Vol.2020-SE-204 No.1
2020/3/2

静的解析ツールの警告に対する自動バグ修正技術の適用と初期評価

浅田 翔[1,a)] 首藤 巧[1,b)] 山手 響介[1,c)] 佐藤 亮介[1,d)] 亀井 靖高[1,e)] 鵜林 尚靖[1,f)]

概要：ソフトウェア開発において幅広く利用されている静的解析ツールの中には，プログラム内のバグになり得る実装（静的解析エラー）を警告として検出するものがある．静的解析エラーを修正することはソフトウェアの品質の向上に貢献するが，静的解析エラーは開発者による修正が困難であるという問題点が存在する．近年のソフトウェア工学では，静的解析エラーやテストを失敗させるバグ（テストエラー）を自動的に修正するための研究が行われている．我々はテストエラーの修正内容と静的解析エラーの修正内容との間に関係性があると考え，テストエラーに対する自動バグ修正技術の枠組みが静的解析エラーにも応用できる可能性について着目をした．本研究では，現在開発中であるテストエラーに対する自動バグ修正ツール jProphet に静的解析エラーを修正する機能を追加し，初期評価として実際にいくつかの OSS で適用実験を行った．実験を行った結果，データセットに含まれる 60 件の静的解析エラーのうち 4 件が jProphet によって修正され，そのうち 1 件は目視調査によって正しい修正であると確認された．実験結果より本研究で提案されている自動バグ修正技術で静的解析エラーを修正するためには，修正パターンが不十分である課題を解決することが必要であると分かった．

キーワード：自動バグ修正, 静的解析ツール

1. はじめに

近年，ソフトウェア開発におけるソフトウェアの品質向上を目的として静的解析ツールが開発されている．静的解析ツールはプログラムのソースコードやバイトコードを入力として受け取り，プログラムの性質を解析するツールである．

静的解析ツールには様々な種類があるが，その中には変則的な実装や処理効率の悪い実装を警告として検出するものがある．（以降，本稿では静的解析ツールの警告内容を静的解析エラーと記述する．）静的解析ツールを利用することで開発の初期段階でプログラムの間違った仕様を生みやすい実装を変更することが可能になり，バグを修正するコストを削減することが出来る．

しかし，静的解析ツールは開発者にあまり受け入れられていないという問題点がある．Johnson らの調査 [1] は静的解析ツールの警告の数が多すぎて開発者が処理しきれないことや一部の警告メッセージは理解が困難であることを指摘している．

静的解析ツールの問題点を解決するために，近年のソフトウェア工学では静的解析エラーを自動的に修正する研究が行われている．これらの研究の多くは，それぞれの静的解析エラーに対応するコード修正パターンを作成する手法を利用しており，実際に OSS の警告を自動的に修正することに成功している．

一方でバグを自動的に修正する研究も盛んに行われており，様々な手法が提案されている．自動バグ修正手法の多くはプログラムの構文は正しいが仕様が間違っているバグ（以降，意味的エラーと記述する）を修正対象にしている．

自動バグ修正技術に関連する研究は様々だが，自動バグ修正技術の修正対象を変更する研究は少ない．静的解析ツールはプログラムの間違った仕様を生みやすい実装を検出していることから，静的解析エラーが存在するコードと意味的エラーが存在するコードに関連性があると考えられる．そこで，我々は意味的エラーに対する自動バグ修正技

1 九州大学
a) asada@posl.ait.kyushu-u.ac.jp
b) shuto@posl.ait.kyushu-u.ac.jp
c) yamate@posl.ait.kyushu-u.ac.jp
d) sato@ait.kyushu-u.ac.jp
e) kamei@ait.kyushu-u.ac.jp
f) ubayashi@ait.kyushu-u.ac.jp

ⓒ 2020 Information Processing Society of Japan

プログラム 1　静的解析エラーを含むコード例

```
public void foo() {
    int x = 3;
    \\ エラー：ローカル変数の自己代入
    x = x;
}
```

