

自動バグ修正研究のためのプラットフォーム jProphetの開発について

首藤 巧^{a)} 山手 響介^{b)} 浅田 翔^{c)} 亀井 靖高^{d)} 鵜林 尚靖^{e)} 佐藤 亮介^{f)}

概要：ソフトウェア開発におけるバグの修正作業のコストを削減するために自動バグ修正の研究が盛んに行われている。自動バグ修正の研究では、修正精度の向上や修正時間の短縮、新しいユースケースへの適用などを目的とした様々な手法の提案が行われている。既存の自動バグ修正ツールの改変や新規の自動バグ修正ツールの開発を各手法の実験ごとに行う場合、多大な労力が必要となる。本稿では著者らが開発している自動バグ修正の研究のためのプラットフォーム jProphet について紹介する。jProphet ではテストケースによるフォールトローカリゼーションと修正テンプレートによる修正パッチの作成、及び修正精度の向上のための機械学習などの機能が用意されており、各機能を容易に変更できるような高い拡張性を備えている。また、jProphet の適用事例として、開発者による手動フォールトローカリゼーションの実証実験では自動バグ修正の実行時間が短縮され、静的解析ツールにおける警告の自動修正実験では、jProphet によって 20 件中 9 件の警告が修正された。jProphet を用いることで、これらの適用事例における手動フォールトローカリゼーション及び静的解析ツールによる検証機能を既存のコードの変更無しに実装できた。

キーワード：自動バグ修正，フォールトローカリゼーション，静的解析ツール

1. はじめに

ソフトウェア開発においてバグ修正は重要であり、多くのコストが費やされる。Gazzola らによると、デバッグ作業がソフトウェア製品の開発コスト全体の約 50 % を占めることが多いと報告されている [1]。通常バグの修正は一部ツールを利用しながらも開発者自身が手動で行っている。バグ修正の作業を自動化することによって開発者の負担を大きく軽減することが期待できるため、修正パッチ (バグのあるプログラムを修正する差分及びコード片) を自動で生成する手法である自動バグ修正の研究が盛んに行われている。

自動バグ修正ではフォールトローカリゼーション、修正パッチ生成、修正パッチの検証などの工程が存在し、それぞれ様々な技術が利用、研究されている。一つの工程の研究を行う場合も、評価のためには自動バグ修正ツール全体

の実行が必要であるため、自動バグ修正ツールの開発や既存のツールの改変と利用が必要となるが、これには多大な労力を要する。

そこで我々は自動バグ修正技術の研究における実装のためのプラットフォーム、jProphet を開発した。jProphet は Prophet[2] をベースとしたテンプレートによる修正パッチ生成技術を用いた Java 用の自動バグ修正ツールである。Prophet は機械学習と修正パッチ生成の工程が明確に分離されており、各工程の拡張、交換が容易であることから本ツールのベースとして採用した。自動バグ修正における各工程の機能において共通のインターフェースを実装させることで、PHONENIX[3]、Deepfix[4]、LEARN2FIX[5] に見られるような、フォールトローカリゼーション及び修正パッチ検証のための新手法の実装及び拡張が容易に可能なツールとなっている。本稿では jProphet の特徴を紹介するとともに、実際に jProphet を用いて行われた二つの研究の事例を紹介する。

以降、2 章では自動バグ修正技術について説明する。3 章では jProphet の機能及び実装について説明をする。4 章では jProphet の適用事例について述べる。5 章では jProphet の有効性について述べ、6 章で結論と今後の課題を述べる。

¹ 九州大学

a) shuto@posl.ait.kyushu-u.ac.jp

b) yamate@posl.ait.kyushu-u.ac.jp

c) asada@posl.ait.kyushu-u.ac.jp

d) kamei@ait.kyushu-u.ac.jp

e) ubayashi@ait.kyushu-u.ac.jp

f) rsato@g.ecc.u-tokyo.ac.jp

2. 自動バグ修正研究

2.1 概要

近年ソフトウェア工学において自動バグ修正の分野は盛んに研究が行われている。自動バグ修正の実現によってソフトウェア開発におけるデバッグ作業が大幅に減ることが期待されるため、様々な手法が研究者によって提案されている [1]。

自動バグ修正では生成と検証による修正パッチの生成が多く用いられている。これはバグのあるプログラムに対してフォールトロカライゼーション技術によりバグの箇所を推定し、その箇所に対する修正パッチの候補の生成と、その修正パッチの検証の二つのフェーズを繰り返し行うことによって修正を試みるアプローチである。

