修正ルールを用いた潜在バグ自動修正手法 REPIX の提案

小関万寿実　安田和矢　伊藤信治　中村知倫　原田真雄[1]

概要：本論文は，静的解析ツールが検出する潜在バグに対し，潜在バグの修正に必要な情報を設定項目として入力する修正ルールをもとに自動修正する方式を提案する．従来手法として，機械学習を用いて修正履歴から修正案を提示する手法，潜在バグごとに対象コードの探索・修正処理をプログラム実装する手法が知られている．一般にシステム開発では，設計方針の違いから，プロジェクトごとに修正内容を変更する必要がある．しかしながら，従来手法では，修正内容が学習データに依存，或いは，修正内容の変更にプログラム修正の必要があり，プロジェクトごとに修正内容を変更することは困難である．提案手法では，修正実施条件・修正操作を修正ルールとして定義することで，静的解析ツールで検出される潜在バグに対する自動修正を実現する．従って，設定項目の値を変更するだけで，修正内容の変更が可能となる．提案手法を実プロジェクトのソースコードに適用した結果，有識者選定の優先対応潜在バグが指摘された 237 箇所の内．232 箇所（97.9%）について修正可能であることを確認した．また，開発有識者判断により，潜在バグ修正内容 52 件の内，49 件（94.2%）がコミット可能であると確認した．

キーワード：プログラム自動修正，潜在バグ，静的解析，ルールベース

1. はじめに

IT 人材不足が深刻化する中，システム開発需要の増加に対応するためには，開発生産性をこれまで以上に高める必要がある．経済産業省の試算によれば，2018 年時点で約 22 万人の IT 人材が不足しており，2030 年には 45 万人程度まで人材の不足規模が拡大すると見込まれている [1]．システム開発の生産性向上に向けた動きとしては，コーディングや不具合を検出する作業の自動化が進んでいる．しかし，実装・テスト工程の 25%を占めるデバッグ作業 [2]については効率化が十分に進んでいない．デバッグ作業は開発者による手作業で行われているため，開発生産性の向上にはデバッグ作業の自動化が必要である．

デバッグ作業の自動化技術の 1 つとして，プログラム自動修正技術（Automated Program Repair: APR）が近年注目されている． 不具合には様々な種類が存在するため，APR は広範な技術を含む．例えば，コンパイル時には構文エラーを，静的解析ではコーディング規約違反や潜在バグを検出するように，開発者は実装・テスト工程の活動ごとに異なる種類の不具合を見つける．APR には，これら不具合を修正する技術がそれぞれ存在する．筆者らはこれらの不具合の内，潜在バグに着目している．

潜在バグは，プログラムの品質，セキュリティなどの低下を招く要因となるため，その解消が求められる．現状，潜在バグの修正は，静的解析の結果をもとに人手での修正を行っているが，指摘数は 1 プロジェクトあたり数百にもおよぶ．そのため，手動での修正作業工数は開発工程全体の大きな負担となる．

潜在バグに対して，修正案の提示や，自動修正を実施する既存手法としては，機械学習を用いる手法 [3] [4]と，プログラム解析用のドメイン固有言語を用いて潜在バグごとに修正内容をプログラム実装する手法 [5]がある．機械学習を用いる手法では，ソースコード履歴から修正パターンを学習し，修正案を提示する．修正内容をプログラム実装する手法では，潜在バグの種類ごとに修正適用条件と修正内容をプログラム実装し，このプログラムをもとに修正を実施する．実際のシステム開発では，設計方針の違いからプロジェクトごとに修正内容を変更する必要があるが，機械学習を用いる手法では，修正内容が学習結果依存となり，プロジェクトによっては修正が適切に実施されない場合が考えられる．修正内容をプログラム実装する手法では，修正内容に関するプログラムの再実装の必要が有り，開発作業の負担となることが考えられる．

本論文では，潜在バグの修正に必要な情報を，設定項目として入力する修正ルールを用いた自動修正手法を提案する．提案手法では，修正対象コードの探索や修正内容についての情報を，修正実施条件・修正操作の項目として修正ルールに定義し，修正ルールに対応した修正機能を呼び出すことで，潜在バグの修正を実施する．提案手法を実プロジェクトのソースコードに適用した結果，有識者選定の優先対応潜在バグが指摘された 237 箇所の内. 232 箇所（97.6%）について指摘が発生しなくなることを確認した．また，開発有識者判断により，潜在バグ修正内容 52 件の内，49 件（94.2%）がコミット可能であると確認した．修正ルールにより，各潜在バグに対する修正適用条件・修正操作の設定をユーザがプログラム実装なしで行えるため，新規プロジェクト等で潜在バグの修正内容のカスタマイズの負担軽減が期待できる．

1 (株)日立製作所
Hitachi, Ltd.

