Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming

Thomas Durieux, Benoit Cornu, Lionel Seinturier, Martin Monperrus

Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming

Thomas Durieux, Benoit Cornu, Lionel Seinturier and Martin Monperrus
University of Lille & Inria, France

***Abstract*—Null pointer exceptions (NPE) are the number one cause of uncaught crashing exceptions in production. In this paper, we aim at exploring the search space of possible patches for null pointer exceptions with metaprogramming. Our idea is to transform the program under repair with automated code transformation, so as to obtain a metaprogram. This metaprogram contains automatically injected hooks, that can be activated to emulate a null pointer exception patch. This enables us to perform a fine-grain analysis of the runtime context of null pointer exceptions. We set up an experiment with 16 real null pointer exceptions that have happened in the field. We compare the effectiveness of our metaprogramming approach against simple templates for repairing null pointer exceptions.**

I. Introduction

Null pointer exceptions are the number one cause of uncaught crashing exceptions in production [11]. Li et al. found that 37.2% of all memory errors in Mozilla and Apache are due to null values [11]. Other studies[1] found that up to 70% of errors in Java production applications are due to null pointer exceptions. It is an inherent fragility of software in programming languages where null pointers are allowed, such as C and Java. A single null pointer can make a request or a task to fail, and in the worst case, can crash an entire application. Program repair [16] is a research field concerned with the generation of patches. There are a number of program repair techniques (eg. [10]) but not one of them is dedicated to null pointer exceptions.

One way of fixing null pointer exceptions is to use a template [8]. For instance, one can reuse an existing local variable as follows:

```
+ if (r == null) {
+   r = anotherVar;
+ }
  r.foo(p);
```

This example illustrates a template that depends on the context (called parametrized template in this paper): the reused variable is the template parameter. When repairing a statically typed language, such as Java that we consider in this paper, the static type of the template parameter has to be compatible with the variable responsible for the null pointer exception. This is done with static analysis of the code.

However, there may be other variables in the context of the null pointer exception, typed by a too generic class, which may be compatible. *The static analysis of the context at the location of the null pointer exception may not give a complete picture of the objects and variables that can be used for fixing a null pointer exception.* There is a need to make a dynamic analysis of the repair context. This is the problem we address in this paper.

To overcome this problem, we propose to perform a dynamic analysis of the repair context. This completely changes the way we perform patch generation: instead of applying templates one after the other, we propose to create a metaprogram that is able to dynamically analyze the runtime context, and to dynamically alter the execution of the program under repair to identify a working patch. The resulting technique is called NPEfix. It takes as input a test case that triggers an uncaught null pointer exception and outputs a patch that fixes it if one exists (or several alternative patches if several valid patches exist). NPEfix is composed of three main phases: first, it generates a metaprogram out of the program under repair, second it compiles the metaprogram, third, it runs the failing test case a large number of times, each time altering the behavior to find a way to avoid the null pointer exception.

NPEfix is based on repair strategies for null pointer exceptions. Reusing an existing variable, as seen above, is one such strategy. NPEfix uses a total of 9 different strategies that are categorized in two groups. The first group is about providing an alternative value when a null pointer exception is about to happen. This value can come from elsewhere in the memory (i.e. a valid value that is stored in another variable), or it can be a new object. The second group of strategies is about skipping the execution of the exception-raising expression. It can be either skipping a single statement or skipping a complete method.

In NPEfix, we use a metaprogramming approach – a code transformation – for each of those strategies. All transformations are compatible one with another and can be used in conjunction. Then, we explore the search space of tentative patches for null pointer exceptions by activating the hooks of the metaprogram which have been injected by transformation. Compared to a template based approach, *the novelty of our approach is to explore the search space of tentative patches purely at run-time, which gives us a way to finely analyze the runtime context of the null pointer exception*.

[1] http://bit.ly/2et3t79, last accessed Oct 19 2016

To evaluate our approach, we build a benchmark of 16 null pointer exceptions that happened in the field for open-source software and were all reported in bug trackers. For each of those bugs, we create the NPEfix metaprogram, and explore the search space to identify patches that are able to avoid the null pointer exception to happen.

To sum up, the contributions of this paper are:

- A taxonomy of 9 alternative execution strategies to repair null pointer exceptions (Section II).
- A set of code transformations for implementing those null-pointer repair strategies in Java (Section III).
- The systematic evaluation of our approach on 16 real null pointer exceptions that we were able to reproduce (Section IV). We compare NPEfix against naive template-based repair.
- A publicly available tool and benchmark for facilitating further research on this topic: https://github.com/Spirals-Team/npefix.

This paper is a reworked and extended version of a working paper published on Arxiv [1]. It is structured as follows. Section II presents a taxonomy of repair strategies for null pointer exceptions. Section III describes our metaprogramming approach to apply those repair strategies. Section IV discusses the evaluation of our work. Section V, VI respectively explores the limitations of our approach and the related work.

