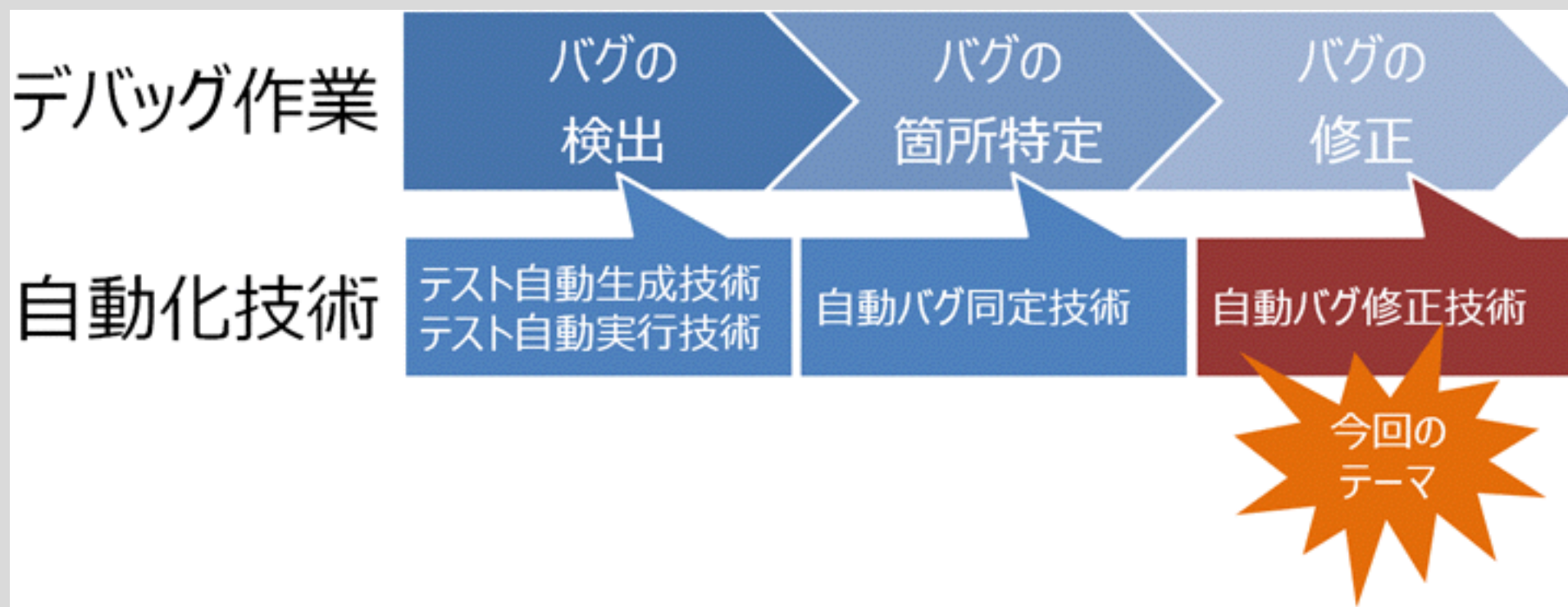


Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim Jaechang Nam
Jaewoo Song Sunghun Kim

デバッグ



自動バグ修正のアプローチ

- 変異を用いた手法
- 意味解析に基づいた手法
- 機械学習を用いた手法

変異を用いた手法

特定されたバグの箇所を変異させ、全てのテストが成功するまで繰り返す手法

代表的なツール

- GenProg (遺伝的アルゴリズムを用いる)
- RSRepair (ランダム探索を用いる)

意味解析に基づいた手法

特定されたバグの箇所が満たすべき制約を導き出し、その制約を満たすプログラムを作り出す手法

比較的人間が理解しやすい修正になる

代表的なツール

- SemFix
- DirectFix

機械学習を用いた手法

大量のバグ修正パッチから学習した規則に従ってプログラムを修正する手法

全く新しい修正方法で一から修正できるわけではない

比較的人間が理解しやすい修正になる

代表的なツール

- Prophet

今回の論文のアプローチ

論文タイトル「Automatic Patch Generation Learned from Human-Written Patches」



- 意味解析に基づいた手法

既存手法の問題

変異を用いた手法は、人間の直感では理解できないような修正を出力することがある

例(修正前)

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920      lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when `getShort(iCode, pc + 1)` is equal to or larger than `strings.length` or smaller than 0.