2. 背景

本章では，本論文での自動修正対象である潜在バグについて説明する．また，潜在バグ自動修正の既存手法について概説し，その課題について述べる．

2.1 潜在バグ

潜在バグとは，不具合が表面化せず，プログラム中に潜んでいるバグであり，一般的にテストで見つけるのは難しいとされている．潜在バグが引き起こす不具合の例としては，処理速度の低下や，ある条件でのプログラムの誤動作などがある．潜在バグを検出するツールとして，SpotBugs [6]や SonarQube*1 [7]が良く用いられている．SpotBugs は，Java*2 言語のプログラムを対象とした静的解析ツールであり，449 種類の潜在バグ項目を指摘可能である．これらの潜在バグ項目は，「セキュリティ」や「悪意のあるコード脆弱性」などの 10 のカテゴリーに分別される．SonarQube は，ソースコードの品質統合管理ツールであり，SpotBugs 等のツールをプラグインとして用いることができる．

2.2 既存手法と課題

既存手法としては，機械学習を用いる手法 [3] [4]と，潜在バグごとにプログラム解析用のドメイン固有言語を用いて修正内容をプログラム実装する手法 [5]がある．

機械学習を用いる手法としては，SpotBugs が検出する潜在バグに対して，ソースコード履歴が保持する潜在バグ修正前ソースコード及び修正後ソースコードから修正パターンを畳み込みニューラルネットワークを用いて学習し，パッチ生成案を提示する手法 [3]や，修正前ソースコードおよび修正後ソースコードを独自のドメイン固有言語に落とし込むことで修正手順を学習し，自動修正を実施する手法がある [4].

修正内容をプログラム実装する手法 [5]では，プログラム解析用の Rascal [8]というドメイン固有言語を用いて，潜在バグの種類ごとに，潜在バグ探索条件と潜在バグ修正内容をプログラム実装し，このプログラムをもとに修正を実施する．

実際のシステム開発では，コーディング規約等の運用方針の違いから，同じ種類の潜在バグでもプロジェクトごとに修正方法が異なる，つまり修正内容のカスタマイズが必要な場合がある．しかし，既存手法では修正内容のカスタマイズが容易に実施できないと考えられる．機械学習を用いる手法では，修正内容が学習結果依存となり，プロジェクトごとに再学習が必要になる．また，修正内容をプログラム実装する手法では，プロジェクトごとにプログラムの再実装が必要となり，開発作業の負担になる．

*1 SonarQube は SonarSource SA.の商標登録.
*2 Java は Oracle America, Inc.の登録商標.

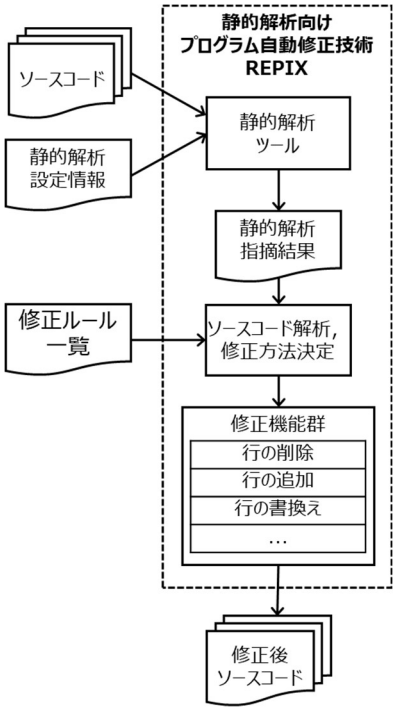

図 1　提案手法 REPIX の全体図

3. 提案手法

本章では，修正実施条件・修正操作の組を修正ルールとして定義し，潜在バグを自動修正する手法 REPIX*3 を提案する．REPIX は，修正ルールに対応した修正機能を呼び出すことで，潜在バグの修正を実施する．修正ルールは各潜在バグの種類に対して定義される．修正ルールに定める修正実施条件により潜在バグの位置および潜在バグの種類を正確に検出し，修正実施条件と組になっている修正操作を実施することで，潜在バグの種類と位置に応じた修正を行う．また，修正実施条件・修正操作の内容を汎用的な設定項目とすることで，ユーザによる修正内容の決定作業を効率化する．

3.1 提案手法の全体図

提案手法の全体図を図 1 に示す．REPIX の入力には，修正対象のソースコードのほかに，静的解析設定情報，修正ルール一覧を用いる．ここで，静的解析設定情報は，静的解析ツールの設定情報であり，例えば SpotBugs では，検出する潜在バグなどを指定するためのフィルタファイルが該当する．修正ルール一覧には，修正ルールとして「修正実施条件」と「修正操作」の組を，静的解析ツールが指摘する潜在バグ項目ごとに定義する．詳しくは，3.2 節にて述べる．また，静的解析設定情報には，静的解析ツールで検出可能な各潜在バグ項目について，検出するか否かの