II. A Taxonomy of Repair Strategies for Null Pointer Exceptions

In this section, we present a taxonomy of run-time repair strategies for null pointer exceptions. It unifies previous work on that topic [4], [6], [14] and proposes new strategies.

When a harmful null pointer exception is going to happen, there are two main ways to avoid it. First, one can replace the null value by a valid object. Second, one can skip the problematic statement. In both cases, no null pointer exception happens. In this paper, we refine those two techniques in 9 different strategies to repair null pointer exceptions. There are grouped along those two families: replacement of the null by an alternative object, and skipping the execution of the statement associated with the null pointer exception.

A. Strategies based on Null Replacement

One way to avoid a null pointer exception to happen is to change the reference into a valid instance. We can provide an existing value (if one can be found) or a new value (if one can be constructed). To facilitate the presentation, *r* is an expression (usually a variable reference). We basically want *r* to reference a valid (non-null) value in order to prevent a null pointer exception when executing r.foo(p). Symbol p is a method parameter.

Table I
NPEfix' repair for null pointer exceptions.

Strategy			Id	Description
Replace	reuse	local	S1a	local reuse of an existing compatible object
		global	S1b	global reuse of an existing compatible object
	creation	local	S2a	local creation of a new object
		global	S2b	global creation of a new object
skipping	line		S3	skip statement
	method	null	S4a	return a null to caller
		creation	S4b	return a new object to caller
		reuse	S4c	return an existing compatible object to caller
			S4d	return to caller (void method)

Reuse (S1b) A first strategy is the case of repairing the null pointer exception with an existing object as follows:

```
+ if (r == null) {
+   r = anotherVar;
+ }
  r.foo(p);
```

Strategy S1b is parameterized by the variable $anotherVar$. The variable is taken from the set S of accessible objects composed of the local variables, the method parameters, the class fields of the current class and all the other static variables. S is further filtered to only select the set of all well typed and non-null values V according to the type of r.

Local Reuse (S1a) A variant of the reuse strategy consists in replacing one null reference by a valid object, without modifying the null variable itself.

```
+ if (r == null) {
+   anotherVar.foo(p);
+ } else {
    r.foo(p);
+ }
```

With local reuse, all the other statements using *r* will still perform their operations on *null*.

Object Creation (S2b) Another strategy consists of creating a new value.

```
+ if (r == null) {
+   r = new Foo();
+ }
  r.foo();
```

Object Creation Local (S2a) A rare possible patch for a null pointer exception consists of providing a disposable object. This is what we call local object creation. This is

interesting if method "foo" changes the state of p based on the method call receiver.

```
+ if (r == null) {
+   new Foo().foo(p);
+ } else {
    r.foo(p);
+ }
```

This sums up in 4 possible strategies for null replacement (see Table I): use an existing value locally (S1a), use an existing value globally (S1b), use a new value locally (S2a) and use a new value globally (S2b).

B. Strategies based on Execution Skipping

We also propose to skip the statement where a null pointer exception would happen. There are different possible ways of skipping.

Line Skipping (S3) First, the straight-forward strategy S3 consists in skipping only the problematic statement and allows to avoid the null pointer exception at this location.

```
+ if (r != null) {
    r.foo(p);
+ }
```

We also propose a family of strategies which consists in skipping the rest of the method. For skipping the rest of the method, several possibilities can be considered. Let us consider that the method returns an object of type "Bar".

Return Null S4a If the method expects a return value, one can return null: this is a reasonable option because it is possible that the caller has a non-null check on the returned object.

```
+ if (r == null) {
+   return null;
+ }
  r.foo(p);
```

Return New Object S4b One can return a new instance of the expected type. As for strategy S2b, this is a strategy parameterized by all possible constructor calls.

```
+ if (r == null) {
+   return new Bar();
+ }
  r.foo(p);
```

Return Variable S4c One can also search in the set of accessible values one which corresponds to the expected return type and return it. As for strategy S1b, this is a strategy parametrized by all possible type-compatible variables.

```
+ if (r == null) {
+   return anotherVar;
+ }
  r.foo(p);
```

Vanille Return S4d When the method does not return anything (`void` return type in Java), inserting a simple "`if (r==null) { return; }`" is a valid option.

All strategies are listed in Table I. The table represents the different dimensions of the analysis: replacement vs skipping, local vs global, reusing objects vs creating new ones. For each strategy, the corresponding code that needs to be injected is shown in the last column.

