対するタスクの指示文章をどう設計するか，あるいは修正対象となるソースコードにどのような情報を持たせるかは，その性能を左右する重要な要素であるといえる．

本稿では，LLM-APR への入力データとなるソースコードが持つ意味的情報に着目する．ここでの意味的情報とはコンパイルや実行には一切寄与しない，開発者によるソースコード理解のための付帯情報のことを指す．例えば，Java における JavaDoc や Python における docstring などの仕様は意味的情報の一つである．これらのドキュメントには，メソッドの目的や振る舞い，利用方法などの意味や解釈が記載されており，対象メソッドの理解に寄与する．同様に，メソッド名はそのメソッドの責務を極めて端的に表現した意味的情報の一つである．さらにソースコードの内部に着目すると，変数名も一種の意味的情報だと見なせる．変数名には対象変数が持つ値の意味や役割が付与されているためである．

本研究の目的は LLM-APR におけるプログラム修正性能の改善であり，そのためにソースコードに付帯する意味的情報の効果を実験的に確かめる．実験では，3 種類の意味的情報（仕様・メソッド名・変数名）の有無を変化させた 4 種類のソー

This article is a technical report without peer review, and its polished and/or extended version may be published eleswhere.
Copyright ©2024 by IEICE

スコードを用意する．さらに，ミューテーション解析によって各ソースコードに人為的なバグを埋め込む．このバグを含むソースコードをプロンプトとして ChatGPT に与えることで，その修正の成否を確かめる．実験の結果，意味的情報を多く含むソースコードほど修正の成功率が向上することが確認できた．また，仕様を併記することで修正成功率が 5% 改善している点から，開発者によるプログラム理解と同様に，LLM に対しても仕様がプログラム理解の手助けになっている可能性が示唆された．

2. 準 備

2.1 大規模言語モデル（LLM）

大規模言語モデル（LLM，Large Language Model）とは大量のテキストデータを用いて学習された言語モデルの一種である．LLM は翻訳，要約，分類などの多くの自然言語処理タスクを実行できる．代表的な LLM として，OpenAI 社によって開発された GPT-3 [6] や，Google 社によって開発された BERT [7] がある．これらの LLM は，従来の畳み込みニューラルネットやリカレントニューラルネットを用いた深層学習と異なり，Transformer モデルを用いて学習を行っている．

LLM はソフトウェア工学の分野でも大きな注目を集めている．LLM を用いたソフトウェア開発支援の一つとして GitHub Copilot(注1)がある．これは LLM の一種である Codex [8] を用いたコーディング支援技術である．Codex とは GPT-3 をソースコードに関するタスクにファインチューニングしたモデルである．GitHub Copilot ではコード生成やコメント生成，プログラム修正，コード補完などのプログラミング作業を支援する．Jesse ら [9] は Codex が補完するコードの品質を調査している．調査の結果，Codex のコード補完タスクでは，正解のソースコードより多くバグを含むソースコードを生成しており，未だ精度改善の余地があると確認されている．

2.2 自動プログラム修正（APR）

自動プログラム修正（APR，Automated Program Repair）とは，バグを含むプログラムから全自動でバグを取り除く技術である．既存の APR 手法として，探索ベースの手法 [1] やテンプレートベースの手法 [10]，意味論ベースの合成手法 [11] などが挙げられる．これらの手法は，バグを含むプログラムとテストを入力として与え，全てのテストケースを通過するプログラムを目標としてプログラム修正する．いずれもソースコードの静的・動的解析やプログラムの修正事例などを用いた，ソースコード特化の手法であるといえる．

2.3 LLM を用いた自動プログラム修正（LLM-APR）

LLM-APR とは LLM を利用した APR 技術であり，バグを含むソースコードとバグ修正の指示文を LLM に与えることで，修正されたソースコードを得る．一般的な APR とは異なり，LLM-APR では自然言語文章を用いた自然かつ対話的なプログラム修正が可能である．