2.2 フォールトロカライゼーション

自動バグ修正手法において、スペクトラムベースドフォールトロカライゼーション (以下 SBFL) [6] と呼ばれるテストケースを用いたバグの位置の推定方法が存在する。SBFL は失敗するテストで実行される文ほどバグの原因である可能性が高いという考えに基づいて、バグのあるプログラムに付随するテストケースを利用してバグの箇所を推測する。

SBFL では各テストケースを一つずつ実行し、そのテストケースの実行経路情報を収集する。成功するテストケースが実行したプログラムの行には低い疑惑値 (バグの存在する可能性) を与え、失敗するテストケースが実行したプログラムの行には高い疑惑値を与える。こうして全てのテストケースを実行し、算出された各行の疑惑値を元にバグの箇所を推定する。

SBFL の疑惑値の計算には様々なアルゴリズムが存在し、Abreu ら [7], [8] は、Jaccard と Ochiai と呼ばれる計算式をフォールトロカライゼーションに応用し、開発者の観点から 9 種類の手法を比較して Ochiai が優れているとした。

自動バグ修正プラットフォームとしての要件 1

フォールトロカライゼーションにおける計算アルゴリズムを変更可能であること

2.3 修正パッチ生成

生成と検証による自動バグ修正手法における代表的な手法として GenProg [9] がある。GenProg は SBFL によって推定されたバグの位置に対して、遺伝的アルゴリズムに基づいてコードの変更を行う。入力として与えられたテストケースに全て通過したプログラムを修正済みのプログラムとして GenProg は出力する。

一方、GenProg のように対象プログラムごとに 0 から修正パッチを生成するのではなく、if 文の追加や変数の置換

のようなテンプレートと呼ばれる予め定義した修正操作を用いて修正パッチを生成する手法が存在する。テンプレートによる修正パッチの生成を行う自動バグ修正手法として代表的なものに ARC [10], SPR [11], Prophet [2] などがある。これらの自動バグ修正における修正パッチは開発者によって定義された修正操作であるテンプレートがベースとなるため、修正可能なバグの種類は遺伝的アルゴリズムを用いた GenProg のような手法よりも少ない。しかし、修正対象のプログラムのドメインに特化したテンプレートを用意することで修正精度や修正速度の向上が見込めるなど、研究の余地が多く残されている。

自動バグ修正プラットフォームとしての要件 2

修正テンプレートの追加・交換が可能であること

2.4 発展的な検証アプローチ

生成と検証によるアプローチにおける自動バグ修正手法ではフォールトロカライゼーションや修正パッチの検証のために開発者が用意したテストケースを用いるのが一般的である。しかし、フォールトロカライゼーション及び修正パッチの検証の工程で、テストケース以外のものを利用する自動バグ修正研究も存在する。

Bavishi らが提案した PHOENIX [3] は静的解析ツールにより検出された警告に対する修正パッチを多数の OSS から収集し、独自の DSL 言語を用いて警告に対する修正手順を学習していく手法である。PHOENIX では生成した修正パッチの検証のために静的解析ツールを再度適用し、警告が解消されるかを確認する。

Gupta らは C 言語のプログラムの構文エラーを修正する Deepfix [4] を提案した。Deepfix は深層学習技術に応用した Seq2Seq モデルを利用して修正パッチを生成する。生成された修正パッチに対して Deepfix はコンパイラによるコンパイルが成功するか否かを元に修正パッチが構文エラーを修正したか否かの検証を行う。

また、Böhme らは LEARN2FIX [5] と呼ばれる半自動バグ修正手法を提案した。LEARN2FIX は、バグレポートを元に自動生成されたテストケースの入力に人間が期待値のラベル付けを行うことで自動的にテストケースを生成するシステムであり、GenProg における自動バグ修正に利用された。

このように自動バグ修正の研究では、細かな手法の改良だけでなく、フォールトロカライゼーションや修正パッチの検証の工程の機能を刷新するような新しい手法の提案が多く行われている。これらの手法の実証を行う場合、既存の自動バグ修正ツールの改変や新規の自動バグ修正ツールの開発が必要となり、多大な労力を要することが予想される。