術を静的解析エラーに対して適用した場合に有効であるかという点について着目した．

本研究では，静的解析エラーに対して自動バグ修正技術を適用するために，既存の自動バグ修正技術の枠組みを利用して静的解析エラーに対する自動修正を行うツールを開発した．そして，ツールの初期評価として実際にいくつかの OSS プロジェクトに対して動作実験を行い，結果を考察した．

本稿では，2 節で静的解析ツールの背景と本研究に関連するいくつかの先行研究を紹介する．3 節では本研究の目的とそのアプローチについて述べる．4 節で自動バグ修正ツールの開発について述べ，5 節で開発した自動バグ修正ツールの初期評価を行った結果を述べる．その後 6 節では，自動バグ修正ツールの今後の改善案について延べ，7 節では，妥当性に対する脅威について述べる．最後に 8 節でまとめについて述べる．

2. 背景

2.1 静的解析ツール

SpotBugs. Pugh らによって開発された静的解析ツール SpotBugs は，Java プログラムコード内に含まれる約 100 種類の静的解析エラーを警告として検出するツールである．SpotBugs が検出する静的解析エラーには以下のような種類がある．

- 明らかなコーディングミス
- 推薦されるコーディング規範からの逸脱
- 誤解を与えやすいコードや間違いを招きやすいコード
- 実行効率が悪くなる可能性のあるコード

これらの静的解析エラーは将来的にバグを発生させる確率が高い．プログラム 1 に SpotBugs によって静的解析エラーが検出される Java コード例を示す．プログラム 1 では，"x = x;"のコード箇所に対して「ローカル変数の自己代入」という警告を出力している．同じ変数の値を代入する処理は無意味であるため，開発者が変数名を打ち間違えている可能性が高いと考えられる．SpotBugs を利用してバグになり得る実装が存在する箇所を特定することにより，プログラムの致命的なバグを事前に予防することが出来る．

静的解析ツールの現状に関する調査. Johnson らは，平均で 10 年の静的解析ツールの利用経験がある 20 人のソフトウェア開発者を対象としてアンケート調査を行った．[1] アンケート調査の結果，ほとんどの開発者は静的解析ツールの誤検知性や修正にかかるコストに対して不満を持っていることが判明し，Johnson らはこれらの点が原因で静的解析ツールが開発者にあまり利用されてないと考察した．また，[1] はどうすれば静的解析ツールの利用者が増えるかについて議論を行い，将来は開発者のワークフローに静的解析ツールを統合することが重要であると結論付けた．

2.2 関連研究

2.2.1 静的解析エラーの自動修正に関する研究

PHOENIX. Bavishi らが提案した PHOENIX[2] は，多数の OSS から静的解析エラーの修正パッチを収集し，独自の言語を用いて静的解析エラーに対する修正手順を学習していく手法である．PHOENIX は，Java で記述された OSS の各リビジョンに対して静的解析ツールを実行することで静的解析エラーを修正するパッチを収集している．修正パッチを収集した後，各修正パッチに独自の言語を適用することで静的解析エラー対する修正手順を学習していく．学習に用いる独自言語は修正パッチをトークンレベルに分割してアルゴリズムを適用しており，各静的解析エラーに対応した細かい修正手順を学習することを可能にしている．

Bavishi らは PHOENIX の性能を評価するために，学習に用いた OSS に対して PHOENIX による静的解析エラーの修正パッチを生成した．そして生成した修正パッチの正当性を確かめるため，Bavishi らは修正パッチを適用した後で静的解析ツールの警告が解消されるかどうかを確認した．その結果，OSS に含まれる全ての静的解析エラーのうち 84%は静的解析エラーを解消する修正パッチが生成され，そのうち 54%が開発者による修正と同等であることが確認された．また，Bavishi らは学習に用いられていない 5 件の GitHub プロジェクトに対しても同様に PHOENIX による自動修正を適用した．その結果，19 個の修正パッチが開発者によって受け入れられたことが分かり，一般のプロジェクトに対しても PHOENIX が有効であることが示された．

