# Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs

Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, Martin Monperrus

# Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs

Jifeng Xuan, Matias Martinez, Favio DeMarco†, Maxime Clément†,
Sebastian Lamelas Marcote†, Thomas Durieux†, Daniel Le Berre, and Martin Monperrus

**Abstract**—We propose NOPOL, an approach to automatic repair of buggy conditional statements (i.e., `if-then-else` statements). This approach takes a buggy program as well as a test suite as input and generates a patch with a conditional expression as output. The test suite is required to contain passing test cases to model the expected behavior of the program and at least one failing test case that reveals the bug to be repaired. The process of NOPOL consists of three major phases. First, NOPOL employs angelic fix localization to identify expected values of a condition during the test execution. Second, runtime trace collection is used to collect variables and their actual values, including primitive data types and objected-oriented features (e.g., nullness checks), to serve as building blocks for patch generation. Third, NOPOL encodes these collected data into an instance of a Satisfiability Modulo Theory (SMT) problem; then a feasible solution to the SMT instance is translated back into a code patch. We evaluate NOPOL on 22 real-world bugs (16 bugs with buggy IF conditions and 6 bugs with missing preconditions) on two large open-source projects, namely Apache Commons Math and Apache Commons Lang. Empirical analysis on these bugs shows that our approach can effectively fix bugs with buggy IF conditions and missing preconditions. We illustrate the capabilities and limitations of NOPOL using case studies of real bug fixes.

**Index Terms**—Automatic repair, patch generation, SMT, fault localization

## 1 INTRODUCTION

AUTOMATIC software repair aims to automatically fix bugs in programs. Different kinds of techniques are proposed for automatic repair, including patch generation [28], [41] and dynamic program state recovery [42], [7].

A family of techniques has been developed around the idea of "test-suite based repair" [28]. The goal of test-suite based repair is to generate a patch that makes failing test cases pass and keeps the other test cases satisfied. Recent test-suite based repair approaches include the work by Le Goues et al. [28], Nguyen et al. [38], Kim et al. [25].

In recent work [32], we have shown that IF conditions are among the most error-prone program elements in Java programs. In our dataset, we observed that 12.5% of one-change commits simply update an IF condition. *This motivates us to study the automatic repair of conditional statements in real-world bugs*.

In this paper, we present a novel automatic repair system called NOPOL.[1] This system fixes conditional bugs in object-oriented programs and is evaluated on real bugs from large-scale open-source programs. For instance, NOPOL can synthesize a patch that updates a buggy IF condition as shown in Fig. 1 or adds a guard precondition as in Fig. 2. Both figures are excerpts of real-world bugs taken from the bug tracking system of Apache Commons Math.

NOPOL takes a buggy program as well as a test suite as input and generates a conditional patch as output. This test suite must contain at least one failing test case that embodies the bug to be repaired. Then, NOPOL analyzes program statements that are executed by failing test cases to identify the source code locations where a patch may be needed.

For each statement, the process of generating the patch consists of three major phases. First, we detect whether there exists a fix location for a potential patch in this statement with a new and scalable technique called "angelic fix localization" (Section 3.2). For one fix location, this technique reveals angelic values, which make all failing test cases pass.

Second, NOPOL collects runtime traces from test suite execution through code instrumentation (Section 3.3). These traces contain snapshots of the program state at all candidate fix locations. The collected trace consists of both primitive data types (e.g., integers and booleans) and object-oriented data (e.g., nullness or object states obtained from method calls).

- *J. Xuan is with the State Key Lab of Software Engineering, School of Computer, Wuhan University, Wuhan, China. Contact author: jxuan@whu.edu.cn.*
- *M. Martinez and M. Monperrus are with the University of Lille & INRIA, Lille, France.*
- *F. DeMarco and S. Lamelas Marcote are with the University of Buenos Aires, Buenos Aires, Argentina.*
- *M. Clément and T. Durieux are with the University of Lille, Lille, France.*
- *D. Le Berre is with the University of Artois & CNRS, Lens, France.*

† *F. DeMarco, M. Clément, S. Lamelas Marcote, and T. Durieux have contributed to this work during their internship at INRIA Lille – Nord Europe.*

1. NOPOL is an abbreviation for "no polillas" in Spanish, which literally means "no moth anymore".

```
- if (u * v == 0) {
+ if (u == 0 || v == 0) {
    return (Math.abs(u) + Math.abs(v));
  }
```

Fig. 1. Patch example of Bug CM5: a bug related to a buggy IF condition. The original condition with a comparison with == is replaced by a disjunction between two comparisons.

```
+ if (specific != null) {
    sb.append(": "); //sb is a string builder in Java
+ }
```

Fig. 2. Patch example of Bug PM2: a precondition is added to avoid a null dereference.

Third, given the runtime traces, the problem of synthesizing a new conditional expression that matches the angelic values is translated into a Satisfiability Modulo Theory (SMT) problem (Section 3.4). Our encoding extends the technique by Jha et al. [20] by handling rich object-oriented data. We use our own implementation of the encoding together with an off-the-shelf SMT solver (Z3 [12]) to check whether there exists a solution.

If such a solution exists, NOPOL translates it back to source code, i.e., generates a patch. We re-run the whole test suite to validate whether this patch is able to make all test cases pass and indeed repairs the bug under consideration.

To evaluate and analyze our repair approach NOPOL, we collect a dataset of 22 bugs (16 bugs with buggy IF conditions and 6 bugs with missing preconditions) from real-world projects. Our result shows that 17 out of 22 bugs can be fixed by NOPOL, including four bugs with manually added test cases. Four case studies are conducted to present the benefits of generating patches via NOPOL and five bugs are employed to explain the limitations.

The main contributions of this paper are as follows.

- The design of a repair approach for fixing conditional statement bugs of the form of buggy IF conditions and missing preconditions.
- Two algorithms of angelic fix localization for identifying potential fix locations and expected values.
- An extension of the SMT encoding in [20] for handling nullness and certain method calls of object-oriented programs.
- An evaluation on a dataset of 22 bugs in real-world programs with an average of 25K executable lines of code for each bug.
- A publicly-available system for supporting further replication and research.
- An analysis of the repair results with respect to fault localization.

This paper is an extension of our previous work [14]. This extension adds an evaluation on a real-world bug dataset, four detailed case studies, a discussion of the limitations of our approach, and a detailed analysis on patches.

The remainder of this paper is organized as follows. Section 2 provides the background of test-suite based repair. Section 3 presents our approach for repairing bugs with buggy IF conditions and missing preconditions. Section 4 details the evaluation on 22 real-world bugs. Section 5 further analyzes the repair results. Section 6 presents potential issues and Section 7 lists the related work. Section 8 concludes.

## 2 BACKGROUND

We present the background on test-suite based repair and the two kinds of bugs targeted in this paper.

### 2.1 Test-Suite Based Repair

Test-suite based repair consists in repairing programs according to a test suite, which contains both passing test cases as a specification of the expected behavior of the program and at least one failing test case as a specification of the bug to be repaired. Failing test cases can either identify a regression bug or reveal a new bug that has just been discovered. Then, a repair algorithm searches for patches that make all the test cases pass.

The core assumption of test-suite based repair is that the test suite is good enough to thoroughly model the program to repair [36]. This is a case when the development process ensures a very strong programming discipline. For example, most commits of Apache projects (e.g., Apache Commons Lang) contain a test case specifying the change. If a commit is a bug fix, the commit contains a test case that highlights the bug and fails before the fix.

Test-suite based repair, which has been popularized by the work of GenProg by Le Goues et al. [28], has become an actively explored research area [38], [25], [44], [45], [30]. The approach presented in this paper, NOPOL, is also an approach to test-suite based repair. Other kinds of repair methods include repair based on formal models [21] and dynamic repair of the program state [42].

### 2.2 Buggy IF Condition Bugs

Conditional statements (e.g., `if (condition){...} else {...}`), are widely-used in programming languages. Pan et al. [39] show that among seven studied Java projects, up to 18.6% of bug fixes have changed a buggy condition in IF statements. A buggy IF condition is defined as a bug in the condition of an `if-then-else` statement.

The bug in Fig. 1 is a real example of a buggy IF condition in Apache Commons Math. This bug is a code snippet of a method that calculates the greatest common divisor between two integers. The condition in that method is to check whether either of two parameters `u` and `v` is equal to 0. In the buggy version, the developer compares the product of the two integers to zero. However, this may lead to an arithmetic overflow. A safe way to proceed is to compare each parameter to zero. This bug was fixed by NOPOL (see Bug CM5 in Table 2).

### 2.3 Missing Precondition Bugs

Another class of common bugs related conditions is the class of missing preconditions. A precondition aims to check the state of certain variables before the execution of a statement. Examples of common preconditions include detecting a null pointer or an invalid index in an array. In software repositories, we can find commits that add preconditions (i.e., which were previously missing).

The bug in Fig. 2 is a missing precondition with the absence of null pointer detection. The buggy version without the precondition throws an exception signaling a null pointer at runtime. NOPOL fixed this bug by adding the precondition (see Bug PM2 in Table 2).

## 3 Our Approach

This section presents our approach to automatically repairing buggy IF conditions and missing preconditions. Our approach is implemented in a tool called NOPOL that repairs Java code.

### 3.1 Overview

NOPOL is a repair approach, which is dedicated to buggy IF conditions and missing preconditions. As input, NOPOL requires a test suite which represents the expected program functionality with at least one failing test case that exposes the bug to be fixed. Given a buggy program and its test suite, NOPOL returns the final patch as output. Fig. 1 and Fig. 2 are two examples of output patches for buggy IF conditions and missing preconditions by NOPOL, respectively.

**How to use NOPOL.** From a user perspective, given a buggy program with a test suite, including failing test cases, the user would run NOPOL and obtain a patch, if any. Before applying NOPOL to the buggy program, the user does not need to know whether the bug relates to conditions. Instead, the user runs NOPOL for any buggy program. If NOPOL finds a patch, then the user would manually inspect and validate it before the integration in the code base. As further discussion in Section 4.4, the user can also add a pre-defined timeout, e.g., 90 seconds as suggested in experiments or a longer timeout like five hours instead of exhaustively exploring the search space.

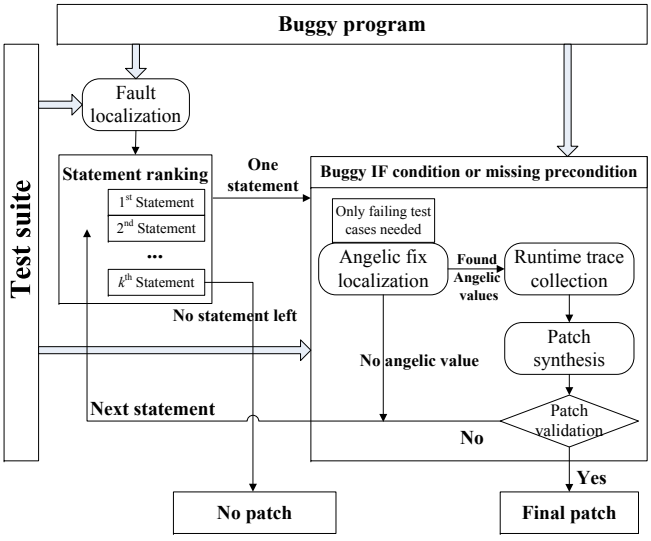


Fig. 3. Overview of the proposed automatic repair approach, NOPOL.

Fig. 3 shows the overview of NOPOL. NOPOL employs a fault localization technique to rank statements according to their suspiciousness of containing bugs. For each statement in the ranking, NOPOL considers it as a buggy IF condition candidate if the statement is an IF statement; or NOPOL considers it as a missing precondition candidate if the statement is any other non-branch or non-loop statement (e.g., an assignment or a method call). NOPOL processes candidate statements one by one with three major phases.

First, in the phase of *angelic fix localization*, NOPOL arbitrarily tunes a conditional value (`true` or `false`) of an IF statement to pass a failing test case. If such a conditional value exists, the statement is identified as a fix location and the arbitrary value is viewed as the expected behavior of the patch. In NOPOL, there are two kinds of fix locations, IF statements for repairing buggy conditions and arbitrary statements for repairing missing preconditions.