*3 REPeat fIX

表 1 提案手法 REPIX の修正ルール設定項目

#	大項目	小項目	必須	説明
1	修正実施条件	静的解析ツール名 (analyzer)	○	SpotBugs 等，利用する静的解析ツール名を設定する．
2		潜在バグ項目名 (check)	○	修正対象となる潜在バグの種別を指定する． 指定項目は，静的解析ツールによる指摘項目名を設定する．
3		静的解析設定条件 (sourceCodeProperty)		静的解析ツールの各指摘項目のプロパティの情報を利用する場合に，静的解析ツールの設定情報を指定する．
4		ソースコード条件 (sourceCodeProperty)		静的解析ツールが検出した潜在バクの箇所を特定する情報を特定するための検索条件を指定する．例えば，潜在バグ箇所を特定する潜在バグ対象文字列(searchString)を設定する．潜在バグ対象文字列の指定には， 表２の表記を用いる．
5	修正操作	修正操作名 (action)	○	コードの挿入，削除，置換等，適用する修正操作名．
6		修正引数 (args)	○	修正操作実行に必要なパラメータ情報を指定する． パラメータ情報には，修正位置(base)，修正前のソースコード情報(oldStr)，修正後ソースコード情報(newStr)等，置換，削除，追加のために必要な情報を設定する． 例えば，文字列置換操作のパラメータには，修正位置，修正前のソースコード情報，修正後のソースコードの情報を設定する．

- ()内は，図２のサンプルでの設定項目との対応を表す．

情報が含まれ，これに加えて，潜在バグ項目を検出すると選択する場合は検出条件が含まれる場合がある．

REPIX では，潜在バグ修正の流れ（潜在バグの特定→潜在バグ内容に適した修正操作の決定・実施）を，修正ルールを用いることによって自動化する．潜在バグ箇所の特定では，修正ルール内の修正実施条件を用いて，静的解析指摘結果と静的解析設定情報をもとに，対象のソースコードを解析し，潜在バグの位置および潜在バグの種類を特定する．潜在バグ内容に適した修正操作の決定・実施では，潜在バグの特定で用いた修正実施条件と組になっている修正操作を選択し実行する．

潜在バグの修正では，修正箇所を特定し，特定した箇所に対し，該当コードの挿入・削除・置換等の修正作業を実施する．修正箇所を特定するために，REPIX では修正ルールに修正箇所の設定項目を設けており，静的解析ツールによる指摘箇所を基準とした相対位置を設定する．例えば，switch-case 文の break 文の挿入漏れに対しては，修正箇所(break 文の挿入箇所)として，該当 case 文ブロックの最終行を修正操作に設定する．また，REPIX では，修正ルール内に，特定した修正箇所に対する修正内容の設定項目を設けている．上記の switch-case 文であれば，「固定文(break 文)を挿入する」という修正内容を，修正作業の設定値とする．

REPIX では，修正箇所の特定および修正内容に関わる情報を，修正実施条件および修正操作として修正ルールに定義し自動修正を実現する．本章では，3.2 節で修正ルールの概要を，3.3 節で自動修正処理の流れを説明する．

3.2 修正ルール

図 1 内に示す修正ルール一覧には，修正ルールとして「修正実施条件」と「修正操作」の組を潜在バグ項目の種類ごとに定義する．「修正実施条件」と「修正操作」は，それぞれ設定項目を保持する．各設定項目の詳細を表 1 に示す．

3.2.1 修正実施条件

修正実施条件は，静的解析ツールで検出される各潜在バグの情報を利用し，各潜在バグに対する修正操作を一意に決定するため設定する．静的解析ツールでは，該当する潜在バグの種類を出力するが，潜在バグのソースコード内の位置情報として出力するのは行番号のみであり，自動修正を実施する際は該当行のステートメント内のどの部分が潜在バグの原因となっているかを探索する必要がある．潜在バグの原因箇所の探索のため，「修正実施条件」では，静的解析ツールに関する情報である，静的解析ツール名（表 1 の#1 に対応），潜在バグ項目名（表 1 の#2 に対応）に加えて，静的解析設定条件（表 1 の#3 に対応），ソースコード条件（表 1 の#4 に対応）を設定項目として含む．

静的解析設定条件には，静的解析設定情報内に設定する各潜在バグ項目の指摘発生条件を指定する．ソースコード条件には，静的解析ツールだけでは判断できない，より詳細な潜在バグ内容を判断するため，ソースコードの状態に関する情報を指定する．