Algorithm 1 TemplateNPE: Exploration of all tentative patches based on parametrized templates

Input: p: a program
Input: t: a test case reproducing a NPE
Input: e: expression that triggers the NPE
Input: S: a set of repair strategies
Output: P: set of tentative patches
Output: Q: set of valid patches

```
1:  compile p
2:  for s in S do
3:    A ← possible parameter value for s in e
4:    for a : A do
5:      x ← apply s on p parametrized by a
6:      recompile x
7:      if x compiles then
8:        add x to P
9:      end if
10:     run t against x
11:     if t succeeds then
12:       add x to Q
13:     end if
14:   end for
15: end for
```

C. Novelty

Among those 9 strategies, some of them have already been explored. Dobolyi et al. [4] have proposed two of them: S2b and S3. Kent [6] have defined S2b, S3, S4a and S4d. Long et al. [14] have explored S3, S4b and S4d. This means that $4/9$ strategies presented in this paper are new: S1a, S1b, S2a, S4c. The novelty lies on the idea of reusing existing objects (S1a, S1b, S4c) and performing local injection (S1a, S2a).

D. A Naive Implementation for NPE Repair

To find patches according to the strategies presented in Section II, a naive implementation consists of exploring all possible strategies one by one. We call this naive implementation TemplateNPE, whose main algorithm is shown in Algorithm 1. The idea of this algorithm is to explore one strategy after each strategy, using source code transformation techniques, and to test whether they repair the null pointer exception under consideration. If the application of a template compiles, we have a *"tentative patch"*. Hence, this algorithm explores the search space of all tentative patches. One sees that this naive implementation requires recompiling one file for each pair $(parameter, strategy)$.

All patch templates are parametrized. The first template parameter contains the Java expression that may be null, it is located in the condition of the template

as follows: `if (<parameter> == null) ...` . The second template parameter refers to the expression that is used to replace the null expression. This expression can be a variable (S1a, S2b), a new instance or a predefined constant (null, 0, 1, "", ' ') for S2a, S2b. This template parameter is not present in S3, S4a and S3). The replacement expressions are statically created based on the static analysis of the context of the line that produces the null dereference. We use code analysis to perform which variables are accessible at the expression causing the null pointer exception, and also to list constructor calls for building new instances.

III. Metaprogramming for Null Pointer Repair

We now present a set of code transformations to embed the 9 strategies presented in Section II in a metaprogram.

In this paper, a metaprogram is a program enriched with behavior modification hooks. By default, all behavior modification hooks are deactivated, which means that by default a metaprogram is semantically equivalent to the original program.

Let us consider the program `x = y + z`. A metaprogram is for instance `x = y - z if HOOK_IS_ACTIVATED else y + z` (functional style) or `x = HOOK_IS_ACTIVATED ? y - z : y + z` (ternary expression, e.g. in Java). Variable HOOK_IS_ACTIVATED is a Boolean global variable which controls the activation of the behavior modification. This metaprogram enables one to transform at run-time an addition into a subtraction. In our context, the metaprogram is automatically created using source-to-source transformation.

The metaprogramming code transformations are realized in a tool for Java called NPEfix. NPEfix is composed of three main phases: first, it generates a metaprogram out of the program under repair, second it compiles the metaprogram, third, it runs the failing test case a large number of times, each time altering the behavior to find a way to avoid the null pointer exception and thus emulate a patch.

A. Core Intercession Hook

To modify the behavior when a null pointer exception happens, we encapsulate each method call and field access as shown in Listing 1.

The call of `doSomething` that is originally present is now made on the result of method `checkForNull`. Method `checkForNull` does the following things. It first assesses whether the object is null, i.e. whether a null pointer exception will occur; if it's not null, the program proceeds with its normal execution. If the object is null, it means a null pointer exception is about to be thrown, and then a strategy is applied. For sake of simplification, this is shown as a switch as in Listing 1, this switch case is the core intercession hook of the metaprogram.

```
//before transformation
Foo o;
o.doSomething();

// after NPEfix transformation
checkForNull(o, Foo.class).doSomething();

// with static method checkForNull
Object checkForNull(Object o, Class c){
  if (o == null) // null pointer exception detected
    switch (STRATEGY) {
      case s1b: return getVar(currentMethod());
      case s2b: return createObject(c);
      ...
    }
  return o;
}
```

Listing 1. Detecting harmful null pointer exceptions With Code Transformation

```
//before transformation
public void method(){
    ...
    Object a = {defaultExpression};
    a = {newValue};
    ...
}

// after NPEfix transformation
public void method(){
    collectField(myField, "myField");
    ...
    Object a = initVar({defaultExpression}, "a");
    a = modifyVar({newValue}, "a");
    ...
}
```

Listing 2. Maintaining a set of variables as pool for replacement at run-time

B. Value Replacement Strategies

There are four strategies based on value replacement (the first half of Table I): S1a, S1b, S2a and S2b.

1) Reuse Variable: For replacing a null value with a variable, the challenge is to maintain a set of variables as a pool for replacement at run-time. Listing 2 shows how we tackle this problem; we use a stack to store all the variables of each method. Each variable initialization and assignment inside the method is registered thanks to the NPEfix' method `initVar`. In addition, at the beginning of each method, we collect all the accessible fields and parameters.

2) Create New Object: Now, let us consider that the strategies that create a new variable (strategies S2a and S2b).

