Context-Aware Patch Generation for Better Automated Program Repair

Ming Wen[1], Junjie Chen[2,3], Rongxin Wu[1], Dan Hao[2,3], Shing-Chi Cheung[1]
[1]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{mwenaa,wurongxin,scc}@cse.ust.hk
[2]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[3]Institute of Software, EECS, Peking University, Beijing, 100871, China
{chenjunjie,haodan}@pku.edu.cn

ABSTRACT

The effectiveness of search-based automated program repair is limited in the number of correct patches that can be successfully generated. There are two causes of such limitation. First, the search space does not contain the correct patch. Second, the search space is huge and therefore the correct patch cannot be generated (i.e., correct patches are either generated after incorrect plausible ones or not generated within the time budget).

To increase the likelihood of including the correct patches in the search space, we propose to work at a fine granularity in terms of AST nodes. This, however, will further enlarge the search space, increasing the challenge to find the correct patches. We address the challenge by devising a strategy to prioritize the candidate patches based on their likelihood of being correct. Specifically, we study the use of AST nodes' context information to estimate the likelihood.

In this paper, we propose CapGen, a context-aware patch generation technique. The novelty which allows CapGen to produce more correct patches lies in three aspects: (1) The fine-granularity design enables it to find more correct fixing ingredients; (2) The context-aware prioritization of mutation operators enables it to constrain the search space; (3) Three context-aware models enable it to rank correct patches at high positions before incorrect plausible ones. We evaluate CapGen on Defects4J and compare it with the state-of-the-art program repair techniques. Our evaluation shows that CapGen outperforms and complements existing techniques. CapGen achieves a high precision of 84.00% and can prioritize the correct patches before 98.78% of the incorrect plausible ones.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; *Software testing and debugging*;

KEYWORDS

Context-Aware, Automated Program Repair, Patch Prioritization

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180233

ACM Reference Format:
Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180233

1 INTRODUCTION

A recent study shows that the global cost of general debugging is 312 billion dollars annually and software developers spend 50% of their time on fixing bugs [6]. The excessively high cost in fixing bugs motivates the researches on repairing programs automatically. Over the years, various Automated Program Repair (APR) techniques have been proposed [16, 19–21, 23–25, 31, 33, 43, 45, 46, 53, 55]. Many of them [16, 19, 20, 29, 36, 45, 46] are known as search-based APR [17] (which may also be referred to as generate-and-validate APR [24, 38, 42] or heuristic-based APR [55]). They generate patch candidates by first applying a predefined set of mutation operators on the fault space determined by Fault Localization (FL) techniques [37, 49–51]. They then deploy some heuristics [20, 36] to search among these candidates for a correct patch that passes all given test cases [16, 19, 20, 25, 35, 36, 43, 46]. Search-based APR techniques have shown to be able to fix a wide range of bugs and be scalable to large programs without requiring extra specifications [20, 27].

Despite their effectiveness, there are two major limitations in existing search-based APR techniques [16, 20, 21, 36]. First, the correct patches are not always in the search space, making it impossible for them to repair bugs successfully. The second limitation arises from the *search space explosion* problem [20, 24]. To enhance the chances of including the correct patches in the search space, increasingly large search spaces (e.g., by using more mutation operators [16, 19, 23, 24]) are being designed. However, the space enlargement greatly increases the search efforts and can result in sharp reduction in repair effectiveness. For example, the large search space results in generating fewer correct patches due to producing plausible (i.e., patches that pass all test cases) but incorrect patches before the correct ones or budget timeout [24].

Finding good fixing ingredients is vital to resolve the first limitation [4, 30, 53]. Fixing ingredients are those existing code elements reused to generate fixing patches [30]. Existing approaches [16, 19, 20, 29, 36, 46] work on Statement level, which are too coarse-grained to find the correct fixing ingredients since previous studies have shown that good fixing ingredients exist more often at a finer granularity than that of statements [4, 30]. This motivates us to leverage fine-granularity fixing ingredients (e.g., expressions)

to generate patches. However, doing so will further increase the search space and hence aggravate the second limitation. This issue was also observed by Barr *et al.* [4] who found that the genetic programming search algorithm is always bogged down within a few generations due to the search space explosion when trying out GenProg [20] with fixing ingredients at the Expression level. Therefore, better strategies are required to prioritize the search space derived from fine fixing ingredients.

In pursuit of an efficient search strategy, we analyze the patch generation process of search-based APR and identify the major factors contributing to the search-space explosion problem [21]. An APR process typically involves generating three spaces. First, it selects a buggy location generated by FL techniques (i.e., fault space). Second, it selects a mutation operator from a predefined set of operations (i.e., operation space) that can be applied to the selected buggy location. Third, it selects a fixing ingredient (i.e., ingredient space) if it is required by the mutation operator. The ingredient space can be a set of code fragments extracted from the same application[20, 23, 36, 45] or across multiple applications[41]. Therefore, the *fault space*, *operation space* and *ingredient space* jointly attribute to the search space explosion problem[21]. Among the three spaces, the fault space is a set of suspicious code elements, which are potentially buggy, derived from a given set of test results. It depends largely on the adequacy of these tests [52, 56]. Following existing practices, we directly leverage the suspicious values returned by FL techniques to prioritize the fault space[15, 16, 20, 33, 36, 43, 46, 48].

In this paper, we approach the search space explosion problem by studying the other two spaces (i.e., operation space and ingredient space), which collectively referred to as the *fix space* [21]. Specifically, we study how to efficiently search for a correct patch, if any, in the fix space of a selected suspicious code element. Existing strategies of search-based APR randomly choose operators and ingredients from the fix space [16, 19, 20, 27, 36, 46]. However, this random strategy is inefficient and subject to high possibilities of producing plausible patches [24, 25, 53]. Therefore, we propose to prioritize the search for correct patches over plausible but incorrect ones based on mutation operators' and fixing ingredients' likelihood to correctly repair the selected suspicious code element.

We find that the context information of the suspicious code and the ingredients can provide rich information about their likelihood to correctly repair the bug. Our finding is supported by our empirical study showing that the correct fixing ingredients and the buggy code elements share high similarity in terms of their contexts. Leveraging the findings, we develop an approach called CapGen, which stands for **C**ontext-**A**ware-**P**atch **Gen**eration. CapGen works on a fine-granularity at the AST node level. It is context-aware in two aspects. First, the mutation operators are prioritized concerning the context information. Specifically, CapGen considers the AST node types of the required ingredient and the suspicious code element when selecting mutation operators. However, whether bugs can be fixed more often under certain contexts than the others remains unknown. To address this challenge, CapGen leverages substantial real bug fixes in open source projects to guide the discovery of such contexts. Second, the fixing ingredients are extracted together with their context information. Specifically, we proposed three models to capture the context information in terms of the ingredients' syntactics and semantics. CapGen leverages these models to guide the application of the ingredients to the correct locations. This is based on our intuition that a fixing ingredient should be applied to the locations with similar contexts compared with the location where it is extracted. To the best of our knowledge, incorporating the context information when selecting mutation operators and prioritizing fixing ingredients is new to program repair.