ソースコード条件には，静的解析ツールの出力を基に具

表 2 任意の変数名・型名・引数の表現形式

#	対象	修正ルールでの表現	変換後の正規表現
1	変数名	$var(number)$	?<var1>(¥w+((¥[.*¥])¥.¥w+)*)
2	型名	$type(number)$	?<type1>(¥w+)
3	引数	$args(number)$	?<args1>(¥s*,¥s*(¥g<args1>))?

```
"conditions" : {
                "analyzer" : "spotbugs",
                "check" : "EI_EXPOSE_REP"
                "souceCodeProperty"  : { "searchString" : "return $var1$" }
              },
"actions" : [
             {
               "action" : "ReplaceStr",
               "args" : {
                          "base" : "buggyStatement",
                          "oldStr" : "$searchStr$",
                          "newStr" : "return ($type_var1$)$var1$.clone()"}
                        }
            ]
```

図 2 修正ルールの例

体的な潜在バグ箇所を特定するための情報を設定する．設定する項目として，例えば，潜在バグ起因要素と潜在バグ対象文字列がある．

潜在バグ起因要素には，該当の潜在バグがどの要素（フィールド変数，ローカル変数，メソッド呼出し）に起因するかを指定する．同じ潜在バグ項目でも，潜在バグ起因要素が異なると，実施する修正操作の内容が異なる場合があり，これに対応するために潜在バグ起因要素を修正実施条件の項目として設定する．潜在バグ箇所がどの要素に起因するかという情報は，静的解析ツールによる潜在バグ指摘結果に含まれる．

潜在バグ対象文字列は，潜在バグ箇所に該当する文字列を指定する．潜在バグ対象文字列に指定された文字列が指摘行にある場合，その文字列の部分が潜在バグ箇所であると判断する．潜在バグ対象文字列には，任意の変数名・型名・引数についての表現を含めることが可能である．任意の変数名・型名・引数についての表現を，表 2 に示す．潜在バグ対象文字列は，ソースコード解析部分にて正規表現に変換され，この正規表現を用いてソースコードとのマッチングを行う．

3.2.2 修正操作

修正操作は，組になっている修正実施条件が満たされる場合に，ソースコードに対して適用される修正内容を設定する．設定内容には修正操作名（表 1 の#5 に対応）と修正操作引数（表 1 の#6 に対応）が含まれる．

修正操作名には，潜在バグに対して実施する修正操作の名称を記載する．潜在バグに対する修正では，該当箇所の削除，不足コードの追加や型情報の補間，別メソッドへの置き換え(文字列の置換)等を行う必要がある．REPIX では，これらの修正操作に対応する修正処理を事前実装しておき，修正操作名には対応する修正処理の名称を設定する．また，修正操作引数には，修正操作実施位置や，修正使用文字列を設定する．

修正操作実施位置には，静的解析指摘結果に含まれる潜在バグ指摘位置を基準とした相対位置を設定する．修正操作実施位置を指定するのは，修正操作を実施する行が，静的解析ツールによる潜在バグ指摘行であるとは限らないためである．例えば，SpotBugs による潜在バグの指摘の内「1 つの case が次の case へと通り抜ける switch 文を見つけた」という指摘では，case 文内に break 文が無いことを潜在バグとして指摘するが，指摘位置は，”case”が記載されている行であり，仮に指摘行に break 文を挿入してしまうと，プログラムは意図した動作を実行しなくなる．この潜在バグの場合は，指摘行のブロックステートメント（つまり case 文のブロック）の最終行を修正位置として設定する．

修正使用文字列には，文字列の挿入・削除・置換などの修正操作で対象とする文字列を設定する．例えば修正操作が「文字列の置換」であれば，置き換え前の文字列と置き換え後の文字列を指定する．文字列の指定には，表 2 の任意の変数名・引数・型名についての表現を含めることが可能である．

3.2.3 修正ルール具体例

修正ルールの具体例を，図 2 を用いて説明する．修正実施条件は，図 2 の“conditions”ブロックが示す．“conditions”ブロック内の “analyzer” では，静的解析ツール SpotBugs を設定している．また “conditions” ブロック内の “check” では，潜在バグ項目名として，SpotBugs が指摘する潜在バグ項目「EI_EXPOSE_REP」（可変オブジェクトへの参照を返すことによって内部表現を暴露するかもしれないメソッド）を設定している．“conditions” ブロック内の “sourcecodeProperty” ではソースコード条件を設定しており，潜在バグ対象文字列“searchString”として“return $var1$”を設定している．ここで，“$var1$” は表 2 で示す任意の変数名の表現である．静的解析設定条件に関しては，潜在バグ項目「EI_EXPOSE_REP」の検出にあたり SpotBugs に対して潜在バグ項目の指摘発生条件が特に存在しないため，省略している．