ChatGPT に代表されるサービス型の生成系 AI の登場に伴い，LLM-APR の実現可能性を確かめる研究が進められるようになってきた．Sobania ら [4] は ChatGPT を用いた LLM-APR のプログラム修正能力を実験的に検証している．この研究では，簡単な指示文章とバグを含むソースコードを LLM への入力に用いている．実験ではプログラミングコンテストから得られた QuixBugs [12] と呼ばれるバグデータセットを用いており，40 個のバグに対し 19 個の修正の成功を確認している．より大規模な LLM-APR 調査として，Xia ら [3] は複数の LLM モデルと複数のバグデータセットを用いた実験を行っている．結果として，QuixBugs データセットに対しては 37 個の修正に成功したと報告している．LLM を用いない従来の APR 手法の比較研究 [13] によると，全 10 種類の APR 手法を統合しても QuixBugs に対する修正成功数は 16 個であるとされている．よって，現時点でも LLM-APR の性能は従来の APR を大幅に上回っているといえる．

また，LLM の活用においてはプロンプトの改善がその性能に大きく影響する [5] ことが知られている．先の Sobania と Xia の研究では，few-shot 学習や one-shot 学習の効果を確かめており，いずれも性能改善に寄与している．

一方で，修正対象となるソースコードにどのような情報をもたせるべきかについては明らかになっていない．ソースコードは自然言語の文章と同様，高い自由度を持っており，同じ内容であっても様々な表現での記述が可能である．例えば，メソッド名や変数名は開発者の考えや癖に強く依存する要素である．これらの識別子名はソースコード理解に強く寄与する [14] [15] ことが知られている．また JavaDoc 記載の仕様はそのメソッドの責務の把握に役立つ．しかしながら，仕様や識別子名が LLM によるバグ修正にも貢献するのか，あるいはどのような仕様や識別子名が LLM に対して有用かについては明らかではない．

3. 調査の目的

本研究の調査目的は，ソースコードの持つ意味的情報が LLM-APR に与える影響の分析である．ここでの意味的情報とはコンパイルや実行には一切寄与しない，開発者のソースコード理解のための付帯情報を指す．より具体的には，仕様，メソッド名及び変数名の 3 種類を意味的情報と見なす．JavaDoc などに記載された仕様は対象メソッドの責務や利用方法，引数などの多くの意味や解釈が記載された意味的情報である．さらにメソッド名は仕様ほど多くの情報を含まないが，そのメソッドの責務を極めて端的に表現しており一種の意味的情報とみなせる．同様に，変数名はメソッド全体の意味や解釈を表現しないが，対象変数が持つ値の意味や役割が付与された意味的情報といえる．これらの情報はプログラム実行には影響を与えないものの，開発者のソースコード理解に大きく寄与すると報告されている [14] [15]．本研究では，これらの開発者のソースコード理解のための情報が，LLM に対しても有効であるかを確認する．

(注1)：https://github.com/features/copilot

4. 調 査 設 計

4.1 調査の流れ

本研究の調査は 図 1 に示す以下 6 ステップから構成される．

1. 対象メソッドを改変し意味的情報の有無を制御する
2. 対象メソッドに人工的にバグを埋め込む
3. 対象メソッドからプロンプトを生成する
4. LLM によりプログラム修正を試みる
5. プログラム修正の成否を確認する
6. ステップ 2 に戻り新たなバグを埋め込む

表 1 に実験に用いたパラメタの一覧を示す．以降では実験の流れに従い，実験題材と実験対象ソースコードの単位，意味的情報の制御方法，人工的なバグの埋込方法，プロンプトの設計，LLM モデル，プログラム修正の成否判定方法についてそれぞれ説明する．