We evaluate CapGen on the Defects4J [14] benchmark. The evaluation results show that CapGen makes two major accomplishments. First, CapGen generates plausible patches for 25 bugs and 21 of them are correct, thus achieving a high precision of 84.00%. It outperforms the current state-of-the-art approaches in terms of the number of bugs repaired correctly and the precision. More specifically, using the context models, the ranks of correct patches has been improved for 85.31% and the number of ties of the rankings has been reduced by 97.59% on average. This significantly alleviates the search-space explosion problem. Second, it generates no incorrect plausible patches for 60.00% (15/25) of the bugs. For those with incorrect plausible patches generated, the correct patches can be ranked in prior to 98.78% of the incorrect ones. It is because that our context-aware models can precisely identify the correct fixing ingredients and thus reduce the possibility of generating incorrect plausible patches. This also significantly alleviates the overfitting problem [42]. Among these 21 bugs correctly repaired, 15 of them have never been repaired by existing APR approaches, showing that CapGen well complements existing approaches. We also evaluated CapGen on the IntroClassJava[10] benchmark, the results of which show the generality of CapGen.

In summary, our paper makes the following novel contributions:

- We propose CapGen, which is a context-aware patch generation technique. To the best of our knowledge, CapGen is the first approach to incorporating the context information to prioritize mutation operators and apply fixing ingredients.
- We design three novel models to capture the context information of fixing ingredients. We also present empirical evidences supporting that the correct fixing ingredients should share high similarities with the buggy element in terms of their contexts.
- We evaluate CapGen on the benchmark Defects4J. The results show that CapGen can generate correct patches with a high precision of 84.00%. Moreover, it outperforms and complements existing state-of-the-art approaches.

2 BACKGROUND AND MOTIVATION

To address the two aforementioned limitations of search-based APR, we study the real bug fixes and make some interesting observations on patch granularities, the operation space and the ingredient space.

- ***Patch Granularities:*** We found that one important reason which hinders existing approaches [16, 20, 29] to generate the correct patches in the search space is that the designed mutation operators work at the statement level, which are too coarse-grained. Martinez *et al.* reported that code redundancy is significantly higher at a finer granularity than statement level [30]. This suggests that it is more likely to find good fixing ingredients at a finer granularity. Figure 1 shows an example, which is a bug caused by a wrong expression in the return statement at line 417. To fix it, developers changed the original expression to `equals(x,y,1)`. The fixing ingredient `equals(x,y,1)` exists at two other places (line 422 and line 442) in the same program. If we apply mutation operators at the

statement level (e.g., replacing the return statement with another) as existing approaches [16, 19, 20, 29], we cannot fix the bug by replacing line 417 with line 422 nor line 442. Another example is shown in Figure 2, where the fixing ingredient `next(pos)` also exists at a granularity finer than statements. These motivate us to extract fixing ingredients and apply mutation operators at a finer granularity, such as the expression level.

```
// 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.

Applying mutation operators with ingredients at a finer granularity increases the likelihood to include the correct patches in the search space [4, 30]. However, a serious side-effect is that it will increase the fix space (i.e., operation and ingredient space) significantly, exacerbating the search space explosion problem [4]. A recommended way to address this is to prioritize the patches based on their likelihood to repair a bug correctly as adopted by recent approaches [19, 53, 55]. We make two key observations on the fix space from existing literature and the motivating examples, which shed lights on how to prioritize the search of correct patches.

```
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.

• ***Observations on the Operation Space:*** Insertion, deletion and replacement are mutation operators commonly used to generate patches [16, 20, 27, 29, 46]. As revealed by existing studies [28, 58], the likelihood of repairing a bug is correlated with the concerned operator and the type[1] of the involved code elements. For instance, *insertion* can correctly repair a bug more frequently with a code element of Expression Statement[2] than other types of code elements [58]. In the example in Figure 2, the bug is fixed by inserting `next(pos)` as an Expression Statement at line 422.

We further observe that the likelihood of correct bug repairing is correlated with the context of the involved code elements. Our observation is in line with a recent finding that the syntactic usage of code elements is highly contextual [39]. For instance, a `switch` statement usually occurs much more often under a `while` statement than a `for` statement [39]. Suppose we want to replace the buggy expression at line 417 in Figure 1 with another compatible expression (i.e., expressions evaluated as `boolean` values). There are many types of ingredients (e.g., Method Invocation, Infix Expression) that can be evaluated as `boolean`. However, under the context of a Return Statement, Method Invocation has a relatively high probability of 24.4% to be selected as an ingredient for the returning value and is more than six times higher than that of type Boolean or Infix Expression [39]. Therefore, expressions of type Method Invocation should be preferentially selected as ingredients to fix this bug in a Return Statement.

These motivate us to prioritize the mutation operators using the context information derived from the AST node type of the involved fixing ingredient and that of the enclosing code element where this ingredient is going to be applied. However, how to select those contexts that can cover as many bugs as possible while keeping a tractable operation space is a challenge. In this study, we leverage substantial open source projects to learn those contexts under which real bugs are frequently fixed, and use such knowledge to prioritize mutation operators (see Section 3.1).

• ***Observations on the Ingredient Space:*** We observe that the context information of the fixing ingredients can guide us to select the correct ingredient to the suspicious buggy location. Consider the return expression at line 417 of the example in Figure 1. Even we have decided to patch the expression using method invocations as aforementioned, we still need to select one of the many method invocations in the program to generate a patch. A random selection is less likely to generate a correct patch. The correct fixing ingredient exists at two other places in the program. If looking at their contexts (e.g., the enclosing statements where they are extracted), we can find that both of them are used in `return` statements, and there are no other usages of `equals(x,y,1)` in this program. Therefore, this fixing ingredient should be preferentially selected over the others to patch the buggy `return` statement at line 417. A similar case is found for the example in Figure 2. The ingredient `next(pos)` is extracted from a statement enclosed by an `if` statement, and thus it should be preferentially selected and inserted under the buggy `if` statement at line 421.

This motivates us to consider the similarity between the context where an ingredient is extracted and the context where the buggy code resides when generating patches. An ingredient with a higher context similarity should be ranked higher. In section 3.2.1, we propose three models to measure the context similarities in terms of an ingredient's and a buggy code's genealogical structures, accessed variables and semantic dependencies. To validate if such measurement can reliably indicate the likelihood of generating correct patches, we conduct an empirical study to collect the supporting evidence in Section 3.2.2.

3 APPROACH

We propose a **C**ontext-**A**ware **P**atch **Gen**eration (CapGen) approach, which works on the Abstract Syntax Tree (AST) of a program. Specifically, all the AST nodes in the category of Type, Expression and Statement [39] are extracted as fixing ingredients. This section first introduces the context-aware prioritization of operations and the fixing ingredients. After that, it introduces how CapGen integrates the fault space, operation space and ingredient space together to prioritize the generated patches.

3.1 Context-Aware Operators Selection

Motivated by our observations, we consider the three basic AST mutation operators, which take a fixing ingredient as a source node and the place (i.e., the buggy code element) where the ingredient is to be applied as a target node.