Sponge Bugs. Marcilio らが提案した Sponge Bugs[3] は，2 種類の静的解析ツールで検出される静的解析エラーのうち 11 種類を修正の対象として選択し，各静的解析エラーに対応する修正テンプレートを適用することによって修正を行う手法である．Marcilio らは Sponge Bugs の修正対象として，開発者によって頻繁に修正されている，かつプログラムの動作を変更せずに修正できることが保証されている警告を選択している．動作実験として Sponge Bugs による警告の自動修正を 12 個の OSS に適用したところ，920 個の修正パッチが生成され，そのうちの 84%が開発者に受

け入れられた．

2.2.2 自動バグ修正に関する研究

サーベイ論文． Gazzola の自動バグ修正に関するサーベイ論文 [4] によると，デバッグ作業はソフトウェアの開発コストの約 50%を占めていると報告されている．バグを修正する作業を自動化することで開発効率が向上すると考えられるため，バグを自動的に修正する研究が盛んに行われている．

[4] は自動バグ修正を行う上で二つのアプローチがあることを示している．そのうちの一つとして，バグがある箇所に対する修正候補の生成と修正候補の検証を反復するアプローチがあり，実際にこのアプローチを利用した自動バグ修正技術が多くあることが示されている．

AVATAR. Kui らが提案した [5] は自動バグ修正技術の性能向上のために，OSS における静的解析エラーの修正履歴から学習した修正パターンを活用している．AVATAR は GitHub 上の多数の OSS に対して各リビジョンに対して静的解析ツールを実行することにより，OSS に含まれる静的解析エラーの修正パッチを収集する．その後収集された修正パッチを数値ベクトルに変換し，CNN による特徴量の学習を行う．そして特徴量に基づいて静的解析エラーの分類を行い，バグに対する抽象化された修正パターンを取得する．そして AVATAR は，抽象化された修正パターンを元にしてプログラムの自動修正を行う．

Kui らは AVATAR の性能を評価するため，Java プロジェクトに含まれる意味的エラーを収集したデータセットである Defects4J[6] を利用した．1 度目の修正として，Defects4J のバグのうち静的解析ツールによって警告が検出されるものを対象に AVATAR による自動修正が行われた．自動修正を行なった結果，修正対象となった 31 件のバグのうち 25 件が修正され，そのうち 6 件が正しい修正パッチを適用していることが分かった．Kui らは次に Defects4J が提供するバグ全てに対して AVATAR による自動修正を行った．その結果，49 個の修正パッチが生成され，そのうち 34 個が正しく修正されていることが分かった．Kui らは AVATAR が Defects4J のバグを修正することを明らかにし，静的解析エラーに対する修正が意味的エラーにも有効であることを示した．

3. 動機

3.1 本研究の目的

静的解析エラーを修正することによりプログラムの仕様の間違いが発生することを防ぐことができることから，意味的エラーを修正するための内容と静的解析エラーを修正するための内容との間に関連性があると考えられる．そこで我々は意味的エラーの修正が副次的に静的解析エラーを修正する可能性について着目をした．

2 節で紹介した AVATAR では，静的解析エラーの修正パッチを学習して意味的エラーの修正を行う研究が行われているが，一つの自動バグ修正技術の枠組みで意味的エラーと静的解析エラーの両方を修正する研究は行われていない．自動バグ修正技術は今後発展していくと考えられるため，修正対象の幅を広げることは有用であると考えられる．本研究では自動バグ修正技術における意味的エラーの修正と静的解析エラーの修正の統一を目的として，意味的エラーに対する自動バグ修正技術が静的解析エラーにも応用できるかどうかを調査する．

3.2 アプローチ

ソフトウェア開発において，プログラムの仕様が正しいか確認するためテストスイートが利用されている．意味的エラーを修正する既存の自動バグ修正技術も多くの場合テストスイートを用いており，以下の処理を行っている．

(1) テストスイートによるバグの位置の特定
(2) 修正パッチ候補の生成
(3) テストスイートによる修正パッチ候補の検証

静的解析ツールはプログラムコードやバイトコードを受け取り静的解析エラーを検出するため，テストスイートは利用されない．自動バグ修正技術を静的解析エラーに適用するためには，上記の処理のうちテストスイートが関与する処理を静的解析ツールを利用する処理に変更する必要がある．