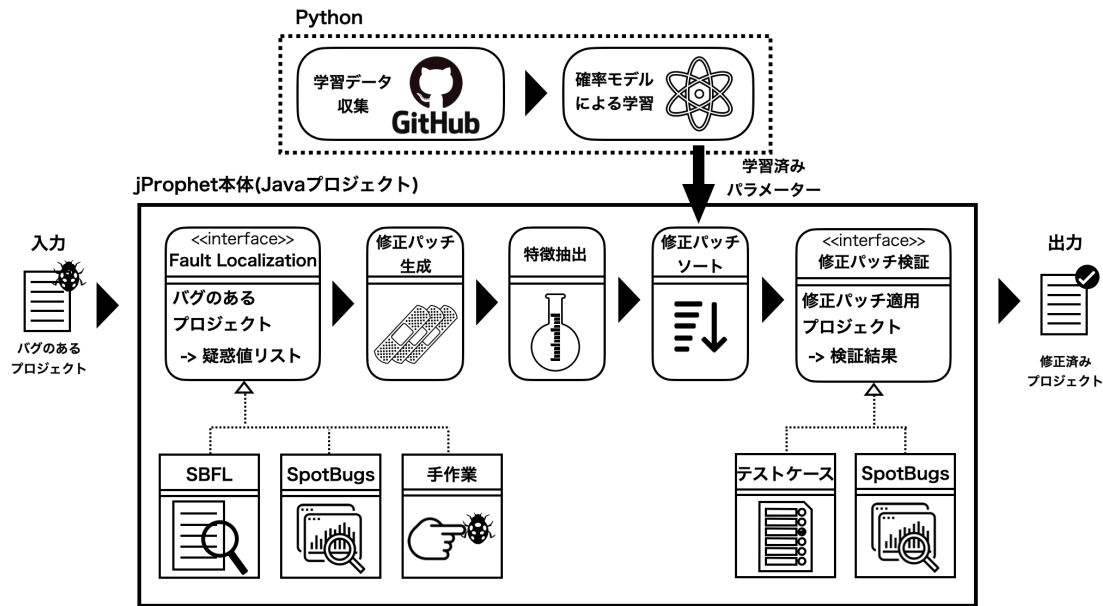


図 1 jProphet の修正工程と実装の概略図

自動バグ修正プラットフォームとしての要件 3
フォルトローカリゼーション及び修正パッチの検証
工程の実装と交換が容易であること

3. jProphet

3.1 概要

jProphet は自動バグ修正技術における各工程の処理を容易に変更、拡張できるように設計された自動バグ修正プラットフォームである。jProphet は Java 言語のプロジェクトを対象とし、テンプレートをを用いた修正パッチ生成アルゴリズムを持つ自動バグ修正を行う。

jProphet は Long らによる Prophet[2] と呼ばれる C 言語を対象とした自動バグ修正手法の修正パッチ生成及び機械学習の工程を参考にして実装されている。Prophet はテンプレートによる修正パッチの生成と、機械学習を利用する工程が完全に分離しているため、各工程の拡張及び交換が容易であることから採用した。また、Maven や Gradle といったビルドやテストのためのプロジェクト管理ツールが多くのプロジェクトに用いられていることや、言語としての利用率の高さから修正対象の言語として Java を選択した。

以降では jProphet における Prophet をベースとした各工程の機能と実装の詳細について図 1 を元に詳しく説明する。

3.2 Prophet をベースとした修正工程

3.2.1 修正パッチ生成

jProphet の修正パッチ生成におけるテンプレートは C 言語を対象とした自動バグ修正ツールである Prophet のテ

ンプレートを参考にして実装されている。テンプレートは以下に示す 6 種類の修正操作を行う。

VariableReplacement : 対象ステートメント中の変数を別の変数で置換する。変数の置換候補は同一スコープに存在するローカル変数や同一クラス内のメンバ変数、対象ステートメントが存在する関数の仮引数である。

MethodReplacement : 対象ステートメントがメソッド呼び出しの場合、メソッド呼び出しの置換を行い、置換後に VariableReplacement 操作によって実引数の置換を行う。置換候補となるメソッドは同一クラス内のメソッドである。

ConditionIntroduction : 対象のステートメントを if ブロックで囲む。

ConditionRefinement : 対象のステートメントが条件式の場合、条件式を変更する。

CopyAndReplacement : 対象のステートメントより前に出現したステートメントをコピーして対象のステートメントの直前に挿入し、挿入したステートメント中の変数に対して VariableReplacement 操作を適用する。

ControlFlowIntroduction : 対象ステートメントの直前に break や return, continue などのフロー制御文を if 文付きで挿入する。

また、ConditionIntroduction, ConditionRefinement, ControlFlowIntroduction 操作では条件式の候補も複数生成される。同一スコープのローカル変数、同一クラスのメンバ変数、同一関数の仮引数を元に以下の二種類の条件式を生成する。

- 全ての変数と null を == と != で比較
- 全ての boolean 型の変数と true を == と != で比較

また、必ず if ブロック内のステートメントが実行されるパターンを用意するために、true (恒真) の条件式も生成される。

これら全てのテンプレートの修正操作は AST ノードのクラスを受け取り、変更を加えた AST ノードのクラスを返すメソッドを持つ共通のインターフェースを実装しているため、自由に交換し組み合わせて利用することが可能である。