[1]Type refers to the AST node type in this study, such as Infix Expression
[2]https://msdn.microsoft.com/en-us/library/s7ytfs2k.aspx

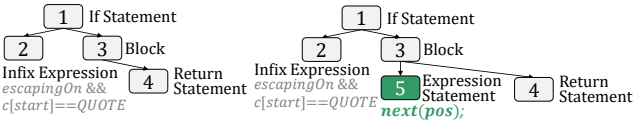


Figure 3: The AST differences of the example in Figure 2

- Replacement: replace *Target Node ($\mathcal{N}_t$)* with *Source Node ($\mathcal{N}_s$)*
- Insertion: insert *Source Node ($\mathcal{N}_s$) under Target Node ($\mathcal{N}_t$)*
- Deletion: delete *Source Node ($\mathcal{N}_s$) under Target Node ($\mathcal{N}_t$)*

The operation space induced by these operators is large [28]. Reduction of the space is a critical issue in APR [21, 24], and we propose to leverage context information to achieve this. Based on our observations in Section 2, we explicitly augment a mutation operator with the context information of the source node's type and the target node's type. Examples of augmented operators are "insert Simple Name under Method Invocation" and "delete Expression Statement under Method Declaration". More examples of augmented mutation operators can be found in Table 1. To reduce the operation space, we identify the probability to correctly fix bugs for each augmented operator and consider only those operators that offer a relatively high probability. We derive the probabilities from the dataset of patches systematically collected by Le Goues *et al.* [19]. The dataset contains over 3,000 real fixing patches from over 700 open source projects and is public available [1]. For ease of presentation, we refer to the dataset as BugFixSet in this paper. For each patch in BugFixSet, we compare the differences between the buggy and the fixed version at the AST node level. For example, Figure 3 gives the AST differences for the bug fix (line 421 to 423) of our motivating example in Figure 2. A node with type Expression Statement (i.e., node 5) is inserted under the target node of If Statement (i.e., node 1). The node with type Block is not considered since it provides little context information for mutation operation. Note that node 5 is a subtree composed of other nodes such as Method Invocation (i.e., `next`) and Simple Name (i.e., `pos`). We regard the operation performed on this subtree as one repair action [58] since the transformation can be completed in one single operation (inserting the whole subtree). Therefore, the augmented mutation operator extracted from this patch is "insert an Expression Statement under an If Statement".

BugFixSet provides a large and representative repository of real patches [19], and this enables us to obtain the knowledge of those augmented mutation operators that are frequently used in real bug fixes. We count the frequency of each augmented mutation operators participating in the patches of BugFixSet to approximate its probability to fix bugs. The higher the frequency, the higher the probability of fixing real bugs. Table 1 shows the top 10 augmented mutation operators with the highest frequencies for replacement, insertion and deletion, respectively. We select only these 30 augmented operators for CapGen to prioritize the operation space. They cover 69.69% of the repair actions participating in the patches of BugFixSet altogether. We use $\mathcal{M}$ and $Freq(\mathcal{M})$ to denote a selected augmented mutation operator and its frequency, respectively. For *replacement*, we utilize the data type (e.g., `int, boolean`) of a node, rather than its node type, to infer context information. It is because that if the data type of a fixing ingredient does not match that of the target node, the mutated program will not be compiled correctly. For example, both the data types of the fixing ingredient (i.e., `equals(x,y,1)`) and the target node (i.e., the returning expression) in Figure 1 are `boolean`.

3.2 Context-Aware Ingredients Prioritization

Existing studies have reported that a significant portion of the fixing ingredients can be found in the same file of the buggy code [4, 30]. Therefore, we extract the ingredients from the file where the target node (i.e., the buggy code element) resides. Unlike existing studies [16, 19, 20, 29, 36, 46], we also extract the context information for each ingredient. Our hypothesis is that the correct fixing ingredients should share high similarity with the target nodes in terms of their contexts. We denote the similarity as $Simi(\mathcal{N}_s, \mathcal{N}_t)$. In the following, we first introduce three aspects of context information embedded in an AST node and its surrounding codes, and then explain how to measure the context similarity between two nodes for each aspect. After that, we validate our hypothesis via an empirical study on the BugFixSet. Finally, we introduce how to prioritize the ingredients for the three types of augmented mutation operators.

3.2.1 *Extracting Context Information.*

We model the context information of an AST node from three different aspects: genealogy, variable and dependency.

First, a node's genealogical structure can provide useful context. For instance, the infix expression `equals(x,y,1)` in Figure 1 is frequently used as the returning value in `return` statements. Therefore, this infix expression should have higher probability to be used as the ingredient for mutating the returning values in Return Statement than the conditions of If Statement. This information of where an ingredient is frequently used can be extracted from its genealogical structures, e.g., the *Ancestor Nodes* of the node.

Second, variable usages in an ingredient can provide useful context information since variables are primary components of an ingredient [55]. The example in Figure 1 shows that the fixing ingredient uses the same variables (e.g., *x, y*) as the target expression to be replaced. Moreover, plausible patches can be generated by simply removing functionalities via replacing complex expressions with constant values (i.e., no variable usages). Patches thus generated might pass a project's test suite due to the weak test problem [42]. These motivate us to rank those ingredients with more similar variable usages compared with the target node at higher positions.

Third, the variables used or defined in a given node are affected by or can affect other nodes. Such semantic contexts in terms of the nodes' dependencies are reported to be helpful in capturing the characteristics of correct patches [5, 25]. Therefore, CapGen extracts this dependency information as an aspect of a node's context to prioritize fixing ingredients.

• ***Modeling Genealogy Contexts:***

CapGen extracts the genealogy contexts of an AST node $\mathcal{N}$ via checking its *Ancestor Nodes* to see $\mathcal{N}$ is used under what types of code elements, and checking its *Sibling Nodes* to see $\mathcal{N}$ is used together with what types of code elements. For ancestor nodes, we traverse from $\mathcal{N}$ to its ancestors until we reach a method declaration. For *Sibling Nodes*, we extract those nodes of category Statement and Expression within the same Block of $\mathcal{N}$. We store the genealogical information in ϕ, which counts the number of occurrences of different node types among all the ancestor and sibling nodes. Algorithm 1 shows the details of this process.

Table 1: Top replacement, insertion and deletion operators augmented with target and source nodes' types

Replacement		*Insertion*			*Deletion*		
Target Node	Freq	Source Node	Target Node	Freq	Source Node	Target Node	Freq
Simple_Name	0.1721	Expression_Statement	Method_declaration	0.0815	Simple_Name	Method_Invocation	0.0191
String_Literal	0.1310	Simple_Name	Method_Invocation	0.0313	Expression_Statement	Method_Declaration	0.0157
Infix_Expression	0.0201	Expression_Statement	If_Statement	0.0304	Infix_Expression	If_Statement	0.0144
Qualified_Name	0.0197	Method_Invocation	Method_Invocation	0.0282	Method_Invocation	Method_Invocation	0.0144
Conditional_Expression	0.0135	If_Statement	If_Statement	0.0141	Expression_Statement	If_Statement	0.0072
Simple_Type	0.0082	Infix_Expression	Method_Invocation	0.0125	Infix_Expression	Method_Invocation	0.0066
Boolean_Literal	0.0075	Qualified_Name	Method_Invocation	0.0075	Prefix_Expression	Infix_Expression	0.0041
Parameterized_Type	0.0038	Expression_Statement	Try_Statement	0.0075	Boolean_Literal	Method_Invocation	0.0038
Primitive_Type	0.0009	If_Statement	Method_Declaration	0.0063	Expression_Statement	Catch_Clause	0.0031
Assignment	0.0003	Boolean_Literal	Method_Invocation	0.0047	Method_Invocation	Infix_Expression	0.0028
	0.3771			0.2240			0.0912