As shown in Listing 1, a call is made to `createObject` that takes as parameter the static type of the dereferenced variable. `createObject` uses reflection to access to all the constructors of the given type. In addition, this method is recursive so as to create complex objects if needed. It tries to create a new instance of the class from each available constructor. Given a constructor, it attempts to create a new instance for each of the

```
// before transformation
value.dereference();

// after NPEfix transformation
if (skipLine(value)){
    value.dereference();
}
boolean skipLine(Object... objs){ // NPEfix framework
    for (Object o : objs) {
        if (o == null && cannotCatchNPE() && doSkip())
            return false
    }
    return true;
}
```

Listing 3. Implementation of Line-based Skipping

parameter recursively. The stopping condition is when a constructor does not need parameters. Note that the primitive types, which don't have constructors, are also handled with default literal values.

C. Skipping Strategies

Now we present how we implement the strategies based on skipping the execution (the second half of Table I).

1) Line skipping: The strategy S3 necessitates to know if a null pointer exception will happen in a line, before the execution of the line. For this, the transformation presented in Listing 1 is not sufficient, because the call to method `checkForNull` implies that the execution of the line has already started. To overcome this issue, we employ an additional transformation presented in Listing 3.

Similarly to `checkForNull`, method `skipLine` assesses, before the line execution, whether the dereferenced value is null or not, and whether it is harmful. Method skipline takes an arbitrary number of objects, the ones that are dereferenced in the statement. This list is extracted statically.

2) Method skipping: The remaining strategies are based on skipping the execution of the rest of the method when a harmful dereference is about to happen: these are strategies S4d, S4a, S4c and S4b (the last part of Table I). We implement those strategies with a code transformation as follows.

A try-catch block is added in all methods, wrapping the complete method body. This try-catch blocks handle a particular type of exception defined in our framework (ForceReturnError). This exception is thrown by the `skipLine` method when one of the method-skipping strategies is activated, as show in Listing 4. This listing also shows a minimalist example of the code resulting from this transformation.

D. Exploration of the Patch Search Space at Runtime

Now that we have a metaprogram that embeds all strategies, we also have a way to explore the search space of null pointer repair purely at run-time. The idea is to first create the metaprogram, then to activate the strategies dynamically by setting the appropriate behavior modification hooks.

```
// before transformation
Object method(){
  ...
  value.dereference();
  ...
  return X;
}
// after NPEfix transformation
Object method(){
  try {
    ...
    if (skipLine(value)){
      value.dereference();
    }
    ...
    return X;
  } catch (ForceReturnError f){
    if (s4a) return null;
    if (s4b) return getVar(Object.class);
    if (s4c) return createObject(Object.class);
  }
}
boolean skipLine(Object... objs){
  if(hasNull(objs) && cannotCatchNPE() && skipMethodActivated()
      )
    throw new ForceReturnError();
  ...
}
```

Listing 4. Metaprogramming for method-based skipping strategies

Algorithm 2 The Exploration Algorithm of NPEfix

Input: p: a program
Input: t: a test case reproducing a NPE
Input: S: a set of repair strategies
Output: P: set of tentative patches
Output: Q: set of valid patches

1: $M_p \leftarrow$ create meta-program of p
2: compile M_p
3: execute t against M_p until null dereference
4: D $\leftarrow$ collects all possible metaprogramming patches based on runtime analysis
5: terminate execution
6: **for** d in D **do**
7: activate d in metaprogram
8: execute t against M_p
9: p$\leftarrow$ patch corresponding to d
10: add p to P
11: **if** t succeeds **then**
12: add p to Q
13: **end if**
14: **end for**

From Hooks to Patches Given a combination of behavioral modification hooks, one can create the corresponding source code patch by reinterpreting the hooks according to the templates presented in Section II.

E. Implementation

All those transformations have been implemented in a tool called NPEfix, which has been made publicly available for sake of reproducible research and open science[2]. The transformations use the Spoon library [18].

IV. Evaluation

We now evaluate NPEfix. We design a protocol to answer the following questions.

- RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches?
- RQ2. Does NPEfix (metaprogramming) produce more valid patches?
- RQ3. What are the reasons explaining the presence of different valid patches?
- RQ4. Is the performance of NPEfix acceptable?

A. Protocol

In order to evaluate our approach of patch generation based on metaprogramming, we build a benchmark of real and reproducible null pointer exceptions in Java programs (see Section IV-C). Then we compare the ability of TemplateNPE and NPEfix to find different patches and repair each bug of the benchmark. TemplateNPE is the template-based implementation of our nine repair strategies that is described in Section II-D. NPEfix uses our metaprogramming approach to fix null pointer exceptions as described in Section III.

