electronics

MDPI

Article

Patch It If You Can: Increasing the Efficiency of Patch Generation Using Context

Jinseok Heo [1,†], Hohyeon Jeong [1,†] and Eunseok Lee [2,*]

[1] Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, Republic of Korea
[2] College of Computing and Informatics, Sungkyunkwan University, Suwon 16419, Republic of Korea
* Correspondence: leees@skku.edu
† These authors contributed equally to this work.

Abstract: Although program repair is a tremendous aspect of a software system, it can be extremely challenging. An Automated Program Repair (APR) technique has been proposed to solve this problem. Among them, template-based APR shows good performance. One of the key properties of the template-based APR technique for practical use is its efficiency. However, because the existing techniques mainly focus on performance improvement, they do not sufficiently consider the efficiency. In this study, we propose EffiGenC, which efficiently explores the patch ingredient search space to improve the overall efficiency of the template-based APR. EffiGenC defines the context using the concept of extended reaching definition from compiler theory. EffiGenC constructs the search space by collecting the ingredient required for patching in the context. We evaluated EffiGenC on the Defects4j benchmark. EffiGenC decreases the number of candidate patches from 27% to 86% compared to existing techniques. EffiGenC also correctly/plausibly fixes 47/72 bugs. For Future work, we will solve the search space problem that exists in multiline bugs using context.

Keywords: software verification and validation; automated program repair; search-based repair; search space; patch ingredient; context

Citation: Heo, J.; Jeong, H.; Lee, E. Patch It If You Can: Increasing the Efficiency of Patch Generation Using Context. *Electronics* **2023**, *12*, 179. https://doi.org/10.3390/electronics12010179

Academic Editor: Sanjay Misra

Received: 1 December 2022
Revised: 22 December 2022
Accepted: 24 December 2022
Published: 30 December 2022

1. Introduction

An automated program repair(APR) can reduce the debugging costs by automatically fixing a buggy code [1,2]. Moreover, the template-based APR technique is one of the techniques showing good performance among the APR techniques [3–5]. It generates a template from the commit history. FixMiner [6] collects patch history from open-source repositories. It used a rich edit script to capture the structure of the AST and then used it to generate the patch pattern. TBar [3] verifies templates from existing template-based APR. It then checks the patches generated using such templates.

For patch generation, template-based APR approaches additionally leverage various context information about the buggy code. ConFix [7] uses the AST node near the modification point as a context to efficiently explore the patch history and changes. CAPGEN [8] uses genetics, variables, and dependency similarities between suspicious codes and candidate patches as context. Furthermore, it utilizes patch prioritization to increase performance.

The main metrics of existing template-based APR methods focus on a performance evaluation [9]. Liu et al. [9] showed that the performance of the APR technique has steadily improved. However, the efficiency, which is a key property for the practical use of the APR technique, has not improved.

To improve efficiency, APR require an effective search strategy for search within a reasonable amount of time. Among the benchmark Defects4j bugs, Figure 1 shows the developer patches for the Lang-24 and Closure-125 bugs. Both patches can be generated using the *Mutate Conditional Expression* template proposed by TBar, one of the latest

template-based APR techniques. However, in the case of TBar, the same patch can be generated only for Lang-24, and not for Closure-125.

Lang 24

src/main/java/org/apache/commons/lang3/math/NumberUtils.java

```
                @@ -1410,7 +1410,7 @@ public static boolean isNumber(String str) {
1410  1410          if (chars[i] == 'l'
1411  1411              || chars[i] == 'L') {
1412  1412              // not allowing L with an exponent or decimal point
1413        -         return foundDigit && !hasExp;
      1413  +         return foundDigit && !hasExp && !hasDecPoint;
1414  1414          }
1415  1415          // last character is illegal
1416  1416          return false;
```

(**a**)

Closure 125

src/com/google/javascript/jscomp/TypeCheck.java

```
                @@ -1658,7 +1658,7 @@ private void visitNew(NodeTraversal t, Node n) {
1658  1658      JSType type = getJSType(constructor).restrictByNotNullOrUndefined();
1659  1659      if (type.isConstructor() || type.isEmptyType() || type.isUnknownType()) {
1660  1660        FunctionType fnType = type.toMaybeFunctionType();
1661        -     if (fnType != null) {
      1661  +     if (fnType != null && fnType.hasInstanceType()) {
1662  1662          visitParameterList(t, n, fnType);
1663  1663          ensureTyped(t, n, fnType.getInstanceType());
1664  1664        } else {
```

(**b**)

Figure 1. Example of search space problem. (**a**) Developer patch of Lang-24; (**b**) Developer patch of Closure-125.

The major difference between the two patches is that, in the case of Lang-24, only one variable *hasDecPoint* is added, and in Closure-125, *fnType.hasInstanceType()*, a method invocation, is added. Compared to Lang-24, Closure-125 has an exponential search space because it requires an additional search of the class and method. If TBar can effectively search the ingredient search space, it will produce a patch equivalent to that of the developer of the closure-125 bug within a given amount of time.

In this paper, to improve the efficiency of the template-based APR, we propose EffiGenC (Increasing the **Effi**ciency of Patch **Gen**eration using **C**ontext) that efficiently explores the search space of ingredients using context. EffiGenC considers the statement related to the target statement as context. For this, we extend the concept of reaching definition in compiler theory. Reaching definition for a given statement is the closest earlier statement whose target variable can reach it without an intervening assignment. EffiGenC obtains the statements and methods that are the context of target statements through reaching definition. It explores the context and collects the patch materials needed to generate a patch. This experimental study on five state-of-the-art template-based APR systems demonstrate that, overall, EffiGenC can reduce the number of candidate patch by up to 86%. Even when we extend the search space from file to project, the number of candidate patches increased by only 29% compared to the exponential increase of ingredients.