本研究では既存の自動バグ修正技術を参考にして新たな自動バグ修正ツール jProphet を開発した．そして，jProphet の処理の一部を変更することにより，静的解析エラーを修正する機能を実現する．また，ツールの初期評価として実際にいくつかの OSS プロジェクトに対して動作実験を行い，静的解析エラーに対して自動バグ修正技術を適用した際の結果や課題点について考察する．

4. 自動バグ修正ツールの開発

4.1 jProphet

jProphet は我々が開発を行っている自動バグ修正ツールである．jProphet は既存の自動バグ修正手法である Prophet[7] を参考に開発されており，Java で記述されたプロジェクトに含まれる意味的エラーの自動修正を行う．jProphet を実装した理由としては，Prophet の手法を Java プロジェクトに対応させることや本研究のために新たな機能を追加することが挙げられる．

4.1.1 自動バグ修正の手順

はじめに意味的エラーに対する jProphet の自動修正の手順を紹介する．jProphet の修正手順の概要を図 1 に示す．

対象プロジェクトが満たすべき条件． jProphet の自動バグ修正機能ではバグを含む Java で記述されたソースコードとテストスイートを入力として受け取る．テストスイート

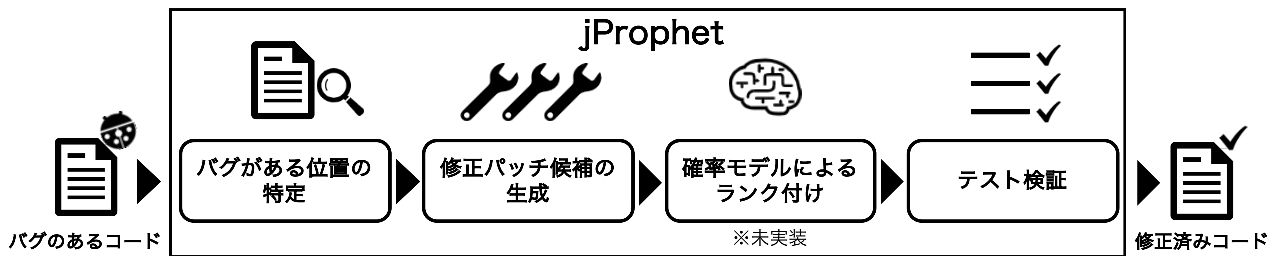

図 1　jProphet の修正手順

には最低でも 1 つのテストが失敗している必要がある．

バグ限局． まず，プログラム内のバグの存在する位置を特定する為に Spectrum-Based Fault Localization[8] と呼ばれる手法を利用する．Spectrum-Based Fault Localization はバグを含むプログラムとテストスイートを入力として受け取り，各テストが辿る実行経路を記録する．そして実行経路情報と各テストの成否を元に各行に対して一定の計算式を適用することでその行がバグを含む確率の高さを算出する．

修正パッチ候補の生成． はじめに対象のプログラムから抽象構文木（AST）を取得する．各 AST に対して，汎用的な変更を適用することでプログラムの修正パッチ候補を生成する．この汎用的な変更は修正テンプレートとして複数定義されており，修正パッチ候補は一つの修正テンプレートにつき複数生成される．jProphet で用いる修正テンプレートの一覧を表 1 に示す．これらの修正テンプレートは Prophet で利用されている PAR[9] と呼ばれる修正テンプレートの分類を参考にして決められている．

確率モデルによるランク付け． 生成された修正パッチに対して機械学習で得られた確率モデルを適用し，そのパッチが開発者による修正に近い確率を算出する．そして確率モデルで得られた確率とバグ限局で得られた疑惑値から修正パッチ候補を優先度が高い順に並び替える．この確率モデルは，OSS 内の開発者によるバグに対する修正履歴を学習データセットとして利用して生成されるものである．なお，現段階では確率モデルを準備できていないため，単に疑惑値が高い順に並び替える仕様となっている．