例(実際に行われた修正)

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+    i = getShort(iCode, pc + 1);
1921+    if (i != -1)
1922+        lhs = strings[i];
1923  }
1924  Scriptable calleeScope = scope;
```

(c) Human-written patch.

例 (GenProgの修正)

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+    lhs = ((Scriptable)lhs).getDefaultValue(null);
1921  }
1922  Scriptable calleeScope = scope;
```

(b) Patch generated by GenProg.

例(提案手法の修正)

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+    if (getShort(iCode, pc + 1) < strings.length &&
        getShort(iCode, pc + 1) >= 0)
1921+    {
1922        lhs = strings[getShort(iCode, pc + 1)];
1923+    }
1924  }
1925  Scriptable calleeScope = scope;
```

(d) Patch generated by PAR.

Fig. 1: Patches created by GenProg, a human developer, and PAR for Mozilla Bug #114493.

※PAR : Pattern-based Automatic program Repair

提案手法の手順

1. 実際に人間が作成した修正パッチから、よくある修正パターンを見つける
2. 各パターンの修正手順を修正テンプレートとしてまとめる
3. 特定されたバグが修正テンプレートの適用可能条件に合致する場合に、その手順で修正を試みる

1. よくある修正パターンを見つける

1. オープンソースプロジェクトから修正パッチを取得（今回はEclipse JDTから62,656個）
2. バグの根本原因と、それに対して修正パッチが行った操作（追加、削除、変更）をもとに分類

よくある修正パターン

TABLE I: Common fix patterns identified from Eclipse JDT's patches.

Fix Patterns
Altering method parameters
Calling another method with the same parameters
Calling another overloaded method with one more parameter
Changing a branch condition
Adding a null checker
Initializing an object
Adding an array bound checker
Adding a class-cast checker

約30%のバグがいずれかのパターンに当てはまった

例 (Adding Null Checker)

```
3561    public ITextHover getCurrentTextHover() {  
3562+        if (fTextHoverManager== null)  
3563+            return null;  
3564        return fTextHoverManager.getCurrentTextHover();  
3565    }
```

Fig. 2: Example of “Null Checker”, a bug-fix pattern for Null Pointer Exception bugs. This patch fixes a bug of **TextViewer.java** described in Eclipse JDT Bug #26028. It inserts an `if` statement to avoid calling `getCurrentTextHover()` when `fTextHoverManager` is **null**.

提案手法の手順(再掲)

1. 実際に人間が作成した修正パッチから、よくある修正パターンを見つける
2. 各パターンの修正手順を修正テンプレートとしてまとめる
3. 特定されたバグが修正テンプレートの適用可能条件に合致する場合に、その手順で修正を試みる

2. 修正テンプレートの作成

プログラムの抽象構文木 (AST) を書き換えるプログラム

1. AST分析 (AST analysis)
2. 文脈チェック (context check)
3. 編集手順 (program editing)

※AST分析と文脈チェックは、このテンプレートで修正可能なバグか調べるための手順

例 (Null Pointer Checker)