3.2.2 機械学習

学習データセット: jProphet では GitHub に存在する Java リポジトリからバグの修正コミットを収集し、学習用のデータセットとして利用する。GitHub のアクセストークンとリポジトリ一覧を記したテキストファイルを用意することでデータセットの収集を自動で行うスクリプトを用意した。デフォルトでは "Fix", "Fixed" などの単語が含まれたコミットメッセージを持つコミットをバグの修正コミットとして抽出する。

特徴抽出: 収集したそれぞれの修正パッチについて jProphet は特徴抽出を行い二値ベクトルへと変換する。抽出する特徴は大きく分けて二種類あり、プログラムの値の特徴及び変更の特徴である。プログラムの値の特徴はプログラムの修正前と修正後で変数や定数がどのように使用されているかをまとめたものであり、変更の特徴は修正の際に行われた変更の種類とその修正が行われた箇所の付近のステートメントの関係性をまとめたものである。jProphet はこれらの特徴を約 1000 次元の二値ベクトルに変換し学習に利用する。

学習モデル: jProphet は特徴抽出により得られた特徴ベクトルから機械学習によって学習モデルを生成する。Prophet の機械学習モデルのアルゴリズムと特徴抽出のアルゴリズムを元に実装しており、このモデルは、学習に利用した実際の開発者が行った修正と近い特徴を持つ修正を行う修正パッチに対してより高いスコアを算出する。

3.2.3 修正パッチのソート

jProphet は生成された修正パッチに対して事前に学習済みの学習モデルによって導き出されたスコアとフォールトローカリゼーションによって算出された疑惑値に基づいてソートが行われ、よりスコアの高い修正パッチから順に検証が行われる。

jProphet のテンプレートでは一つのバグに対して数千の修正パッチが生成されるため、実行時間の問題から一定数の修正パッチの検証しか行うことができない。また、修正の成否判定をテストケースで行う場合、テストケースに適合するだけの無意味な修正パッチが生成される可能性が高くなる。このソートの工程によって無意味な修正パッチを排除し、開発者が実際に行うような正しい修正パッチを生成する確率が向上すると考えられる。

3.3 拡張・交換可能な修正工程

フォールトローカリゼーション: jProphet ではバグの位置を推定するフォールトローカリゼーションの手法として、テストケースを利用した SBFL 技術、プログラムの実行を伴わずにソースコードの解析を行う静的解析ツールである SpotBugs[12] を利用してエラーを検出する手法、及び開発者が手動で直接指定したバグの推定範囲を利用する手法の三つが実装されている。全く異なるフォールトローカリゼーション手法を実装するために、バグのある対象プロジェクトの情報をまとめたクラスを受け取り、ソースコードの各行に対する疑惑値のリストを返すメソッドを持つ FaultLocalization クラスをインターフェースとして用意し、三つの手法を実装したクラスがそのインターフェースを実装する形を取っている。また、SBFL を用いる手法では Jaccard, Ochiai, Tarantula[13] の 3 つの疑惑値の計算式を選択可能であり、コンストラクタから計算式を依存クラスとして注入し動作を決定する。この実装により、トップレベルのクラスで jProphet の動作設定を一元管理している。

学習モデル: 図 1 に示すように学習モデルは jProphet 本体である Java プロジェクトからは切り離されており Python によって実装されている。学習モデルによる学習結果は学習済みパラメータをファイルとして渡すことで jProphet 本体で利用される構造となっており、他のあらゆる言語、プロジェクトで実装された学習モデルを利用することが可能である。

修正パッチの検証: jProphet では一般的な生成と検証による自動バグ修正技術と同様のテストケースを用いた検証機能、及び静的解析ツールである SpotBugs を用いた検証機能の二つが現在利用可能である。それぞれの検証のためのクラスは、修正パッチが適用されたプロジェクトを受け取り、修正結果の成否を返すメソッドを持ったインターフェースを実装しているため、容易な切り替えや新規の手法の実装が可能になっている。

4. 適用事例

4.1 開発者による手動フォールトローカリゼーション

4.1.1 概要

山手ら [14] によって行われた研究で、フォールトローカリゼーションにおいて開発者の直感による推定を行った場合の修正精度の調査を jProphet を用いて行った。

生成と検証による自動バグ修正手法において、フォールトローカリゼーションでバグの箇所が正しく推定できなければ自動バグ修正の精度の低下及び修正時間が増加する可能性が増大する。実際の開発においてバグが判明した際に開発者はある程度のバグの位置の見当が付くのではないかと、という仮説から、バグの位置の直接指定可能な機能を jProphet のフォールトローカリゼーションに組み込み、修

