ICSE ‘18

Context-Aware Patch Generation for Better Automated Program Repair

Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, Shing-Chi Cheung

Apresentado por **Eduardo Souza**

Setembro 2018

I4Soft
Intelligence for Software

UFG

Problem

The effectiveness of search-based automated program repair is limited in the number of correct patches.

1. The search space does not contain the correct patch;
2. The search space is huge and therefore the correct patch cannot be generated.

Naive approaches

Search space enlargement:

- Increase granularity;
- Low-order mutations.

Better strategies are required to prioritize the search space derived from fine fixing ingredients.

Major factors in search space explosion factor

Three spaces:

1. A buggy location generated by FL techniques
2. A mutation operator from a predefined set of operations (applied to buggy location)
3. Fixing ingredient (if required by 2)

1 + 2 + 3 = 

*[...] we approach the search space explosion by studying the **operation** space and **ingredient** space (**fix** space).*

=> how to effectively search for a correct patch?

Paper proposal

To prioritize the search for correct patches over plausible but incorrect ones base on mutation operators' and fixing ingredients' likelihood.

The context information of the suspicious code and the ingredients can provide rich information about their likelihood to correctly repair a bug.

The correct fixing ingredients and the buggy code elements share high similarity in terms of their contexts.

```
// Buggy statement
417:- return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
// Fixed statement
417:+ return equals(x, y, 1);
//The ingredients for fixing the bug
422: return (Double.isNaN(x) && Double.isNaN(y)) || equals(x, y, 1);
442: return equals(x, y, 1) || FastMath.abs(y - x) <= eps;
```

Figure 1: Buggy statement, fixed statement and the fixing ingredients of bug Math 63 in DEFECTS4J.

```
421: if (escapingOn && c[start] == QUOTE) { // Buggy statement
422: +     next(pos);   // Fixed statement (inserted)
423:       return appendTo == null ? null : appendTo.append(c);

// The ingredients for fixing the bug
170: if (c[pos.getIndex()] == START_FMT) {
171:       format = parse(pattern, next(pos));
```

Figure 2: Buggy statement, fixed statement and the fixing ingredients of bug Lang 43 in DEFECTS4J.

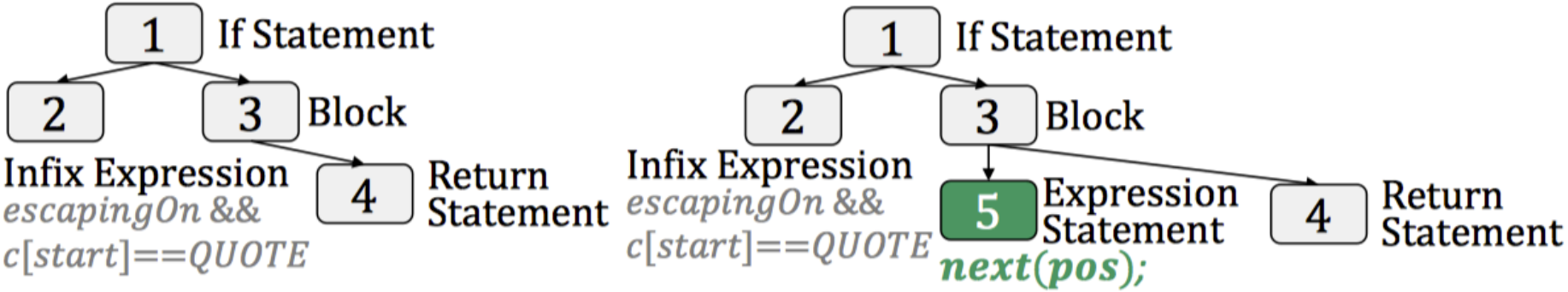


Figure 3: The AST differences of the example in Figure 2

Approach

Approach

- **C**ontext-**A**ware **P**atch **Gen**eration (CapGen)
- Works on AST: Type, Expression, and Statement (fixing ingredients).