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
```

When applying a source node $\mathcal{N}_s$ with context ϕ_S under a target node with context ϕ_T, we compute its feasibility via comparing the similarities $f(\phi_S, \phi_T)$ between these two contexts:

$$f(\phi_S, \phi_T) = \frac{\sum_{t \in \mathcal{K}} \min(\phi_T(t), \phi_S(t))}{\sum_{t \in \mathcal{K}} \phi_T(t)} \quad (1)$$

where $\mathcal{K}$ denotes a set of all distinct AST node types captured by ϕ_T. Here, Equation 1 measures the proportion of the contexts in ϕ_T that can be covered by the contexts of the source node $\mathcal{N}_s$.

• ***Modeling Variable Contexts:***

Given an AST node $\mathcal{N}$, CapGen extracts a set of variables (including local variables and fields) that are accessed (i.e., read/write) in this node, which is denoted as θ. Each of the elements in set θ is denoted as ϵ, which is a tuple ⟨Type, Name⟩ recording the data type and the name of the variable. We use the Jaccard distance to measure the similarity between the variable contexts of two nodes θ_T and θ_S. Besides, to avoid plausible patches generated by involving trivial ingredients, we give more weights to those complex ingredients. Therefore, we use the number of variables involved in $\mathcal{N}_s$ to represent the weight of the ingredient.

$$g(\theta_S, \theta_T) = |\theta_S| * \frac{|\theta_S \bigcap \theta_T|}{|\theta_S \bigcup \theta_T|}. \quad (2)$$

Two elements are the same only if both the type and the name are matched. However, for nodes like Simple Name (i.e., variables), when we replace it with another, their names must be different. Therefore, we only require the data type is the same for such cases.

• ***Modeling Dependency Contexts:***

Given an AST node $\mathcal{N}$, CapGen extracts its dependency contexts via investigating those nodes affecting $\mathcal{N}$ and those nodes affected by $\mathcal{N}$. Specifically, we conduct intra-procedure backward slicing and forward slicing based on the `definition` and `use` (also known as `def-use`) of variables extracted from a given node [47]. Lines 2 to 11 in Algorithm 2 describe the details of extracting the context information by leveraging backward slicing. For each of the variables used in the node, CapGen slices a set of statements which affect the values of the variable. We then extract the AST nodes in these statement that are instances of Expression or Statement and store such information in a map ψ to count the number of occurrences of each node type in the sliced statements. Such context information is also extracted by leveraging forward slicing.

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      */
```

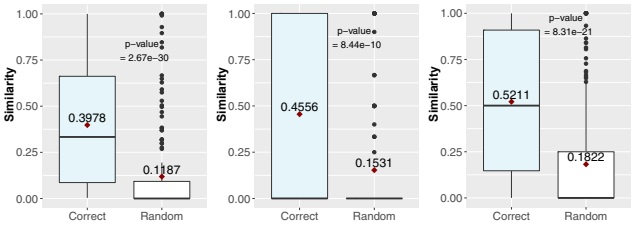
The feasibility of applying a source node $\mathcal{N}_s$ with dependency context ψ_S under a target node with context ψ_T is similar to how we measure the feasibility in terms of the genealogy contexts. Specifically, we measure the proportion of the target node' contexts that can be covered by the contexts of the source node.

$$f(\psi_S, \psi_T) = \frac{\sum_{t \in \mathcal{K}} \min(\psi_T(t), \psi_S(t))}{\sum_{t \in \mathcal{K}} \psi_T(t)} \quad (3)$$

where $\mathcal{K}$ denotes a set of all AST node types captured by ψ_T.

3.2.2 *Empirical Evidences.*

We designed three different models to capture the context similarities between the fixing ingredients $\mathcal{N}_s$ and the target node $\mathcal{N}_t$. Our intuition is that the required fixing ingredients are supposed to share high similarities with the target node to be mutated in terms of their contexts. To validate the intuition, we collected empirical evidences from the BugFixSet dataset. Among 12.65% of the bug fixes in BugFixSet, fixing ingredients exist in their associated buggy program. This proportion is consistent with existing empirical studies [4, 30]. For each of these bug fixes, we first extract the correct fixing ingredient as $\mathcal{N}_s$ together with its context information from the associated buggy program. We then compare it with the target node $\mathcal{N}_t$ at the buggy location for each of the three aforementioned context models and denote the similarity as S_s. For comparison, we also *randomly* select another ingredient $\mathcal{N}_r$ whose type is the same as $\mathcal{N}_s$ and can also be applied to the target location. For example, if $\mathcal{N}_s$ is a node of Infix Expression with type `boolean`, we also randomly select another infix expression with type `boolean` whose variables can be resolved at the target location. This process is similar to the process of selecting ingredients randomly by existing approaches [16, 20, 36, 38]. We denote the context similarity between $\mathcal{N}_r$ and $\mathcal{N}_t$ as S_r. For each bug, we



(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

repeat the random process 50 times, and S_r is the averaged context similarity value over the 50 runs. We then compare S_s and S_r to see if S_s is greater than S_r significantly.

Figure 4 shows the results in terms of the genealogy, variable and dependency context models. Here, for the variable context model, we only keep the JACCARD term, which measures similarity. The similarities of all models are normalized to a range from 0 to 1. The results show that for all the three models, the similarity between $\mathcal{N}_t$ and $\mathcal{N}_s$ is significantly higher than that between $\mathcal{N}_t$ and $\mathcal{N}_r$ ($p < 0.001$) using the one-sided Mann-Whitney U-Test [26]. These results empirically support our intuition that correct fixing ingredients should share high similarities with the target node in terms of their context information.

Actually, existing studies have also investigated where to find fixing ingredients [4, 12, 30]. However, their investigations focus on finding ingredients at the file level (i.e., in the same file or in different files) [30] or at the commit level (i.e., in the current version or in the previous versions) [4]. For example, Barr *et al.* [4] found that a significant portion of fixing ingredients exist in the same file. Our empirical study takes a further step and searches the fixing ingredients at a finer granularity within the same file (i.e., those code elements with higher context similarities). In particular, our findings are also useful for future search-based as well as semantics-based APR, both of which need to search fixing ingredients in their patch generation process.

3.2.3 Ingredients Prioritization.

We integrate the three models to prioritize the extracted ingredients. Those ingredients with higher context similarities (i.e., with a larger value of $Simi(\mathcal{N}_s, \mathcal{N}_t)$) are ranked higher. However, we have different treatments to the three different mutation operators. For *replacement*, we combine all the three context models, and thus the $Simi(\mathcal{N}_s, \mathcal{N}_t)$ is specifically defined as:

$$Simi_R(\mathcal{N}_s, \mathcal{N}_t) = f(\phi_S, \phi_T) * f(\psi_S, \psi_T) * g(\theta_S, \theta_T) \quad (4)$$

However, for *insertion*, the $\mathcal{N}_t$ is the parent node of $\mathcal{N}_s$ after inserting $\mathcal{N}_s$ under $\mathcal{N}_t$, and thus it makes no sense to include the variable model in this case since $\mathcal{N}_t$ and $\mathcal{N}_s$ serve different functionalities. The same case for *deletion*. Therefore, for *insertion*:

$$Simi_I(\mathcal{N}_s, \mathcal{N}_t) = f(\phi_S, \phi_T) * f(\psi_S, \psi_T) \quad (5)$$

For *deletion*, the case is more complicated since we are going to delete $\mathcal{N}_s$ from $\mathcal{N}_t$. In this case, we first locate the nodes in the other places $\mathcal{N}_o$ which are identical to $\mathcal{N}_s$. We compute the context similarity between $\mathcal{N}_o$ and $\mathcal{N}_t$. If there is a high context similarity between $\mathcal{N}_o$ and $\mathcal{N}_t$, which means the source node $\mathcal{N}_s$ is used in similar contexts at the other places. In this case, we give lower probability to delete $\mathcal{N}_s$ under $\mathcal{N}_t$. Therefore, for *deletion*, $Simi(\mathcal{N}_s, \mathcal{N}_t)$ is defined as followings specially:

$$Simi_D(\mathcal{N}_s, \mathcal{N}_t) = (1 - f(\phi_O, \phi_T)) * (1 - f(\psi_O, \psi_T)) \quad (6)$$

where $\mathcal{N}_o \equiv \mathcal{N}_s$. If there is no other nodes which are identical to $\mathcal{N}_t$, then $Simi(\mathcal{N}_s, \mathcal{N}_t)$ would simply be 1.