Second, in the phase of *runtime trace collection*, NOPOL runs the whole test suite in order to collect the execution context of each fix location. The context includes both variables of primitive types (e.g., booleans or integers) and a subset of object-oriented features (nullness and certain method calls); then such runtime collection will be used in synthesizing the patch in the next phase.

Third, in the phase of *patch synthesis*, the collected trace is converted into a Satisfiability Modulo Theory (SMT) formula. The satisfiability of SMT implies that there exists a program expression that preserves the program behavior and fixes the considered bug. That is, the expression makes all test cases pass. If the SMT formula is satisfiable, the solution to the SMT is translated as a source code patch; if unsatisfiable, NOPOL goes to the next statement in the statement

ranking, until all statements are processed.

After the above three phases, the whole test suite is re-executed to validate that the patch is correct. This validation could be skipped if the SMT encoding is proven to be correct. Indeed, theoretically, if the SMT solver says "satisfiable", it means that a patch exists. However, there could be an implementation bug in the trace collection, in the SMT problem generation, in the off-the-shelf SMT solver, or in the patch synthesis. Consequently, we do make the final validation by re-executing the test suite.

### 3.2 Angelic Fix Localization

In NOPOL, we propose to use value replacement [19] to detect potential fix locations. Value replacement [19] comes from fault localization research. It consists in replacing at runtime one value by another one. More generally, the idea is to artificially change the program state for localizing faults. There are a couple of papers that explore this idea. For instance, Zhang et al. [55] use the term "predicate switching" and Chandra et al. [9] use the term "angelic debugging".

NOPOL replaces conditional values in IF statements. We refer to conditional values that make test cases pass as *angelic values*.

**Definition (Angelic Value)** An angelic value is an arbitrarily-set value at a given location during test execution, which enables a failing test case to pass.

To facilitate the description of our approach, we follow existing work [18] to introduce the concept of locations. A *location* is an integer value, which identifies the absolute position of a statement in source code.

**Definition (Angelic Tuple)** An angelic tuple is a triplet $(loc, val, test)$, where the statement at a location $loc$ is evaluated to a value $val$ to make a failing test case $test$ pass.

In this paper, we refer to the technique of modifying the program state to find the values for angelic tuples $(loc, val, test)$ as *angelic fix localization*. If an angelic tuple $(loc, val, test)$ is found, there may exist a patch in the location $loc$ in source code. In the phase of angelic fix localization, only failing test cases are needed, not the whole test suite.

A single test case $test$ may evaluate the statement at the location $loc$ several times. Consequently, according to our definition, the value $val$ is fixed across all evaluations of a given statement for one test case. This is the key point for having a tractable search space (will be discussed in Section 3.2.3). On the other hand, one angelic value is specific to a test case: for a given location $loc$, different failing test cases may have different angelic values.

#### 3.2.1 For Buggy IF Conditions

For buggy IF conditions, angelic fix localization works as follows. For each IF condition that is evaluated during test suite execution, an angel forces the IF condition to be `true` or `false` in a failing test case. An angelic tuple $(loc, val, test)$, i.e., (IF condition location, boolean value, failing test case), indicates that a fix modifying this IF condition may exist (if the subsequent phase of patch synthesis succeeds, see Section 3.4).

**Input** :
$stmt$, a candidate IF statement;
$T_f$, a set of failing test cases.
**Output**:
$R$, a set of angelic tuples.

```
1  R ← ∅;
2  Initialize two sets T_true ← ∅ and T_false ← ∅;
3  Let cond be the condition in stmt and let loc be
   the location of cond;
4  Force cond to true and execute test cases in T_f;
5  foreach failing test case t_i ∈ T_f do
6      if t_i passes then
7          T_true ← T_true ∪ {t_i};
8      end
9  end
10 Force cond to false and execute test cases in T_f;
11 foreach failing test case t_i ∈ T_f do
12     if t_i passes then
13         T_false ← T_false ∪ {t_i};
14     end
15 end
   // All test cases in T_f are passed
16 if (T_f \ T_true) ∩ (T_f \ T_false) = ∅ then
17     foreach t_i ∈ T_true do
18         R ← R ∪ {(loc, true, t_i)};
19     end
20     foreach t_i ∈ T_false do
21         R ← R ∪ {(loc, false, t_i)};
22     end
23 end
```

**Algorithm 1:** Angelic Fix Localization Algorithm for Buggy IF Conditions

Algorithm 1 is the pseudo-code of angelic fix localization for buggy IF conditions. For a given IF statement *stmt* and its condition *cond*, both `true` and `false` are set to pass originally failing test cases at runtime. Lines 4 to 9 and Lines 10 to 15 describe how to set *cond* to be `true` and `false`, respectively. If all failing test cases are passed, angelic tuples are collected, i.e., Lines 17 to 22, for further patch synthesis; otherwise, there exists no angelic value for the test case and the location under consideration. The same idea of forcing the execution can be used to identify angelic values for loop conditions [31].

**Input** :
$stmt$, a candidate non-IF statement;
$T_f$, a set of failing test cases.
**Output**:
$R$, a set of angelic tuples.

1 $R \leftarrow \emptyset$;
2 Initialize a test case set $T_{pre} \leftarrow \emptyset$;
3 Let $loc$ be the location of a potential precondition of $stmt$;
4 Force $stmt$ to be skipped and execute $T_f$;
5 **foreach** *failing test case* $t_i \in T_f$ **do**
6 | **if** $t_i$ *passes* **then**
7 | | $T_{pre} \leftarrow T_{pre} \cup \{t_i\}$;
8 | **end**
9 **end**
`// All test cases in` $T_f$ `are passed`
10 **if** $T_{pre} = T_f$ **then**
11 | **foreach** $t_i \in T_{pre}$ **do**
12 | | $R \leftarrow R \cup \{(loc, \texttt{false}, t_i)\}$;
13 | **end**
14 **end**

**Algorithm 2:** Angelic Fix Localization Algorithm for Missing Preconditions

#### 3.2.2 For Missing Preconditions

Angelic fix localization for missing preconditions is slightly different from that for IF conditions. For each non-branch and non-loop statement that is evaluated during test suite execution, an angel forces to skip it. If a failing test case now passes, it means that a potential fix location has been found. The oracle for repair is then "false"; that is, the added precondition must be `false`, i.e., the statement should be skipped. Then, an angelic tuple $(loc, val, test)$ is (precondition location, `false`, failing test case).

Algorithm 2 is the pseudo-code of this algorithm. Given a non-IF statement *stmt*, we skip this statement to check whether all failing test cases are passed, i.e., Lines 4 to 9. If yes, the location *loc* of the precondition as well as its angelic value `false` is collected, i.e., Lines 11 to 13. If skipping the statement does not pass all the failing test cases, no angelic values will be returned. This technique also works for missing preconditions for entire blocks since blocks are just specific statements in Java. In our implementation, we only consider adding missing preconditions for single statements rather than blocks. Manual examination on the dataset in Section 4.4 will show that our dataset does not contain missing preconditions for blocks.

#### 3.2.3 Characterization of the Search Space

We now characterize the search space of angelic values. If an IF condition is executed more than once in a failing test case, there may exist a sequence of multiple different angelic values resulting in a passing test case. For example, a buggy IF condition that is executed three times by one failing test case may require a sequence of three different angelic values to pass the test case.

**Search space for buggy IF conditions**. In general, if one failing test case executes a buggy IF condition for $t_c$ times, the search space of all sequences of angelic values is $2^{t_c}$. To avoid the problem of combinatorial explosion, NOPOL assumes that, for a given failing test case, the angelic value is the same during the multiple executions on one statement. The search space size becomes 2 for one failing test case instead of $2^{t_c}$. Under this assumption, the search space is shown as follows.

For buggy IF condition, the search space is $2 \times n_c$ where $n_c$ is the number of executed IF statements by a given failing test case.

**Search space for missing preconditions**. Similarly to angelic fix localization for buggy IF conditions, if a statement is executed several times by the same failing test case, angelic fix localization directly adds a precondition (with a `false` value) and completely skips the statement for a given test case.

For missing precondition bugs, the search space size is $n_p$, where $n_p$ is the number of executed statements by test cases. It is not $2 \times n_p$ because we only add a precondition and check whether the `false` value passes the failing test case.

NOPOL does not decide a priority between updating existing conditions or adding new preconditions. A user can try either strategy, or both. There is no analytical reason to prefer one or the other; our evaluation does not give a definitive answer to this question. In our experiment, we perform both strategies for statements one by one (see Section 3.1).

If no angelic tuple is found for a given location, there are two potential reasons. First, it is impossible to fix the bug by changing the particular condition (resp. adding a precondition before the statement). Second, only a sequence of different angelic values, rather than a single angelic value, would enable the failing test case to pass. Hence, NOPOL is incomplete: there might be a way to fix an IF condition by alternating the way of finding angelic values, but we have not considered it in this paper.

### 3.3 Runtime Trace Collection for Repair

Once an angelic tuple is found, NOPOL collects the values that are accessible at this location during program execution. Those values are used to synthesize a correct patch (in Section 3.4). In our work, different kinds of data are collected to generate a patch.

#### 3.3.1 Expected Outcome Data Collection

As mentioned in Section 3.2, an angelic value indicates that this value enables a failing test case to pass. To generate a patch, NOPOL collects the expected

outcomes of conditional values to pass the whole test suite: angelic values for failing test cases as well as actual execution values for the passing ones.

Let $O$ be a set of expected outcomes in order to pass all test cases. An expected outcome $O_{loc,m,n} \in O$ refers to the value at location $loc$ during the $m$-th execution in order to pass the $n$-th test case. NOPOL collects $O_{loc,m,n}$ for all executions of location $loc$.

**For buggy IF conditions**. $O_{loc,m,n}$ is the expected outcome of the condition expression at $loc$. For a failing test case, the expected outcome is the angelic value; for a passing test case, the expected outcome is the runtime value $eval(loc)$, i.e., the result of the evaluation during the actual execution of the IF condition expression.

$$O_{loc,m,n} = \begin{cases} eval(loc), & \text{for passing test cases} \\ \text{angelic value} & \text{for failing test cases} \end{cases}$$

**For missing preconditions**. $O_{loc,m,n}$ is the expected value of the precondition at $loc$, i.e., `true` for passing test cases and `false` for failing test cases. The latter comes from angelic fix localization: if the precondition returns `false` for a failing test case, the buggy statement is skipped and the test case passes.

$$O_{loc,m,n} = \begin{cases} true & \text{for passing test cases} \\ false & \text{for failing test cases} \end{cases}$$

Note that not all the bugs with missing preconditions can be fixed with the above definition. Section 4.6.3 will present the limitation of this definition with a real example.

#### 3.3.2 Primitive Type Data Collection

At the location of an angelic tuple, NOPOL collects the values of all local variables, method parameters, and class fields that are typed with a basic primitive type (booleans, integers, floats, and doubles).

Let $C_{loc,m,n}$ be the set of collected values at location $loc$ during the $m$-th execution of the $n$-th test case. In order to synthesize conditions that use literals (e.g., `if (x > 0)`), $C_{loc,m,n}$ is enriched with constants for further patch synthesis. First, NOPOL collects static values that are present in the program.[2] Second, we add three standard values {0, -1, 1}, which are present in many bug fixes in the wild (for instance for well-known off-by-one errors). Based on these standard values, other values can be formed via wiring building blocks (in Section 3.4.2). For example, a value 2 in a patch can be formed as $1 + 1$, if 2 is not collected during runtime trace collection.

2. Besides collecting static fields in a class, we have also tried to collect other class fields of the class under repair, but the resulting patches are worse in readability than those without collecting class fields. Hence, no class fields other than static fields are involved in data collection.

#### 3.3.3 Object-Oriented Data Collection

NOPOL aims to support automatic repair for object-oriented programs. In particular, we would like to support nullness checks and some particular method calls. For instance, NOPOL is able to synthesize the following patch containing a method call.

```
+ if (obj.size() > 0) {
      compute(obj);
+ }
```

To this end, in addition to collecting all values of primitive types, NOPOL collects two kinds of object-oriented features. First, NOPOL collects the nullness of all variables of the current scope. Second, NOPOL collects the output of "state query methods", defined as the methods that inspect the state of objects and are side-effect free. A state query method is an argument-less method with a primitive return type. For instance, methods `size()` and `isEmpty()` of `Collection` are state query methods. The concept of state query methods is derived from "argument-less boolean queries" of "object states" by Pei et al. [41].