The contributions of this study are as follows.

- New context concept through extended reaching definition.
- An APR technique to efficiently explore the patch ingredient search space.
- Evaluation of APR performance and efficiency through Real java dataset.

The rest of this paper is organized as follows. The following Section 2 summarizes the terms for understanding the proposed approach. Section 3 presents the detailed process of the proposed technique. Sections 4 and 5 present our experimental setup and results. Section 6 discusses the limitation of our approaches. After surveying the related studies in Section 7, we provide some concluding remarks in Section 8.

2. Terminology

Sensical patch versus nonsensical patch. A sensical patch can successfully compile a buggy program. A nonsensical patch cannot successfully compile a buggy program [9].

Plausible patch versus in-plausible patch. A plausible patch allows a buggy program to successfully compile and pass all test cases in the available test suite. An in-plausible patch still allows the buggy program to successfully compile but fail to pass certain test cases in the available test suite [9].

Correct patch versus incorrect patch. A correct patch is semantically equivalent to the developer-provided patch, based on a manual examination. An incorrect patch is a patch that is incorrect [10].

Patch ingredient. APR use identifiers or operators to create a template for a concrete patch. For example, variables and method names.

3. Approach

Figure 2 shows the overall process of EffiGenC. The first step is a fault localization. EffiGenC calculates a list of suspicious statements for the buggy project in this step. The next step is to select a fix template. Next, EffiGenC constructs the context of the suspicious statement using the buggy project. The fourth step is patch generation by exploring the context to obtain the patch ingredients and make the template into a concrete patch. The last step is the validation, which runs the preparation of the test suite and obtains the valid patch.

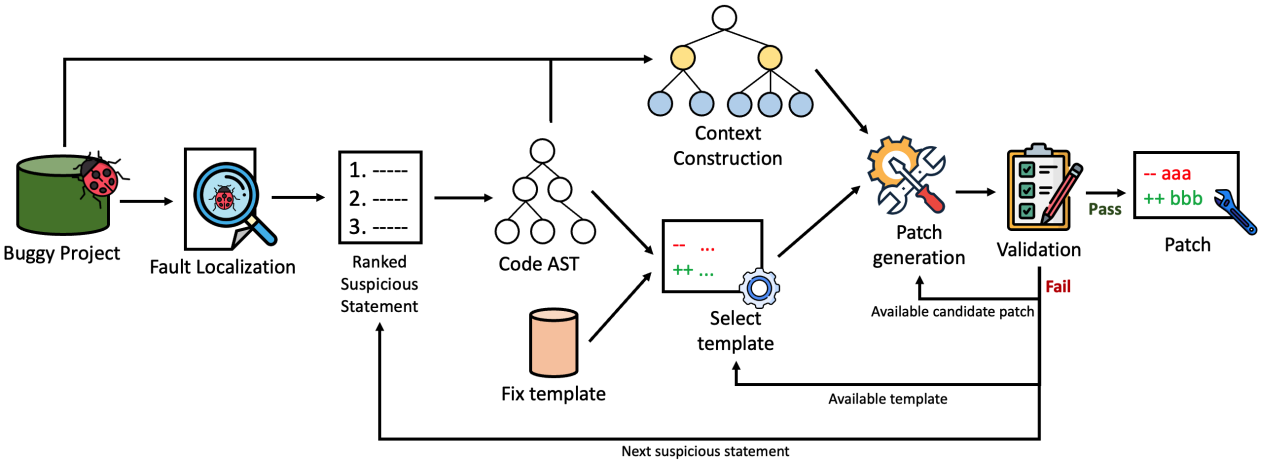

Figure 2. Overall process of EffiGenC.

3.1. Fault Localization

In the fault localization step, EffiGenC derives a ranked list of suspicious statements using test cases for the buggy project. Among the different fault localization techniques, EffiGenC then uses the spectrum-based fault localization technique *Ochiai* [11]. APR studies [3,7,9,12] used *Ochiai* to calculate suspicious statements.

3.2. Select Fix template

In this step, EffiGenC selects the fix template for patch generation. EffiGenC uses 15 templates introduced in the existing studies on template-based APR [3]. EffiGenC selects a template by exploring the AST of the suspicious statement. EffiGenC identifies the node type for each AST node. It selects an available template based on whether it matches the node type of the template. In addition, it selects templates for all nodes belonging to the AST of suspicious statement. During the patch generation, EffiGenC generates a candidate patch by applying a template from the root node.

3.3. Context Construction

EffiGenC constructs the context for the suspicious statement from the buggy project. Among the existing APR techniques, there is a technique [13] that identifies the part to be fixed together using reaching definition. Reaching definition is one of data-flow analysis. It can statically determine which definitions may reach a given point in the source code. EffiGenC extends this concept to collect ingredients related to suspicious statements.

- **Definition 1** (Context Element). Context Element(e) is an identifier that appears in the target statement(S_T) for which context information is to be obtained.
- **Definition 2** (Extended Reaching definition). If statement(S) contains e and exists in the same File(F) with S_T, then $REACH_+(S, e) = true$.
- **Definition 3** (Related Statement). Related statement(S_R) is a statement that satisfy $REACH_+(S_R, e) = true$, $S_R, S_T \in F$ and $S_R \neq S_T$
- **Definition 4** (Related Method). Related Method(M_R) is a method that a related statement(S_R) appears.