Context-Aware Operators Selection

- **Replacement:** replace Target Node (Nt) with Source Node (Ns)
- **Insertion:** insert Source Node (Ns) under Target Node (Nt)
- **Deletion:** delete Source Node (Ns) under Target Node (Nt)

Context-Aware Ingredient Selection

- [hypothesis] correct fixing ingredients should share high similarity with the target nodes in terms of their contexts.
- Simi(Ns, Nt)

Extracting Context Information

- Model the context information of an AST node: genealogy, variable, and dependency.

Extracting Context Information - Modeling genealogy context

Algorithm 1: Modeling Genealogy Context

```
   input  : N: A given node
   output : Map φ : NodeType ↦ Integer: the extracted context information
 1 /* encoding ancestor nodes                                          */
 2 parent ← N
 3 while parent ≠ null & parent ≠ MethodDeclaration do
 4 |   if parent ≠ Block then
 5 |   |   φ(GetNodeType(parent)) += 1
 6 |   end
 7 |   parent ← GetParentNode(parent)
 8 end
 9 /* encoding sibling nodes                                           */
10 parent ← GetParentNode(N);
11 while parent ≠ Block do
12 |   parent ← GetParentNode(parent)
13 end
14 children ← parent.FilterChildrenOfType (Statement | Expression)
15 foreach node ∈ children do
16 |   φ(GetNodeType(parent)) += 1
17 end
```

Extracting Context Information - Modeling genealogy context

$$f(\phi_S, \phi_T) = \frac{\sum_{t \in \mathcal{K}} \min(\phi_T(t), \phi_S(t))}{\sum_{t \in \mathcal{K}} \phi_T(t)}$$

Extracting Context Information - Modeling variable context

$$g(\theta_S, \theta_T) = |\theta_S| * \frac{|\theta_S \cap \theta_T|}{|\theta_S \cup \theta_T|}.$$

Extracting Context Information - Modeling dependency context

Algorithm 2: Modeling Dependency Context

```
   input  : 𝒩: A given node
   output: Map ψ: NodeType ↦ Integer: the extracted context information
 1 /* context information extracted from backward slicing        */
 2 variableSet ← GetUseVariable(𝒩)
 3 foreach variable ∈ variableSet do
 4 |   contextStatements ← BackwardSlicing(variable)
 5 |   foreach stmt ∈ contextStatements do
 6 |   |   chs ← stmt.FilterChildrenOfType (Statement |Expression)
 7 |   |   foreach node ∈ chs do
 8 |   |   |   ψ(GetNodeType(node)) += 1
 9 |   |   end
10 |   end
11 end
12 /* context information extracted from forward slicing         */
13 variableSet ← GetDeforSetVariable(𝒩);
14 /* forward slicing is conducted the same on variableSet       */
```