NOPOL is manually fed with a list of such methods. The list is set with domain-specific knowledge. For instance, in Java, it is easy for developers to identify such side-effect free state query methods on core library classes such as `String`, `File` and `Collection`. For each object-oriented class $T$, those predefined state query methods are denoted as $sqm(T)$.

NOPOL collects the nullness of all visible variables and the evaluation of state query methods for all objects in the scope (local variables, method parameters, and fields) of a location where an angelic tuple exists. Note that this incorporates inheritance; the data are collected based on the polymorphism in Java. For instance, when the value of `obj.size()` is collected, it may be for one implementation of `size()` based on array lists and for another implementation of `size()` based on linked lists. This means that a patch synthesized by NOPOL can contain polymorphic calls.

#### 3.3.4 On the Size of Collected Data

Let us assume there are $u$ primitives values and a set $O$ of $w$ objects in the scope of an angelic tuple. In total, NOPOL collects the following values:

- $u$ primitive variables in the scope;
- $w$ boolean values corresponding to the nullness of each object;
- $\sum_{o \in O} |sqm(class(o))|$ values corresponding to the evaluation of the state query methods of all objects available in the scope, where $class(o)$ denotes the class of the object $o$;
- constants, i.e., 0, -1, and 1 in our work.

### 3.4 Patch Synthesis: Encoding Repair in SMT

The patch synthesis of buggy IF conditions and missing preconditions consists of synthesizing an expres-

sion $exp$ such that

$$\forall_{loc,m,n} \quad exp(C_{loc,m,n}) = O_{loc,m,n} \tag{1}$$

As a synthesis technique, NOPOL, as SemFix [38], uses a variation of oracle-guided component-based program synthesis [20], which is based on SMT.

The solution of the SMT problem is then translated back to a boolean source code expression $exp$ representing the corrected IF condition or the added precondition.

#### 3.4.1 Building Blocks

We define a *building block* (called a *component* in [20]) as a type of expression that can appear in the boolean expression to be synthesized. For instance, the logical comparison operator "<" is a building block. As building block types, we consider comparison operators ($<$, $>$, $\leq$, $\geq$, $=$, and $\neq$), arithmetic operators ($+$, $-$, and $\times$),[3] and boolean operators ($\wedge$, $\vee$, and $\neg$). The same type of building blocks can appear multiple times in one expression.

Let $b_i$ be the $i$th building block ($i = 1, 2, \ldots, k$). Then $b_i$ is identified as a tuple of a set of input variables $I_i$, an output variable $r_i$, and an expression $\phi_i(I_i, r_i)$ encoding the operation of the building block. That is $b_i = (\phi_i(I_i, r_i), I_i, r_i)$ ($b_i = (\phi_i, I_i, r_i)$ for short). For example, given a boolean output $r_i$ and an input $I_i = \{I_{i,1}, I_{i,2}\}$ consisting of two boolean values, a building block could be $b_i = (\phi_i, \{I_{i,1}, I_{i,2}\}, r_i)$, where $\phi_i$ is implemented with the operator $\wedge$, i.e., $I_{i,1} \wedge I_{i,2}$.

Let $r$ be the final value of the synthesized patch. Hence there exists one building block $b_i$ whose output is bound to the return value $r_i = r$.

Suppose we are given a set $B$ of building blocks and a list $CO$ of pairs $(C_{loc,m,n}, O_{loc,m,n})$, i.e., pairs of collected values and expected values at the location $loc$ during the $m$-th execution of the $n$-th test case. $C_{loc,m,n}$ includes values of different types: BOOL, INT, or REAL.[4] A patch is a sequence of building blocks $< b_1, b_2, ..., b_k >$ with $b_i \in B$, whose input values are taken from either $C_{loc,m,n}$ or other building blocks.

#### 3.4.2 Wiring Building Blocks

The problem of patch synthesis is thus to wire the input of building blocks $< b_1, b_2, ..., b_k >$ to the input values $I_0$ or the output values of other building blocks. To synthesize a condition, we need to make sure that the types of the variables are valid operands (e.g., an arithmetic operator only manipulates integers).

**Example**. Let us assume that $C_{loc,m,n}$ has three values, an integer variable $i_0$, a boolean constant $c_1 \leftarrow False$, and an integer constant $c_2 \leftarrow 3$. Assume we have two building blocks, BOOL $\leftarrow f_1$(BOOL) and BOOL $\leftarrow f_2$(INT, INT). Then the goal of patch synthesis is to find a well formed expression, such as $False$, $f_1(False)$, $f_2(i_0, 3)$, $f_2(3, 3)$, and $f_1(f_2(3, 3))$; meanwhile, one of these expressions is expected to match the final output $r$.

#### 3.4.3 Mapping Inputs and Outputs with Location Variables

Let $I = \cup I_i$ and $O = \cup\{r_i\}$ be the sets of input and output values of all building blocks $b_i \in B$. Let $I_0$ be the input set $\{C_{loc,m,n}\}$ and let $r$ be the output in the final patch. We define $IO$ as $IO = I \cup O \cup I_0 \cup \{r\}$. We partition the elements of $IO$ according to their types in BOOL, INT, and REAL.

The SMT encoding relies on the creation of location variables. A *location variable* $l_x \in L$ represents an index of an element $x \in IO$. Note that the concept of location variables in SMT encoding is different from the concept of locations in Section 3.2. A location variable $l_x$ indicates a relative position, i.e., an index, of $x$ in a patch while a location $loc$ indicates an absolute position of a statement in a source file. A *value variable* $v_x \in V$ represents a value taken by an elements $x \in IO$. Values of location variables are actually integers ($L \subseteq$ INT); value variables are of any supported type, i.e., BOOL, INT, or REAL.

Informally, a location variable $l_x$ serves as an index of $x \in IO$ in a code fraction while a value variable $v_x$ indicates its value during test execution. Section 3.4.4 will further illustrate how to index the code via location variables. Location variables are invariants for the execution of all test cases: they represent the patch structure. Value variables are used internally by the SMT solver to ensure that the semantics of the program is preserved.

#### 3.4.4 Domain Constraints

Let us first define the domain constraints over the location variables. Given the input set $I_0$ and the building block set $B$, let $p$ be the number of possible inputs and $p = |I_0| + |B|$. The location variables of the elements of $I_0$ and $r$ are fixed:

$$\phi_{FIXED}(I_0, r) = (\wedge_{i=1}^{|I_0|} \; l_{I_0,i} = i) \bigwedge l_r = p$$

Given building blocks $b_i \in B$, the location variable $l_{r_i}$ for the output $r_i$ ($r_i \in O$) of $b_i$ belongs to a range of $[|I_0| + 1, p]$:

$$\phi_{OUTPUT}(O) = \bigwedge_{i=1}^{|O|} (|I_0| + 1 \leq l_{r_i} \leq p)$$

**Handling types**. Only the location variables corresponding to the values of the same type are allowed. Suppose that $type(x)$ returns the set of elements with the same type of $x$ among BOOL, INT, and REAL. Then we can restrict the values taken by the location

3. Adding the division is possible but would require specific care to avoid division by zero.

4. In the context of SMT, we use BOOL, INT, and REAL to denote the types of booleans, integers, and doubles as well as floats in Java, respectively.

variables of the input values of building blocks using the following formula:

$$\phi_{INPUT}(I) = \bigwedge_{x \in I} \bigvee_{y \in type(x), x \neq y} (l_x = l_y)$$

Recall the example in Section 3.4.2, we have the input $I_0 = \{i_0, c_1, c_2\}$, the output $r$, and two building blocks $f_1$ and $f_2$. We assume that each building block is involved in the patch for at most once for simplifying the example; in our implementation, a building block can be used once or more in a synthesized patch. Then we have the location variables as follows. The location variables of $i_0$, $c_1$, and $c_2$ are 1, 2, and 3; the locations variables of building blocks are 4 and 5, respectively. Based on the types, candidate values of location variables of $I_{f_1,1}$, $I_{f_2,1}$, and $I_{f_2,2}$ are calculated.

| | |
|---|---|
| $l_{i_0} = 1$ | input variable, integer |
| $l_{c_1} = 2$ | boolean constant, $False$ |
| $l_{c_2} = 3$ | integer constant, 3 |
| $l_{r_{f_1}} \in [4, 5]$ | output of $f_1$, boolean |
| $l_{r_{f_2}} \in [4, 5]$ | output of $f_2$, boolean |
| $l_r = 5$ | expected output value, boolean |
| $l_{I_{f_1,1}} \in \{l_{c_1}, l_{r_{f_1}}, l_{r_{f_2}}\}$ | the parameter of $f_1$, boolean |
| $l_{I_{f_2,1}} \in \{l_{i_0}, l_{c_2}\}$ | the first parameter of $f_2$, integer |
| $l_{I_{f_2,2}} \in \{l_{i_0}, l_{c_2}\}$ | the second parameter of $f_2$, integer |

The following additional constraints are used to control the values of location variables. First, we ensure that each output of a building block is mapped to one distinct input (wires are one-to-one).

$$\phi_{CONS}(L, O) = \bigwedge_{x,y \in O, x \neq y} l_x \neq l_y$$

Second, we need to order the building blocks in such a way that its arguments have already been defined.

$$\phi_{ACYC}(B, L, I, O) = \bigwedge_{(\phi_i, I_i, r_i) \in B} \bigwedge_{x \in I_i} l_x < l_{r_i}$$

Then, we combine all constraints together.

$$\phi_{WFF}(B, L, I, O, I_0, r) = \phi_{FIXED}(I_0, r) \wedge \phi_{OUTPUT}(O) \wedge \phi_{INPUT}(I) \wedge \phi_{CONS}(L, O) \wedge \phi_{ACYC}(B, L, I, O)$$

An assignment of $L$ variables respecting the predicate $\phi_{WFF}(B, L, I, O, I_0, r)$ corresponds to a syntactically correct patch.

Value variables corresponding to the input and the output of a building block are related according to the functional definition of a predicate $pb_i$. Given a building block $b_i = \{\phi_i, I_i, r_i\}$, let $value(I_i)$ be a function that returns the value for the input $I_i$. For a value variable $v_{r_i}$, let $pb_i(value(I_i), v_{r_i}) = true$ iff $\phi_i(I_i) = r_i$. Given $V_{IO} = \{v_x | x \in I \cup O\}$, we define the following constraint.

$$\phi_{LIB}(B, V_{IO}) = \bigwedge_{(\phi_i, I_i, r_i) \in B, v_{r_i} \in V_{IO}} pb_i\Big(value(I_i), v_{r_i}\Big)$$

The location variables and the value variables are connected together using the following rule which states that elements at the same position should have the same value. Note that we need to limit the application of that rule to values of the same type because in our case, input or output values can be of different types. Such a limitation to the elements of the same type is valid since the domain of the location variables are managed using constraints $\phi_{INPUT}(I)$.

$$\phi_{CONN}(L, V_{IO}) = \bigwedge_{S \in \{\text{BOOL,INT,REAL}\}} \bigwedge_{x,y \in S} l_x = l_y \Rightarrow v_x = v_y$$

Let the notation $\alpha[v \leftarrow x]$ mean that the variable $v$ in the constraint $\alpha$ has been set to the value $x$. For a given location $loc$, the patch for a given input $I_0$ and a given output $r$ is preserved using the following existentially quantified constraint.

$$\phi_{FUNC}(B, L, C_{loc,m,n}, O_{loc,m,n}) = \exists V_{IO}\Big(\phi_{LIB}(B, V_{IO}) \bigwedge \phi_{CONN}(L, V_{IO})[value(I_0) \leftarrow C_{loc,m,n}, v_r \leftarrow O_{loc,m,n}]\Big)$$

Finally, finding a patch which satisfies all expected input-output pairs $(C_{loc,m,n}, O_{loc,m,n})$ requires to satisfy the following constraint.

$$\phi_{PATCH}(B, I, O, CO, I_0, r) = \exists L\Big(\bigwedge_{(C_{loc,m,n}, O_{loc,m,n}) \in CO} \phi_{FUNC}(B, L, C_{loc,m,n}, O_{loc,m,n}) \bigwedge \phi_{WFF}(B, L, I, O, I_0, r)\Big)$$

### 3.4.5 Complexity Levels of Synthesized Expressions

Ideally, we could feed SMT with many instances of all kinds of building blocks (see Section 3.4.1). Only the required building blocks would be wired to the final result. This is an inefficient strategy in practice: some building blocks require expensive computations, e.g., a building block for multiplication (which is a hard problem in SMT).