EffiGenC constructs context for the suspicious statement. Algorithm 1 shows the context construction process. EffiGenC extracts the identifier by exploring the AST of the suspicious statement (Line 1). It checks whether the context element appears in the AST of the statements appearing in the buggy file to which the suspicious statement belongs (Line 4–7). If appears, the corresponding statement is added to the list (Line 7–10). Based on the configured statement list, EffiGenC collects the method name and parameter information to which the statement belongs (Line 13–19). If the list already has the method information, do not include it to avoid duplicates. Finally, it returns the statement list and method list as the context.

Algorithm 1 Context Construction

Require: S_s: Suspicious statement, F: Buggy File
Ensure: Context

```
1:  ContextElementList = extractIdentifier(S_S)
2:  StatementList = { }
3:  MethodList = { }
4:  for Statement S ∈ F do
5:      for Element e ∈ ContextElementList do
6:          ElementList = extractIdentifier(S)
7:          if e ∈ ElementList then
8:              StatementList.add(S)
9:              Break
10:         end if
11:     end for
12: end for
13: for Statement S ∈ StatementList do
14:     M = getMethodInfo(S)
15:     if MethodList.contain(M) then
16:         Continue
17:     end if
18:     MethodList.add(M)
19: end for
20: Context = (StatementList, MethodList)
21: return Context
```

Figure 3 is an example of constructing context. Figure 3a shows the developer patch of closure-10. And Figure 3b is an example of constructing context for the 1417th line where the patch is applied. In this example, context element is *allResultsMatch*, *n* and *MAY_BE_STRING_PREDICATE*. Based on these, EffiGenC compute the related statement in the file and present the four statements in the example. Also, EffiGenC computes related methods.

statements #4 is an assignment of the global variable, so the example shows only three methods list. EffiGenC generates a patch using a total of seven lists including the statement and method as context.

Closure 10

src/com/google/javascript/jscomp/NodeUtil.java

```
@@ -1414,7 +1414,7 @@ static boolean mayBeString(Node n) {
1414 1414
1415 1415   static boolean mayBeString(Node n, boolean recurse) {
1416 1416     if (recurse) {
1417      -       return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
     1417 +       return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
1418 1418     } else {
1419 1419       return mayBeStringHelper(n);
1420 1420     }
```

(**a**)

Target Statement	return allResultsMatch(n, MAY_BE_STRING_PREDICATE)
Context Element	{**allResultsMatch**, **n**, **MAY_BE_STRING_PREDICATE**}
[Related Statement List]	
#1. return **allResultsMatch** (**n**, BOOLEAN_RESULT_PREDICATE) • Variable = {n, BOOLEAN_RESULT_PREDICATE} • Method = {allResultsMatch} • Expression = ∅	
#2. Node parent = **n**.getParent() • Variable = {parent, n} • Method = {getParent} • Expression = {n.getParent}	
#3. return anyResultsMatch(**n**.getLastChild(), p) • Variable = {n, p} • Method = {anyResultsMatch, getLastChild} • Expression = {n.getLastChild}	
#4. **MAY_BE_STRING_PREDICATE** = new MayBeStringResultPredicate() • Variable = {MAY_BE_STRING_PREDICATE} • Method = {anyResultsMatch, getLastChild} • Expression = {n.getLastChild}	
[Related Method]	
#1: isBooleanResult(Node n) • Variable = {n} • Method = {isBooleanResult} • Expression = ∅	
#2: getAssignedValue(Node n) • Variable = {n} • Method = {getAssignedValue} • Expression = ∅	
#3: anyResultsMatch(Node n, Predicate<Node> p) • Variable = {n, p} • Method = {anyResultsMatch} • Expression = ∅	
#4: GLOBAL_VARIABLE • Variable = Method = Expression = ∅	
[Search Space]	
Variable	{n, parent, p, BOOLEAN_RESULT_PREDICATE, MAY_BE_STRING_PREDICATE}
Method	{allResultsMatch, getParent, anyResultsMatch, getLastChild, isBooleanResult, getAssignedValue}
Expression	{n.getParent, n.getFirstChild, n.getLastChild}

(**b**)

Figure 3. Concept of context and searching patch ingredient. (**a**) Developer patch of Closure-10; (**b**) Example of constructing context and extracting patch ingredient.

3.4. Patch Generation

In the patch generation step, EffiGenC generates a patch using a template and context for a suspicious statement. Algorithm 2 shows the patch generation process. EffiGenC first checks the template requires ingredients (Line 1). Among the types of templates, *MoveStatement* and *MutateDataType*, do not require the ingredient. Therefore, in the case of templates that are not required, it is possible to generate candidate patches only with

suspicious statements and templates (Line 13). If the ingredient is required, initialize the ingredient set of variables, methods, and expressions(Line 2–4). After extracting variables, methods, and expressions from statements belonging to Context, sets V, M, and E are generated, respectively (Line 5–8). EffiGenC searches the AST tree of the statement, checks each node type, and includes it in the set. Ingredient consists of three sets (variable, method, expression), and finally, by inserting ingredients into the template, it goes through the concretization process to make concrete patches (Line 10–11).

Algorithm 2 Patch generation using Context

Require: S_s: Suspicious statement, T: Template, C: Context
Ensure: Candidate Patch
1: **if** isNeedIngredient(T) **then**
2: $V = \{\}$
3: $M = \{\}$
4: $E = \{\}$
5: **for all** Sstatement $S \in C$ **do**
6: V = extractVariable(S)
7: M = extractMethod(S
8: E = extractExpression(S)
9: **end for**
10: Ingredient = (V, M, E)
11: CandidatePatch = Concretization(S_s, T, Ingredient)
12: **else**
13: CandidatePatch = Concretization(S_s, T)
14: **end if**
15: **return** CandidatePatch

Figure 3b shows how to extract variables, methods, and expressions. There are a total of eight lists including statements and methods. For example, in the case of statement #2, *n* and *parent* are variable. We can check the method *getParent*. Finally, *n.getParent*, which is a *MethodInvocation*, is extracted as an expression. In the concretization process, EffiGenC generate a patch using the whole expression rather than splitting it (e.g., update the expression) Combining the list of ingredients for each statement results in a set like the bottom of the example. We can observe *anyResultsMatch* is included in the method list, which is necessary when generating the correct patch.