修正操作は，図 2 の“actions”ブロックが示す．“actions”ブロック内の “action” では，修正操作として文字列置換操作“replaceStr”を設定している．“actions”ブロック内の“args”ブロック内では，文字列置換機能に必要な情報である修正操作位置，置換前文字列，置換後文字列の情報をそれぞれ

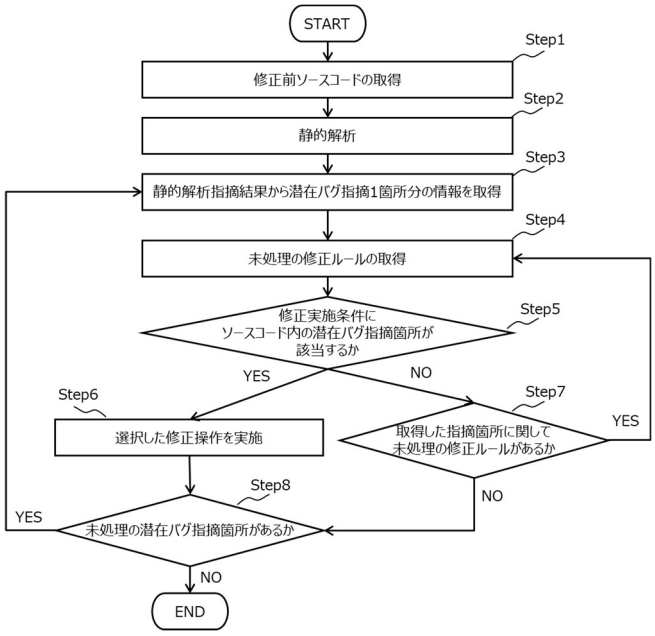

図 3　提案手法 REPIX による潜在バグ修正の
フローチャート

3 つの引数“base”，“oldStr”，“newStr”に設定している．修正操作位置を示す引数“base”は修正対象行を指定する引数である．図 2 では“base”に“buggyStatement”を設定しているが，これは「静的解析ツールで指摘された行」を表す．また，置換前文字列を示す引数“oldStr”には“$searchString$”を設定しているが，これは修正実施条件でソースコード条件として設定する潜在バグ対象文字列を意味する．つまり，置換前文字列は，“return $var1$”である．また，置換後文字列を示す引数“newStr”には，“return ($type_var1$)$var1$.clone()”を設定している．“$type_var1$”は“$var1$”が指す任意の変数の型名を意味している．

3.3 潜在バグ修正の流れ

提案手法 REPIX での潜在バグ修正の流れを表したフローチャートを図 3 に示す．REPIX では，潜在バグ箇所を特定するために，静的解析指摘結果をもとに，修正ルール内の修正実施条件を用いて対象のソースコードを解析し，潜在バグの位置および潜在バグの種類を特定する．そして，潜在バグの特定で用いた修正実施条件と組になっている修正操作を選択し実行することで，潜在バグの種類に合った修正操作を実施する．

REPIX による潜在バグ修正の流れを，具体例を用いて説明する．修正ルール(json)の例を図 2 に，潜在バグが検出されるソースコードとその修正例を図 4 に示す．図 4 で示す潜在バグは,「可変オブジェクトへの参照を返すことによって内部表現を暴露するかもしれないメソッド」として SpotBugs にて検出されるバグで，“EI_EXPOSE_REP”という潜在バグ名で検出される．以降では，図 3 で示す潜在バグ修正の流れを，図 2，図 4 に示した例と共に概説する．

修正前ソースコード例

```
1     public class ComponentDefinitionsBeen{
2         private Date _dataUpdateDatime;
3
4         public Date getDataUpdateDatime() {
5             return _dataUpdateDatime;
6         }
7     }
```

修正後ソースコード例

```
1     public class ComponentDefinitionsBeen{
2         private Date _dataUpdateDatime;
3
4         public Date getDataUpdateDatime() {
5             return (Date)_dataUpdateDatime.clone();
6         }
7     }
```

図 4　潜在バグの修正例

潜在バグ修正に向けて，REPIX では，修正前のソースコードを取得し(Step1)，取得したソースコードに対し静的解析をかける(Step2)．その指摘結果から潜在バグ 1 箇所分の情報を取得する(Step3). そして, 修正ルール一覧の中から，取得した潜在バグ箇所に関して精査していない未処理の修正ルールを取得する(Step4).