修正パッチ候補の検証． 確率モデルによって並び替えられた修正パッチ候補を上位から順番に修正対象のプログラムに適用し，テストスイートを実行する．全てのテストに成功するパッチが見つかった時点で検証を終了し，その修正パッチを jProphet の最終的な出力とする．

4.2　jProphet による静的解析エラーの自動修正

4.1.1 節で紹介した手法は，プログラムの意味的エラーを修正対象としている．本研究では自動バグ修正機能の他に静的解析エラーを自動修正する機能を追加した．静的解析エラーの自動修正機能は，前節で説明した自動バグ修正機能の手順と基本的に同じであるが，一部の内容を変更している．本節では静的解析エラーの自動修正機能の追加にあたって jProphet に変更を施した部分について紹介する．

表 1　jProhet における修正テンプレート

修正テンプレート	説明
文の複製	対象箇所の前のコードを対象箇所の直前に挿入し，さらに変数の置換を行う．
代入文やメソッドの引数の変更	対象箇所に含まれる変数の代入文やメソッドの引数を別の変数に置き換える．
メソッドの置換	対象箇所に含まれるメソッドを別のメソッドに置き換える．
if 文で囲む	対象箇所を if 文で囲む．
条件式の変更	対象箇所に含まれる条件式を変更する．
コントロールフロー文の挿入	特定の条件下で return 文や break 文などを実行するような処理を対象箇所の直前に挿入する．

対象プロジェクトが満たすべき条件． 静的解析ツールで検出される静的解析エラーはテストスイートにより発見されるバグとは異なり，テストが全て成功していて正常に動作するプログラムにも含まれる可能性がある．静的解析エラーの自動修正機能では後述する修正パッチ候補の検証の関係上，全てのテストが成功しているプロジェクトを修正対象としている．

バグ限局． 静的解析エラーの自動修正機能でのバグ限局では静的解析ツール SpotBugs が出力する警告を利用する．対象のプログラムに SpotBugs を適用してプログラムに含まれる静的解析エラーの箇所を取得し，その箇所全てに対して疑惑値を高く設定する．

修正パッチ候補の検証． 各修正パッチ候補が静的解析エラーを修正するかどうかを調べるために再び SpotBugs を利用する．SpotBugs による検証では，各修正パッチ候補によって修正されたプログラムに再度 SpotBugs を適用し，静的解析エラーのリストを取得する．修正前のプログラムから出力されたリストと比較して 1 つ以上が修正される，かつ修正後で新たなエラーが発生しなかった場合を成功とみなす．SpotBugs による検証の後，修正パッチ候補に対してテストスイートを実行し，全て成功するかどうかを確認する．そして SpotBugs による検証とテストスイートによる検証の両方で成功したものを採用し，対象のプログラムに含まれるそれぞれの警告に対して，修正に成功した修正パッチが生成された時点で検証を終了する．

ⓒ 2020 Information Processing Society of Japan

表 2 実験対象のプロジェクト

プロジェクト	概要	ファイル数	LOC
Math	基本的な数値計算	398	39,464
Lang	文字列に関する処理	108	21,788
Time	時間情報の取得	157	27,801

表 3 データセットに対する jProphet の適用結果

プロジェクト	エラー数	修正対象エラー	修正されたエラー
Math	36	34	2
Lang	31	12	1
Time	23	14	1
合計	90	60	4

検証に用いるテストスイートは修正の前後でプログラムの仕様が変わっていないことを確認するために利用する．修正前のプロジェクトに失敗するテストが存在する場合，新たに仕様を変更するような修正を検知出来ない可能性があるため，対象のプロジェクトは全てのテストが成功している必要がある．

5. 初期実験

5.1 データセット

修正対象のプロジェクトとして Defects4j の 3 プロジェクトを用いる．修正対象のプロジェクトの一覧を表 2 に示す．なお，Defects4j の他の 3 プロジェクトに関しては，プロジェクトの形式が jProphet に対応していないことやコンパイルができないなどの理由から修正対象から除外している．