In this paper, we study "test-suite adequate" patches [15] We consider a patch as test-suite adequate when the failing test case that reproduces the null dereference passes as follows: 1) no exception (a null pointer exception or another one) is uncaught and crashes the test case and 2) whether the assertions that come at the end of the test case reproducing the null pointer exception pass. For sake of readability, we refer to test-suite adequate patches as "valid" patches (hence we use test-suite adequate and "valid" interchangeably in this paper)

Note that test-suite adequate patches mare be conceptually incorrect, even if they pass the test suite. Since we use as only oracle the test cases, which can miss correctness assertions, a valid patch can be considered incorrect beyond the test-suite satisfaction correctness criterion [12], [21].

B. Evaluation Metrics

After each bug and each repair technique, we collect three metrics: the number of tentative patches (whether valid or not); the number of valid patches (that pass the test case); the execution time required to explore the search space of patches.

[2]https://github.com/Spirals-Team/npefix

Table II
Descriptive statistics of the subjects suffering from null pointer exceptions. Size measured by cloc version 1.60.

Subject	Domain	Size
COLL	Collection library	21,594 LOC
LANG	Utility functions	18,970 LOC
MATH	Math library	90,771 LOC
PDF	PDFBox library	64,375 LOC
Felix	Felix library	33,057 LOC
SLING	Sling library	583 LOC
Total	16 bugs	229,350 LOC

We interpret those metrics as follows. A larger number of tentative patches means that the search space is richer. As shown in previous work [15], [12], there are often multiple different patches that are able to make a failing test case passing. A larger number of generated valid patches is better, it means that the developer is given more choices to get a really good choice.

The results of this experimentation, incl. all tentative patches are publicly available at https://github.com/Spirals-Team/npefix-experiments

C. Benchmark

To build a benchmark of real null pointer exceptions in Java, we consider two inclusion criteria. First, the bug must be a real bug reported on a publicly available forum (e.g. a bug tracker). Second, the bug must be reproducible. This point is very challenging since it is really difficult to reproduce field failures, due to the absence of the exact crashing input, or the exact configuration information (versions of dependencies, execution environment, etc.). As a rule of thumb, it takes one day to find and reproduce a single null pointer exception bug. We consider bugs in the Apache Commons set of libraries (e.g. Apache Commons Lang) because they are well-known, vastly used and their bug trackers are public, easy to access and to be searched. Also, thanks to the strong software engineering discipline of the Apache foundation, a failing test case is often provided in the bug report. We have not rejected a single reproducible field null pointer exception.

As a result, we have a benchmark that contains 16 null pointer exceptions (1 from collections, 3 from lang and 7 from math, 3 from PDFBox, 1 from Felix, 1 from Sling). It is publicly available for future research (https://github.com/Spirals-Team/npe-dataset). The main strength of this benchmark is that it only contains real null pointer exception bugs and no artificial or toy bugs. Table II shows the size of applications for which we have real field failures.

Table III
COMPARISON OF THE THE TEMPLATE APPROACH (TEMPLATENPE) AND THE META-PROGRAMMING APPROACH (NPEFIX) ON THREE KEY METRICS.

	# Tentative Patches		# Valid Patches		Execution Time	
Bug ID	Template	NPEfix	Template	NPEfix	Template	NPEfix
collections360	7	**10**	0	0	0:00:20	0:02:32
felix-4960	9	7	4	4	0:00:43	0:03:22
lang304	44	**77**	43	**65**	0:00:11	0:00:26
lang587	21	**28**	12	**28**	0:00:23	0:00:23
lang703	16	15	0	**7**	0:00:09	0:00:20
math1115	8	**11**	6	5	0:00:47	0:02:08
math1117	8	**11**	0	0	0:00:41	0:01:52
math290	9	**10**	3	**4**	0:00:18	0:00:42
math305	2	**4**	1	**3**	0:00:08	0:00:40
math369	15	**16**	14	14	0:00:26	0:00:43
math988a	17	17	11	11	0:01:02	0:01:38
math988b	17	**25**	17	17	0:01:39	0:01:48
pdfbox-2812	6	**14**	2	2	0:00:28	0:01:46
pdfbox-2965	4	3	4	3	0:00:16	0:01:34
pdfbox-2995	4	**5**	3	1	0:00:13	0:01:35
sling-4982	18	**20**	7	**11**	0:00:05	0:00:06
Total	205	273	127	175	0:07:57	0:21:44
Average	12.81	17.06	7.94	10.94	0:00:29	0:01:21
Median	9.00	12.50	4.00	4.50	0:00:22	0:01:35

D. RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches?

Table III presents the results of our experiment. The first column contains the bug identifier. The second column contains the number of tentative patches for each bug (ie. the size of the search space). This column is composed of two sub-columns: the number of tentative patches using the template-based approach (TemplateNPE) and the number of patches generated by the metaprogramming approach (NPEfix). For example, TemplateNPE identifies 7 tentative patches and NPEfix identifies 10 tentative patches. The remaining top-level columns will discussed below in Section IV-E and Section IV-G.

In 12/16 (in bold) of the case NPEfix explores more tentative patches than TemplateNPE. This validates that the static analysis and dynamic analysis of the repair context differs, and that the latter is potentially richer. The difference in the number of tentative patches between TemplateNPE and NPEfix is explained as follows.