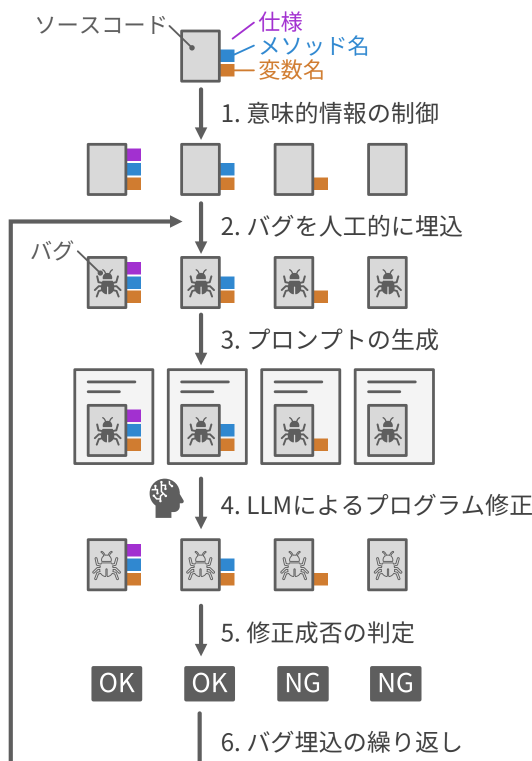

図 1: 調査の流れ

表 1: 実験に用いたパラメタ

パラメタ	値
実験題材	java.util.Arraylist の全 41 メソッド
仕様	JavaDoc をそのまま使用
メソッド名と変数名の制御方法	構文解析によるマスキング
バグの埋込方法	ミューテーション解析(注2)
バグを含むメソッドの数	150 個
LLM モデル	gpt-3.5-turbo

(注2)：https://github.com/kusumotolab/Mutanerator

4.2 実験題材と実験対象ソースコードの単位

実験題材として Java 標準ライブラリに含まれる `java.util.ArrayList` クラスを採用する．本クラスは全てのメソッドに JavaDoc が記載されており，メソッド名や変数名も適切に記述されている．よって，これらの意味的情報がメソッド理解に役立つ内容であると判断したためである．

プロンプトに与えるソースコードは，個々のメソッドを用いる．すなわち 図 1 に示すソースコードは全てメソッドに該当する．`ArrayList` クラスは 41 個のメソッドを含んでおり，全メソッドを実験の対象とする．

4.3 意味的情報の制御方法

意味的情報の一つである仕様は，JavaDoc に既に記載されている内容を改変せずそのまま用いる．JavaDoc 記載の内容を利用することで，著者らの主観を排除した実験が可能となる．なお，JavaDoc にはメソッドの著者`@author` や，リンクの参照`@link` などのメタ情報のほかに，API ドキュメント生成時の HTML の修飾情報なども含まれる．これらはメソッドの責務の理解には寄与しないため，プログラム修正への影響はないと考えられるが，本調査ではこのメタ情報や修飾情報の排除はしていない．

メソッド名と変数名の制御は構文解析によるマスキングを用いる．`ArrayList` クラスの全てのメソッド名と変数名は適切な内容が付与されている．よってこれらを`$1` や`$2` という意味を持たない文字列に正規化し，意味的情報の有無を制御する．正規化の際には，コンパイルやプログラム実行に一切の影響を与えないように処理する．

4.4 人工的なバグの埋込方法

バグの埋込方法として，ミューテーション解析(注2)を用いる．これは，生成規則に基づいてソースコードを改変し，バグを人工的に埋め込む技術である．例えば，`<`を`<=`に変換する関係演算子の境界の変更や，`+` を`-`に変換する算術演算子の書き換えなどの生成規則がある．ミューテーション解析は人工的にバグを埋め込むため，バグの種類や位置を正確に把握できる．また生成規則に従って大量のバグを埋め込むことが可能である．本調査では，ミューテーション解析を用いて単一のバグを持つソースコードを生成する．実験題材である `ArrayList` の全 41 メソッドにミューテーション解析を適用し，単一のバグを含むソースコードを 150 個生成した．この 150 個のバグを含むソースコードを全て実験に用いる．

4.5 プロンプトの設計

LLM への入力となるプロンプトの基本構造は指示文章とソースコードのペアである．指示文章には *“The method has a bug. Please fix the bug.”* のような，ソースコードにバグが含まれている旨，及びタスクがバグ修正である旨が含まれる．よって想定する利用シーンは，開発者がバグの存在に気づいており，かつその修正方法が不明という状況である．

プロンプトの取り得る種類は 3 種類の意味的情報の有無の組み合わせであり，全 8（2^3）通りとなる．以降では，この 8 通りのプロンプトをプロンプトパターンと呼び，$P_{--/--/--}$ の形式で表現する．まず，最も多くの意味的情報を持つパターン

は $P_{sp/mn/vn}$ であり，仕様（**sp**ecification），メソッド名（**m**ethod **n**ame），変数名（**v**ariable **n**ame）の全てを含む．また，仕様を持たずメソッド名と変数名を含む場合は $P_{--/mn/vn}$ となり，これが 2.3 節で述べた Sobania ら [4] が実施していた調査のプロンプトとなる．この $P_{--/mn/vn}$ を実験のベースラインと見なす．