次に，Step4 で取得した修正ルールが保持する修正実施条件に，潜在バグ箇所が該当するかを判定する(Step5)．例えば，Step4 で図 2 で示す修正ルールが取得されたとすると，Step5 では，静的解析ツール名の条件“analyzer”=“spotbugs”，指摘項目名の条件“check”=“EI_EXPOSE_REP”，ソースコード条件“sourceCodeProperty”内の小項目“searchString”=“return $var1$”を満たすかを判定する．

ここで，“sourceCodeProperty”について説明する．図 2 のソースコード条件“sourceCodeProperty”に属する検索文字列“searchString”は，“return $var1$”を指定している．これを正規表現に変換すると，表 2 より，“?<var1>(¥w+((¥[.*¥])|¥.¥w+)*) ?<var1>(¥w+((¥[.*¥])|¥.¥w+)*)”となる．この正規表現に当てはまる文字列を図 4 の修正前ソースコードの指摘行内のステートメントを対象に探索する．その結果，“return _dataUpdateDatime”が該当する．このとき，“var1”=“_dataUpdateDatime”と取得できる．

続いて，Step4 で取得した修正ルールが保持する修正操作を実施する(Step6)．例えば，Step4 で図 2 の修正ルールが取得されたとすると，Step6 では，修正操作名“action”=“ReplaceStr”が示す修正操作である「文字列置換操作」が，修正操作引数“args”の情報をもとに実施される．

ここで，修正ルール図 2 の修正操作引数に基づいた文字列置換操作について，図 4 の修正前ソースコードを例に説明する．図 2 内の修正操作引数“args”として，修正位置を表す引数“base”，置換前文字列を示す引数“oldStr”，置換後文字列を示す引数“newStr”が定義されている．図 2 では，修正位置を表す引数“base”に“buggyStatement”

が設定されている．この値から，図 4 の修正前ソースコードの場合，修正位置は，指摘位置である 5 行目であると求められる．また，置換前文字列を示す引数“oldStr”に“searchStr”が設定されている．この値は，置換前の文字列が，修正ルール “condition” -“sourceCodeProperty”-“searchString”に該当する文字列であることを示す．図 4 の修正前ソースコードの場合，置換前文字列は，“return _dataUpdateDatime”と求められる．一方，置換後文字列を示す引数“newStr”には，“return ($type_var1$)$var1$.clone()”が設定されている．ここで “$type_var1$” は「変数 var1 の型名」を表す文字列を，ソースコードから解析して挿入することを示す．ここで，図 4 の修正前ソースコードについて，Step4 にて，“var1” = “_dataUpdateDatime”が取得されているため，“newStr”が示す文字列の“$type_var1$”には，変数_dataUpdateDatime の型名である“Date”が入る．また，var1 に該当する変数名は，“_dataUpdateDatime”であるため，置換後文字列は “newStr” = “return (Date)_dataUpdateDatime.clone()”となる．

図 3 のフローチャートの Step5 にて，修正実施条件に潜在バグ箇所が該当しないと判定された場合は，修正ルール一覧の中で，取得した潜在バグ箇所に関して精査していない未処理の修正ルールがあるか判定する(Step7)．未処理の修正ルールがある場合は，再度 Step4 を実施する．未処理の修正ルールが無い場合は，静的解析指摘結果が示す潜在バグ箇所の中で，未処理の潜在バグ指摘箇所があるか判定する(Step8)．全未処理の潜在バグ指摘箇所がある場合は Step3 に戻り，静的解析指摘結果から潜在バグ箇所の情報を取得する．未処理の潜在バグ指摘箇所がない場合は，修正処理を終了する．

4. 評価

4.1 評価観点

本評価では，2 つの観点から提案手法を評価する．

- 修正網羅性評価
  静的解析ツールが指摘する潜在バグの内，修正対象とする潜在バグが何件修正されたかを評価する．本評価では，評価題材で指摘された 237 件の修正対象潜在バグの自動修正率を評価する．
- 修正内容評価
  自動修正による修正結果の妥当性を評価する．具体的には，開発有識者によるレビューを行うことで，修正結果がコミット可能かを判定する．評価実験では，修正結果のソースコード記述が重複しない 52 件を抽出し，開発有識者により，修正内容を判定する．

なお，対象とする潜在バグ，及び，評価対象については次節にて説明する．

4.2 評価対象

4.2.1 評価題材

本評価の評価題材として，企業で開発された実システムのソースコードを用いた．題材システムは Java 言語で実装されており，プロダクトコードは 1045 クラス，約 111,000 LoC (Lines of Code) からなる．

4.2.2 評価対象の潜在バグ

提案方式の評価のため，潜在バグを検出する静的解析ツールとして SpotBugs を用いる．SpotBugs では，10 カテゴリ，449 種類の潜在バグ項目が存在するが，本評価ではプロジェクトでの実用性を考慮し，運用方針から重要度の高い項目に対し，次の観点に基づき絞り込みを行った．

(1) 最終的な納品物の品質確保の上で守るべきもの
(2) 後工程で見つかった場合，工数を要するもの

上記評価観点のもと，開発有識者により，潜在バグ項目 31 項目を選定した．この内，プログラムだけでなく仕様の変更が必要な項目などを除いた 25 項目を本評価の修正対象とした．