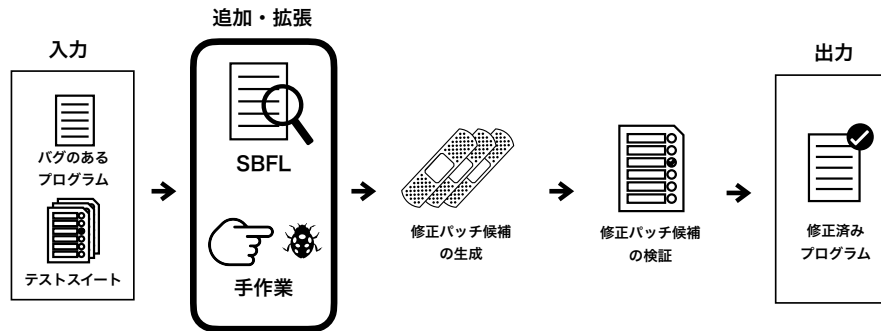


図 2 手動フォールトローカリゼーションによる実験

表 1 テストされた修正パッチ数

	P72-1			P72-2			P83		
	SBFL	one	multi	SBFL	one	multi	SBFL	one	multi
パッチ数	346/7/1	209/2/1	608/22/1	1419/6/1	416/2/1	1387/6/1	9019/0/1	582/0/1	2494/0/1

$x/y/z$: x は全てのパッチ数, y は無意味なパッチ数, z は正しい修正を行うパッチ数.

正精度の上昇及び修正時間の減少が見られるかを調査した。

4.1.2 実験

修正対象のプログラムとして, Defects4J[15] の Math プロジェクトに含まれる一部のバグを利用した。jProphet で正常に動作し, かつ, jProphet で修正可能なバグのみを実験対象にし, 複数箇所にバグがあるものは分離して実験を行った。条件を満たすバグは Math の 72 番と 80 番のバグで, 72 番はバグが二箇所に存在したため, 分離した。以降それぞれのバグを含むプログラムを P72-1, P72-2, P80 と記す。

実験の流れを図 2 に示す。この実験では jProphet におけるフォールトローカリゼーションの手法を SBFL, 手作業の 2 通りに切り替えて修正を行った。SBFL によるフォールトローカリゼーションでは, 疑惑値の算出に Jaccard と Ochiai を用いた。それぞれの手法で生成された修正パッチを疑惑値の大きさを基に並びかえ, 疑惑値の大きい順にテストを行う。最初にテストに通った修正パッチと同じ疑惑値を持つ行を修正する修正パッチ全てのテストが終了するまで実行される。実行終了までにテストされた修正パッチ数と, 無意味な修正パッチ数を分析する。

手作業によるフォールトローカリゼーションには以下の 2 通りの指定方式を用いた。

one-line: バグの箇所の疑惑値を最大の 1.00 に設定する。

multi-lines: 複数行の疑惑値を最大の 1.00 に設定する (バグの箇所を含む if 文や for 文全体などを指定)。

いずれの手法も, 疑惑値を最大にした箇所以外は 0 を付与する。なお, multi-lines は, 開発者が if 文の中のどこかにバグの原因があるのではないかと予想した場合を想定している。

4.1.3 実験結果

この実験では Jaccard と Ochiai を用いて疑惑値を算出

```

1 - if (Math.abs(yMin) <=
    functionValueAccuracy) {
2 + if (Math.abs(yMin) <=
    functionValueAccuracy && !(f != null)) {
3     setResult(min, 0);
4     return result;

```

図 3 P72-1 で生成された無意味な修正パッチ

したが, 各行に付与された疑惑値の順位に差はなかった。SBFL によるフォールトローカリゼーションで, バグの箇所の疑惑値が最大になったものは P72-1 と P72-2 であり, P83 はバグの箇所より大きい疑惑値を持つ文が 4 つあった。

修正パッチの数を表 1 に示す。表 1 の x における修正パッチ数は生成された全ての修正パッチ数ではなく, 実行終了までにテストされた修正パッチの数である。P72-1, P72-2, P83 それぞれのプログラムについて, SBFL によるフォールトローカリゼーションを行った際の結果を基準として, one-line, multi-lines の結果を評価する。

(1) **P72-1**: one-line では, テストされた修正パッチ数は約半分に, 無意味な修正パッチ数は約 3 分の 1 に減少した。一方 multi-lines では, テストされた修正パッチ数が約 2 倍に, 無意味な修正パッチ数は約 3 倍に増加した。

(2) **P72-2**: one-line では, テストされた修正パッチ数は 3 分の 1 以下に, 無意味な修正パッチ数は 3 分の 1 に減少した。一方 multi-lines では, テストされた修正パッチ数, 無意味な修正パッチ数共に大きな変化は見られなかった。