- During execution, more objects are detected as compatible with the type of the null expression. With the template approach, we do not know the actual runtime type of all variables.
- The number of different new objects created varies because NPEfix detects at runtime more compatible constructors.
- Some strategies cannot be applied at certain locations with the template approach. For example, TemplateNPE cannot apply the skip-line strategy (S3) on a local variable. This case is naturally handled in the metaprogramming approach.
- In 3/16 cases, the template-based approaches identifies more tentative patches. The reason is that NPEfix filters out equivalent patches by on the runtime variable value. This is further discussed in Section IV-F2 and Section IV-F3.

RQ1. What is the impact of NPEfix' runtime analysis of the repair context and TemplateNPE's static analysis of the repair context on the number of explored tentative patches? NPEfix explores 273 tentative patches, which is 68 more than the template-based approach. In other words, the search space of the metaprogramming technique is larger. This validates our intuition that the runtime analysis of the repair context is valuable in certain cases.

E. RQ2. Does NPEfix (metaprogramming) produce more valid patches?

Table III also gives the number of tentative patches that are valid, ie. that make the failing test case passing. This is is shown in the 4th and 5th columns under the top-level header "# Valid Patches" presents the results of this experiment. For example, for bug Collection-360, neither TemplateNPE nor NPEfix identifies a valid patch.

For lang304, TemplateNPE identifies 43 valid patches, while NPEfix finds 65 valid patches. As we can see there is a correlation between the size of the explored search space and number of valid patches identified. This means that it is worth exploring more tentative patches to identify more valid patches.

RQ2. Does NPEfix (metaprogramming) produce more valid patches? NPEfix finds 175 patches that avoid the null pointer exception and make the test case passing. Among those valid patches, 48 patches of them are uniquely found by the metaprogramming approach thanks to the runtime analysis of the repair context. Those patches are only test-suite adequate, and if the test suite is weak, they may be incorrect.

F. RQ3. What are the reasons explaining the presence of different valid patches?

We answer to this research question with 3 case studies.

1) Math-305: We now discuss the patches found for bug Math-306, where the null pointer exception is thrown during the computation of a clustering algorithm called Kmeans. The null pointer exception is triggered when a point is added to the nearest cluster. When the library first computes the nearest cluster, it fails because the current point is at a distance largest than Integer.MAXVALUE. Then, the nearest cluster is set to null and a null pointer exception is thrown when the library tries to add the current point to it.

NPEfix succeeds to identify 4 different patches for this bug. They are all presented in Figure 1. The top three first patches (in green) of the figure are test-suite adequate: they all avoid the null pointer exception to be thrown. As we see, they have the same behaviour: they skip the line that produces the null pointer exception. The first and the third patches create a new cluster that is never used in the application: this is useless but it works. The second patch skips the line that produces the null pointer exception, resulting in the point not being added to the cluster. The last patch is invalid because it produces a division per zero later in the execution: the test case still fails (not with the original null pointer exception but with a division-by-zero exception).

This case study illustrates the fact many different valid patches indeed exist. However, while they are all equivalent according to the test case oracle, they are not equivalent for the developers: a developer would likely discard a patch that creates a temporary object only to avoid a null pointer exception.

2) Felix-4960: Felix is an implementation of the OSGI component model. Real bug Felix-4960 is about a null pointer exception that is thrown in method "getResourcesLocal(name)" which is dedicated to searching a resource in a path. The null pointer exception appears when Felix does not succeed to get a list of resources in the path. In this case Felix tries to iterate on a list that is null, which triggers the null pointer exception.

```
--- org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
@@ -90,3 +90,7 @@
            Cluster<T> cluster = getNearestCluster(
    clusters, p);
-           cluster.addPoint(p);
+           if (cluster == null) {
+                   new Cluster(null).addPoint(p);
+           } else {
+                   cluster.addPoint(p);
+           }
        }
```

```
--- org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
@@ -90,2 +90,5 @@
            Cluster<T> cluster = getNearestCluster(
    clusters, p);
+           if (cluster == null) {
+                   cluster = new Cluster(null);
+           }
            cluster.addPoint(p);
```

```
--- org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
@@ -90,3 +90,5 @@
            Cluster<T> cluster = getNearestCluster(
    clusters, p);
-           cluster.addPoint(p);
+           if (cluster != null) {
+                   cluster.addPoint(p);
+           }
        }
```

```
--- org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
+++ org/apache/commons/math/stat/clustering/
    KMeansPlusPlusClusterer.java
@@ -90,2 +90,5 @@
            Cluster<T> cluster = getNearestCluster(
    clusters, p);
+           if (cluster == null) {
+                   return null;
+           }
            cluster.addPoint(p);
```

Figure 1. The generated patches for the bug Math-305, the patches that have a green border are valid patches and the patch that has a red border is the invalid patch.