To overcome this issue, we use the same technique of complexity levels as SemFix [38]. We first try to synthesize an expression with only one instance of easy building blocks ($<$, $\leq$, $\neq$, and $=$)[5]. Then, we add new building blocks (e.g., building blocks of logical operators and arithmetic operators, successively) and eventually we increase the number of instances of building blocks. We refer to those successive SMT satisfaction trials as the "SMT level".

### 3.4.6 Patch Pretty-Printing

NOPOL translates a solution to a patch in source code if there is a feasible solution to the SMT problem. Since NOPOL repairs bugs with buggy IF conditions and

5. $>$ and $\geq$ are obtained by symmetry, e.g., $a \geq b$ as $b \leq a$.

**Input** :
$L$, an assignment of location variables, i.e., an SMT solution;
$r$, a final and expected output variable of patch;
**Output**:
$patch$, a source code patch.

```
1  Find a location variable l_x = l_r;
2  patch = traverse(l_x);
3  Function traverse(l_x)
4      if x ∈ O then   // Output of a building block
5          Find the expression φ_x(I_x, x) and
           I_x = (I_{x,1}, I_{x,2}, ...);
6          return code(φ_x(traverse(I_{x,1}), traverse(I_{x,2}), ...));
7      else if x ∈ I then // Input of a building block
8          Find y for l_y = l_x;   // l_y ∈ O ∪ I_0
9          return traverse(l_y);
10     else   // x ∈ I_0, from collected runtime trace
11         return code(x);
12     end
13 end
```

**Algorithm 3:** Translation Algorithm from an SMT Solution to a Source Code Patch.

missing preconditions, the patch after translation is a conditional expression, which returns a boolean value.

The translation is obtained with a backward traversal starting at the final output location $l_r$, as explained in Algorithm 3. A function $traverse$ returns the traversal result according to the location variables while a function $code$ converts a variable into source code. For example, for a variable $a$, $code(a)$ is translated to $a$; if $\phi$ denotes the conjunction of boolean values, $code(\phi(traverse(a), traverse(b)))$ is translated to $a \wedge b$. As shown in Algorithm 3, patch translation from a SMT solution to a patch is a deterministic algorithm, which generates an identical patch for a given SMT solution. Once a patch is translated from the SMT solution, NOPOL returns this patch to developers as the final patch.

Here is a possible solution to the SMT instance for our running example (in Section 3.4.2): $l_{i_0} = 1$, $l_{c_1} = 2$, $l_{c_2} = 3$, $l_r = 5$, $l_{r_{f_1}} = 4$, $l_{r_{f_2}} = 5$, $l_{I_{f_1,1}} = 2$, $l_{I_{f_2,1}} = 1$, $l_{I_{f_2,2}} = 1$.

In our example, the output is bound to $l_r = 5$ that is the output of $f_2$. Then $f_2$ takes the integer input value $i_0$ in $l_{i_0}$ as a parameter. The final patch is thus the expression $f_2(i_0, i_0)$ which returns a boolean. This patch could be the repair of a bug, i.e., a fixed IF condition or an added precondition. In this example, $f_1$ is never used.

### 3.5 Fault Localization

NOPOL uses an existing fault localization technique to speed up finding an angelic value, if one exists. In fault localization, statements are ranked according to their suspiciousness. The *suspiciousness* of a statement measures its likelihood of containing a fault.

In NOPOL, a spectrum-based ranking metric, Ochiai [4], is used as the fault localization technique. Existing empirical studies [47], [52] show that Ochiai is more effective on localizing the root cause of faults in object-oriented programs than other fault localization techniques. In Section 5.2, we will compare the effectiveness among different fault localization techniques.

Given a program and a test suite, the suspiciousness $susp(s)$ of a statement $s$ is defined as follows.

$$susp(s) = \frac{failed(s)}{\sqrt{total\_failed * (failed(s) + passed(s))}}$$

where $total\_failed$ denotes the number of all the failing test cases and $failed(s)$ and $passed(s)$ respectively denote the number of failing test cases and the number of passing test cases, which cover the statement $s$. Note that $0 \leq susp(s) \leq 1$ where $susp(s) = 1$ indicates the highest probability of localizing the bug and $susp(s) = 0$ indicates there is no likelihood between this statement and the bug.

We rank all the statements based on their suspiciousness in descending order. For all the statements with the suspiciousness over zero, we detect whether this statement is an IF statement or not. As previously mentioned in Section 3.1, for an IF condition, NOPOL tries to synthesize a new condition while for a non-IF statement, NOPOL tries to add a precondition.

## 4 Automatic Repair of Real-World If Bugs

We now evaluate our repair approach, NOPOL, on a dataset of 22 real-world bugs. First, we describe our evaluation methodology in Section 4.1; second, we introduce the setup of our dataset in Section 4.2 and the implementation details in Section 4.3; third, we present the general description of the synthesized patches in Section 4.4; fourth, four bugs are employed as case studies in Section 4.5 and five bugs are used to illustrate the limitations in Section 4.6.

### 4.1 Evaluation Methodology

Our evaluation methodology is based on the following principles.

**P1**. We evaluate our tool, NOPOL, on real-world buggy programs (Section 4.4).

**P2**. For bugs that NOPOL can fix, we examine the automatically generated patches, and compare them with human-produced patches (Section 4.5).

**P3**. For bugs that NOPOL cannot correctly fix, we check the details of these bugs and highlight the reasons behind the unrepairability (Section 4.6). When the root cause is an incorrect test case (i.e., an incomplete specification), we modify the test case and re-run NOPOL.

**P4**. We deliberately do not compute a percentage of repaired bugs because this is a potentially unsound measure. According to our previous investigation [36], this measure is sound if and only if 1) the dataset is only composed of bugs of the same kind and 2) the distribution of complexity within the dataset reflects the distribution of all in-the-field bugs within this defect class. In our opinion, the second point is impossible to achieve.

We have not quantitatively compared our approach against existing repair approaches on the same dataset because 1) either existing approaches are inapplicable on this dataset (e.g., GenProg [28] and SemFix [38] are designed for C programs); 2) or these approaches are not publicly available (e.g., PAR [25] and mutation-based repair [13]).

### 4.2 Dataset of Real-World Bugs

NOPOL focuses on repairing conditional bugs, i.e., bugs in IF conditions and preconditions. Hence, we build a dataset of 22 real-world bugs of buggy IF conditions and missing preconditions. Since our prototype implementation of NOPOL repairs Java code, these 22 bugs are selected from two open-source Java projects, Apache Commons Math[6] and Apache Commons Lang[7] (*Math* and *Lang* for short, respectively).

Both Math and Lang manage source code using Apache Subversion[8] (SVN for short) and manage bug reports using Jira.[9] Jira stores the links between bugs and related source code commits. In addition, these projects use the FishEye browser to inspect source code and commits.[10]

In our work, we employ the following four steps to collect bugs for the evaluation. First, we automatically extract small commits that modify or add an IF condition using Abstract Syntax Tree (AST) analysis [16]. We define a *small commit* as a commit that modifies at most 5 files, each of which introduces at most 10 AST changes (as computed by the analytical method, GumTree [16]). In Math, this step results in 161 commits that update IF conditions and 104 commits that add preconditions; in Lang, the commits are 165 and 91, respectively. The lists of commits are available at the NOPOL project [1]. Second, for each extracted commit, we collect its related code revision, i.e., the source program corresponding to this commit. We manually check changes between the code revision and its previous one; we only accept changes that contain an IF condition or a missing precondition and do not affect other statements. Those commits could also contain other changes that relate to neither a bug nor a patch, such as a variable renaming or the addition of a logging statement. In this case, changes of the patch are separated from irrelevant changes. Third, we extract the test suite at the time of the patch commit, including failing test cases.[11] Fourth, we manually configure programs and test suites to examine whether bugs can be reproduced. Note that the reproducibility rate is very low due to the complexity of the projects Math and Lang.

Table 1 summarizes the 22 bugs in two categories, i.e., bug types of buggy IF conditions and missing preconditions. We index these bugs according to their types and projects. A *bug index* (Column 3) is named based on the following rule. Letters *C* and *P* indicate bugs with buggy IF conditions and missing preconditions, respectively; *M* and *L* are bugs from Math and Lang, respectively. For instance, CM1 refers to a bug with a buggy IF condition in the project Math. We also record the number of executable Lines of Code (LoC, i.e., the number of lines that exclude empty lines and comment lines) for each source program (Column 6). Moreover, we show the number of classes, the number of methods in the buggy program, and the number of unit test cases (Columns 7-9). For each method that contains the buggy code, we describe the functionality of this method and record its Cyclomatic Complexity (Columns 10 and 11). The *Cyclomatic Complexity* [34] is the number of linearly independent paths through the source code of a method. This complexity indicates the testability of a method and the difficulty of understanding code by developers.

As shown in Table 1, the dataset contains 16 bugs with buggy IF conditions and 6 bugs with missing preconditions. Among these bugs, 12 bugs are from Math and 10 bugs are from Lang. In average, a buggy program consists of 25.48K executable lines of code. The average complexity is 8.6; that is, a buggy method consists of 8.6 independent paths in average. Note that the method complexity of Bug PM1 is 1 since its buggy method contains only one `throw` statement (which misses a precondition); the method complexity of Bug CM9 is 38 and its buggy method contains 30 IF statements.

### 4.3 Implementation Details

Our approach, NOPOL, is implemented with Java 1.7 on top of Spoon 3.1.0.[12] Spoon [40] is a library for transforming and analyzing Java source code. It is used for angelic fix localization, instrumentation, and final patch synthesis in our work. Fault localization is implemented with GZoltar 1.1.0.[13] GZoltar [6] is a fault localization library for ranking faulty statements.

6. Apache Commons Math, http://commons.apache.org/math/.
7. Apache Commons Lang, http://commons.apache.org/lang/.
8. Apache Subversion, http://subversion.apache.org/.
9. Jira for Apache, http://issues.apache.org/jira/.
10. FishEye for Apache, http://fisheye6.atlassian.com/.
11. In considered commits, bug fixes are always committed together with originally failing test cases (which are passed after fixing the bugs). This is a rule in Apache development process [3].
12. Spoon 3.1.0, http://spoon.gforge.inria.fr/.
13. GZoltar 1.1.0, http://gzoltar.com/.