本研究では 8 通りのプロンプトパターンの内，表 2 に示す 4 種類のプロンプトパターンを用いる．各意味的情報のソースコード理解への寄与度は仕様・メソッド名・変数名の順に高くなると考えられる．仕様はメソッド全体の責務を自然言語で表現しており，メソッド名はその責務を簡略化し表現している．変数名は処理内部の理解には役立つものの，メソッド全体の理解には直接的に貢献しないためである．よって今回は仕様・メソッド名・変数名の順に有無を制御したプロンプトパターンを用いる．

表 2: 調査に用いる 4 種類のプロンプトパターン

	仕様	メソッド名	変数名
$P_{sp/mn/vn}$	○	○	○
$P_{--/mn/vn}$ *	×	○	○
$P_{--/--/vn}$	×	×	○
$P_{--/--/--}$	×	×	×

* 実験のベースラインとなるプロンプト

```
### prompt ###
The method has a bug. Please fix the bug.

### method ###
public boolean $1() {
    return $2 != 0;
}
```

(a) $P_{--/--/--}$ の例

```
### prompt ###
The method follows the specification.
And it has a bug.
Please fix the bug.

### specification ###
Returns true if this list contains no elements

### method ###
public boolean isEmpty() {
    return size != 0;
}
```

(b) $P_{sp/mn/vn}$ の例

図 2: プロンプトパターンの一例

2 つのプロンプトパターンの具体例を 図 2 に示す． 図 2(a) は $P_{--/--/--}$ であり，図 2(b) は $P_{sp/mn/vn}$ に対応している． 図 2(a) を確認すると，全ての意味的情報がマスクされた状態であり，修正対象メソッドの責務や変数（この場合フィールド）の役割が読み取れない．そのためこれだけの情報でプログラム修正は困難である．他方， 図 2(b) はメソッド名（`isEmpty`）と変数名（`size`）が付与されており，何らかの集合のサイズが空であるかを確認するメソッドだと推測できる．さらに仕様から読み取れる情報より，その推測が正しいと窺える．今回の例では 図 2(b) の `return size != 0;` にバグが含まれており，`return size == 0;` に修正する必要がある．各意味的情報から読み取れる内容より，$P_{--/--/--}$ と比較してメソッドの責務が明確であるため，プログラム修正の難易度も下がっていると思われる．なお，プロンプトの指示文章については，$P_{sp/mn/vn}$ のときのみ *"The method follows the specification."* という，修正対象メソッドが仕様に従うという旨が追加されている．

4.6 LLM モデル

実験に使用する LLM のモデルは gpt-3.5-turbo である．入力としてテキストを受け取り，テキストを出力するモデルである．gpt-3.5-turbo には出力トークン数の上限が存在しているが，本研究において出力トークン数の上限を上回ることは無かったため問題はない．

gpt-3.5-turbo を呼び出す際のパラメータとして，`temperature=0` としている．`temperature` は，LLM の出力のランダム性に関係する．gpt-3.5-turbo では，0～1 で指定され，0 に近いほど程ランダム性が小さくなる．ランダム性を小さくし再現性を確保するため，Xia ら [3] は 0 を採用している．同様の理由で本研究でも 0 を用いる．

4.7 プログラム修正の成否判定方法

LLM-APR によるプログラム修正の成否判定としては，目視により振る舞いを考慮した確認を行う．先述の通りソースコードは様々な記述方法が可能であるため，バグを埋め込む前のソースコードとのテキスト的な一致の確認では不十分となる．ここでは，LLM-APR の出力した全ソースコードを目視で確認し，意味的に正しい修正が行われているかを調べる．

5. 結果と考察

5.1 プログラム修正の成否結果

各プロンプトパターンのプログラム修正の成否結果を 図 3 に示す．プログラム修正の成功数と失敗数を積み上げ式で記述している．ここで成功数と失敗数の合計が各プロンプトパターンで一致していないのは，gpt-3.5-turbo の呼び出しにおいてタイムアウトエラーが発生し，解答を得られなかった場合があるからである．