```
1 [Null Pointer Checker]
2 P = program
3 B = fault location
4
5 <AST Analysis>
6 C ← collect object references (method invocations,
   field accesses, and qualified names) of B in P
7
8 <Context Check>
9 if there is any object references in C ⇒ continue
10 otherwise ⇒ stop
11
12 <Program Editing>
13 insert an if() statement before B
14
15 loop for all objects in C {
16     insert a conditional expression that checks whether a
       given object is null
17 }
18 concatenate conditions by using AND
19
20 if B includes return statement {
21     negate the concatenated conditional expression
22     insert a return statement that returns a default value
       into THEN section of the if() statement
23     insert B after the if() statement
24 } else {
25     insert B into THEN section of the if() statement
26 }
```

Fig. 4: Null pointer checker fix template. This template inserts an **if**() statement checking whether objects are null.

修正テンプレート(全10個)

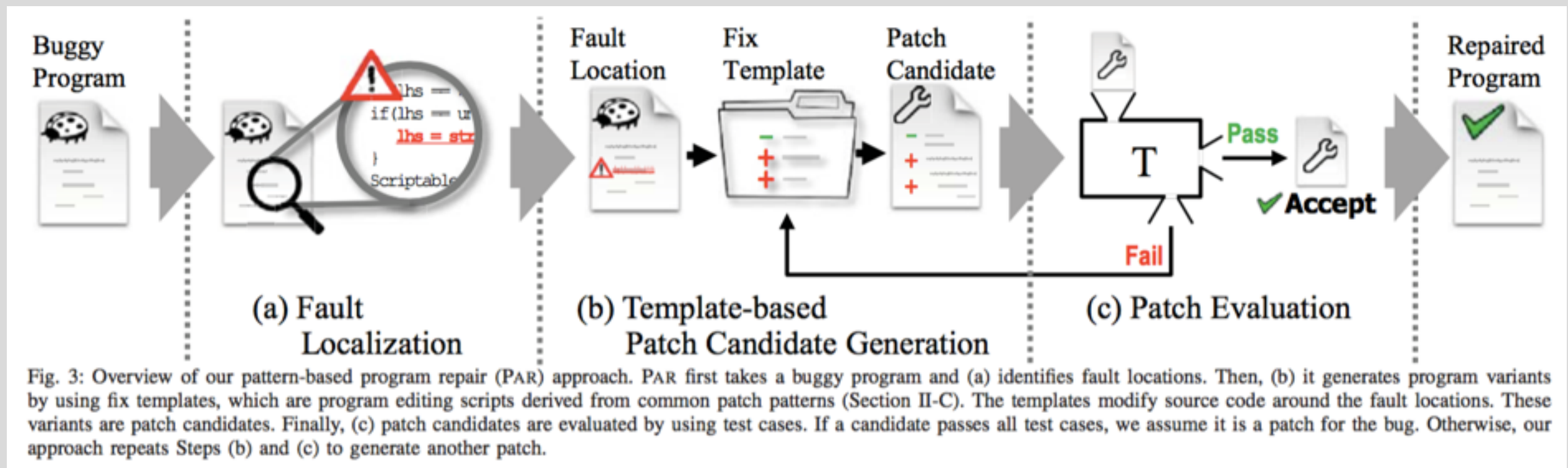
TABLE II: Fix templates derived from patterns in Section II-C.

Template Name	Description
Parameter Replacer	For a method call, this template seeks variables or expressions whose type is compatible with a method parameter within the same scope. Then, it replaces the selected parameter by a compatible variable or expression.
Method Replacer	For a method call, this template replaces it to another method with compatible parameters and return type.
Parameter Adder and Remover	For a method call, this template adds or removes parameters if the method has overloaded methods. When it adds a parameter, this template search for compatible variables and expressions in the same scope. Then, it adds one of them to the place of the new parameter.
Expression Replacer	For a conditional branch such as <code>if()</code> or ternary operator, this template replaces its predicate by another expression collected in the same scope.
Expression Adder and Remover	For a conditional branch, this template inserts or removes a term of its predicate. When adding a term, the template collects predicates from the same scope.
Null Pointer Checker	For a statement in a program, this template adds <code>if()</code> statements checking whether an object is <code>null</code> only if the statement has any object reference.
Object_INITIALIZER	For a variable in a method call, this template inserts an initialization statement before the call. The statement uses the basic constructor which has no parameter.
Range Checker	For a statement with array references, this template adds <code>if()</code> statements that check whether an array index variable exceeds upper and lower bounds before executing statements that access the array.
Collection Size Checker	For a collection type variable, this template adds <code>if()</code> statements that check whether an index variable exceeds the size of a given collection object.
Class Cast Checker	For a class-casting statement, this template inserts an <code>if()</code> statement checking that the castee is an object of the casting type (using <code>instanceof</code> operator).

提案手法の手順(再々掲)

1. 実際に人間が作成した修正パッチから、よくある修正パターンを見つける
2. 各パターンの修正手順を修正テンプレートとしてまとめる
3. 特定されたバグが修正テンプレートの適用可能条件に合致する場合に、その手順で修正を試みる

修正手順



修正例 (Null Pointer Checker)

```
01  if (kidMatch != -1) return kidMatch;
02  for (int i = num; i < state.parenCount; i++)
03  {
04      state.parens[i].length = 0;
05  }
06  state.parenCount = num;
```

(a) **Buggy Program:** the underlined statement is a fault location.

↓
<Null Pointer Checker>

INPUT: `state.parens[i].length = 0;`

1. Analyze: Extract obj refer → `state, state.parens[i]`
2. Context Check: object references?: PASS
3. Edit: INSERT

```
...
+ if( state != null && state.parens[i] != null ) {
+     state.parens[i].length = 0;
+ }
...
```

OUTPUT: a new program variant

↓