3.5. Validation

After patch generation, EffiGenC validates the candidate patch by running the test suite. If the candidate patch passes all of the prepared test cases, EffiGenC treats the candidate patch as a valid patch, and the EffiGenC is terminated.

If the candidate patch cannot pass the test cases, EffiGenC discards the patch and validates the next candidate patches. If all candidate patches fail to pass the test suite, EffiGenC generates the patch from the next template. If there is no other template to apply the patch, EffiGenC applies the next rank of the suspicious statement. The EffiGenC is terminated if a valid patch is generated, the program execution time reaches the specified timeout, or the number of generated candidate patches reaches the specified maximum number of candidate patches.

4. Experimental Setup

4.1. Research Question

The following research questions are investigated:

- RQ1. What quality does the proposed context have?
- RQ2. How does EffiGenC perform in terms of efficiency?
- RQ3. How effective is the ingredient search space reduction based on context?
- RQ4. How does EffiGenC perform against state-of-the-art APR techniques?

4.2. Metric: Hit Ratio

We propose a hit ratio metric to evaluate whether our proposed context is of high quality. The hit ratio is a metric that checks whether the ingredient pool contains the ingredients required for the correct patch. Figure 4a shows the developer patch of the Chart-20 bug. To generate the same patch as the developer patch, *outlinePain* and *outlineStroke* variables are required. Figure 4b presents an example of extracting ingredients for a suspicious statement, context, and file. For suspicious statements, the hit ratio is zero because there are no ingredients required for the correct patch. In the case of the context and file, each contains one and two, and thus the hit ratio will be 0.5 and 1, respectively.

Chart-20

source/org/jfree/chart/plot/ValueMarker.java

```
@@ -92,7 +92,7 @@ public ValueMarker(double value, Paint paint, Stroke stroke) {
92 92    */
93 93    public ValueMarker(double value, Paint paint, Stroke stroke,
94 94                       Paint outlinePaint, Stroke outlineStroke, float alpha) {
95      -    super(paint, stroke, paint, stroke, alpha);
   95   +    super(paint, stroke, outlinePaint, outlineStroke, alpha);
96 96        this.value = value;
97 97    }
98 98
```

(**a**)

$$\text{Hit ratio} = \frac{\text{\# of correct ingredients in the group}}{\text{\# of correct ingredients}}$$

The correct ingredients: (**outlinePaint**, **outlineStroke**)
$group_{sus.}$: (paint, stroke, alpha)
$group_{context.}$: (..., **outlinePaint**, ...)
$group_{file}$: (..., **outlinePaint**, **outlineStroke**, ...)

Hit ratio of $group_{sus.}$ = 0
Hit ratio of $group_{context}$ = 0.5
Hit ratio of $group_{file}$ = 1

(**b**)

Figure 4. Example of Calculating Hit ratio. (**a**) Developer patch of Chart-20; (**b**) Formula of hit ratio and example.

4.3. Evaluation Dataset

For the evaluation, we used Defects4j 2.0.0 [14]. Defects4j is a framework that collects real bugs of Java projects and is used for evaluation in many existing studies [3,7,8]. For the same comparison with previous studies, we experimented on 6 projects and 395 bugs among the bugs of Defects4j 2.0.0. Table 1 shows a list of the projects and the number of bugs per project. Column #*Bugs* shows the number of buggy versions in the project.

Column #*Tests* and *LOC* refer to the number of JUnit tests and lines of code available within the latest version of each project.

4.4. Implementation

For this experiment, we implemented EffiGenC on top of TBar. EffiGenC leverages the GZoltar [15] framework to automate the execution of the test cases for each buggy program. We use the *Ochiai* metric to compute the suspiciousness scores of the statements for fault localization. We set the maximum number of candidate patches to 20,000. The timeout is three hours. We run the experiment on Ubuntu 20.04. We use an Intel Core i5-10600 @3.30 GHz CPU and 32 GB of RAM.

Table 1. Details of Defects4j.

Identifier	Name	#Bugs	#Tests	LOC
Chart	JFreeChart	26	2205	96 K
Closure	Closure compiler	133	7927	90 K
Lang	Commons-lang	65	2245	22 K
Math	Commons-math	106	3602	85 K
Mockito	Mockito framework	38	1366	23 K
Time	Joda-Time	27	4130	28 K
Total		**395**	**21,475**	**344 K**

5. Result

5.1. RQ1: Quality of Context

To verify the quality of our proposed context, we evaluated the context for 125 bugs [9] that the existing APR techniques could generate a patch for among the Defects4j bugs. We construct the context based on the statement to which the developer patch is applied. In the same way as the patch ingredient extraction of EffiGenC, we construct an ingredient pool for suspicious statements and files.

Number of Patch ingredient. Table 2 shows the average number of ingredients in each group. $group_{sus.}$ had an average of 3.1 patch ingredients, $group_{context}$ had an average of 38.7, and $group_{file}$ had an average of 176.1 patch ingredients. Taking $group_{file}$ as 100% and calculating the proportions of each group, $group_{sus.}$ was 1.8%, and $group_{context}$ was the only 22%. Figure 5 shows the distribution of patch ingredients in each group for 125 bugs as a box plot. We found that $group_{file}$ had the most patch ingredient, and $group_{context}$ had less distribution overall than $group_{file}$.