For Felix-4960, TemplateNPE is surprisingly able to generate more tentative patches: NPEfix generates 7 tentative patches and TemplateNPE generates 9 tentative patches that compile. The reason is that template generates tentative patches based on reusing variables, however those patches are meaning less. TemplateNPE fills the template parameters with `m_activationIncludes` and `m_declaredNativeLibs`. However, those variables are null because not initialized at this location in the code. In other words, TemplateNPE generates a patch

```
--- pdfbox/pdmodel/interactive/form/PDAcroForm.java
+++ pdfbox/pdmodel/interactive/form/PDAcroForm.java
@@ -250,2 +250,5 @@

+    if (fields == null) {
+        return retval; // reval is null
+    }
     for (int i = 0; i < fields.size() && retval == null; i
      ++)
```

Listing 5. The additional patch of TemplateNPE for the bug PDFBbox-2965

that replaces a null by a null, which obviously results in the same null pointer exception as before.

On the contrary, NPEfix works at runtime, and hence knows that the actual value of `m_activationIncludes` and `m_declaredNativeLibs` are null. Hence, it does not even tries them, because it knows in advance that such a tentative patch would be invalid. In this case, the runtime analysis of the context is interesting to discard incorrect patches early in the process.

3) PDFBbox-2965: PDFBbox is a PDF rendering library. This library allows one to read the properties of a PDF file. PDFBbox-2965 happens when PDFBox searches for a specific field on a PDF that contains no PDF form. Internally PDFBox iterates on all form fields of the PDF and compare the name of the field against the searched field. But when the PDF contains no form, the list of fields is null, and a null pointer exception is thus triggered.

NPEfix generates 3 tentative patches and TemplateNPE generates tentative 4 patches for this bug. They are all valid according to the test case. As we see, TemplateNPE generates one additional valid patches. This patch is shown in Listing 5. It consists of returning variable `retval` if the `fields` variable is null. This patch is of low-quality because `retval` is always null in this case. In other words, a simple `return null;` is a more explicit patch, and would better help the developer. This explains why NPEfix does not generate this patch, as for Felix-4960, thanks to runtime analysis, NPEfix knows that `retval` is null, and that `return retval;` is equivalent to `return null;` (strategy S4a).

G. RQ4. Is the performance of NPEfix acceptable?

The last column of Table III presents the execution time required to explore the whole search space of tentative patches for fixing the null pointer exception. For example, for the bug Collections-360, TemplateNPE explores the search space in 20 seconds while NPEfix requires 2 minutes 32 seconds.

In most cases, the template based approach is faster. The reason is that creating the metaprogram takes a lot of time due to the complexity of code transformations. Also, the additional code injected in the metaprogram slows down each repair attempt, ie slows down each execution of the failing test case. For NPEfix, the complete exploration of the search space takes at most 3 minutes 22 seconds. We consider this acceptable since a developer can wait for 3 minutes before offered a set of automatically tentative patches.

> **RQ4. Is the performance of NPEfix acceptable?** The complete exploration of the search space of tentative patches is faster with TemplateNPE. This highlights a trade-off between the number of patches explored found and the time to wait.

V. Discussion

A. Patch Readability and Templates

One concern of automatic patch generation is the readability of the generated patches and its impact on the maintainability of applications [5]. Let us discuss the readability of NPEfix patches. NPEfix patches repair one specific type of bug: null dereference. We observe that most developers fix these bugs by adding a null check (`if(... != null) something()`) before the null expression. Most NPEfix strategies resemble those human-written patches and thus and can be as easily understood and maintained as human patches. Our experience with NPEfix suggests that template-inspired patches enables one to encode – if not enforce – readable patches.

B. Genetic Improvement and Metaprogramming

Genetic improvement [9] refers to techniques that change the behavior of a program in order to improve a specific metric, for example the execution time. There is an interesting relation between genetic improvement and NPEfix. Both are based on dynamically changing the behavior of an application. In this perspective, making failing tests pass can be considered as a functional metric to be optimized. Indeed, we think that metaprogramming techniques similar to that proposed in this paper could be used for genetic improvement by creating a space of different program behaviours to explore.

C. Threat to Validity

A bug in the implementation of TemplateNPE or NPEfix is the threat to the validity of the findings based on quantitative comparison presented in Table III.

Our benchmark is only composed of real bugs due to null pointer exceptions which is a strength. However, since they are all in Java and from 6 projects, there is a threat to the external validity of our findings. Other bugs and an implementation of TemplateNPE and NPEfix in another language may uncover other behavioral differences.

D. Limitations

We now summarize the main limitations of NPEfix that we identified during our experiment. First, NPEfix is complex. As we have seen, the template-based approach is quite straightforward, while the metaprogramming

approach is complex to debug and maintain. Due to the richness and complexity of Java, there may be cases where the metaprogramming approach slightly changes the behaviour of the application. Second, as we have seen in Section IV-G, NPEfix is slower than the template based approach. Third, as for all test-suite based patch generation technique, there is a risk of generating test-suite adequate, yet incorrect patches as suggested by the Math-305 example.