4.3 結果

4.3.1 修正網羅性評価

修正網羅性評価について，結果を表 3 に示す．評価題材に SpotBugs を適用し，本評価の修正対象の潜在バグ項目の指摘が発生した 237 件(潜在バグ項目 13 項目）の内．232 件（97.9%）について，提案手法での修正適用により指摘があった潜在バグを修正可能であることを確認した．

4.3.2 修正内容評価

4.3.1 節で述べた，提案手法で修正可能な 232 件の内，指摘対象のソースコード中の変数名やステートメントが重複する箇所等を除いた 52 件（潜在バグ項目 13 項目）に対して修正内容がコミット可能かの判定を行った．結果，52 件のうち，49 件(94.2%)がコミット可能であることを確認した．修正内容評価について，開発有識者がコミット可能と判定した潜在バグ修正件数は，評価対象の潜在バグ修正内容 52 件の内，49 件（94.2%）であることを確認した．

5. 考察

5.1 修正網羅性

4.3.1 節の修正網羅性評価にて，修正未実施のケースが 5 件あった．これは，提案手法のプロトタイプの仕様が，修正によりコンパイルエラーが起こる箇所に対し自動で修正を取り消す仕様となっていることが理由と考えられる．修正によりコンパイルエラーが発生するのは，try-catch 文において，「catch 文が指定する例外が発生する可能性のある

表 3　修正網羅性評価の結果

#	修正対象 潜在バグ項目	潜在バグ項目　説明	潜在バグ数	修正完了潜在バグ数	修正未実施潜在バグ数
1	EI_EXPOSE_REP2	可変オブジェクトへの参照を取り込むことによって内部表現を暴露するかもしれないメソッド	83	83	0
2	DLS_DEAD_LOCAL_STORE	ローカル変数への無効な代入	46	44	2
3	EI_EXPOSE_REP	可変オブジェクトへの参照を返すことによって内部表現を暴露するかもしれないメソッド	46	46	0
4	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	null ではないことがわかっている値の冗長な null チェック	32	32	0
5	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	インスタンスメソッドから static フィールドへの書き込み	8	6	2
6	MS_SHOULD_BE_FINAL	final にすべきフィールド	6	6	0
7	BC_UNCONFIRMED_CAST	未チェック/未確認のキャスト	5	5	0
8	PS_PUBLIC_SEMAPHORES	公開インタフェースで同期化とセマフォを暴露するクラス	3	3	0
9	ICAST_INTEGER_MULTIPLY_CAST_TO_LONG	整数乗算の結果を long にキャストしている	2	2	0
10	NP_DEREFERENCE_OF_READLINE_VALUE	readLine メソッドの結果が null なのか確かめないで値を利用している	2	2	0
11	RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE	null とわかっている値の冗長な null チェック	2	1	1
12	NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE	null になっている可能性があるメソッドの戻り値を利用している	1	1	0
13	SF_SWITCH_FALLTHROUGH	1 つの case が次の case へと通り抜ける switch 文を見つけた	1	1	0
計			237	232	5

try 文内のステートメント」を削除してしまう場合であった．削除の操作は，潜在バグに対する修正として，不要な null チェックの if 文や不要な値の代入文に対して実施していた．上記のように修正実施内容とソースコードの設計意図が矛盾するケースを提案手法で解決するためには，修正ルールの項目追加や修正機能の拡充が必要である．ただし，ユーザによる修正ルール設定を複雑化させないような項目や修正機能の選定が課題となる．

5.2 修正内容

4.3.2 節の修正内容評価にて，修正内容のコミット不可判断のケースが 3 件あった．この内 1 件は，表 3 の#13 が示す「1 つの case が次の case へと通り抜ける switch 文を見つけた」という潜在バグに対する修正であった．この潜在バグは，case 文に break 文がないことを示し，修正内容として，「case 文内の適切な箇所への break 文の挿入」を修正ルールに設定していた．しかし，開発有識者がコミット不可と判断したソースコードでは，開発者が意図的に挿入していない break 文を挿入する修正を行っていた．

また，他の 2 件は，表 3 の#2 が示す「ローカル変数への無効な代入」という潜在バグに対する修正であった．この潜在バグは不要な値の代入を示す．修正内容として，指摘箇所が変数宣言と値の代入を兼ねている場合は，変数宣言だけを残して値の代入は削除する．開発有識者からコミット不可判定されたソースコードでは，修正により残った変数宣言の位置が不適切であると判断された．

コミット不可判定されたケースを提案手法で解決するためには，5.1 節で述べた修正実施内容とソースコードの設計意図が矛盾するケースへの対策と同様に，修正ルールの項目追加や修正機能の拡充が必要である．ただし，ユーザによる修正ルール設定を複雑化させないような項目や修正機能の選定が課題となる．