Hit Ratio. Table 3 presents the average hit ratio for each group. The average hit ratio of the $group_{sus.}$ was 25.2%, $group_{context}$ was 61.3%, and $group_{file}$ was 73.2%. There was a patch ingredient that it could not find even if it looked at the entire file. Because some patches need identifier belonging to other packages or classes, or it needs new variables. As a result of calculating the hit ratio considering only the case where $group_{file}$ was able to find it, $group_{sus.}$ reached 34.4%, and $group_{context}$ reached 83.7%.

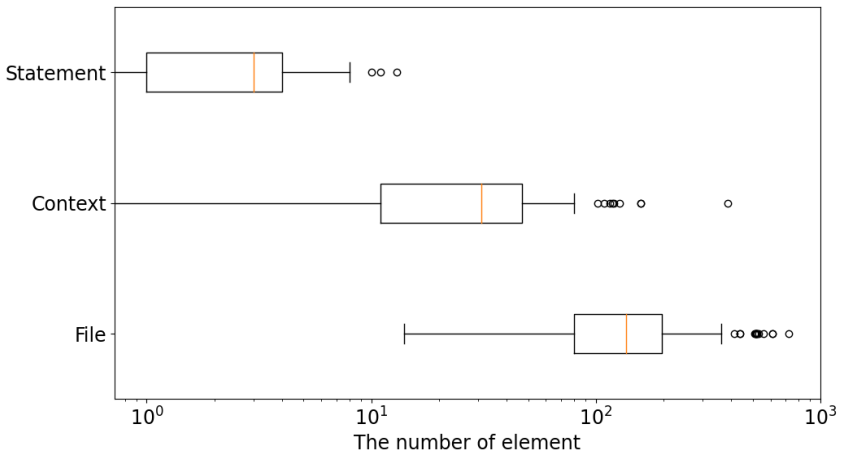

Figure 5. Distribution of element.

Table 2. Number of elements in ingredient pool.

Group Id	$group_{sus.}$	$group_{context}$	$group_{file}$
Avg. Number of Elements	3.1	38.7	176.1
Ratio about $group_{file}$	1.8%	22.0%	100%

Table 3. Result of hit ratio.

Group Id	$group_{sus.}$	$group_{context}$	$group_{file}$
Average Ratio	25.2%	61.3%	73.2%
Ratio about $group_{file}$	34.4%	83.7%	100%

Ratio of Perfect Case. In order to generate a correct patch, APR requires all ingredients for patch generation. We additionally calculated the frequency of perfect cases; the cases that had all the ingredients for the correct patch for each group. Table 4 shows the number of perfect cases for each group and the ratio of each group to $group_{file}$. There were 20, 62, and 77 perfect cases for each group, and when converted to a percentage for $group_{file}$, $group_{sus.}$ was 26%, and $group_{context}$ was 80.5%.

Finding 1. Through the context, even a small ingredient pool can be sufficient to include correct ingredients. Moreover, the perfect case is 80.5%. It can be effective in reducing the patch ingredient search space.

Table 4. Number of Perfect Case.

Group Id	$group_{sus.}$	$group_{context}$	$group_{file}$
Number of Perfect Case	20	62	77
Ratio about $group_{file}$	26.0%	80.5%	100%

5.2. RQ2: Efficiency of EffiGenC

Following the previous study, [9], we compared the efficiency of publicly available template-based APR techniques [3,6,12,16,17]. We use the $Numberofpatchcandidate$ (NPC) as an efficiency metric, in which the existing study presented as an APR efficiency comparison [9]. We calculate the NPC score as the sum of the number of nonsensical patches, in-plausible patches, and valid patches. For the results of the existing technique, we refer to existing studies [9].

Figure 6a shows the NPC score comparison results between EffiGenC and the template-based APR techniques through a boxplot. In this experiment, we computed the number of candidate patches until a valid patch was generated. The number of candidate patches on the x-axis is a log scale. EffiGenC generated lower candidate patches compared to all template-based APR techniques. When we compare the average values, EffiGenC reduced the NPC score from a minimum of 27% to as much as 86%, compared to existing techniques. In addition, we can observe that EffiGenC is effective in most cases because the overall distribution is decreased, not just the mean or average value.

Figure 6b shows the result of comparing the number of nonsensical patches. EffiGenC generates the lowest number of nonsensical patches except for SimFix. When we compare the average value, it reduces the nonsensical patches from at least 53% to 87%. Figure 6c shows the result of the number of in-plausible patches. EffiGenC generates fewer in-plausible patches than kPAR, SimFix, and TBar.

However, it still produces more in-plausible patches than AVTAR and FixMiner. We can see that EffiGenC is implemented based on TBar. EffiGenC can be extended to any other template-based APR.