3.3 Patch Prioritization

In summary, to generate candidate patches, CAPGEN first selects a buggy node $\mathcal{N}_t$, prioritized by its suspicious value $FL(\mathcal{N}_t)$ generated by FL techniques. Note that existing FL techniques assign suspicious values at the line level, CAPGEN applies these FL techniques by assigning a line's suspicious value to each of its AST nodes in the same line. The suspicious value of those AST nodes (e.g., IF STATEMENT) that span across multiple lines are averaged over these lines. Then, for the selected $\mathcal{N}_t$, CAPGEN selects a set of mutation operators that can be applied to $\mathcal{N}_t$, and each mutation operators $\mathcal{M}$ has the corresponding operation probability $Freq(\mathcal{M})$ by considering the contexts of $\mathcal{N}_t$. Afterwards, CAPGEN prioritizes all compatible source nodes $\mathcal{N}_s$, which are used as ingredients required by $\mathcal{M}$, based on their context similarity $Simi(\mathcal{N}_s, \mathcal{N}_t)$ compared with the target node $\mathcal{N}_t$. Finally, patches are generated, and each of them is denoted as $P = \langle \mathcal{N}_t, \mathcal{M}, \mathcal{N}_s \rangle$. CAPGEN prioritizes those patches by integrating the fault space, the operation space and the ingredient space together. The final score of P is denoted as $Score(\langle \mathcal{N}_t, \mathcal{M}, \mathcal{N}_s \rangle)$, which is calculated as shown in Equation 7.

$$Score(\langle \mathcal{N}_t, \mathcal{M}, \mathcal{N}_s \rangle) = FL(\mathcal{N}_t) * Freq(\mathcal{M}) * Simi(\mathcal{N}_s, \mathcal{N}_t) \quad (7)$$

Currently, we use multiplication to combine all the models to filter out those patches generated with ingredients sharing no context similarities. Designing different ways to combine those components together is left for future work as discussed in Section 5.1.

4 EVALUATION

4.1 Research Questions

We implemented CAPGEN in Java based on a code transformation library Spoon [34] and the fault localization tool GZoltar [2]. Specifically, CAPGEN leverages GZoltar with the algorithm Ochiai [37] to retrieve the fault space and the suspicious values. We use the program analysis tool Understand [3] for code slicing. Our experiments are run on a CentOS server with 2x Intel Xeon E5-2450 Core CPU @2.1GHz and 192GB physical memory. We set the time budget for each bug as 90 minutes following existing practices [16, 19]. Our evaluation aims to answer the following 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?

The implementation of CAPGEN and the details of our evaluation results are publicly available at:

https://github.com/justinwm/CapGen.

4.2 The Subject Programs

To answer the three RQs, we run CAPGEN on the benchmark datasets for program repair: DEFECTS4J [14]. The benchmark was built to facilitate controlled experiments in software debugging and testing researches [14], which has been widely adopted by recent studies on APR[19, 27, 29, 40, 54]. Following existing practices [27, 40, 53], we use DEFECTS4J of *version 1.0.1* and exclude the Closure Compiler

Table 2: Subjects for evaluation

Subject	#Bugs	KLOC	Test KLOC	#Test Cases
Commons Lang	65	22	6	2,245
JFreeChart	26	96	50	2,205
Commons Math	106	85	19	3,602
Joda-Time	27	28	53	4,130
Total	224	231	128	12,182

project because it uses a customized testing format instead of JUnit test [53]. As a result, four projects with a total of 224 real bugs are used as subjects for experiment. Their demography is given in Table 2. Evaluations on another benchmark IntroClassJava [10, 22] are also conducted (see discussion in Section 5.2).

4.3 RQ1: Performance of CapGen on Real Bugs

To answer RQ1, we apply CapGen to the Defects4J benchmark. For each bug, we validate the generated patches prioritized by their $Score(\langle \mathcal{N}_t, \mathcal{M}, \mathcal{N}_s \rangle)$ one by one. We first run the failing test cases to see if they pass on the generated patch. If so, we then run the regression test cases to see whether this patch introduces new bugs. We skip to validate the next patch otherwise. CapGen keeps generating and validating the candidate patches within the time budget. After all the patches have been validated or the budget timeout, we collect the *plausible* patches which pass both the failing test cases and the regression test cases. We then manually analyze these patches to see if they are semantically equivalent to the patches submitted by developers. We mark them as *correct* patches if they are, and *incorrect plausible* patches otherwise.

CapGen generates plausible patches for 25 bugs in total among all the 224 bugs. We identify the correct patches for 22 of these 25 bugs successfully. Table 3 shows the details of these 22 bugs. Column *Tot* shows the number of the patches generated while *Crt* shows the number of the correct ones among them. The total number of patches generated is affected by the size of the fault space and the size of buggy source file (since more fixing ingredients will be extracted if the buggy source file is larger and contains more statements). There are cases that CapGen generates more than one correct patch. For example, CapGen generates 2 correct patches for Math 53, which requires inserting a statement to fix the bug. We find that both the two patches are equivalent to the developers' patch since the required statement can be inserted at two adjacent places. Columns *Rank* and *Tie* in Table 3 show the rank of the first correct patch and the number of patches, including wrong patches (i.e., those did not pass the test suite) and plausible patches, that are ranked in tie with the first correct patch respectively. The value of *Rank* is the rank of the first position if there are ties. Column *Time* shows that the time required for CapGen to generate the correct patches ranges from 0.18 to 50.32 minutes. Columns *P-B*, *P-T* and *P-A* show the number of incorrect plausible patches grouped by their relative ranks (i.e., before, in tie with and after) compared with the correct patches. We can see that CapGen generates incorrect plausible patches for 40.00% (10/25) of the bugs. However, even though CapGen might generate incorrect plausible patches, it ranks them mostly after the correct ones. Table 3 shows that 21 out of the 22 bugs, CapGen ranks no incorrect plausible patches before or in tie with the first correct patches. The only exception is Math 80, for which CapGen generates two incorrect plausible patches that are ranked before the first correct patch.

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**.

For the 21 correctly repaired bugs, 90.48% of them require a fixing granularity finer than the statement level. Examples are given in Figures 1 and 5. Some of the cases requiring fixing ingredients with finer granularities can be handled by specifically designed mutation operators. For example, for bug Chart 1, by replacing the infix expression `dataset!=null` with another `dataset==null`, CapGen can repair it. HDRepair also repaired it by including a mutation operator that negates boolean expressions [19]. Currently, CapGen includes 30 augmented mutation operators, and seven of them can repair at least one bug correctly.