5.2 実験内容

データセットの各プロジェクトに対して jProphet による静的解析エラーの自動修正を行う．jProphet の出力として，データセットに含まれる静的解析エラーに対して修正に成功した修正パッチが得られる．自動修正を行った後，静的解析エラーの修正に成功した修正パッチが開発者に受け入れられる内容であるかどうかを目視により確認する．

5.3 実験結果

実験を行った結果を表 3 に示す．データセットにおいて検出された静的解析エラーは合計 90 件であり，それらを内容ごとに分類したところ 29 種類に分けられた．表 4 にデータセットに含まれる 29 種類の静的解析エラーのうち 4 件以上検出されたものを示す．検出頻度が最も高い「内部表現を暴露するかもしれないメソッド」は，Java のクラスにおいて変更可能なフィールド値を外部に渡すメソッドを追加した場合に検出される．

検出された 90 件の静的解析エラーのうち jProphet が正常に動作しなかったものを除くと 60 件であった．60 件の静的解析エラーのうち jProphet で修正パッチが出力されたのは 4 件であった．さらに修正に成功した 4 件の修正パッチについて，修正の内容および目視で確認を行った結果を表 5 に示す．

表 4 データセットで検出頻度の高い静的解析エラーと修正例

エラー内容	検出された数			
	Math	Lang	Time	合計
内部表現を暴露するかもしれないメソッド	23	0	2	25
default がない switch 文	0	7	3	10
String オブジェクトを == や != を使用して比較している	0	8	0	8
デフォルトエンコーディングへの依存	5	1	1	7
戻り型が Boolean のメソッドが明示的に null を返す	0	6	0	6
hashCode メソッドを定義していないクラス	0	0	4	4
浮動小数点同士の比較	4	0	0	4

表 5 jProphet によって修正された静的解析エラー

エラーの内容	適用したテンプレート	開発者に受け入れられるか
String オブジェクトを==や!=を使用して比較している	条件式の変更	×
変数の自己代入	コントロールフロー文の挿入	×
内部表現を暴露するかもしれないメソッド	コントロールフロー文の挿入	×
効率の悪い String コンストラクタの呼び出し	代入文やメソッド引数の変更	○

修正に成功した 4 件の修正パッチのうち，開発者に受け入れられる修正を行っていたのは 1 件であった．開発者に受け入れられる修正をパッチ A，それ以外の 3 件のうち 1 つの修正をパッチ B とし，それぞれプログラム 2 とプログラム 3 に示す．

パッチ A では，静的解析エラーとして検出された効率の悪い String コンストラクタの呼び出し部分を String 型変数の呼び出しに変更しており，これは開発者に受け入れられる修正だと考えられる．一方パッチ B では変数の自己代入が静的解析エラーとして検出されているが，修正パッチでは対象の箇所の直前にコントロールフロー文を挿入することで対象の箇所以降の処理が必ず実行されない修正を行っており，これは開発者に受け入れられない修正と考えられる．他の 2 件についても同様の修正を行っていた．

5.4 考察

本節では 2 つの観点から考察を行う．

なぜほとんどの静的解析エラーが修正されなかったのか． データセット内のほとんどの静的解析エラーが修正されなかった理由として，修正テンプレートを 1 箇所適用するだ

プログラム 2　パッチ A

```
  String sub = token.substring(1);
  if (sub.length() == 1) {
    builder.appendLiteral(sub.charAt(0));
  } else {
    // エラー：効率の悪い文字列クラスの
    // コンストラクタの呼び出し
-   builder.appendLiteral(new String(sub));
+   builder.appendLiteral(sub);
  }
```

プログラム 3　パッチ B

```
  dest.m1 = source.m1;
  dest.dev = source.dev;
+ if (true)
+   return;
  //エラー：変数の自己代入
  dest.nDev = dest.nDev;
```

けで修正できるものが限られていることが考えられる．例えば SpotBugs が出力する警告に「メソッド名は小文字から始めるべき」というものがある．これに対する修正案として，メソッド名の先頭を小文字にする変更が考えられるが，正しく修正を行うためにはメソッドを参照している箇所全てに対して変更を行う必要がある．