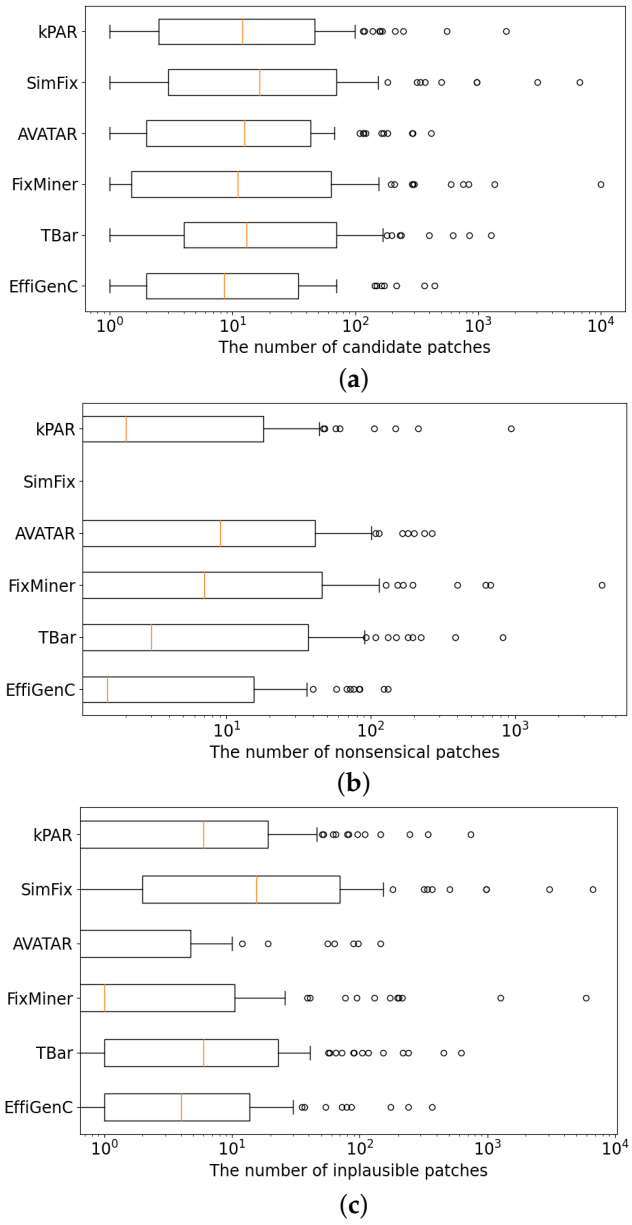

Figure 6. Result of NPC score. (**a**) Distribution of number of candidate patches; (**b**) distribution of number of nonsensical patches; (**c**) distribution of number of in-plausible patches.

Therefore, if we extend the EffiGenC to FixMiner and AVATAR, we can check the search space reduction based on the context. Although SimFix does not generate nonsensical patches, it is less efficient than EffiGenC because all patches it generates are in-plausible.

Finding 2. EffiGenC can generate valid patches with only a small number of computations through the proposed context. The average NPC scores, nonsensical and in-plausible patches are smaller than most template-based APR techniques. Therefore, EffiGenC can increase the efficiency of patch generation through the context.

5.3. RQ3: EffiGenC Space Reduction

To check the search efficiency of EffiGenC, we compare the number of NPCs and correct patches according to the search space. The vanilla version of EffiGenC selects the

related statement and related method in the search space in which the suspicious statement is included. We expand this scope to the package and project.

Table 5 shows the NPC score and number of correct patches according to the search space. The rows present the NPC score results. The last row presents performance results. We manually examine the patches generated by EffiGenC and consider a patch correct if it is semantically the same as the developer patches. When we set the search space to File, EffiGenC generates an average of 37.1 candidate patches. Moreover, it can generate the correct patch for 47 bugs. When we expand the search space to package and project, the number of candidate nonsensical patches and in-plausible increased. In addition, EffiGenC can increase the number of correct patches than existing techniques.

Figure 7 shows the total amount of ingredients that can be extracted from context according to the search space. We calculated the ingredient for bugs that EffiGenC can generate valid patches. We calculated the total amount of ingredients as the sum of the number of variable, method, and expression set elements. The x-axis represents the number of ingredients on the log scale.

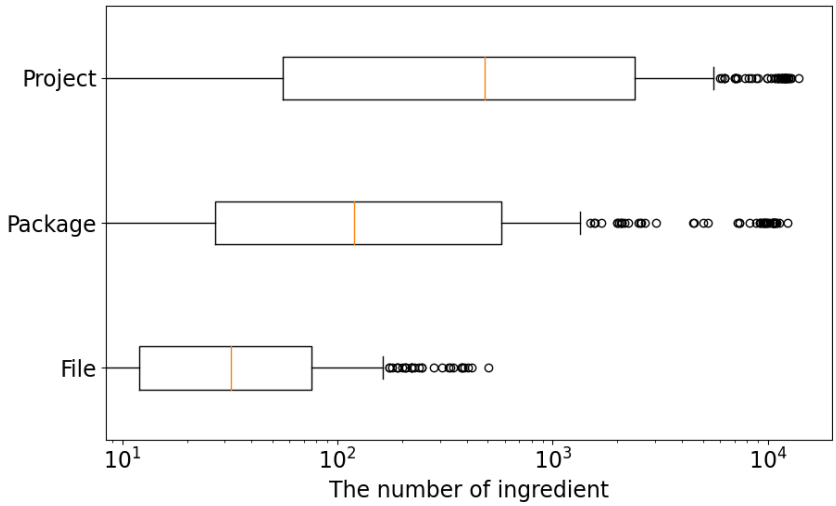

Figure 7. Distribution of ingredient pool.

Table 5. Efficiency and performance changes according to the ingredient pool.

	File	*Package*	*Project*
Avg. # Candidate	37.1	40.7	48
Avg. # Nonsensical	14.7	15.6	21.7
Avg. # in-plausible	21.4	25.1	26.3
# Correct patch	47	48	51

When the only file was targeted, the number of ingredients was the smallest, followed by the package and project. The average number of ingredients was 60.2 for the file, 1210.1 for the package, and 2189.7 for the project. We confirmed that the most optimized version is the vanilla version when considering the search space, efficiency and performance. Also, when the search space is increased to package and project, even if the ingredient increases, EffiGenC can generate a small number of candidate patches through efficient exploration.

Finding 3. Although EffiGenC has expanded the scope of collecting patch ingredients to package and project, we can observe that it explores efficiently through context compared to the growing search space. Moreover, we suggest the best search space as the file when both computation cost and performance are considered.