E. Reflections on TemplateNPE

We would like to emphasize the fact TemplateNPE works surprisingly well. Initially, TemplateNPE was built as baseline comparison for NPEfix. However, over time, we realized that it is much simpler to implement and maintain, while keeping comparatively good results, and while being faster. This is an important lesson learned for us and for the metaprogramming research community: advanced code transformations and behavior modifications at runtime is not necessarily the best option. Simpler may be better.

VI. Related Work

A. Patch Generation

The literature on patch generation is growing very fast. We only present a brief overview of notable contributions and refer to [16] for a comprehensive overview. GenProg by [10] applies genetic programming to the AST of a buggy program and generates patches by adding, deleting, or replacing AST nodes. Debroy and Wong [2] propose a mutation-based repair method inspired from mutation testing. This work combines fault localization with program mutation to exhaustively explore a space of tentative patches. SemFix by [17] is a constraint based repair approach. It provides patches for assignments and conditions by combining symbolic execution and code synthesis. Nopol by [3], [21] is also a constraint based method, which focuses on repairing bugs in if-conditions and missing preconditions. SPR [13] defines a set of staged repair operators so as to early discard many candidate repairs that cannot pass the supplied test suite and eventually to exhaustively explore a small and valuable search space.

We have discussed in this paper a template-based patch generation approach. PAR by [8] uses 10 patch templates for common programming errors, Relifix [20] defines templates specifically for regression bugs. All of those approaches require a regression test suite to validate the patch, none of them leverage production traffic to assess the absence of regressions.

B. Metaprogramming for Repair

Rinard et al. [19] presented a technique called "failure oblivious computing" to avoid illegal memory accesses. Their idea is to create a metaprogram that adds additional code around each memory operation during the compilation process. For example, the additional code verifies at run-time whether an array is accessed out of his bounds. If the access is outside the allocated memory, the access is ignored (akin line skipping presented in this paper) instead of crashing with a segmentation fault.

Dobolyi and Weimer [4] present a technique to tolerate null pointer exceptions using metaprogramming. Using code transformation, they introduce hooks to a recovery framework. This framework is responsible for forward recovery of the form of creating a default object of an appropriate type of skipping instructions. Their strategies are a small subset of ours, and they do not explore the search space as we do in this paper.

The closest related work is by Kern and Esparza [7] use a metaprogram that integrates all possible mutations according to a mutation operator. The mutations that are actually executed are driven by meta-variables. A repair is a set of values for those meta-variables. The meta-variables are valued using symbolic execution. Both the metaprogram and the kind of faults are completely different from what we have presented in this paper.

VII. Conclusion

In this paper, we have presented a novel and original technique to explore the repair search space of null pointer exception bugs. Our technique, called NPEfix, is based on 9 strategies that are specific to null pointer exceptions, and uses metaprogramming to explore the search space of all possible strategies. We have evaluated our technique on 16 real null pointer exceptions: NPEfix is able to successfully explore the search space of null pointer repair, and finds 175 valid patches over all considered bugs.

Future work is required to see whether other patch templates can be implemented using metaprogramming. For instance, we are confident that even a generic technique as Genprog [10] could be implemented in a metaprogramming way, which may be key for performance. Future work is also required to optimize the transformations and to speed-up the execution of metaprograms.

References

[1] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. NPEFix: Automatic Runtime Repair of Null Pointer Exceptions in Java. Technical Report 1512.07423, Arxiv, 2015.

[2] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of ICST*, pages 65–74, 2010.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. 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*, pages 30–39. ACM, 2014.

[4] K. Dobolyi and W. Weimer. Changing Java's Semantics for Handling Null Pointer Exceptions. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium On*, pages 47–56, 2008.

[5] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[6] S. Kent. Dynamic Error Remediation: a Case Study with Null Pointer Exceptions. 2008.

[7] C. Kern and J. Esparza. Automatic Error Correction of Java Programs. In *Formal Methods for Industrial Critical Systems*, pages 67–81. 2010.

[8] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811, 2013.

[9] W. B. Langdon, B. Y. H. Lam, M. Modat, J. Petke, and M. Harman. Genetic improvement of gpu software. *Genetic Programming and Evolvable Machines*, pages 1–40, 2016.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[11] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[12] F. Long and M. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713, 2016.

[13] F. Long and M. C. Rinard. Staged program repair with condition synthesis. In *Proceedings of ESE/FSE*, 2015.

[14] F. Long, S. Sidiroglou-Douskos, and M. C. Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*,, 2014.

[15] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset. *Springer Empirical Software Engineering*, 2016.

[16] M. Monperrus. Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille, 2015.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, 2013.

[18] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. Technical report, 2015.

[19] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, volume 4, pages 21–21, 2004.

[20] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *37th International Conference on Software Engineering*. IEEE, 2015.

[21] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.