図 3 より，各プロンプトパターンの修正成功率は 64%，59%，50% 及び 24% であった．全ての意味的情報を含む $P_{sp/mn/vn}$ の精度が最も高く，一切含まない $P_{--/--/--}$ の精度が最も低かった．よって，LLM-APR において 3 種類の意味的情報全てがその性能向上に寄与していると考えられる．またベースラインとなる $P_{--/mn/vn}$ と仕様を加えた $P_{sp/mn/vn}$ を比較すると，5% の改善が見受けられる．開発者によるプログラム理解と同様に，LLM に対しても仕様がプログラム理解の手助けになっている可能性がある．

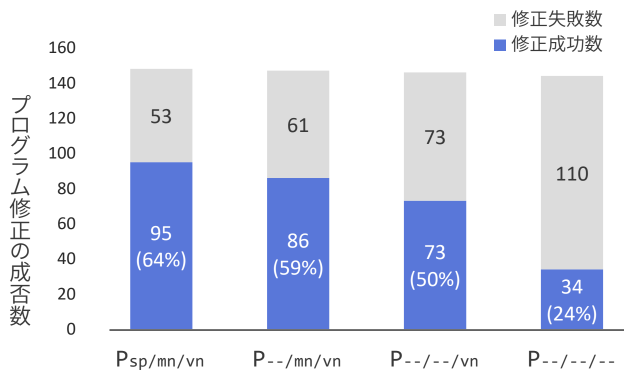

図 3: プロンプトパターンごとのプログラム修正の成否数

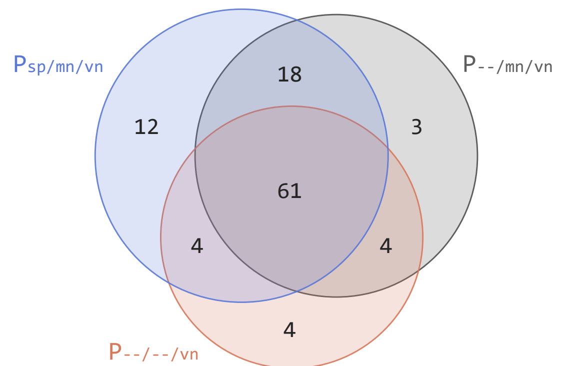

図 4: 修正に成功したバグの包含関係（$P_{--/--/--}$ を除く）

$P_{sp/mn/vn}$ と $P_{--/mn/vn}$ の修正成功率の差は 5%，$P_{--/mn/vn}$ と $P_{--/--/vn}$ の修正成功率の差は 9%，及び $P_{--/--/vn}$ と $P_{--/--/--}$ の修正成功率の差は 26% となっていることから，LLM-APR の修正精度に最も影響を与えている意味的情報は変数名だと推測できる．ただし，今回の実験では 3 種類の意味的情報の全組み合わせである 8 通りのプロンプトパターンを用いて実験をしていない．今後，プロンプトパターンを拡張しそれぞれの意味的情報の影響について調査する必要がある．

5.2 プログラム修正に成功したバグの包含関係

最も精度の低かった $P_{--/--/--}$ を除く，3 プロンプトパターンの修正成功バグの包含関係を 図 4 に示す．図より 61 個のバグは 3 種のプロンプトパターン全てで修正に成功しており，多くのバグは包含関係を持つ．これはバグの種類によって修正の難易度が異なるためだと考えられる．これら共通集合となる 61 個のバグは LLM にとって修正難易度の低いバグであり，意味的情報の有無にかかわらず修正しやすい可能性がある．一方で，意味的情報を多く含むプロンプトが含まないプロンプトを完全に内包していないことも確認できる．例えば，仕様を与えないベースライン手法のみで成功するバグが 3 個（図の右上）あり，仕様もメソッド名も与えない手法のみで成功するバグが 4 個（図の下）存在している．この理由は LLM のランダムな挙動に起因すると考えられる．本稿では各プロンプト・各バグに対して 1 試行のみしか実験しておらず，複数試行による実験が必須である．なお，意味的情報の過多が性能低下の要因となっている可能性も考えられる．この点については続く 6.1 節で議論する．