TABLE 1
The Evaluation Dataset of NOPOL. It contains 22 bugs related to buggy IF conditions and missing preconditions.

| Bug type | Project | Bug description | | | Executable LoC | #Classes | #Methods | #Unit tests | Buggy method | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Index | Commit ID† | Bug ID‡ | | | | | Description | Complexity |
| Buggy IF condition | Math | CM1 | 141003 | - | 4611 | 153 | 947 | 363 | Returns a specified percentile from an array of real numbers | 7 |
| | | CM2 | 141217 | - | 5539 | 212 | 1390 | 543 | Returns an exact representation of the Binomial Coefficient | 8 |
| | | CM3 | 141473 | - | 6633 | 191 | 1504 | 654 | Returns the natural logarithm of the factorial for a given value | 3 |
| | | CM4 | 159727 | - | 7442 | 206 | 1640 | 704 | Computes the a polynomial spline function | 5 |
| | | CM5 | 735178 | Math-238 | 25034 | 468 | 3684 | 1502 | Gets the greatest common divisor of two numbers | 12 |
| | | CM6 | 791766 | Math-280 | 37918 | 632 | 5123 | 1982 | Finds two real values for a given univariate function | 11 |
| | | CM7 | 831510 | Math-309 | 38841 | 650 | 5275 | 2105 | Returns a random value from an Exponential distribution | 3 |
| | | CM8 | 1368253 | Math-836 | 64709 | 975 | 7374 | 3982 | Converts a double value into a fraction | 11 |
| | | CM9 | 1413916 | Math-904 | 70317 | 1037 | 7978 | 4263 | Computes a power function of two real numbers | 38 |
| | | CM10 | 1453271 | Math-939 | 79701 | 1158 | 9074 | 4827 | Checks whether a matrix has sufficient data to calculate covariance | 3 |
| | Lang | CL1 | 137231 | - | 10367 | 156 | 2132 | 781 | Replaces a string with another one inside | 4 |
| | | CL2 | 137371 | - | 11035 | 169 | 2240 | 793 | Removes a line break at the end of a string | 4 |
| | | CL3 | 137552 | - | 12852 | 173 | 2579 | 994 | Gets a sub-string from the middle of a string from a given index | 5 |
| | | CL4 | 230921 | - | 15818 | 215 | 3516 | 1437 | Finds the first matched string from the given index | 10 |
| | | CL5 | 825420 | Lang-535 | 17376 | 86 | 3744 | 1678 | Extracts the package name from a string | 6 |
| | | CL6 | 1075673 | Lang-428 | 18884 | 211 | 3918 | 1934 | Checks whether the char sequence only contains unicode letters | 4 |
| Missing precondition | Math | PM1 | 620221 | Math-191 | 16575 | 396 | 2803 | 1168 | Checks the status for calculating the sum of logarithms | 1 |
| | | PM2 | 1035009 | - | 44347 | 745 | 5536 | 2236 | Builds a message string from patterns and arguments | 3 |
| | Lang | PL1 | 504351 | Lang-315 | 17286 | 233 | 4056 | 1710 | Stops a process of timing | 3 |
| | | PL2 | 643953 | Lang-421 | 17780 | 240 | 4285 | 1829 | Erases a string with the Java style from the character stream | 19 |
| | | PL3 | 655246 | Lang-419 | 18533 | 257 | 4443 | 1899 | Abbreviates a given string | 9 |
| | | PL4 | 1142389 | Lang-710 | 18974 | 218 | 3887 | 1877 | Counts and translates the code point from an XML numeric entity | 19 |
| Average | | | | | 25480.6 | 399.1 | 3960.4 | 1784.6 | | 8.6 |
| Median | | | | | 17585.0 | 225.5 | 3818.5 | 1694.0 | | 5.5 |

† A commit ID is an identifier that indicates the commit of the patch, in both SVN and the FishEye system. According to this commit, we can manually check relevant patched code and test cases. For instance, the commit of Bug CM1 can be found at https://fisheye6.atlassian.com/changelog/commons?cs=141003.
‡ For some bugs, bug IDs are not obviously identified in the bug tracking system. These bugs can be found in the version control system. For example, Apache projects previously used Bugzilla as a bug tracking system before moving to Jira. The Bugzilla system is not available anymore.

The SMT solver inside NOPOL is Z3 4.3.2.[14] We generate SMT-LIB[15] files using jSMTLIB.[16] jSMTLIB [10] is a library for checking, manipulating, and translating SMT-LIB formatted problems. The test driver is JUnit 4.11. For future replication of the evaluation, the code of NOPOL is available on GitHub [2].

All experiments are run on a PC with an Intel Core i7 2.70 GHz CPU and an Ubuntu 12.04 operating system. The maximum heap size of Java virtual machine was set to 2.50 GB.

### 4.4 Main Research Questions

We present the general evaluation of NOPOL on the dataset via answering six Research Questions (RQs).

> RQ1: Can NOPOL fix real bugs in large-scale Java software?

In test-suite based repair, a bug is *fixed* if the patched program passes the whole test suite [28]. Table 2 presents the evaluation of patches on 22 bugs. Column 3 shows the buggy code (the condition for each bug with a buggy IF condition and the statement for each bug with a missing precondition). Column 4 shows the patches that were manually-written by developers as found in the version control system: the updated condition for each bug with a buggy IF condition and the added precondition for each bug with a missing precondition. Column 5 presents the generated patches by NOPOL. Column 6 is the result of our manual analysis of the correctness of the patches (will be explained in RQ2). Finally, Column 7 shows whether we had to modify existing test cases: "A" stands for additional test cases, "T" for transformed test cases, and "D" for deleted test cases. The purpose of test case modification is to yield a correct repair (will be explained in RQ3).

As shown in Table 2, among 22 bugs, NOPOL can fix 17 bugs: 13 out of 16 bugs with buggy IF conditions and 4 out of 6 bugs with missing preconditions. Meanwhile, four out of five unfixed bugs are related to timeout. In our work, the execution time of NOPOL is limited to up to five hours. We will empirically analyze the fixed bugs in Section 4.5 and explore the limitations of our approach as given by the five unfixed bugs in Section 4.6.

Table 2 also shows that patches generated by NOPOL consist of both primitive values and object-oriented features. For the object-oriented features, two major types can be found in the generated patches: nullness checking (patches of Bugs CL4 and PM2) and the `length()` method of strings (patches of Bugs CL1, CL2, CL3, and CL5).

Note that four bugs (Bugs CM2, CM3, CM6, and CM10) with buggy IF conditions are fixed by adding preconditions rather than updating conditions. One major reason is that a non-IF statement is ranked above the buggy IF statement during the fault localization; then NOPOL adds a patch, i.e., a precondition to this non-IF statement. Hence, the condition inside the buggy IF statement cannot be updated. This shows that those two kinds of patches intrinsically relate to

14. Z3, http://z3.codeplex.com/.
15. SMT-LIB, http://smt-lib.org/.
16. jSMTLIB, http://sourceforge.net/projects/jsmtlib/.

TABLE 2
Buggy code, manually-written patches, and generated patches for the bugs of the dataset.

| Bug type | Bug index | Buggy code | Patch written by developers | Patch generated by NOPOL | Correctness | Test case modification |
|---|---|---|---|---|---|---|
| Buggy IF condition | CM1 | `pos > n` | `pos >= n` | `length <= fpos` | Correct | A ‡ |
| | CM2 | `n <= 0` | `n < 0` | `if (n < 0) { ... †` | Correct | T |
| | CM3 | `n <= 0` | `n < 0` | `if (MathUtils.ZS != n) { ...` | Correct | A T |
| | CM4 | `v < knots[0] \|\| v >= knots[n]` | `v < knots[0] \|\| v > knots[n]` | `v < knots[0] \|\| knots[n] < v` | Correct | A |
| | CM5 | `u * v == 0` | `u == 0 \|\| v == 0` | `v == 0 \|\| MathUtils.ZB == u` | Correct | A T |
| | CM6 | `fa * fb >= 0.0` | `fa * fb > 0.0` | `if (-1 == b) { ...` | Incorrect | T D |
| | CM7 | `mean < 0` | `mean <= 0` | `mean <= 0.0` | Correct | T D |
| | CM8 | `p2 > overflow \|\| q2 > overflow` | `FastMath.abs(p2) > overflow \|\| FastMath.abs(q2) > overflow` | – (Timeout in SMT) | - | - |
| | CM9 | `y >= TWO_POWER_52 \|\| y <= -TWO_POWER_52` | `y >= TWO_POWER_53 \|\| y <= -TWO_POWER_53` | – (Timeout in test execution) | - | - |
| | CM10 | `nRows < 2 \|\| nCols < 2` | `nRows < 2 \|\| nCols < 1` | `if (1 != nCols) { ...` | Correct | - |
| | CL1 | `text == null` | `text == null \|\| repl == null \|\| with == null \|\| repl.length() == 0` | `with.length() != 0` | Incorrect | D |
| | CL2 | `lastIdx == 0` | `lastIdx <= 0` | `lastIdx < blanks.length()` | Correct | T |
| | CL3 | `pos > str.length()` | `len < 0 \|\| pos > str.length()` | `len <= -1 \|\| str.length() < pos` | Correct | T |
| | CL4 | `startIndex >= size` | `substr == null \|\| startIndex >= size` | `(startIndex >= size \|\| substr == null) && size != -1` | Correct | T D |
| | CL5 | `className == null` | `className == null \|\| className.length() == 0` | `className.length() == 0` | Incorrect | - |
| | CL6 | `cs == null` | `cs == null \|\| cs.length() == 0` | – (Timeout due to a `null` pointer) | - | - |
| Missing precondition | PM1 | `throw new IllegalStateException("")` | `if (getN() > 0) { ...` | – (No angelic value found) | - | - |
| | PM2 | `sb.append(": ")` | `if (specific != null) { ...` | `if (specific != null) { ...` | Correct | - |
| | PL1 | `stopTime ← System.currentTimeMillis()` | `if (this.runningState == STATE_RUNNING) { ...` | `if (StopWatch.STATE_RUNNING == runningState) {...` | Correct | - |
| | PL2 | `out.write('\\')` | `if (escapeForwardSlash) { ...` | `if (escapeSingleQuote) { ...` | Incorrect | - |
| | PL3 | `lower ← str.length()` | `if (lower > str.length()) { ...` | – (Timeout in SMT) | - | T |
| | PL4 | `return 0` | `if (start == seqEnd) { ...` | `if (start == seqEnd) { ...` | Correct | T |

† An added precondition is in a form of "`if( ) { ...`" to distinguish with an updated condition.
‡ For test case modification, "A" stands for additional test cases, "T" for transformed test cases, and "D" for deleted test cases.

each other. To further understand this phenomenon, we have performed repair only in the mode of "condition" in NOPOL: the four bugs could also be fixed via only updating IF conditions.

> RQ2: Are the synthesized patches as correct as the manual patches written by the developer?

In practice, a patch should be more than making the test suite pass since test cases may not be enough for specifying program behaviors [36], [45]. In this paper, a generated patch is *correct* if and only if the patch is functionally equivalent to the manually-written patch by developers.

For each synthesized patch, we have followed Qi et al. [45] to perform a manual analysis of its correctness. The manual analysis consists of understanding the domain (e.g., the mathematical function under test for a bug in the project Math), understanding the role of the patch in the computation, and understanding the meaning of the test case as well as its assertions.

As shown in Table 2, 13 out of 17 synthesized patches are as correct as the manual patches. Among these 13 correct patches, four patches (for Bugs CM1, CM3, CM4, and CM5) are generated based on not only the original test suite but also additional test cases. The reason is that the original test suite is too weak to drive the synthesis of a correct patch; then we had to manually write additional test cases (all additional test cases are publicly-available on the companion website [1][17]). This will be discussed in next research question.

For four bugs (Bugs CM6, CL1, CL5, and PL2), we are not able to synthesize a correct patch, even if we can write additional test cases. This will be further discussed in Section 5.4.

> RQ3: What is the root cause of test case modification?

As shown in Table 2, some bugs are correctly repaired only after the test case modification (including test case addition, transformation, and deletion). The most important modification is test case addition. Four bugs (Bugs CM1, CM3, CM4, and CM5) with additional test cases correspond to too weak specifications. We manually added test cases for these bugs to improve the coverage of buggy statements. Without the additional test case, the synthesized patch is degenerated. A case study of Bug CM1 (Section 4.5.4) will further illustrate how additional test cases help to synthesize patches. Note that all additional test cases appear in the bugs, which are reported in the early stage of the project Math. One possible reason is that the early version of Math is not in test-driven development and the test suites are not well-designed.

In test case transformation (Bugs CM2, CM3, CM5,

17. Additional test cases, http://sachaproject.gforge.inria.fr/nopol/dataset/data/projects/math/.

CM6, CM7, CL2, CL3, CL4, PL3, and PL4), we simply break existing test cases into smaller ones, in order to have one assertion per test case. This is important for facilitating angelic fix localization since our implementation of NOPOL has a limitation, which collects only one runtime trace for a test case (Section 3.2). We note that such test case transformation can even be automated [51].

The reason behind the four bugs with deleted test cases (Bugs CM6, CM7, CL1, and CL4) is accidental and not directly related to automatic repair: these deleted test cases are no longer compatible with the Java version and external libraries, which are used in NOPOL.

RQ4: How are the bugs in the dataset specified by test cases?