These results show the effectiveness of CapGen. Specifically, it repairs 21 bugs successfully (i.e., the first patch passing all tests is correct) and achieves a high precision of 84.00% (21/25), which outperforms the existing state-of-the-art approach ACS [53] with a precision of 78.30%. Precision is vital for APR techniques because much debugging effort would be wasted if the provided patches are plausible but incorrect [53]. This high precision is achieved via patch prioritization guided by our context-aware models. Specifically, 98.78% (164/166) of the incorrect plausible patches are ranked after the correct ones (see Section 4.5 for detailed analysis).

4.4 RQ2:Comparison with Existing Approaches

We compare CapGen with five APR approaches, ACS [53], HDRepair [19], jGenProg [27], Nopol [54] and PAR [16, 19], which have been evaluated on the Defects4J benchmark within our knowledge. Among them, ACS [53] and HDRepair [19] are the state-of-the-art approaches. Similar to CapGen, HDRepair [19] and PAR [16] set the time budget as 90 minutes. Nopol [54] and jGenProg [27] set the searching time as 3 hours. ACS [53] sets the budget as 30 minutes since it only targets at condition synthesis and thus the corresponding search space is much smaller. Table 4 shows the comparison results. The baselines' results are directly extracted from existing literature [16, 20, 53, 54]. Compared with these techniques, CapGen outperforms all of them in terms of the number

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.

of correctly repaired bugs and precision. By analyzing the overlap of repaired bugs between CapGen and the compared techniques, CapGen actually complements the-state-of-art techniques. 15 of the 21 bugs repaired by CapGen have never been repaired by existing approaches [16, 19, 20, 53, 54].

Six of the 21 bugs repaired by CapGen can be repaired by the state-of-the-art search-based approach HDRepair [19]. This happens when the required fixing ingredient is at the statement level (e.g., Math 53) or can be handled by specific designed mutation operations (e.g., Chart 1). The dominant reason why CapGen can repair 15 new bugs is that CapGen works at a finer granularity (e.g., Figure 2). For the 4 bugs that can be fixed by HDRepair [19] but not CapGen, the major reason is that the fixing ingredients do not exist in our searching scope. For example, for Math 34, the correct fix is to replace the method invoking expression `chromosimes` in `chromosomes.iterator()` with `getChromosomes()`. However, the fixing ingredient `getChromosomes()` does not exist in the buggy program, and thus CapGen failed to repair the bug. To handle this case, HDRepair [19] designed a mutation operator that replaces the name of a method call (or a method invoking expression) by another method name (or expression) with compatible types. By including more mutation operators or increasing the scope to extract fixing ingredients, CapGen can handle more cases (see Section 5.1 for more discussions).

```
1135: int j = 4 * n - 1;  // Buggy statement
1135: int j = 4 * (n - 1);  // Fixed statement
 // The ingredients for fixing the bug
1133: if (1.5 * work[pingPong] < work[4 * (n - 1) + pingPong])
 941: double d = work[4 * (n - 1) + pingPong];
```

Figure 5: Buggy statement, fixed statement and the fixing ingredients of bug Math 80 in Defects4J.

Comparing with ACS [53], CapGen can repair 19 different bugs. This is because ACS only targets at synthesizing predicate conditions, and thus other bugs in general are out of the repairing capabilities of ACS. For instance, our motivating example shown in Figure 1 which replaces the return value of another expression. Another motivating example in Figure 2 can not be repaired by ACS neither, which is an omission bug and requires inserting a method invocation to repair it. Actually, these kind of bugs can hardly been repaired successfully by ACS, as well as semantics-based APR [23, 32, 33, 53], since they mainly focus on synthesizing conditions or right-hand side of assignments. This embodies the indispensability of search-based APR in program repair which targets at fixing bugs in general. We also notice that CapGen successfully repaired two bugs that can be repaired by ACS which are caused by wrong conditions. For example, by replacing the condition in `if (fa * fb >= 0.0)` with `fa * fb > 0.0`, CapGen can repair Math 85.

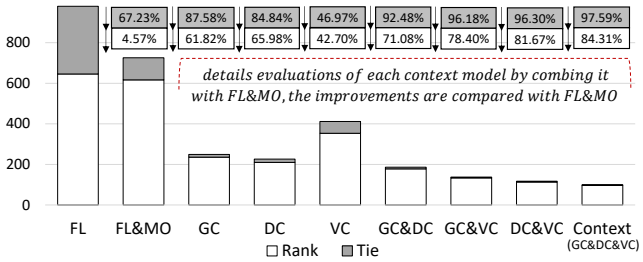


Figure 6: The average rank of first correct patch.

The same case for Chart 1. However, CapGen requires the existence of the fixing ingredients in the program. Those cases whose fixing ingredients do not exist in the program might be handled by ACS.

4.5 RQ3: Contributions of Each Model

We use the average results of the 22 bugs shown in Table 3 to answer this question. Specifically, we conducted nine experiments to examine the contribution of each component of CapGen. Figure 6 shows the results. Each bar has two heights. The white part indicates the first correct patch's average rank, and their height difference (gray colored) gives the number of other patches that are ranked in tie with the correct one. In the first setup, we use only the suspicious value of the fault space (i.e., $FL(\mathcal{N}_t)$) to rank all the patches generated. All mutation operators, which are denoted as MO, bear the same weights, and no context models are considered. In the second setup, we study the prioritization performance with the mutation operators' weights combined (i.e., $FL(\mathcal{N}_t) * Freq(\mathcal{M})$). In the remaining seven setups, we evaluate the contribution of each of our context models and their combinations one by one. Specifically, we integrate a single context model to the final ranking (i.e., GC for Genealogy Context, DC for Dependency Context and VC for Variable Context) and the arbitrary combinations of them. In Figure 6, bars from position 3 to 9 show the results by combining the context models with FL and MO (e.g., VC means CapGen leverages FL, MO and the VC model to produce the final ranking).

As shown in Figure 6, each individual model contributes to improve the rank of correct patches. If we use only the FL model, the patches are simply ranked by the suspicious values of the buggy locations return by GZoltar. Due to inadequacy of test cases [42], the ranking of the correct buggy location is low and contains many ties. Besides, many possible patches can be generated for each location. These two reasons account for the poor ranks (e.g., , the low rank and many ties) of the correct patches. By combining the weights of the mutation operators, CapGen can significantly break the ties and reduce the number of ties by 67.23%. The performance of the first rank has also been improved by 4.57%. By further introducing all the context models of fixing ingredients, the rank is further improved by 84.31% and the number of ties are further reduced by 97.59% with respect to the ranking of using $FL(\mathcal{N}_t) * Freq(\mathcal{M})$. Let us illustrate this using the bug shown in Figure 5. The buggy expression is in an Assignment under an If Statement (line 1133). The correct fixing ingredient also exists in an Assignment and a condition expression in an If Statement. By capturing such context information, we can prioritize the correct fixing ingredients at high positions. Specifically, the correct patch is ranked at 6758 with 3767 ties if only the FL model is used. By including the MO model, it can be ranked at 5807 with 1002 ties. By including all the three

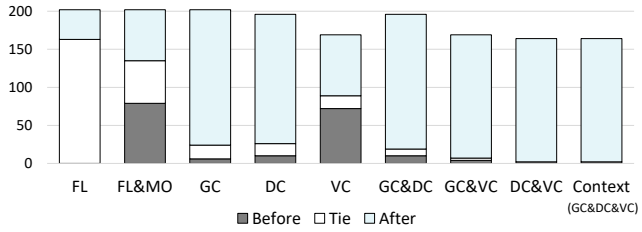


Figure 7: Total number of plausible patches

context models, it can be ranked at 221 with 35 ties, which can be searched within 30 minutes. For the rankings of each individual model and the combinations of them, the improvement is computed against the ranking of using $FL(\mathcal{N}_t) * Freq(\mathcal{M})$. The results show that the three models collectively achieve the best performance, followed by the combination of the two models of DC and VC.