(3) **P83**: one-line では, テストされた修正パッチ数は約 20 分の 1 に減少し, 無意味なパッチ数は変化しなかった。一方 multi-lines では, テストされた修正パッチ数は約 4 分の 1 に減少し, 無意味な修正パッチ数は変化しなかった。

SBFL によるフォールトローカリゼーションで, バグの

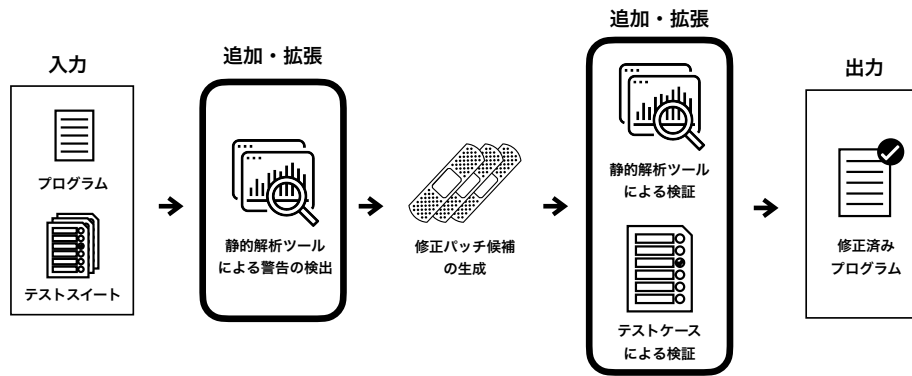


図 4 静的解析ツールによる実験

箇所の疑惑値が最大になった P72-1 と P72-2 について、自動バグ修正の性能が向上したと言えるのは one-line の場合のみで、multi-lines の場合は性能が低下したと言える。実際に P72-1 において multi-lines によって生成された無意味な修正パッチの一つを図 3 に示す。この修正パッチでは null になることのない変数 f が null の場合に実行されるようにコードが書き換えられているため、if 文の中は実行されない。P72-1 の失敗するテストは、プログラムのバグの箇所を通らなければ成功してしまうテストであった。P72-1 のバグの箇所は if 文の中にあるため、if 文の条件式が必ず偽になるような書き換えを行う修正パッチはテストに通り、無意味な修正パッチとなる。

multi-lines では if 文の条件式の疑惑値が、バグの箇所の疑惑値と同じであったため、このような無意味な修正パッチが多く生成された。if 文の条件式のようにバグの原因箇所のごく近い箇所であっても、無意味な修正パッチが生成されやすい箇所が存在すると考えられる。SBFL によるフォールトロカライゼーションで、バグの箇所の疑惑値が最大にならなかった P83 については、one-line と multi-lines の両方で自動バグ修正の性能が向上したと言える。

4.1.4 本実験を踏まえた jProphet の課題

本実験では、開発者による手作業でのバグの位置の推定結果を利用するため、外部から入力された疑惑値のリストを直接利用するフォールトロカライゼーションの実装クラスを新しく実装した。

今回の実験では、開発者によるバグの推定における中率は考慮されていないため、実際に開発者がバグの位置の推定し、自動バグ修正を実行した場合の修正率の調査も必要となる。

今後、開発者の利用を想定した実験を行う場合、実験の効率の面から GUI の必要性が高まると考える。実際の利用ケースを想定した実験の幅を増やすためにも、jProphet を直接 Eclipse や VSCode などのエディター及び IDE に組み込む必要性が高いだろう。

4.2 静的解析ツールの利用

4.2.1 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には、プログラムの実行を伴わずにプログラム内のバグになり得る箇所を静的解析エラーとして検出するものがある。静的解析ツールにより検出され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ため、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ための研究が行われている。

浅田ら [16] が行った研究では、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した。この実験では図 4 に示すように jProphet のフォールトロカライゼーション工程において、テストケースの代わりとして静的解析ツールである SpotBugs[12] を利用した。また、修正パッチの検証工程において、SpotBugs によって静的解析エラーが検出されるかどうかを基準とする検証機能を追加し、初期評価として実際に複数の OSS で適用実験を行った。

4.2.2 実験

jProphet はテストケースによって検出されるバグを修正するための修正テンプレートである Prophet のテンプレートを元に実装されている。この実験ではこのテンプレートが静的解析エラーに対してどれほど有効であるかを調査した。

また、静的解析エラーに対する修正がどれほど有効であるかを測る指標として、修正することができるエラータイプの数や、検出されたエラー全体に対する修正率に注目した。

修正対象のプロジェクトとして Defects4J[15] の 3 プロジェクトと SpotBugs の論文でデータセットとして用いられていた 2 プロジェクトを用いた。