Extracting Context Information - Modeling dependency context

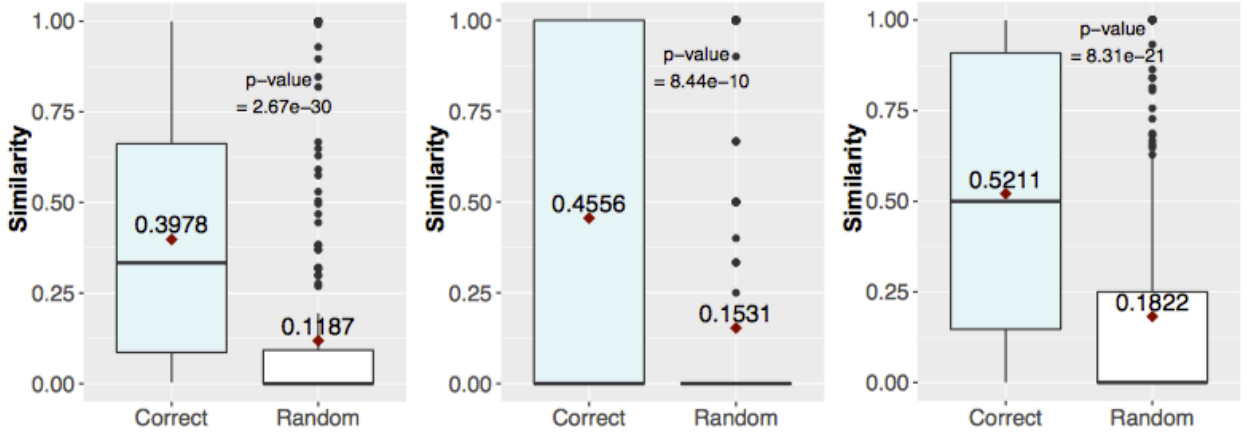
$$f(\psi_S, \psi_T) = \frac{\sum_{t \in \mathcal{K}} \min(\psi_T(t), \psi_S(t))}{\sum_{t \in \mathcal{K}} \psi_T(t)}$$

Empirical evidences

Empirical evidences

“Our intuition is that the required fixing ingredientes are supposed to share **high similarities** with the target node to be mutated in terms of their contexts.”

Empirical evidences



(a) Genealogy (b) Variable (c) Dependency

Figure 4: Comparisons of the contexts similarities between the correct fixing ingredients and the fixing ingredients with the same type as the correct one but randomly selected

Ingredients prioritization

$$Simi_R(N_s, N_t) = f(\phi_S, \phi_T) * f(\psi_S, \psi_T) * g(\theta_S, \theta_T)$$

$$Simi_I(N_s, N_t) = f(\phi_S, \phi_T) * f(\psi_S, \psi_T)$$

$$Simi_D(N_s, N_t) = (1 - f(\phi_O, \phi_T)) * (1 - f(\psi_O, \psi_T))$$

Patch prioritization

To generate candidate patches:

1. Select a buggy node *Nt* [prioritized by suspiciousness *FL(Nt)*]
2. Select a set o mutation operations that can be applied to Nt.
    1. Each mutation operator *M* has a operation probability *Freq(M)*, considering the context of *Nt*
3. Prioritize all compatible source nodes *Ns* [ingredients required by *M*] based on context similarity *Simi(Nt, Ns)*
4. *Patch = <Nt, M, Ns>*
5. *Score(Patch) = FL(Nt) * Freq(M) * Simi(Ns, Nt)*

Evaluation

Materials and methods

- Java
- Spoon
- GZoltar / Ochiai
- Understand (program analysis)
- 2x Intel Xeon + 192GB RAM
- 90 minutes
- Defects4J
- IntroClassJava

Research Questions

- **RQ1:** How effectively does CapGen fix real world bugs?
- **RQ2:** Can CapGen outperform existing approaches?
- **RQ3:** How is the contribution of each model used by CapGen?

RQ1: How effectively does CapGen fix real world bugs?

- Defects4J
- Generate and validate (Prophet)
- Ranked by *Score(Patch)*
- First run failing TCs, then regression
- Keeps generating and validating within the time budget
- Manually analyses the plausible to check if they are correct

- Bigger files => increased search space

- **25 plausible out of 224. 22 correct.**

RQ1: How effectively does CapGen fix real world bugs?