To further understand the repair process of bugs by NOPOL, Tables 3 and 4 show the detailed analysis for patched statements in 17 fixed bugs and buggy statements in five non-fixed bugs, respectively. Table 3 gives the following information: whether a synthesized patch is at the same location as the one written by the developer, the number of failing ($e_f$) and passing ($e_p$) test cases executing the patched statements, the fault localization metrics (the rank of the patched statement and the total number of suspicious statements), the overall execution time of NOPOL, and the SMT level (see Section 3.4.5). In Table 4, $e_f$ and $e_p$ denote the number of failing and passing test cases executing the buggy statements while the rank of the buggy statement is listed.

Tables 3 and 4 show the numbers of failing ($e_f$) and passing ($e_p$) test cases that execute the patched or buggy statements. Such numbers reflect to which extent the statement under repair is specified by test cases. As shown in Table 3, the average of $e_f$ and $e_p$ are 1.5 and 7.3 for 17 bugs with synthesized patches. In Table 4, the average of $e_f$ and $e_p$ are 5.2 and 27.4 for five bugs without patches.

For all 22 bugs under evaluation, only one bug has a large number of failing test cases ($e_f \geq 10$): Bug PL3 with $e_f = 18$. For this bug, although the buggy statement is ranked at the first place, NOPOL fails in synthesizing the patch. This failure is caused by an incorrectly identified output of a precondition. Section 4.6.3 will explain the reason behind this failure.

RQ5: Where are the synthesized patches localized? How long is the repair process?

A patch could be localized in a different location from the patch which is manually-written by developers. We present the details for patch locations for all the 17 patched bugs in Table 3. For 13 out of 17 fixed bugs, the patched statements (i.e., the location of the fix) are exactly the same as the buggy ones. For the other four bugs, i.e., Bugs CM2, CM3, CM6, and CM10, NOPOL generates patches by adding new preconditions rather than updating existing conditions, as mentioned in Table 2.

For 17 fixed bugs in Table 3, the average execution time of repairing one bug is 27.5 seconds while for five non-fixed bugs in Table 4, four bugs are run out of time and the other one spends 41 seconds. The execution time of all the 22 bugs ranges from 7 to 88 seconds. We consider that such execution time, i.e., fixing one bug within 90 seconds, is acceptable.

In practice, if applying NOPOL to a buggy program, we can directly set a timeout, e.g., 90 seconds (over 88 seconds as shown in Table 3) or a longer timeout like five hours in our experiment. Then for any kind of buggy program (without knowing whether the bug is with a buggy condition or a missing precondition), NOPOL will synthesize a patch, if it finds any. Then a human developer can check whether this patch is correct from the user perspective.

TABLE 3
Analysis of the 17 fixed bugs (patched statement).

| Bug type | Bug index | Patch location | #Test cases $e_f$† | #Test cases $e_p$† | **Patched** statement rank | #Suspicious statements ‡ | Execution time (seconds) | SMT level |
|---|---|---|---|---|---|---|---|---|
| Buggy IF condition | CM1 | Same as dev. | 1 | 7 | 3 | 28 | 10 | 2 |
| | CM2 | Different | 1 | 1 | 171 | 584 | 17 | 2 |
| | CM3 | Different | 1 | 1 | 2 | 12 | 7 | 2 |
| | CM4 | Same as dev. | 4 | 2 | 71 | 116 | 23 | 3 |
| | CM5 | Same as dev. | 1 | 2 | 1 | 2 | 50 | 3 |
| | CM6 | Different | 1 | 1 | 36 | 189 | 57 | 2 |
| | CM7 | Same as dev. | 3 | 10 | 36 | 74 | 64 | 2 |
| | CM10 | Different | 2 | 0 | 1 | 104 | 18 | 1 |
| | CL1 | Same as dev. | 2 | 4 | 2 | 9 | 33 | 2 |
| | CL2 | Same as dev. | 1 | 8 | 2 | 4 | 9 | 2 |
| | CL3 | Same as dev. | 2 | 10 | 4 | 5 | 13 | 3 |
| | CL4 | Same as dev. | 2 | 23 | 1 | 20 | 12 | 3 |
| | CL5 | Same as dev. | 1 | 37 | 1 | 2 | 39 | 1 |
| Missing precondition | PM2 | Same as dev. | 1 | 1 | 1 | 12 | 11 | 1 |
| | PL1 | Same as dev. | 1 | 14 | 6 | 22 | 88 | 2 |
| | PL2 | Same as dev. | 1 | 1 | 1 | 18 | 8 | 1 |
| | PL4 | Same as dev. | 1 | 2 | 1 | 26 | 8 | 2 |
| Median | | | 1 | 2 | 2 | 20 | 17 | 2 |
| Average | | | 1.5 | 7.3 | 20.0 | 72.2 | 27.5 | 2 |

† $e_f$ and $e_p$ denote the number of failing and passing test cases that execute the **patched** statement.
‡ #Suspicious statements denotes the number of statements whose suspiciousness scores by fault localization are over zero.

TABLE 4
Analysis of the 5 non-fixed bugs (buggy statement).

| Bug type | Bug index | #Test cases $e_f$† | #Test cases $e_p$† | **Buggy** statement rank | #Suspicious statements ‡ | Execution time (seconds) |
|---|---|---|---|---|---|---|
| Buggy IF condition | CM8 | 1 | 51 | 1206 | 1296 | - |
| | CM9 | 2 | 73 | 625 | 705 | - |
| | CL6 | 1 | 10 | 4 | 4 | - |
| Missing precondition | PM1 | 4 | 0 | 3 | 132 | 41 |
| | PL3 | 18 | 3 | 1 | 16 | - |
| Median | | 2 | 10 | 4 | 132 | - |
| Average | | 5.2 | 27.4 | 367.8 | 430.6 | - |

† $e_f$ and $e_p$ denote the number of failing and passing test cases that execute the **buggy** statement.
‡ #Suspicious statements denotes the number of statements whose suspiciousness scores by fault localization are over zero.

RQ6: How effective is fault localization for the bugs in the dataset?

Fault localization is an important step in our repair approach. As shown in Table 3, for patched statements in 17 fixed bugs, the average fault localization rank is 20.0. In four out of 17 bugs (Bugs CM2, CM4, CM6, and CM7), patched statements are ranked over 30. This fact indicates that there is room for improving the fault localization techniques. Among the five unfixed bugs, the buggy statements of Bugs CM8 and CM9 are over 600. Section 5.2 will further compare the effectiveness of six fault localization techniques on 22 bugs.

Note that in Tables 3 and 4, Bugs CM10 and PM1 have no passing test cases. Bug PM1 cannot be fixed by our approach while Bug CM10 can be still fixed because the two failing test cases give a non-trivial input-output specification. The reason behind the unfixed Bug PM1 is not $e_p = 0$, but the multiple executions of the buggy code by one test case. This reason will be discussed in Section 4.6.1.

### 4.5 Case Studies for Fixed Bugs

We conduct four case studies to show how NOPOL fixes bugs with buggy IF conditions and missing preconditions. These bugs are selected because they highlight different facets of the repair process. The patch of Bug PL4 (Section 4.5.1) is syntactically the same as the manually-written one; the patch of Bug CL4 (Section 4.5.2) is as correct as the manually-written patch (beyond passing the test suite), but different from the manually-written one; the patch of Bug CM2 (Section 4.5.3) is correct by adding a pre-condition rather than updating the buggy condition, as written by developers; and the patch of Bug CM1 (Section 4.5.4) requires additional test cases.

#### 4.5.1 Case Study 1, Bug PL4

NOPOL can generate the same patches for four out of 22 bugs as the manually-written ones. We take Bug PL4 as an example to show how the same patch is generated. This bug is fixed by adding a precondition. Fig. 4 shows a method `translate()` at Line 1 and the buggy method `translateInner()` at Line 9 of Bug PL4. The method `translate()` is expected to translate a term in the regular expression of `&#[xX]?\d+;?` into codepoints, e.g., translating the term "&#x30" into "\u0030".

To convert from `input` to codepoints, the characters in `input` are traversed one by one. Note that for a string ending with "&#x", no codepoint is returned. Lines 15 to 20 in Fig. 4 implement this functionality. However, the implementation at Lines 17 to 19 ignores a term in a feasible form of "&#[xX]\d+", e.g., a string like "&#x30". A precondition should be added to detect this feasible form, i.e., the comment at Line 18 of `start == seqEnd`.

```
1  String translate(CharSequence input, int index) {
2    int consumed = translateInner(input, index);
3    if(consumed == 0)
4      ... // Return the original input value
5    else
6      ... // Translate code points
7  }
8
9  int translateInner(CharSequence input, int index) {
10   int seqEnd = input.length();
11   ...
12   int start = index + 2;
13   boolean isHex = false;
14   char firstChar = input.charAt(start);
15   if(firstChar == 'x' || firstChar == 'X') {
16     start++;
17     isHex = true;
18 // FIX: if(start == seqEnd)
19     return 0;
20   }
21   int end = start;
22   //Traverse the input and parse into codepoints
23   while(end < seqEnd && ... )
24   ...
25 }
```

Fig. 4. Code snippet of Bug PL4. The manually-written patch is shown in the `FIX` comment at Line 18. Note that the original method `translate` consists of three overloaded methods; for the sake of simplification, we use two methods `translate` and `translateInner` instead.

TABLE 5
Sample of test cases for Bug PL4

| Input | | Output, `translate(input)` | | Test result |
|---|---|---|---|---|
| `input` | `index` | Expected | Observed | |
| "Test &#x" | 5 | "Test &#x" | "Test &#x" | Pass |
| "Test &#X" | 5 | "Test &#X" | "Test &#X" | Pass |
| "Test &#x30 not test" | 5 | "Test \u0030 not test" | "Test &#x30 not test" | Fail |

The buggy code at Line 19 is executed by two passing test cases and one failing test case. Table 5 shows these three test cases. For the two passing test cases, the behavior of the method is expected not to change the variable `input` while for the failing test case, the `input` is expected to be converted. In the passing test cases, the value of the precondition of the statement at Line 19 is expected to be `true`, i.e., both `start` and `seqEnd` equal to `8`, while in the failing test case, the condition is expected to be `false`, i.e., `start` and `seqEnd` are `8` and `19`, respectively. The `false` value is the angelic value for a missing precondition.

According to those expected precondition values for test cases, NOPOL generates a patch via adding a pre-condition, i.e., `start == seqEnd`, which is exactly the same as the manually-written patch by developers. Besides the patch of Bug PL4, patches of Bugs CM4,

CM7, and PM2 are also syntactically the same as the patch written by developers, among 22 bugs in our dataset.

#### 4.5.2 Case Study 2, Bug CL4

For several bugs, NOPOL generates patches that are literally different from the manually-written patches, but these generated patches are as correct as manully-written patches. In this section, we present a case study where NOPOL synthesizes a correct patch for a bug with a buggy IF condition. Bug CL4 in Lang fails to find the index of a matched string in a string builder. Fig. 5 presents the buggy method of Bug CL4: to return the first index of `substr` in a parent string builder from a given basic index `startIndex`. The condition at Line 4 contains a mistake of `startIndex >= size`, which omits checking whether `substr == null`. A variable `size` is defined as the length of the parent string builder. The manually-written fix is shown at Line 3.

The buggy code at Line 4 in Fig. 5 is executed by 23 passing test cases and two failing test cases. One of the passing test cases and two failing test cases are shown in Table 6. For the passing test case, a value `-1` is expected because no matched string is found. For the two failing test cases, each input `substr` is a `null` value, which is also expected to return a non-found index `-1`. This requires the checking of `null` to avoid `NullPointerException`, i.e., the condition at Line 3.

For the passing test case in Table 6, the condition at Line 4 is `false`. For the two failing test cases, NOPOL extracts the angelic value `true` to make both failing test cases pass. According to these condition values, a patch of `(startIndex >= size || substr == null) && size != -1` can be synthesized. This patch is different from the manually-written one at Line 3. The difference is the additional `size != -1`. However, from the context in the method, the length of the parent string builder is no less than zero, i.e., `size >= 0` holds. The value of `size != -1` is always `true`. Then the synthesized patch is equivalent to `startIndex >= size || substr == null`, which is correct. The reason for the unnecessary expression in the patch (i.e., `&& size != -1`) is that a constant value `-1` is collected either as runtime data or as standard primitive data by NOPOL (see Section 3.3.2). This value is then encoded in the SMT instance. The resulted expression based on the solution to the SMT does not weaken the repairability of synthesized patches since it does not result in more failing test cases. A recent method for finding simplified patches, proposed by Mechtaev et al. [35], could be used to avoid such a redundant expression.