4.2.3 実験結果

実験結果を表 2 に示す。データセットにおいて、jProphet

表 2 データセットに対する jProphet の適用結果

プロジェクト	修正対象エラー	修正されたエラー
Math	1	0
Lang	1	0
Time	1	1
db-preservation-toolkit	13	5
mssql-jdbc	4	3
合計	20	9

プログラム 1

```

1 String sub = token.substring(1);
2 if (sub.length() == 1) {
3     builder.appendLiteral(sub.charAt(0));
4 } else {
5     // エラー：効率の悪い文字列クラスの
6     // コンストラクタの呼び出し
7 -   builder.appendLiteral(new String(sub));
8 +   builder.appendLiteral(sub);
9 }

```

プログラム 2

```

1 + if(type == null) return;
2 // エラー：null 値を利用している
3 // 可能性がある
4 String sql99 = type.getSql99TypeName();
5 if (StringUtils.isNotBlank(sql99)) {
6     sql99 = sql99.toUpperCase(Locale.ENGLISH);
7 } else {
8     sql99 = "(unknown)";
9 }

```

のテンプレートによる修正操作によって修正可能と判断したエラーは 20 件であった。20 件のエラーに対して jProphet を適用したところ、修正に成功したエラーは 9 件であった。修正に成功したエラーのうち、エラータイプが異なる 2 つの修正例を、それぞれプログラム 1 とプログラム 2 に示す。

プログラム 1 では、String 型クラスのコンストラクタ呼び出しが効率の悪い処理として SpotBugs により検出された。この修正パッチの適用によってエラー内容を正しく解決するような修正が行われており、修正パッチの適用前後でプログラム挙動は変わらない。よってこの修正パッチは開発者に受け入れられる修正を行っていると考えられる。「効率の悪い文字列クラスのコンストラクタの呼び出し」に関するエラーにおいて、置換されるべき変数は他のコード箇所にも必ず記述されているため、この種類のエラーに対する修正は必ず成功すると考えられる。この処理に対して、jProphet は代入文の変更テンプレートを適用して適切な変数に置換している。

プログラム 2 では、null 値が代入されている可能性のある変数を利用している処理が SpotBugs により検出された。jProphet はこのコード箇所の直前に条件文付きのコント

ロールフロー文を挿入するテンプレートを適用しており、変数に null 値が代入されている場合はそれ以降の処理が実行されないようにしている。ControlFlowIntroduction テンプレートによる修正操作によってエラーは解消されたが、修正後のプログラムの挙動が本来想定されているものとは異なる可能性がある。一般的に、プログラム内で参照される変数が null 値であった場合の対処方法は様々であるが、現状の jProphet のテンプレートでは個別の詳細な対処を行うことはできない。よって、「null 値を利用する可能性がある」に関するエラーに対して jProphet が生成した修正パッチが必ずしも開発者に受け入れられるとは限らない。

4.2.4 本実験を踏まえた jProphet の課題

本実験では jProphet におけるテストケースに代わる修正パッチの検証の方法として、静的解析ツールである SpotBugs に対応し、静的解析エラーを jProphet のテンプレートによって修正可能かどうか検証した。静的解析エラーの修正率をより向上させるためには、静的解析エラーに特化したテンプレートが必要である。jProphet におけるテンプレートの追加をより容易にするような設計が求められると考える。

一方で、本実験の実装によって、プログラムに対して実行を伴わない解析を行うツールを jProphet に組み込むことはより容易になった。今後、同種の静的解析ツールを利用した自動修正の研究も可能である。

5. jProphet の有効性について

jProphet は主にフォールトローカライゼーション及び修正パッチの検証の工程に対して機能の追加・交換が容易になるよう設計された。

2.4 節で紹介した Deepfix[4] におけるコンパイラの構文エラーを基準とした修正パッチの検証機能は jProphet における修正パッチの検証工程として実装、交換することによって実現可能である。しかし、Deepfix の深層学習を利用した修正パッチの生成機能は jProphet 上での実装は困難である。現状の jProphet では機械学習の工程と修正パッチ生成の工程が完全に分離しているため、修正パッチ生成の途中の過程で機械学習を利用するような複雑な処理は行うことができない。

また PHOENIX[3] における静的解析ツールを用いた修正パッチの検証機能は本稿で紹介した適用事例によって同等のものを実装できた。しかし、PHOENIX における修正パッチの生成方法はテンプレートによる修正パッチ生成方法と大きく乖離しており実装におけるコストの比重が大きいため、jProphet で実装を行うメリットは少ないと考えられる。

以上より、現状の jProphet において実装可能な自動バグ修正手法は、テンプレートベースの手法であること、また、jProphet の各工程の機能を複合的に利用しない手法で