6. 既存手法との比較

6.1 開発者による修正内容の受入

REPIX では，4.3.2 節の修正内容評価より，開発有識者がコミット可能と判定した潜在バグ修正件数の割合は 94.2%（潜在バグ修正内容 52 件中，49 件）であった．対して既存手法に関して，機械学習を用いて潜在バグの自動修正を実施する手法について，文献 [4]では 20.2%（94 件中 19 件）の修正が開発者によって受け入れられたと述べられている．また，修正内容を事前に実装する手法について，文献 [5]では，84% （920 件中 775 件）の修正が開発者によって受け入れられたと述べられている．

上記の実験結果は，機械学習を用いた手法が学習データに依存するのに対し，提案手法 REPIX，及び，事前実装する既存手法は，潜在バグごとに修正内容の設定が可能であり，プロジェクト方針に沿った修正が実施されるためと考える．

6.2 修正内容のカスタマイズ性

実際のシステム開発において，同じ種類の潜在バグでもプロジェクトごとに修正方法が異なる，つまり修正内容のカスタマイズが必要な場合がある．しかし，2.2 節でも述べ

たように，既存手法では修正内容のカスタマイズが容易に実施できないと考えられる．機械学習を用いる手法 [3] [4] では，修正内容が学習結果依存となり，プロジェクトごとに再学習が必要になる．また，潜在バグごとに修正内容をプログラム実装する手法 [5]では，プロジェクトごとに修正適用条件と修正操作のカスタマイズ，つまりプログラムの再実装が必要となり，開発作業の負担となる．

提案手法では，各潜在バグ種類に対する修正適用条件・修正操作を項目化しているため，ユーザはプロジェクトの方針に沿って，項目に対応する値を設定するだけで良く，プログラム実装を実施することなく修正ルールを設定することが可能である．そのため，既存手法と比較して，提案手法ではプロジェクトごとでの潜在バグ修正内容のカスタマイズの効率化が期待できる．ただし，提案手法では修正ルールとして修正実施条件および修正操作の設定項目を設けているが，潜在バグの修正内容によっては本論文で述べた設定項目では不十分な場合も考えられるため，その際は設定項目の種類を増やすなどして修正ルールを改善する必要が有る．

7. おわりに

本論文では，静的解析ツールが検出する潜在バグをルールベースで自動修正する方式を提案した．提案手法では潜在バグに対する修正実施条件と修正操作の設定項目を設け，修正ルールとして定義し，この修正ルールと静的解析設定情報をもとに修正対象ソースコードを解析し，潜在バグの修正を実施する．提案手法を実プロジェクトのソースコードに適用した結果，有識者選定の優先対応潜在バグが指摘された 237 箇所の内．232 箇所（97.9%）について修正可能であることを確認した．また，開発有識者判定により，潜在バグ修正内容 52 件の内，49 件（94.2%）がコミット可能であることを確認した．提案手法により，各潜在バグ種類に対する修正適用条件・修正操作を項目ごとに設定することが可能となるため，プロジェクトごとのカスタマイズの効率化が期待できる．今後の課題として，静的解析による潜在バグ指摘箇所に対する修正内容と，プログラムの設計意図との矛盾を解消することが挙げられる．課題解決に向けて，潜在バグの修正とプログラムの設計意図が矛盾する場合のパターンを列挙し，修正ルールに設定する修正実施条件の項目を拡充することが必要になると考える．

参考文献

[1] 経済産業省，“IT 人材受給に対する調査,” 2019.

[2] T. Britton, J. Lisa, C. Graham and C. Paul, "Reversible Debugging Software," University of Cambridge, 2013.

[3] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo and Y. Le Traon, "Mining Fix Patterns for FindBugs Violations," *IEEE Transactions on Software Engineering,* vol. 1, no. 1, 2018.

[4] R. Bavishi, H. Yoshida and M. R. Prasad, "Phoenix: Automated Data-Driven Synthesis of Repairs for Static Analysis Violations," *In proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* pp. 613-624, 2019.

[5] D. Marcilio, C. A. Furia, R. Bonifacio and G. Pinto, "Automatically generating fix suggestions in response to static code analysis warnings," *In proceedings of 19th IEEE International Working Conference on Source Code Analysis and Manipulation,* pp. 34-44, 2019.

[6] "SpotBugs," [Online]. Available: https://spotbugs.github.io/.

[7] "SonarQube," [Online]. Available: https://www.sonarqube.org/.

[8] P. Klint, T. V. D. Storm , J. Vinju, “Rascal: A domain specific language for source code analysis and manipulation,” *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE,* pp. 168-177, 2009.