#### 4.5.3 Case Study 3, Bug CM2

In this section, we present Bug CM2, a correctly patched bug via adding a precondition, rather than updating an existing condition, as written by developers. The buggy method in Bug CM2 is to calculate the value of Binomial Coefficient by choosing `k`-element subsets from an `n`-element set. Fig. 6 presents the buggy method. The input number of elements `n` should be no less than zero. But the condition at Line 4 reads `n <= 0` instead of `n < 0`. The manually-written patch by developers is in the `FIX` comment at Line 4.

The buggy code at Line 4 in Fig. 6 is executed by one passing test case and one failing test case. Table 7 shows these two test cases. For the passing test case, an expected exception is observed; for the failing test case, an `IllegalArgumentException` is thrown rather than an expected value.

To fix this bug, NOPOL generates a patch via adding a missing precondition `n < 0` to the statement at Line 5. Then this statement owns two embedded preconditions, i.e., `n <= 0` and `n < 0`. Hence, the generated patch is equivalent to the manually-written patch, i.e.,

```
1  int indexOf(String substr, int startIndex) {
2    startIndex = (startIndex < 0 ? 0 : startIndex);
3  // FIX: if (substr == null || startIndex >= size) {
4    if (startIndex >= size) {
5      return −1;
6    }
7    int strLen = substr.length();
8    if (strLen > 0 && strLen <= size) {
9      if (strLen == 1)
10       return indexOf(substr.charAt(0), startIndex);
11     char[] thisBuf = buffer;
12     outer:
13     for (int i = startIndex; i < thisBuf.length
14               − strLen; i++) {
15       for (int j = 0; j < strLen; j++) {
16         if (substr.charAt(j) != thisBuf[i + j])
17           continue outer;
18       }
19       return i;
20     }
21   } else if (strLen == 0) {
22     return 0;
23   }
24   return −1;
25 }
```

Fig. 5. Code snippet of Bug CL4. The manually-written patch is shown in the `FIX` comment at Line 3, which updates the buggy IF condition at Line 4.

TABLE 6
Sample of test cases for Bug CL4

| Input | | | Output, `indexOf(substr, startIndex)` | | Test result |
|---|---|---|---|---|---|
| parent | substr | startIndex | Expected | Observed | |
| abab | z | 2 | -1 | -1 | Pass |
| abab | (String) null | 0 | -1 | `NullPointerException` | Fail |
| xyzabc | (String) null | 2 | -1 | `NullPointerException` | Fail |

```
1  long binomialCoefficient(int n, int k) {
2    if (n < k)
3      throw new IllegalArgumentException(...);
4    if (n <= 0) // FIX: if (n < 0)
5      throw new IllegalArgumentException(...);
6    if ((n == k) || (k == 0))
7      return 1;
8    if ((k == 1) || (k == n - 1))
9      return n;
10   long result = Math.round(
11       binomialCoefficientDouble(n, k));
12   if (result == Long.MAX_VALUE)
13     throw new ArithmeticException(...);
14   return result;
15 }
```

Fig. 6. Code snippet of Bug CM2. The manually-written patch is shown in the `FIX` comment at Line 4.

TABLE 7
Sample of test cases for Bug CM2

| Input | | Output, `binomialCoefficient(n,k)` | | Test |
|---|---|---|---|---|
| `n` | `k` | Expected | Observed | result |
| -1 | -1 | Exception | Exception | Pass |
| 0 | 0 | 1 | Exception | Fail |

updating the condition at Line 4 from `n <= 0` to `n < 0`. The reason of adding a precondition instead of updating the original condition is that the statement at Line 5 is ranked prior to the statement at Line 4. This has been explained in Section 4.4. Consequently, the generated patch of Bug CM2 is correct and syntactically equivalent to the manually-written patch.

#### 4.5.4 Case Study 4, Bug CM1

Insufficient test cases lead to trivial patch generation. We present Bug CM1 in Math, with a buggy IF condition. This bug cannot be correctly patched with the original test suite due to the lack of test cases. In our work, we add two test cases to support the patch generation. Fig. 7 presents the buggy source code in the method `evaluate()` of Bug CM1. This method returns an estimate of the percentile `p` of the values stored in the array `values`.

According to the API document, the algorithm of `evaluate()` is implemented as follows. Let `n` be the length of the (sorted) array. The algorithm computes the estimated percentile position `pos = p * (n + 1) / 100` and the difference `dif` between `pos` and `floor(pos)`. If `pos >= n`, then the algorithm returns the largest element in the array; otherwise the algorithm returns the final calculation of percentile. Thus, the condition at Line 15 in Fig. 7 contains a bug, which should be corrected as `pos >= n`.

As shown in Table 3, this bug is executed by seven passing test cases and one failing test case.

```
1  double evaluate(double[] values, double p) {
2    ...
3    int length = values.length;
4    double n = length;
5    ...
6    double pos = p * (n + 1) / 100;
7    double fpos = Math.floor(pos);
8    int intPos = (int) fpos;
9    double dif = pos - fpos;
10   double[] sorted = new double[n];
11   System.arraycopy(values, 0, sorted, 0, n);
12   Arrays.sort(sorted);
13   if (pos < 1)
14       return sorted[0];
15   if (pos > n) // FIX: if (pos >= n)
16       return sorted[n - 1];
17   double lower = sorted[intPos - 1];
18   double upper = sorted[intPos];
19   return lower + dif * (upper - lower);
20 }
```

Fig. 7. Code snippet of Bug CM1. The manually-written patch is shown in the `FIX` comment at Line 15.

TABLE 8
Two original test cases and two additional test cases for Bug CM1

| Input | | Output, `evaluate(values,p)` | | Test |
|---|---|---|---|---|
| `values` | `p` | Expected | Observed | result |
| Two original test cases | | | | |
| {0,1} | 25 | 0.0 | 0.0 | Pass |
| {1,2,3} | 75 | 3.0 | Exception | Fail |
| Two additional test cases | | | | |
| {1,2,10} | 75 | 10.0 | Exception | Fail |
| {1,2,10,15,75} | 100 | 75.0 | 75.0 | Pass |

Table 8 shows one of the seven passing test cases and the failing test case. In the failing test case, an `ArrayIndexOutOfBounds` exception is thrown at Line 16. For the passing test case, the value of the condition at Line 15 is equal to the value of the existing condition `pos > n`, i.e., `true`; for the failing test case, setting the condition to be `true` makes the failing test case pass; that is, the angelic value for the failing test case is also `true`. Thus, according to these two test cases, the generated patch should make the condition be `true` to pass both test cases.

With the original test suite, NOPOL generates a patch as `length == intPos`, which passes all test cases. This patch is incorrect. To obtain a correct patch (the one shown in Table 2), we add two test cases of `values` ← `{1,2,10}`, `p` ← `75` and `values` ← `{1,2,10,15,75}`, `p` ← `100`, as shown in Table 8. Then the expected values of `evaluate()` are `10.0` and `75.0`, respectively. After running NOPOL, a patch of `length <= fpos`, which is different from the

TABLE 9
Summary of five limitations.

| Bug index | Root cause | Result of repair | Reason for the unfixed bug |
|---|---|---|---|
| PM1 | Angelic fix localization | No angelic value found. Termination before runtime trace collection and patch synthesis | One failing test case executes the missing precondition for more than once. |
| CM9 | Angelic fix localization | Timeout during test suite execution | An infinite loop is introduced during the trial of angelic values. |
| PL3 | Runtime trace collection | Timeout during SMT solving | The expected value of a precondition is incorrectly identified. |
| CM8 | Patch synthesis | Timeout during SMT solving | A method call with parameters is not handled by SMT. |
| CL6 | Patch synthesis | Timeout during SMT solving | A method of a `null` object yields an undefined value for SMT. |

manually-written one (`pos >= n`). However, from the source code at Line 7, `fpos` is the `floor()` value of `pos`, i.e., `fpos` is the largest integer that no more than `pos`. That is, `fpos <= pos`. Meanwhile, `n == length` holds according to Line 4. As a result, the generated patch `length <= fpos` implies the manually-written one, i.e., `pos >= n`. We can conclude that NOPOL can generate a correct patch for this bug by adding two test cases.

### 4.6 Limitations

As shown in Section 4.4, we have collected five bugs that reveal five different limitations of NOPOL. Table 9 lists these five bugs in details. We analyze the related limitations in this section.

#### 4.6.1 No Angelic Value Found

In our work, for a buggy IF condition, we use angelic fix localization to flip the boolean value of conditions for failing test cases. For Bug PM1, no angelic value is found as shown in Table 2. The reason is that both `then` and `else` branches of the IF condition are executed by one failing test case. Hence, no single angelic value (`true` or `false`) can enable the test case to pass. As discussed in Section 3.2.3, the search space of a sequence of angelic values is exponential and hence discarded in our implementation of NOPOL.

To mitigate this limitation, a straightforward solution is to discard the failing test case, which leads to no angelic values (keeping the remaining failing ones). However, this may decrease the quality of the generated patch due to the missing test data and oracles. Another potential solution is to refactor test cases into small snippets, each of which covers only `then` or `else` branches [51]. A recent proposed technique SPR [30] could help NOPOL to enhance its processing of sequential angelic values.

#### 4.6.2 Performance Bugs Caused by Angelic Values

NOPOL identifies angelic values as the input of patch synthesis. In the process of angelic fix localization, all failing test cases are executed to detect conditional values that make failing test cases pass (see Section 3.2). However, sometimes the trial of angelic fix localization (forcing to `true` or `false`) may result in a performance bug. In our work, Bug CM9 cannot be fixed due to this reason, i.e., an infinite loop caused by angelic fix localization.

A potential solution to this issue is to set a maximum execution time to avoid the influence of performance bugs. But a maximum execution time of test cases may be hard to be determined according to different test cases. For instance, the average execution time of test cases in Math 3.0 is much longer than that in Math 2.0. We leave the setting of maximum execution time as one piece of future work.

#### 4.6.3 Incorrectly Identified Output of a Precondition

As mentioned in Section 3.3.1, the expected output of a missing precondition is set to be `true` for a passing test case and is set to be `false` for a failing one. The underlying assumption for a passing test case is that the `true` value keeps the existing program behavior. However, it is possible that given a statement, both `true` and `false` values can make a test case pass. In these cases, synthesis may not work for bugs with missing preconditions.

This is what happens to Bug PL3. Fig. 8 shows a code snippet of Bug PL3. The manually-written patch is a precondition at Line 3; Table 10 shows one passing test case and one failing test case. Based on the angelic fix localization in Algorithm 2, the expected precondition values of all passing test cases are set to be `true`. However, in the manually-written patch, the precondition value by the passing test case in Table 10 is `false`, i.e., `lower > str.length()` where `lower` is `0` and `str.length()` is `10`. Thus, it is impossible to generate a patch like the manually-written one, due to a conflict in the input-output specification. Consequently, in the phase of patch synthesis, the SMT solver executes with timeout.

The example in Bug PL3 implies that for some bugs, the assumption (i.e., a missing precondition is expected to be `true` for passing test cases) can be violated. For Bug PL3, we have temporarily removed this assumption and only used the failing test cases to synthesize a patch. The resulting patch is `if(lower >= str.length()) lower = str.length()`, which has the same program behavior as the manually-written patch (`lower > str.length()`). In NOPOL, we are conservative and assume that the expected value of a precondition by passing test cases is `true` (in Section 3.2.2).

```
1  String abbreviate(String str, int lower, int upper){
2     ...
3     // FIX: if (lower > str.length())
4     lower = str.length();
5
6     if (upper == −1 || upper > str.length())
7       upper = str.length();
8     if (upper < lower)
9       upper = lower;
10    StringBuffer result = new StringBuffer();
11    int index = StringUtils.indexOf(str, "␣", lower);
12    if (index == −1)
13      result.append(str.substring(0, upper));
14    else ...
15    return result.toString();
16 }
```

Fig. 8. Code snippet of Bug PL3. The manually-written patch is shown in the `FIX` comment at Line 3.