Table 3: Performance of CapGen on Defects4J

Project	Bug Id	Crt	Tot	Rank	Tie	P-B	P-T	P-A	Time
Chart	1	1	503	99	0	0	0	0	5.89
Chart	8	1	731	5	4	0	0	47	0.37
Chart	11	1	105	27	0	0	0	0	7.48
Chart	24	1	179	14	0	0	0	0	0.99
Lang	6	2	771	216	0	0	0	0	10.93
Lang	26	1	5,094	371	0	0	0	0	50.32
Lang	43	3	438	9	1	0	0	1	5.96
Lang	57	3	18,370	30	0	0	0	0	5.78
Lang	59	1	986	59	0	0	0	19	1.74
Math	5	1	1,166	150	0	0	0	3	5.33
Math	30	1	669	172	0	0	0	0	6.18
Math	33	1	9,412	43	10	0	0	0	21.51
Math	53	2	553	129	1	0	0	0	11.06
Math	57	1	421	78	0	0	0	0	19.07
Math	58	1	1,104	456	0	0	0	0	27.74
Math	59	1	225	9	0	0	0	0	4.61
Math	63	1	463	4	3	0	0	8	2.17
Math	65	1	31,152	3	2	0	0	0	0.18
Math	70	1	262	23	0	0	0	0	3.02
Math	75	1	464	7	2	0	0	0	0.85
Math	80	1	69,252	221	35	2	0	81	22.86
Math	85	1	566	3	0	0	0	3	3.42

Crt denotes the number of correct patches. ***Tot*** denotes the total number of patches generated. ***Rank*** is the rank of the first correct patch. ***P-B***, ***P-T*** and ***P-A*** denotes the number of the plausible patches that are ranked *before,in tie with* and *after* the first correct patch respectively. ***Time*** stands for the time required to find the first correct patch in **minutes**.

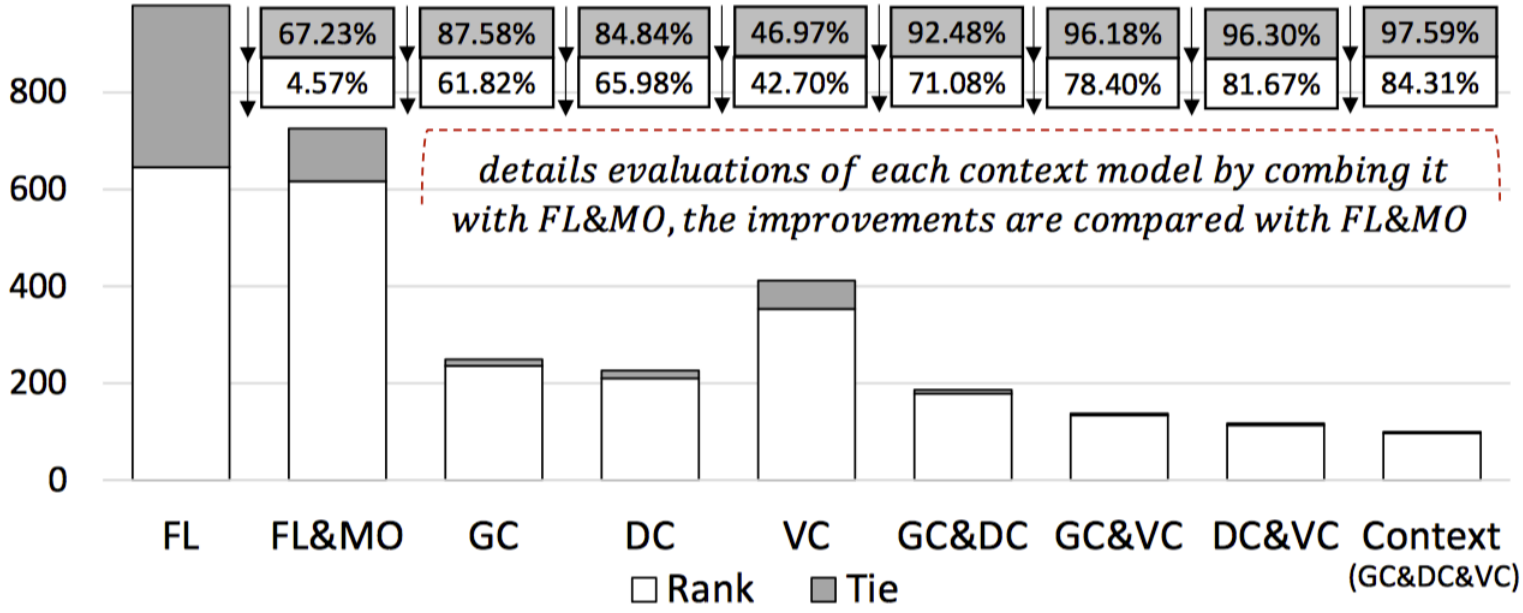
RQ2: Can CapGen outperform existing approaches?