5.4. RQ4: Comparison with the State-of-the-Art

In this section, we investigate the overall performance of our default EffiGenC. We manually validate the correct patch, such as RQ2. For the results of the existing technique, the results of the previous study [3,9] were referred to.

Table 6 shows the number of correct and plausible patches each technique generated for the Defects4j project. In the table, the first number of result denotes the number of correct patches generated by the technique, and second number denotes the number of plausible patches. EffiGenC generated 47 correct patches, which was the largest number of correct patches.

Table 6. Number of Defects4j bugs that are correctly/plausibly fixed by APR tools. "C, CL, L, M, Moc, T" represent Chart, Closure, Lang, Math, Mockito and Time.

Technique	C	CL	L	M	Moc	T	Total
kPAR	3/10	5/9	1/8	7/18	1/2	1/2	18/49
SimFix	4/8	6/8	**9**/13	14/26	0/0	1/1	34/56
AVATAR	5/12	8/12	5/11	6/13	**2**/2	1/3	27/53
FixMiner	5/8	5/5	2/3	12/14	0/0	1/1	25/31
TBar	**9**/14	8/12	5/14	19/36	1/2	1/3	43/81
EffiGenC	**9**/13	**9**/14	5/8	**21**/31	1/2	**2**/4	**47**/72

Figure 1b is a case in which the existing APR techniques cannot generate correct patches, but EffiGenC succeeds. In the case of the bug, $fnType$ is the context element, and EffiGenC computes the related statements based on this. Of the many ingredients in the file, only the necessary ingredients including $fnType.hasInstanceType()$ were extracted, so EffiGenC can generaete the correct patch.

> **Finding 4.** EffiGenC was able to efficiently explore patch ingredient search space, and generate the correct patches for bugs that the existing template-based APR techniques failed to generate.

6. Threats to Validity

Benchmark overfitting patch. The validity can be threatened by the benchmarks used in the evaluation. Although Defects4j is a high-quality Java project bug framework, there is a threat in which the patches generated by each APR only overfit that bug [18], and there is a risk because the framework does not cover all bug types. However, many APR studies have evaluated the performance of patch generation using benchmarks [14,19,20].

Additional computing cost. EffiGenC can efficiently generate patches by reducing the patch ingredient search space. We also show this through the NPC score. The computational cost of constructing the context and collecting ingredients from such context does not appear in NPC score. However, as a result of running TBar and EffiGenC in the same environment, it took an average of 661 s for TBar and 580 s for EffiGenC to generate the correct patch. Therefore, we can observe that the context construction and patch generation process of EffiGenC are sufficiently efficient.

Scalability. For the experiment, we implemented EffiGenC on TBar. Therefore, it can be observed that the patch ingredient search space construction method of EffiGenC is limited to TBar. The context construction of EffiGenC is a method that can be applied to any technique that uses suspicious statements regardless of TBar. In addition, if it is template-based APR, the concretization process that inserts ingredients to make the template a candidate patch is a common process. Therefore, the process of extracting the patch ingredient of EffiGenC can also be sufficiently generalized. EffiGenC can efficiently generate patches regardless of the technique.

7. Related Work

Research related to APR has been actively conducted [1]. APR is largely divided into search-based APR and semantic-driven APR. A search-based APR generates a candidate patch by defining and exploring a space in which a candidate patch exists. GenProg [21] generates a candidate patch by manipulating the existing buggy source code using genetic programming. By contrast, ARJA [10] generates candidate patches for Java programs using multi-objective genetic programming. Unlike genetic programming, which uses stochastic elements, EffiGenC generates candidate patches based on templates collected from previous patch history.

Semantic-driven APR is a technique for generating correct patches using semantic information such as a symbolic execution or the satisfiability modulo theory. SemFix [22] generates a correct patch using symbolic execution, constraint solving, and program synthesis. Angelix [23] generates a patch by introducing the concepts of an angelic path and an angelic forest. Furthermore, Angelix alleviates the problem of scalability, which is a problem in semantic-driven APR.

There are studies using the patch history to increase the number of correct patches PAR [24] generates a new correct patch for the target project that fails to generate an existing correct patch by creating a template with the pattern found by manually analyzing the patch manually generated patch. Prophet [25] generates a machine-learning model that extracts the correct patch characteristics from a human-written patch in an open-source software repository project. The model was used to prioritize candidate patches and increase the rank of the correct patch. EffiGenC also creates patches by exploring the search space more efficiently through the context, rather than using only the patch history.

As research on search-based APR remains active, empirical analyses of the search space and algorithms have been conducted. Wen et al. [26] revealed that the quality of the search space significantly influences the performance of search-based APR when analyzing the search space explored through existing APR techniques. In addition, the quality of the patch is dependent on test cases. A technique for sampling only good test cases is needed to generate the correct patch with a high performance and high efficiency. Fan Long et al. [25] analyzed the density of plausible and correct patches in a space explored through the APR approach and showed that there are plausible patches other than the correct patch.

Owing to the problematic performance and efficiency of search-based APR, studies using context have continued to efficiently explore the search space. SimFix [22] extracts high-level abstract changes from the past patch histories. Based on this, the correct patch is generated by applying a patch to a suspicious statement. In addition, CapGen [8] proposed a patch-prioritization technique to generate more correct patches with an efficient patch validation. To prioritize the patch, the genealogy, variable, and dependency context scores between the suspicious statement and the past patch history were calculated and prioritized based on this technique. ConFix [7] considered the context by extracting the parent and sibling nodes from the previous patch history. When a suspicious statement identified, ConFix extracts the context and applies only the change in the same context existing in the database to more efficiently generate the correct patch. EffiGenC uses the same context as previous techniques. However, we redefine the context using an extended reaching definition. In addition, EffiGenC effectively reduces the ingredient search space required for patch generation.