```
01  if (kidMatch != -1) return kidMatch;
02  for ( ... )
03  {
04+    if( state != null && state.parens[i] != null)
05        state.parens[i].length = 0;
06  }
07  state.parenCount = num;
```

(b) **After applying a fix template:** a patch generated by PAR. As shown in the fix template, corresponding statements have been edited.

Fig. 5: Real example of applying a fix template to `NativeRegExp.java` to fix Rhino Bug #76683.

評価実験

RQ1

どのくらいの数のバグを正しく修正できるのか（修正能力）

RQ2

より受け入れられやすい修正パッチを生成できるのは、どのアプローチか（受容性）

実験対象

TABLE III: Data set used in our experiments. “LOC” (Lines of code) and “# statements” represent the size of each subject. “# test cases” is the number of test cases used for evaluating patch candidates generated by PAR.

Subject	# bugs	LOC	# statements	# test cases	Description
Rhino	17	51,001	35,161	5,578	interpreter
AspectJ	18	180,394	139,777	1,602	compiler
log4j	15	27,855	19,933	705	logger
Math	29	121,168	80,764	3,538	math utils
Lang	20	54,537	40,436	2,051	helper utils
Collections	20	48,049	35,335	11,577	data utils
Total	119	483,004	351,406	25,051	

実験結果：RQ1（修正能力）

TABLE IV: Patch generation results. Among 119 bugs, PAR successfully fixed 27 bugs while GenProg was successful for only 16 bugs. Note that 5 bugs were fixed by both approaches. We used these 5 bugs in our comparative study for acceptability evaluation.

Subject	# bugs	# bugs fixed by GenProg	# bugs fixed by PAR	# bugs fixed by both
Rhino	17	7	6	4
AspectJ	18	0	9	0
log4j	15	0	5	0
Math	29	5	3	1
Lang	20	1	0	0
Collections	20	3	4	0
Total	119	16	27	5

Genprogの16個に対して、

PARは27個の修正パッチを生成できた

実験方法：RQ2(受容性)その1

学生(17人)と開発者(68人)を集めて、

- 人間、PAR、GenProgの修正パッチを、アプローチを伏せた状態で見せて、レビュアーの立場になって順位付けしてもらう
- PARもGenProgも修正できた5つのバグについて行う

実験結果：RQ2(受容性)その1

TABLE V: Average rankings evaluated by 17 students (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	1.33 (0.62)	2.27 (0.59)	2.40 (0.83)
Rhino #114493	2.00 (0.54)	1.33 (0.62)	2.67 (0.72)
Rhino #192226	1.47 (0.64)	1.67 (0.62)	2.67 (0.72)
Rhino #217379	1.69 (0.70)	1.50 (0.63)	2.81 (0.40)
Rhino #76683	2.13 (0.51)	1.07 (0.26)	2.80 (0.41)
Average	1.72 (0.67)	1.57 (0.68)	2.67 (0.64)

TABLE VI: Average rankings evaluated by 68 developers (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	1.92 (0.76)	2.00 (0.82)	2.08 (0.95)
Rhino #114493	1.60 (0.63)	2.40 (0.74)	2.00 (0.93)
Rhino #192226	2.00 (0.68)	1.79 (0.98)	2.21 (0.80)
Rhino #217379	1.62 (0.77)	1.69 (0.63)	2.69 (0.63)
Rhino #76683	1.92 (0.64)	1.23 (0.43)	2.85 (0.38)
Average	1.81 (0.70)	1.82 (0.80)	2.36 (0.90)

実験方法：RQ2(受容性)その2

学生(72人)と開発者(96人)を集めて、

- 人間の修正パッチとPAR(またはGenProg)の修正パッチを、アプローチを伏せた状態で見せて、レビュアーの立場になって受理できるか考えてもらう
- PAR(またはGenProg)が生成できた全てのバグについて行う

実験結果：RQ2(受容性)その2

TABLE VII: Indirect patch comparison results.

Selection	# response
PAR	130 (21%)
Both	175 (28%)
Human	229 (37%)
Not Sure	87 (14%)
Total	621 (100%)

(a) PAR comparison results.

Selection	# response
GenProg	68 (20%)
Both	40 (12%)
Human	176 (51%)
Not Sure	60 (17%)
Total	344 (100%)

(b) GenProg comparison results.

PARの方が、GenProgよりも受け入れられやすい修正パッチを生成できた

まとめ

- PARは27個の修正パッチを生成できた (Genprogは16個)
- PARの方が、GenProgよりも受け入れられやすい修正パッチを生成できた

所感

- 前半がほぼ人力だったことに驚いた
- リファクタリングパターンでも同じようなことができる?(すでにされているだろうな…)