6. 議　　論

6.1 意味的情報の効果を左右するソースコードの要素

意味的情報の有無が LLM-APR の性能に与える影響は，修正対象メソッドの責務や行数の多さに左右されると推測している．仕様が修正成功率の改善に寄与するバグ（ 図 4 の左上）は，責務が多く長いメソッドに多く含まれる傾向があった．責務が多く長いメソッドほど，バグの候補となりうる箇所が多くプログラム修正の精度は下がる．そのため仕様の追加による改善幅が大きかったのだと考えられる．実際，`ArrayList` クラス内で最も行数の多い 30 行のメソッドにおいて，$P_{sp/mn/vn}$ と $P_{--/mn/vn}$ の修正成功率は 55% と 32% であった．よって修正成功率の差は 23% であり，`ArrayList` の全メソッドを対象とする場合と比較してその改善幅は大きい．一方，責務が多く長いメソッドではメソッド名を含まないことが性能改善に寄与すると考えている．実際，メソッド名をマスクする手法のみで成功するバグ（図の下）は，責務が多く長いメソッドに含まれる傾向があった．`ArrayList` 内の最も行数の多いメソッドにおいて $P_{--/--/vn}$ の修正成功率は 59% であり，これは $P_{sp/mn/vn}$ の 55% と比較しても高い精度である．責務が多く長いメソッドは，そのすべての責務を完全に表現できるメソッド名を命名しづらい．よってメソッド名が表現できていない範囲にバグが存在するとき，メソッド名が LLM に対してミスリードを招き，プログラム修正が失敗する可能性がある．ただしこれは判断材料となる数が少ないため，今後メソッドの責務の多さを定量的に表現し，その関係を調査していく必要がある．

6.2 LLM-APR に効果的な仕様

本稿では仕様として JavaDoc を用いたが，LLM-APR に利用できる仕様の選択肢は他にもある．選択肢の一つとして，自動コード要約が考えられる．コード要約により生成された要約を仕様として用い LLM-APR の性能向上を図る．コード要約が性能向上に寄与する場合，開発者が JavaDoc を作成しなくとも高い精度のプログラム修正が期待できる．ただし実際の利用シーンを考えると，バグを含むソースコードから要約する必要があるため，要約の内容自体がバグの影響を受けると考えられる．2.1 節で述べた Jesse ら [9] は，バグを含むソースコードから生成されたコメントをプロンプトに組み込み，LLM によるコード補完性能の検証をしている．調査の結果，コメントが無い場合より Codex によるコード補完の性能が向上したと報告している．同様に，LLM-APR においてもバグを含むソースコードから作成された要約が性能向上に寄与する可能性はあると考えられる．

さらに他の仕様として，入出力例やテストケースがある．入出力例は対象メソッドのとるべき挙動を示し，テストケースは対象メソッドの取る挙動が想定通りか確認する．どちらも対象メソッドの振る舞いを示しており一種の仕様といえる．また実際の開発において，開発者は入出力例やテストケースを持っているため利用シナリオにも即しており，これらの影響

の調査は残された調査課題の一つである．

6.3 LLM-APR に効果的なメソッド名と変数名

メソッド名と変数名の命名方法は様々であり，それぞれの名前が持つ意味によってソースコードの可読性は大きく変わる．Schankin ら [15] は，開発者のソースコード理解に役立つ識別子名について調査している．調査の結果，短く端的な識別子名に比べて長く複雑な識別子名の方がプログラマのソースコード理解は早くなると報告されている．複雑な識別子名はコード行が増え可読性を損ねる恐れがあるが，それ以上に名前の持つ意味情報の利益が大きいと推測できる．LLM-APR も同様に，複雑な識別子名が持つ意味情報が有効に働き，高い性能を示すと考えている．端的な識別子名と複雑な識別子名が LLM-APR の性能に与える影響の調査はさらなる研究課題である．

6.4 LLM のデータリーケージ

LLM の学習データに今回の修正対象となったソースコードの情報が存在しており，LLM-APR の正しい性能評価ができない可能性がある．本稿で使用した gpt-3.5-turbo モデルの学習データは公開されていない．そのためデータリーケージの問題を完全に検証することは不可能である．

本稿の調査では，同一メソッドに対して LLM-APR を適用した場合でも，埋め込んだバグが異なると修正成否の結果が異なる場合があった．例えば，$P_{\texttt{sp/mn/vn}}$ の場合，全 41 メソッドの内 15 メソッドにおいて，修正成功と修正失敗が混在している．これは LLM-APR がそれらの 15 メソッドにおいてはデータリーケージによる出力をしていないと考えられる．データリーケージの対策は，公開されていないソースコードを用いる，またモデルの学習データの期間が公開されている場合は，その期間から外れたソースコードを用いるなどがある．