8. Conclusions

The existing template-based APR did not sufficiently consider the search space for patch ingredients. We presented the concept of a context based on the extended reaching definition to contain patch information for the target statement. We proposed EffiGenC, which generates patches based on the proposed context. The proposed context contained enough patch ingredients to generate the correct patch. Experiments with Defects4j showed that EffiGenC produced fewer candidate patches. We can see that EffiGenC can explore efficiently a large patch ingredient search space. For future work, we plan to use context to

solve the search space problem that exists in the multiline bug, and study techniques for generating patches for more complex bugs. We are currently focusing only on the identifier as an ingredient. We will conduct research that can reduce the search space for identifiers and change actions. In addition, there is an issue about performance deterioration due to out-of-vocabulary in deep learning-based patch generation. We try to solve this issue by collecting identifiers related to bugs from other projects through the proposed context.

Author Contributions: Conceptualization, J.H.; Data curation, J.H.; Formal analysis, J.H. and H.J.; Funding acquisition, E.L.; Investigation, J.H. and H.J.; Methodology, J.H. and E.L.; Project administration, J.H.; Resources, J.H. and H.J.; Software, J.H.; Supervision, J.H. and E.L.; Validation, J.H. and H.J; Visualization, J.H.; Writing—original draft, J.H.; Writing—review & editing, J.H., H.J. and E.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1A2C2006411).

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gazzola, L.; Micucci, D.; Mariani, L. Automatic software repair: A survey. *IEEE Trans. Softw. Eng.* **2017**, *45*, 34–67. [CrossRef]
2. Kim, M.; Kim, Y.; Heo, J.; Jeong, H.; Kim, S.; Chung, H.; Lee, E. An Empirical Study of Deep Transfer Learning-based Program Repair for Kotlin Projects. *Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* **2022**, *accepted*.
3. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. Tbar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 31–42.
4. Jiang, N.; Lutellier, T.; Tan, L. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In Proceedings of the 43rd International Conference on Software Engineering, Madrid, Spain, 22–30 May 2021; pp. 1161–1173.
5. Kim, M.; Kim, Y.; Kim, K.; Lee, E. Multi-objective Optimization-based Bug-fixing Template Mining for Automated Program Repair. *Int. Conf. Autom. Softw. Eng.* **2022**, *accepted*.
6. Koyuncu, A.; Liu, K.; Bissyandé, T.F.; Kim, D.; Klein, J.; Monperrus, M.; Le Traon, Y. Fixminer: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* **2020**, *25*, 1980–2024. [CrossRef]
7. Kim, J.; Kim, S. Automatic patch generation with context-based change application. *Empir. Softw. Eng.* **2019**, *24*, 4071–4106. [CrossRef]
8. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S. Context-aware patch generation for better automated program repair. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–11.
9. Liu, K.; Wang, S.; Koyuncu, A.; Kim, K.; Bissyandé, T.F.; Kim, D.; Wu, P.; Klein, J.; Mao, X.; Traon, Y.L. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In Proceedings of the 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 615–627.
10. Yuan, Y.; Banzhaf, W. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Trans. Softw. Eng.* **2018**, *46*, 1040–1067. [CrossRef]
11. Meyer, A.D.S.; Garcia, A.A.F.; Souza, A.P.D.; Souza, C.L.D., Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize. *Genet. Mol. Biol.* **2004**, *27*, 83–91. [CrossRef]
12. Jiang, J.; Xiong, Y.; Zhang, H.; Gao, Q.; Chen, X. Shaping program repair space with existing patches and similar code. In Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis, Amsterdam, The Netherlands, 16–21 July 2018; pp. 298–309.
13. Saha, S. Harnessing evolution for multi-hunk program repair. In Proceedings of the 41st International Conference on Software Engineering, Montreal, QC, Canada, 25–31 May 2019; pp. 13–24.
14. Just, R.; Jalali, D.; Ernst, M.D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 437–440.
15. Campos, J.; Riboira, A.; Perez, A.; Abreu, R. Gzoltar: An eclipse plug-in for testing and debugging. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 378–381.
16. Liu, K.; Koyuncu, A.; Bissyandé, T.F.; Kim, D.; Klein, J.; Le Traon, Y. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In Proceedings of the 12th IEEE conference on software testing, validation and verification, Xi'an, China, 22–27 April 2019; pp. 102–113.
17. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering, Hangzhou, China, 24–27 February 2019; pp. 1–12.

18. Smith, E.K.; Barr, E.T.; Le Goues, C.; Brun, Y. 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, Bergamo, Italy, 30 August–4 September 2015; pp. 532–543.
19. Lin, D.; Koppel, J.; Chen, A.; Solar-Lezama, A. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In Proceedings of the Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Vancouver, BC, Canada, 22–27 October 2017; pp. 55–56.
20. Kim, M.; Kim, Y.; Lee, E.; Denchmark: A Bug Benchmark of Deep Learning-related Software. In Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories, Madrid, Spain, 22–30 May 2021; pp. 540–544.
21. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **2011**, *38*, 54–72. [CrossRef]
22. Nguyen, H.D.T.; Qi, D.; Roychoudhury, A.; Chandra, S. Semfix: Program repair via semantic analysis. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 772–781.
23. Mechtaev, S.; Yi, J.; Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In Proceedings of the 38th international conference on software engineering, Austin, TX, USA, 14–22 May 2016; pp. 691–701.
24. Kim, D.; Nam, J.; Song, J.; Kim, S. Automatic patch generation learned from human-written patches. In Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 802–811.
25. Long, F.; Rinard, M. An analysis of the search spaces for generate and validate patch generation systems. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 702–713.
26. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S.C. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv* **2017**, arXiv:1707.05172.