もう一つの理由として，修正対象のエラーの中には，その内容に特化した変更を必要とするものがあることも考えられる．SpotBugs が出力する警告に「equals メソッドを定義していないクラス」があるが，これを修正するためには元のソースコードに存在しないメソッド名を利用した変更を行う必要がある．このような変更は汎用的な修正テンプレートとして実現することが困難である．

なぜ開発者に受け入れられない修正パッチが生成されたのか．開発者に受け入れられない 3 件の修正パッチについて調べたところ，いずれもコントロールフロー文の挿入や条件式の変更を適用していた．これらの修正テンプレートは恒真あるいは恒偽の条件式を生成することで対象の箇所を実行させないようにする修正を行う．

SpotBugs による検証が成功する理由としては，恒真あるいは恒偽の条件式を生成することで一部の処理が必ず実行されなくなることが考えられる．SpotBugs はプログラムのバイトコードを読み取って解析を行うが，必ず実行されないことが明確であるコード箇所に対しては解析を行わない仕様になっている．そのため，解析が行われないコード箇所に静的解析エラーが含まれる場合，修正パッチを適用した後で SpotBugs の警告が出力されなくなる状況になってしまう．

また，テストスイートによる検証が成功する理由としては，修正パッチを適用した後で実行されなくなる箇所に対するテストが不十分であることが考えられる．開発者に受け入れられない修正パッチの生成を防ぐためには，一部の修正テンプレートの改善やデータセットのテストの洗練が必要であると考えられる．

6. 改善案

6.1 修正テンプレートの改良や追加

初期実験の結果から現状の汎用的なテンプレートでは警告を修正するのに不十分であることが考えられるため，静的解析エラーの修正率を上げるための改善案としては，修正テンプレートの内容の改良や修正テンプレートの種類を増やすことが挙げられる．

表 6 に，データセットに含まれる検出頻度の高い警告に対する修正案を示す．また各修正案に対して，将来的に対応できると考えられる jProphet の修正テンプレートについても示す．例えば「対象の条件式を equals メソッドに置換する」に関しては，条件式を Boolean 型のメソッドに置換するように改良することで実現可能になると考えられる．今後の方針としては，表 6 に示した静的解析エラーを修正可能にすることを目標にする．

6.2 修正対象の絞り込み

本研究で用いた jProphet は，SpotBugs で検出が可能な全ての警告に対して修正を行っているが，一部の静的解析エラーは汎用的な修正テンプレートで修正が出来ないことが明らかである．プログラム 4 に汎用的な修正テンプレートで修正が出来ない Java コード例を示す．プログラム 4 では 2 つの浮動小数点値の比較を行う処理が記述されている．浮動小数点の計算は丸め誤差によって違う結果になる可能性あるため，SpotBugs はこの処理に対してエラーを検出する．2 つの浮動小数点値の差を計算する内容に変更することがエラーを修正するための内容として考えられる．この修正内容は Math ライブラリを参照したり具体的な数値を利用しているため，汎用的な修正テンプレートで生成することは出来ない．修正出来ないことが明らかである警告に対して修正を行う時間は結果が得られないため無駄であるため，修正対象にするエラーをあらかじめ絞り込むことが必要だと考えられる．

6.3 確率モデルの導入

本研究では，意味的エラーを修正するための修正内容を用いて静的解析エラーを修正する機能を実現した．今後は，意味的エラーと静的解析エラーそれぞれに対応する修正テンプレートを統合することを目標としていく．これにより，静的解析エラーを修正するための内容による意味的

ⓒ 2020 Information Processing Society of Japan

表 6　データセットで検出頻度の高い警告と修正案

エラー	修正案	対応する修正テンプレート
内部表現を暴露するかもしれないメソッド	可変オブジェクトを扱う処理をオブジェクトのコピーを扱う処理に変更する	代入文やメソッドの引数の変更
default がない switch 文	対象の switch 文に default 文を追加する	無し
String オブジェクトを == や != を使用して比較している	対象の条件式を equals メソッドに置換する	メソッドの変更
デフォルトエンコーディングへの依存	エンコーディング設定を変更する処理を追加する	無し
戻り型が Boolean のメソッドが明示的に null を返している	null を返す箇所を True や False に置換する	代入文やメソッドの引数の変更
hashCode メソッドを定義していないクラス	hash メソッドを定義し，適切な hash 値を返すようにする	無し
浮動小数点同士の比較	浮動小数点同士の差を計算するように変更	無し