7. おわりに

本研究では LLM-APR におけるソースコードの持つ意味的情報の影響について調査した．本稿での意味的情報とはコンパイルや実行には寄与しない開発者のソースコード理解のための情報のことで，仕様，メソッド名及び変数名の 3 つである．調査の結果，3 種類全ての意味的情報が LLM-APR の性能向上に寄与すると分かった．これは，開発者のソースコード理解と同様に，LLM も意味的情報からソースコード理解をし，プログラム修正している可能性を示唆している．

今後取り組むべき課題として，JavaDoc の適切な整形と全プロンプトパターンの実施が挙げられる．本稿では JavaDoc に含まれるメタ情報の排除をしていない．これらが LLM-APR に与える影響は不明だが，仕様の影響を正しく調査するために適切な整形は必要である．また，本稿では 3 種類の意味的情報の組み合わせである 8 通りのプロンプトパターンについて調査していない．それぞれの意味的情報の影響を正しく検証するためには必須な項目であり今後の課題である．

さらに 2 つの発展的な研究課題への取り組みを検討している．一つ目は，意味的情報の影響を左右するソースコードの特徴の調査である．修正対象とするメソッドの責務の多さにより，意味的情報の影響は左右されると考えている．メソッドの責務を行数などで定量的に表現し，意味的情報との関係を調査する必要がある．2 つ目は，LLM-APR に効果的な意味的情報の調査である．自動コード要約や入出力例，テストケースを仕様として用いて LLM-APR を行い，JavaDoc を必要としない LLM-APR の性能向上について調査する．また，端的な識別子名と複雑な識別子名による LLM-APR の精度を比較調査し，LLM にとって適した識別子名を明らかにする．

謝 辞 本研究の一部は，JSPS 科研費（JP20H04166, JP21K18302, JP21K11829, JP21H04877, JP22H03567, JP22K11985）による助成を受けた．

文 献

[1] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, “kGenProg: A high-performance, high-extensibility and high-portability apr system,” In Proceedings of Asia-Pacific Software Engineering Conference, pp.697–698, 2018.
[2] M. Martinez and M. Monperrus, “ASTOR: a program repair library for java,” In Proceedings of Internatinal Symposium on Software Testing and Analysis, pp.441–444, 2016.
[3] C.S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” In Proceedings of International Conference on Software Engineering, pp.1482–1494, 2023.
[4] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An analysis of automatic bug fixing performance of chatgpt,” In Proceedings of International Workshop on Automated Program Repair, pp.23–30, 2023.
[5] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D.C. Schmidt, “A prompt pattern catalog to enhance prompt engineering with chatgpt,” arXiv, p.arXiv:2302.11382, 2023.
[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., “Language models are few-shot learners,” In Proceedings of International Conference on Neural Information Processing Systems, pp.1877–1901, 2020.
[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” arXiv, p.arXiv:1810.04805, 2018.
[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., “Evaluating large language models trained on code,” arXiv, p.arXiv:2107.03374, 2021.
[9] K. Jesse, T. Ahmed, P.T. Devanbu, and E. Morgan, “Large language models and simple, stupid bugs,” In Proceedings of International Conference on Mining Software Repositories, pp.563–575, 2023.
[10] K. Liu, A. Koyuncu, D. Kim, and T.F. Bissyandé, “TBar: Revisiting template-based automated program repair,” In Proceedings of International Symposium on Software Testing and Analysis, pp.31–42, 2019.
[11] S. Jha, S. Gulwani, S.A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” In Proceedings of International Conference on Software Engineering, pp.215–224, 2010.
[12] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge,” In Proceedings of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, pp.55–56, 2017.
[13] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, “A comprehensive study of automatic program repair on the quixbugs benchmark,” In Proceedings of International Workshop on Intelligent Bug Fixing, pp.1–10, 2019.
[14] R. Cates, N. Yunik, and D.G. Feitelson, “Does code structure affect comprehension? on using and naming intermediate variables,” In Proceedings of International Conference on Program Comprehension, pp.118–126, 2021.
[15] A. Schankin, A. Berger, D.V. Holt, J.C. Hofmeister, T. Riedel, and M. Beigl, “Descriptive compound identifier names improve source code comprehension,” In Proceedings of Conference on Program Comprehension, pp.31–40, 2018.