Table 4: Comparisons with existing tools on Defects4J

Project	CapGen	HDRepair	jGenProg	PAR	ACS	Nopol
Chart	4	2/2	0/0	0/-	2/0	1/0
Lang	5	3/2	0/0	1/-	3/0	3/0
Math	12	4/2	5/3	2/-	12/2	1/1
Time	0	1/0	0/0	0/-	1/0	0/0
Total	21	10/6	5/3	3/-	18/2	5/1
Precision	84.0%	56.5%	18.5%	-	78.3%	14.3%

X/Y: X is the number of bugs repaired by the approach; Y is the number of bugs that repaired by CapGen and also by the approach. '-' means the results are not available due to the PAR did not present the ID of the repaired bugs.

RQ3: How is the contribution of each model used by CapGen?



RQ3: How is the contribution of each model used by CapGen?

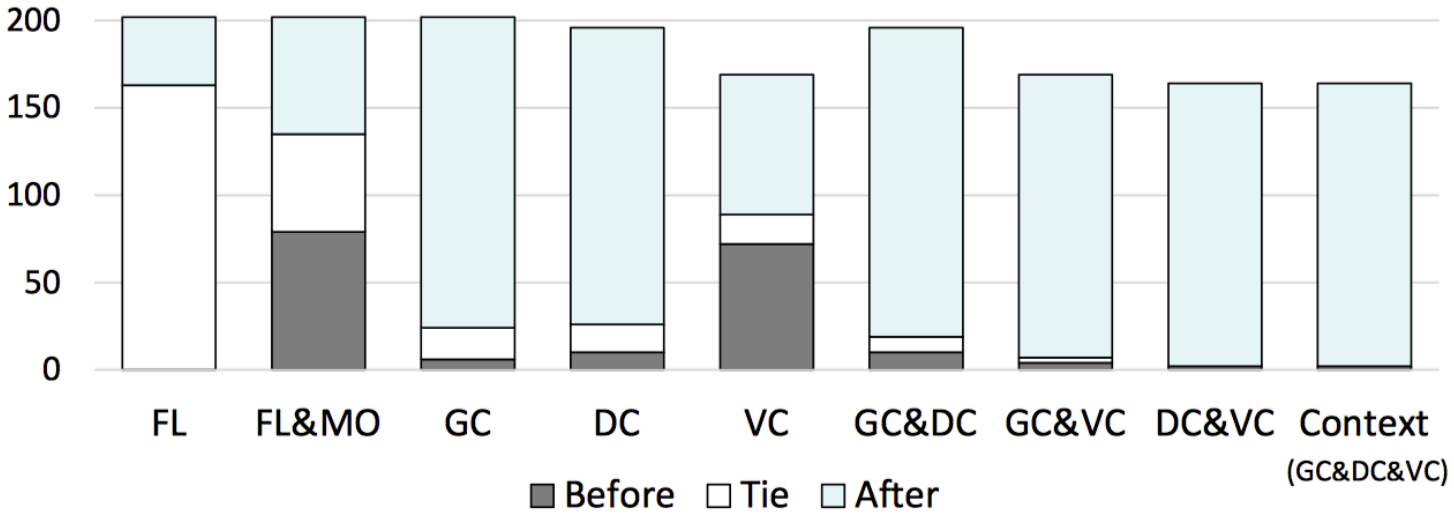


Figure 7: Total number of plausible patches


That's all Folks!