Another crucial function of our context-ware models is that they can prioritize the correct patches before the incorrect plausible ones. The majority of the patches generated which pass all test cases are plausible but incorrect due to the weak tests provided [42, 52, 56]. Previous techniques [16, 19, 20, 29, 36, 46] use random techniques to search among these generated patches, and therefore, they are more likely to produce the incorrect plausible patches [38]. This problem can be significantly alleviated by prioritizing the patches using our context models. Figure 7 shows the distributions of the total number of incorrect plausible patches ranked *before*, *in tie with* and *after* the correct patches under the nine analyses. The use of FL model alone in the first analysis leads to 80.69% of the incorrect plausible patches generated being ranked in tie with or before the first correct one. The number has been reduced by 17.18% by introducing the MO model, and 98.52% in a further step by combining our context models. Finally, only 2 plausible patches are ranked before or in tie with the correct ones. We also present the results in detail for each of our context model. For example, by integrating the GC, DC and VC model alone can reduce the number by 82.22%, 80.74% and 34.07% respectively comparing with the results generated by leveraging FL and MO.

Let us further illustrate the effectiveness of our context models by case analysis. For our motivating example shown in Figure 1, CapGen can rank the correct patch at position 4 before all other 8 plausible bugs. Replacing the variable `x` in the function `isNaN(x)` with field `MathUtils.TWO_PI` is one example of incorrect plausible patches. However, due to the low context similarity shared by `x` and `MathUtils.TWO_PI`, the final probability of this patch is only 4.97-*e*4, and thus ranked after the correct patch. Replacing `Double.isNaN(x) && Double. isNaN(y)` with expression `x==0` can also pass the test suite. However, our variable context model favors those ingredients which involve more similar variables with the target node. Therefore, this patch is also ranked after the correct one. Similar cases for the other bugs.

CapGen fails to rank the correct patch before all incorrect plausible ones for Math 80 shown in Figure 5. Due to weak test cases provided, 81 incorrect plausible patches have been generated and 78 of them are ranked after the correct one using our context models. However, there are still 2 exceptions. For example, replacing the variable `n` at line 1135 with another variable `pingPong` can pass all the test cases, and CapGen fails to rank this one after the correct patch. It is because these two variables are used together a lot in the program and thus sharing a high context similarity. Besides, our variable context model can not filter out the rank of this patch since it only requires their types to be the same if both the ingredient and the target node are Simple Name as described in Section 3.2.1.

These results show that each of our proposed model can help rank the correct patches at top positions and, more importantly, before the incorrect plausible ones.

5 DISCUSSIONS

5.1 On the Extensions of CapGen

CapGen performs a single mutation to generate patches, which indicates CapGen can only target at fixing those bugs requiring a single repair action currently. However, this type of bugs is common in practice [28, 58]. For example, a existing study found that around 30% of the bugs can be fixed by a single repair action [58]. In the Defects4J benchmark, 31.70% of the bugs (71/224) can be repaired by a single repair action. Note that, patches generated via a simple repair action might also be complex. For example, CapGen inserts a whole If Statement, which across three lines, to fix bug Math 53. In the future, we will include performing mutations at multiple places in CapGen . At the same time, we will also try to reduce the increase of the search space that it incurs. One possible direction is to perform the same operation at multiple buggy target nodes simultaneously in the fault space if they are identical to each other. For example, an buggy expression `return isZero?NaN:INF` exists at multiple places for Math 46, repairing this bug requires changing all these buggy expressions to the correct ones.

In the future, we also plan to include more mutation operators to increase the chances of repairing more bugs, which could be easily integrated by design. Furthermore, CapGen currently searches only in the same program file where the target node resides to find fixing ingredients. Among the 71 bugs which require only one repair action, only 27 of them whose all required fixing ingredients can be found in the same program. Therefore, extending the searching scope is a promising extension for CapGen, which can increase the chances to find the correct fixing ingredients for more bugs [30]. For example, if we search among the whole application, we can find the ingredients for two more bugs among the 71 bugs. However, including more mutation operators and enlarging the scope to search the ingredients will inevitably increase the search space and thus make it more challenge to identify the correct patch. Therefore, better searching strategies are desired. Currently, CapGen combines FL, MO and the three context models by multiplication to prioritize the search space. In the future, CapGen will try better strategies such as learning-to-rank techniques to integrate these models.

5.2 Threats to Validity

One potential threat to validity is the generality of the benchmark Defects4J, which means that our evaluation results may not be generalized to other dataset. We planed to use BugFixSet for cross validation. However, it only provides the patch for each bug without the corresponding test cases. Therefore, it makes it impossible to validate the correctness of the patches generated by CapGen automatically. The other benchmark in Java besides Defects4J is IntroClassJava [10, 22] within our knowledge. To alleviate the threat of generality, we also evaluated CapGen on this benchmark. IntroClassJava is the Java version of the IntroClass benchmark

originally proposed by Le Goues *et al.* [22]. This benchmark contains 297 bugs from student-written homework assignments. These bugs feature complicated fixes although they are small programs [55]. We follow exiting techniques [18, 57] to check whether a generated patch is correct. We run CapGen on this benchmark and found that it can successfully repair 25 bugs in total. The state-of-the-art semantics-based APR techniques, JFix [18] and Angelix [32], which work on the IntroClassJava benchmark, are able to repair 19 and 7 bugs respectively [18]. Specifically, CapGen can repair 14 new bugs. It is because neither Angelix nor JFix can handle floating point or strings related bugs [55] limited by the capability of the constraint solving techniques. Therefore, they have only evaluated on a subset of the IntroClassJava benchmark which only includes integer and boolean-related fixes. CapGen has no such limitations and are evaluated on the whole dataset. This result also shows that CapGen complements well to existing approaches.

Besides, CapGen is designed independent of the dataset used for evaluation. Specifically, the mutation operators are prioritized based on substantial real bug fixes extracted from open source projects. For the three context models proposed to prioritize the ingredients, we conduct an empirical study and the results shows that the correct fixing ingredients indeed share high context similarity with the target nodes measured by our models. This finding is general, and can facilitate the design of selecting fixing ingredients for both search-based and semantics-based APR techniques in the future.

6 RELATED WORK

Automated program repair techniques can be broadly classified into two categories. The first category is search-based APR [16, 19, 20, 36, 38, 43, 45, 46]. The most representative of this category is GenProg [20, 46], which searches for correct patches via genetic programming algorithm. RSRepair [36] adopts the same fault space and operation space as GenProg. It differs from GenProg in that it searches among all the candidates randomly instead of using genetic programming. Reducing the number of candidate patches and reducing the test cases required for validating a patch are two important aspects to boost the efficiency of search-based APR techniques [11]. Motivated by this, AE [45] is proposed to reduce the total cost required to find a correct patch. In order to generate candidates that are more likely to be the correct patch, PAR [16] proposes to leverage fixing templates learned from human patches to generate possible candidates. Each template is able to fix a common type of bugs. The design of mutation operators of CapGen is also guided by substantial real bug fixes. However, CapGen only leverages the syntactic information to design the mutation operators (i.e., inserting a type of code element under another type of code element). PAR designs mutation operators considering the semantic information such as adding a null pointer checker. Designing with such specific information makes PAR less generalizable than CapGen. The most recent work is HDRepair [19], which leverages historical data to search for correct patches.