プログラム 3

```
1  int patch = 0;
2  if (patch == 0) {
3      x = a;          // 修正パッチ1
4  }
5  if (patch == 1) {
6      x = b;          // 修正パッチ2
7  }
8  if (patch == 2) {
9      x = c;          // 修正パッチ3
10 }
```

あることを満たす必要がある。

6. まとめと今後の課題

本研究では、著者らが開発している自動バグ修正の研究のためのプラットフォーム jProphet について紹介した。また、jProphet を実際に活用した研究事例として、フォールトローカリゼーションに手動のバグ推定機能を追加した実験と、テストケースに加えて静的解析ツールによるエラーの検出を修正パッチの検証手段に加えることで静的解析ツールの検出するエラーの修正を試みる実験の二つを紹介した。

今後の jProphet の課題として考えられるのが高速化である。テストケースを用いた修正パッチの検証を行う場合、修正パッチごとに、修正パッチ適用後のソースコードのコンパイル及び各テストケースの実行が必要となる。そのため、実験には多くの時間が費やされ、大量のバグに対して実験を行いデータを収集することが難しい。

ソースコードのコンパイルに要する時間を削減する方法として現在検討中であるのが、if 文による修正パッチの動的な切り替えである。プログラム 3 に示すように、修正パッチをそれぞれ if ブロックで囲み、条件式の変数の値によって分岐を行うようなプログラムとして対象プログラムに変更を加える。この例では本来 3 回のビルドが必要であるが、1 回のビルドで済ませることが可能となり、ビルドに要する時間を短縮することができる。なお、実行時の条件式の値の切り替え方法としては Java コンパイラの生成したバイナリファイルを直接編集することで条件式の変数の値を切り替える方式を実装する予定である。

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。

参考文献

- [1] Gazzola, L., Micucci, D. and Mariani, L.: Automatic software repair: A survey, *IEEE Transactions on Software Engineering*, Vol. 45, pp. 34–67 (2017).
- [2] Long, F. and Rinard, M.: Automatic patch generation by learning correct code, *ACM SIGPLAN Notices*, Vol. 51,

- No. 1, pp. 298–312 (2016).
- [3] Bavishi, R., Yoshida, H. and Prasad, M. R.: Phoenix: Automated data-driven synthesis of repairs for static analysis violations, *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).
- [4] Gupta, R., Pal, S., Kanade, A. and Shevade, S.: Deepfix: Fixing common c language errors by deep learning, *Proceedings of the 2017 Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345–1351 (2017).
- [5] Böhme, M., Geethal, C. and Pham, V.-T.: Human-In-The-Loop Automatic Program Repair, *Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, pp. 274–285 (2020).
- [6] Wong, W. E., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A survey on software fault localization, *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740 (2016).
- [7] Abreu, R., Zoetewij, P. and Van Gemund, A. J.: On the accuracy of spectrum-based fault localization, *Proceedings of the 2007 Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, IEEE, pp. 89–98 (2007).
- [8] Abreu, R., Zoetewij, P., Golsteijn, R. and Van Gemund, A. J.: A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software*, Vol. 82, No. 11, pp. 1780–1792 (2009).
- [9] Weimer, W., Nguyen, T., Le Goues, C. and Forrest, S.: Automatically finding patches using genetic programming, *Proceedings of the 31st International Conference on Software Engineering*, IEEE, pp. 364–374 (2009).
- [10] Kelk, D., Jalbert, K. and Bradbury, J. S.: Automatically repairing concurrency bugs with ARC, *Proceedings of the 2013 International conference on multicore software engineering, performance, and tools*, Springer, pp. 73–84 (2013).
- [11] Long, F. and Rinard, M.: Staged program repair with condition synthesis, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 166–178 (2015).
- [12] Hovemeyer, D. and Pugh, W.: Finding bugs is easy, *Acm sigplan notices*, Vol. 39, No. 12, pp. 92–106 (2004).
- [13] Jones, J. A., Harrold, M. J. and Stasko, J. T.: Visualization for fault localization, *Proceedings of the 2001 ICSE Workshop on Software Visualization*, Citeseer (2001).
- [14] 山手響介, 首藤巧, 浅田翔, 佐藤亮介, 亀井靖高, 鶴林尚靖: 自動バグ修正における開発者によるバグ限局の効果-Defects4J を対象にした初期評価-, ソフトウェア高額の基礎研究会 (FOSE), p. (to appear) (2020).
- [15] Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M. and de Almeida Maia, M.: Dissection of a bug dataset: Anatomy of 395 patches from Defects4J, *Proceedings of the 2018 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, pp. 130–140 (2018).
- [16] 浅田翔, 首藤巧, 山手響介, 佐藤亮介, 亀井靖高, 鶴林尚靖: 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, 研究報告ソフトウェア工学 (SE), Vol. 2020, No. 1, pp. 1–8 (2020).