プログラム 4　汎用的な修正テンプレートで修正できないエラーとその修正例

```
  float a = 0.1;
  float b = 1.0;
    // エラー：浮動小数点同士の比較
- if (a * 10 == b) {
+ if (Math.abs(a * 10 - b) < 0.000001) {
      ...
  }
```

エラーの修正も可能になる．ただし，双方のエラーに対応する修正が競合する可能性があるため，一つのエラーに対して得られた複数の修正パッチのうちどれが適切かを判定する必要がある．この問題を解決するためには，確率モデルの導入が重要であると考えられる．確率モデルの導入によって開発者によって受け入れられないパッチを生成する可能性が下がると予測される．

7. 妥当性に対する脅威

7.1 内的妥当性

本研究の初期実験では SpotBugs が検出する全ての静的解析エラーに対して検証を行なっていない．修正対象にならなかったエラーの中にも修正されるものが含まれる可能性があるため，可能な限り多くの種類のエラーを含むようにデータセットを収集して調査を行う必要がある．

また，本研究では自動バグ修正手法や静的解析ツールの種類について考慮されていない．自動バグ修正技術が修正可能としているバグは手法によって異なると考えられる．また静的解析ツールの種類によっても，検出される内容は様々である．よって今後は複数の自動バグ修正手法および静的解析ツールについての調査が必要だと考えらえる．

静的解析ツールに関するもう一つの妥当性に対する脅威として，静的解析ツールの検出性能が挙げられる．静的解析ツールは誤検知を引き起こす可能性があるため，間違った結果が生じる場合がある．

7.2 外的妥当性

jProphet は，生成された修正パッチがプログラムの機能を変更しないかを確認するために，テストスイートを利用している．そのため，テストを行うプログラムの品質が結果に影響を与える可能性がある．テストを行うプログラムの品質が悪い場合，プログラムの機能を変更する修正がテストスイートによって検出されない状況が起こる可能性がある．よって，修正パッチがプログラムの機能を変更しないかを人の目によって確認することが必要になる．

また，本研究の初期実験では Defects4J のプロジェクトを対象としているが，他のプロジェクトに対しても同様の結果を得られる保証が無い．ソフトウェアの開発チームの多くはコーディング規約を決めており，プログラムコードの書き方は異なる．今後は，一般的な OSS に対しても調査を行うことが必要となる．

8. おわりに

本研究では，自動バグ修正技術の枠組みで警告の自動修正を行うツールを開発し，いくつかのプロジェクトに対して初期実験を行うことで汎用的な修正テンプレートに基づく自動バグ修正技術がどれほど警告を修正するかについて調査した．実験の結果から，本研究で提案されている自動バグ修正技術で静的解析エラーを修正するためには修正パターンが不十分である課題を解決することが必要であると分かった．今後は jProphet の機能の改良および新たなデータセットの収集をした上で，同じ枠組みでの静的解析エラーと意味的エラーの自動修正の実現を目標とする．

謝辞　本研究の一部は JSPS 科研費 JP18H04097・JP18H03222，および，JSPS・国際共同研究事業の助成を受けた．

ⓒ 2020 Information Processing Society of Japan

参考文献

[1] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineerin*, pp. 672–681, 2013.

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

[3] Diego Marcilio, Carlo Furia, Rodrigo Bonifacio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *2019 19th International Working Conference on Source Code Analysis and Manipulation*, pp. 34–44, 2019.

[4] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, Vol. 45, No. 1, pp. 34–67, 2017.

[5] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*, pp. 1–12, 2019.

[6] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, 2014.

[7] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 298–312, 2016.

[8] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792, 2009.

[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pp. 802–811, 2013.

ⓒ 2020 Information Processing Society of Japan