The second category is semantics-based APR, which synthesizes a repair patch directly using semantic information via symbolic execution and constraint solving [9, 13, 15, 31–33, 44, 54]. Staged Program Repair (SPR) [23] is a hybrid of search-based and semantics-based automated program repair technique. It leverages traditional mutation operators to generate candidate patches [8], and it is also capable of synthesizing conditions via symbolic execution. Prophet [25] was proposed based on SPR, which is capable of prioritizing candidate patches via learning from correct patches automatically using machine learning techniques. SPR and Prophet only synthesize program elements involving conditions. Nopol is also capable of synthesizing conditions [54]. SemFix [33] is capable of synthesizing right hand side of assignments besides conditions. Angelix [32] was proposed to address the scalability issue concerning semantics-based techniques using a novel lightweight repair constraint *angelic forest*. S3 was recently proposed to synthesize patches leveraging program-by-examples techniques [55].

Both search-based APR and semantics-based APR techniques have their advantages and disadvantages. Search-based APR is simple, intuitive and generalizable to fix all types of bugs, which make them more effective. However, the efficiency is greatly compromised by the search space explosion problem [24]. On the other hand, semantics-based APR is more effective since the search space is more tractable by using program synthesis with restricted components. However, the effectiveness could be limited by the capability of constraint solving and program synthesis. Besides, both of these two categories suffer from the overfitting problem [42, 52].

In order to generate high-quality patches, many approaches propose to rank the patches based on their likelihood to be correct recently [7, 19, 25, 31, 53, 55]. HDRepair prioritizes patches based on their similarities with previous fix patterns that are mined from software histories [19]. Prophet [25] also compares the generated patch with existing human patches to investigate its probability of being correct but leverages different features as HDRepair [19]. ACS leverages the information mined from other projects and Java documentations to prioritize the variables and predicates used in the synthesized conditions [53]. S3 prioritizes the generated patches by comparing their similarities with the original buggy program in terms a set of features [55]. Different from them, CapGen directly leverages the context information extracted from the buggy code elements and the fixing ingredients to prioritize the generated patches. To the best of our knowledge, incorporating the context information of fixing ingredients is new to program repair.

7 CONCLUSION

We present a novel search-based APR technique, CapGen, which works on the AST node level to increase the chances of including the correct patches in the search space. The key novelty of CapGen which allows it to generate and search the correct patches efficiently is that the patch generation process is context-aware, specifically, in the following two aspects: 1). The mutation operators are prioritized considering the context information guided by large historical data. 2). Fixing ingredients are extracted together with their context information, and such information is leveraged to prioritize the generated patches. We evaluate CapGen on Defects4J, and the results show that CapGen outperforms and complements existing state-of-the-art techniques. More importantly, CapGen can achieve a precision of 84.00% and can prioritize the correct patches in prior to 98.78% of the plausible but incorrect ones.

ACKNOWLEDGMENTS

The work was supported by the Hong Kong RGC/GRF Grant No. 16202917, the MSRA Collaborative Research Award, and the National Natural Science Foundation of China under Grant No. 61522201.

REFERENCES

[1] 2017. https://github.com/xuanbachle/bugfixes. (2017). Accessed: 2017-03-22.
[2] 2017. http://www.gzoltar.com. (2017). Accessed: 2017-03-22.
[3] 2017. Understand. https://scitools.com. (2017). Accessed: 2017-03-22.
[4] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE'14*. ACM, 306–317.
[5] Marcel Böhme, Ezekiel Olamide Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is it Fixed? An Experiment with Practitioners. In *ESEC/FSE'2017 (ESEC/FSE 2017)*. 1–11.
[6] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013).
[7] Loris DâĂŹAntoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.
[8] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 65–74.
[9] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. ACM, 30–39.
[10] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. Ph.D. Dissertation. Universite Lille 1.
[11] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *GECCO'2009*. ACM, 947–954.
[12] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *FSE'10*. ACM, 147–156.
[13] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 389–400.
[14] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA'2014*. ACM, 437–440.
[15] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (T). In *ASE'2015*. IEEE, 295–306.
[16] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE'2013*. IEEE Press, 802–811.
[17] Xuan-Bach D Le. 2016. Towards efficient and effective automatic program repair. In *ASE'2016*. ACM, 876–879.
[18] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *ISSTA'17*. ACM, 376–379.
[19] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER'2016*, Vol. 1. IEEE, 213–224.
[20] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *ICSE'2012*. IEEE, 3–13.
[21] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software Quality Journal* 21, 3 (2013), 421–443.
[22] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.
[23] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE'2015*. ACM, 166–178.
[24] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE'2016*. ACM, 702–713.
[25] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 298–312.
[26] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.
[27] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2016. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empirical Software Engineering* (2016), 1–29.
[28] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.
[29] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA, Demonstration Track*.
[30] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 492–495.
[31] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *ICSE'2015*, Vol. 1. IEEE, 448–458.
[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE'2016*. ACM, 691–701.
[33] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *ICSE'2013*. IEEE Press, 772–781.
[34] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. DOI : http://dx.doi.org/10.1002/spe.2346
[35] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, and others. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 87–102.
[36] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE'2014*. ACM, 254–265.
[37] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *ISSTA'2013*. ACM, 191–201.
[38] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA'2015*. ACM, 24–36.
[39] Dong Qiu, Bixin Li, Earl T Barr, and Zhendong Su. 2017. Understanding the syntactic rule usage in java. *Journal of Systems and Software* 123 (2017), 160–172.
[40] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *ASE'2015*. IEEE, 201–211.
[41] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 43–54.
[42] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.
[43] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated repair of software regressions. In *ICSE'2015*. IEEE Press, 471–482.
[44] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *ISSTA'2010*. ACM, 61–72.
[45] Westley Weimer, Zachary P Fry, and Stephanie Forrest. 2013. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'2013*. IEEE, 356–366.
[46] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE'2009*. IEEE Computer Society, 364–374.
[47] Mark Weiser. 1981. Program slicing. In *ICSE'1981*. IEEE Press, 439–449.
[48] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An Empirical Analysis of the Influence of Fault Space on Search-Based Automated Program Repair. *arXiv preprint arXiv:1707.05172* (2017).
[49] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *ASE'2016*. ACM, 262–273.
[50] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2009. A survey on software fault localization. (2009).
[51] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2017. ChangeLocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* (2017), 1–35.
[52] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA,17*. ACM, 226–236.
[53] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE'17*. IEEE Press, 416–426.
[54] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. (2016).
[55] D Le Xuan-Bach, Chu Duc-Hiep, Lo David, Goues Claire, Le, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *FSE'2017*. ACM, to appear.
[56] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *FSE'17*. ACM, 831–841.
[57] Luciano Zemín, Simón Gutiérrez Brida, Ariel Godio, César Cornejo, Renzo Degiovanni, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2017. An analysis of the suitability of test-based patch acceptance criteria. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*. IEEE Press, 14–20.
[58] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *ICSE'15*. IEEE Press, 913–923.