TABLE 10
Sample of test cases for Bug PL3

| Input | | | Output, `abbreviate(str,lower,upper)` | | Test result |
|---|---|---|---|---|---|
| `str` | `lower` | `upper` | Expected | Observed | |
| "0123456789" | 0 | -1 | "0123456789" | "0123456789" | pass |
| "012 3456789" | 0 | 5 | "012" | "012 3456789" | fail |

#### 4.6.4 Complex Patches using Method Calls with Parameters

In our work, we support the synthesis of conditions that call unary methods (without parameters). However, our approach cannot generate a patch if a method with parameters has to appear in a condition. For example, for Bug CM8, the patch that is written by developers contains a method `abs(x)` for computing the absolute value. Our approach cannot provide such kinds of patches because methods with parameters cannot be directly encoded in SMT. Then the lack of information of method calls leads to the timeout of an SMT solver.

A workaround would generate a runtime variable to collect existing side-effect free method calls with all possible parameters. For example, one could introduce a new variable `double tempVar = abs(x)` and generate a patch with the introduced variable `tempVar`. However, this workaround suffers from the problem of combinatorial explosion.

#### 4.6.5 Unavailable Method Values for a Null Object

Our repair approach can generate a patch with objected-oriented features. For example, a patch can contain state query methods on Java core library classes, such as `String.length()`, `File.exists()` and `Collection.size()`. We map these methods to their return values during the SMT encoding. However, such methods require that the object is not `null`; otherwise, a `null` pointer exception in Java is thrown.

Let us consider Bug CL6, whose manually-written patch is `cs == null || cs.length() == 0`. For this bug, one passing test case detects whether the object `cs` is `null`. For this test case, the value of `cs.length()` is undefined and not given to SMT. Thus, it is impossible to generate a patch, which contains `cs.length()` if `cs` is `null` by at least one test case. Consequently, the SMT solver times-out because it tries to find a complex patch that satisfies the constraints.

A possible solution is to encode the undefined values in the SMT. Constraints should be added to ensure that the unavailable values are not involved in the patch. This needs important changes in the design of the encoding, which is left to future work.

## 5 Discussions

We now discuss NOPOL features with respect to four important aspects.

### 5.1 Differences with SemFix

As mentioned in Section 3.4, NOPOL uses the same technique in the phase of patch synthesis as SemFix [38], i.e., component-based program synthesis [20]. However, there exist a number of important differences between NOPOL and SemFix.

First, SemFix does not address missing preconditions. As shown in Table 2, adding preconditions enables us to repair more bugs than only updating conditions. We think that it is possible to extend the SemFix implementation to support repairing missing preconditions via adding the encoding strategy as in NOPOL.

Second, NOPOL does not use symbolic execution to find an angelic value. We have not evaluated the impact of this choice because SemFix is not publicly available. But it is known that symbolic execution may have difficulties with the size and complexity of analyzed programs [29]. According to our work, we make the following observations. The angelic value in angelic fix localization is possible only when the domain is finite. For booleans, the domain of variables is not only finite but also very small. This results in a search space that can be explored dynamically and exhaustively as in NOPOL. Symbolic execution as done in SemFix is capable of also reasoning on integer variables, because the underlying constraint solver is capable of exploring the integer search space. To sum up, our analytical answer is that for boolean domains, angelic fix localization is possible and probably much faster (this is claimed, but not empirically verified). For integer domains, only symbolic execution is appropriate. Meanwhile, NOPOL can also handle the right-hand side of assignments as in SemFix. If we encode the synthesis as in SemFix, the right-hand side of assignments can be directly processed by NOPOL.

Third, NOPOL supports object-oriented code. We have adapted the code synthesis technique so that the

generated patch can contain null checks and method calls.

Finally, the evaluation of NOPOL is significantly larger than that of SemFix. We have run NOPOL on larger programs and real bugs. In SemFix, 4/5 of subject programs have less than 600 LoC and the bugs are artificially seeded. In the evaluation in our paper, the average number of lines of code per subject program is 25K LoC and the bugs are extracted from real programs that happened in practice.

### 5.2 Effectiveness of Fault Localization Techniques

Fault localization plays an important role during the repair process. In our approach, a fault localization technique ranks all the suspicious statements and NOPOL attempts to generate patches by starting with analyzing the most suspicious statement first. We use Ochiai [4] as the fault localization technique. For our dataset, we wonder whether there is a difference between Ochiai and other techniques. In this section, we study the accuracy of different fault localization techniques on the bugs of our dataset.

We employ the absolute wasted effort to measure fault localization techniques. The wasted effort is defined as the ranking of the actual buggy statement. Given a set $S$ of statements, the wasted effort is expressed as follows,

$$effort = |\{susp(x) > susp(x^*)\}| + 1$$

where $x \in S$ is any statement, $x^*$ is the actual buggy statement, and $|\cdot|$ calculates the size of a set. A low value indicates that the fault localization technique is effective.

In our experiment, we compare six well-studied fault localization techniques: Ochiai [4], Tarantula [22], Jaccard [4], Naish2 [37], Ochiai2 [37], and Kulczynski2 [50]. Table 11 presents the comparison on the two types of bugs considered in this paper.

As shown in Table 11, for bugs with buggy IF conditions, Tarantula obtains the best average wasted effort while all techniques except Kulczynshi2 get the same median value. For bugs with missing preconditions, Ochiai, Jaccard, and Naish2 obtain the best average and median values. Those results assess that, according to our dataset of real bugs, the sensitivity of NOPOL with respect to fault localization is not a crucial point and Ochiai is an acceptable choice.

### 5.3 Potential Parallelization of NOPOL

Our method NOPOL is originally implemented to not perform parallelization. Based on the design of this method, it is possible to enhance the implementation by parallelizing NOPOL to reduce the execution time. Indeed, the core algorithms of NOPOL are highly parallelizable.

TABLE 11
Wasted effort comparison among six fault localization techniques.

| Fault localization technique | Buggy IF condition | | Missing precondition | |
|---|---|---|---|---|
| | Average | Median | Average | Median |
| Ochiai | 172.31 | **7.50** | **2.17** | **1.00** |
| Tarantula | **146.06** | **7.50** | 2.33 | 1.50 |
| Jaccard | 159.63 | **7.50** | **2.17** | **1.00** |
| Naish2 | 180.56 | **7.50** | **2.17** | **1.00** |
| Ochiai2 | 171.06 | **7.50** | 2.33 | 1.50 |
| Kulczynshi2 | 159.13 | 12.00 | 7.50 | 4.50 |

First, during angelic fix localization, two feasible ways of parallelization can be performed: over test cases and over potential locations. Let us assume that there are 3 failing test cases with respectively 50, 100 and 200 buggy IF conditions executed. Since the search space of buggy IF conditions is $2 \times n_c$ ($n_c$ is the number of executed IF statements by one test case, see Section 3.2.3), we could automatically run in parallel $(50 + 100 + 200) \times 2 = 700$ sessions of angelic fix localization on many different machines.

Second, our synthesis technique is based on different SMT levels (see Section 3.4.5). Synthesis at each SMT level corresponds to one independent SMT instance. Hence, the synthesis can also be run in parallel. However, we should mention that parallelizing the synthesis may lead to multiple resulted patches. Synthesis at a low SMT level can generate a simple patch; for the same bug, synthesis at a higher SMT level may not generate a better patch and may waste the running cost.

### 5.4 Insight on Test-Suite Based Repair

NOPOL is a test-suite based repair approach, as other existing work ([28], [38], [25], [44], etc.) in the field of automatic software repair. However, the foundations of test-suite based repair are little understood. Our experience with NOPOL enables us to contribute to better understanding the strengths and the weaknesses of test-suite based repair.

There are two grand research questions behind test-suite based repair. The first one is about the quality of test suites [36]: do developers write good-enough test suites for automatic repair? Qi et al. [45] have shown that the test suites considered in the GenProg benchmark are not good enough, in the sense that they accept trivial repairs such as directly removing the faulty code. The experiments we have presented in this paper shed a different light. For nine bugs considered in this experiment, the test suite leads to a correct patch. For four additional bugs, a slight addition in the test suite allows for generating a correct patch. We consider this as encouraging for the future of the field. However, there is a need for future work on recommendation systems that tell when to

add additional test cases for the sake of repair, and what those test cases should specify.

The second grand research question goes beyond standard test suites such as JUnit ones and asks whether repair operators do not overfit the inputs of input-output specifications [46]. For instance, for Bug CM6, one of the test inputs always equals to $-1$ when the buggy code is executed. As a result, the patch simply uses this value (corresponding to the number of rows) to drive the control flow, which is wrong. On the other hand, there are other cases when the repair operator yields a generic and correct solution upfront. This fact indicates that the same repair operator may overfit or not according to different bugs. It is necessary to conduct future research on the qualification of repair operators according to overfitting.

# 6 Threats to Validity

We discuss the threats to the validity of our results along four dimensions.

## 6.1 External Validity

In this work, we evaluate our approach on 22 real-world bugs with buggy IF conditions and missing preconditions. One threat to our work is that the number of bugs is not large enough to represent the actual effectiveness of our technique. While the number of bugs in our work is fewer than that in previous work [28], [38], [25], the main strength of our evaluation is twofold. On one hand, our work focuses on two specific types of bugs, i.e., buggy IF conditions and missing preconditions (as opposed to general types of bugs in [25]); on the other hand, our work is evaluated on real-world bugs in large Java programs (as opposed to bugs in small-scale programs in [38] and bugs without object-oriented features in [28]). We note that it is possible to collect more real-world bugs, with the price of more human labor. As mentioned in Section 4.2, reproducing a specific bug is complex and time-consuming.

## 6.2 Single Point of Repair

As all previous works in test-suite based repair, the program under repair must be repaired at one single point. In the current implementation of NOPOL, we do not target programs with multiple faults, or bugs which require patches at multiple locations.

## 6.3 Test Case Modification

In our work, we aim to repair bugs with buggy IF conditions and missing preconditions. Test cases are employed to validate the generated patch. In our experiment, several test cases are modified to facilitate repair. As mentioned in Section 4.4, such test case modification consists of test case addition, transformation, and deletion. The answer to RQ3 analyzes the root causes of test case modification. All test case modifications are listed in our project website [1].

## 6.4 Dataset Construction

We describe how to construct our dataset in Section 4.2. The manually-written patches of conditional statements are extracted from commits in the version control system. However, it is common that a commit contains more code than the patch in buggy IF conditions and missing preconditions. In our work, we manually separate these patch fragments. In particular, the fixing commit of Bug PM2 contains two nested preconditions within a complex code snippet. We manually separate the patch of this bug according to the code context and keep only one precondition. Hence, there exists a potential bias in the dataset construction.

# 7 Related Work

We list related work in four categories: approaches to test-suite based repair, repair besides test-suite based repair, empirical foundations of test-suite based repair, and related techniques in NOPOL.

## 7.1 Test-Suite Based Repair

**GenProg**. Test-suite based repair generates and validates a patch with a given test suite. Le Goues et al. [28] propose GenProg, an approach to test-suite based repair using genetic programming for C programs. In GenProg, a program is viewed as an Abstract Syntax Tree (AST) while a patch is a newly-generated AST by weighting statements in the program. Based on genetic programming, candidate patches are generated via multiple trials. The role of genetic programming is to obtain new ASTs by copying and replacing nodes in the original AST. A systematic study by Le Goues et al. [26] shows that GenProg can fix 55 out of 105 bugs in C programs. The difference between NOPOL and GenProg are as follows. NOPOL targets a specific defect class while GenProg is generic; NOPOL uses component-based program synthesis while GenProg only copies existing code from the same code base; NOPOL uses a four-phase repair approach (fault localization, angelic fix localization, runtime trace collection, and patch synthesis) while GenProg uses a different two-phase approach (fault localization and trial); NOPOL is designed for object-oriented Java programs while GenProg is for C.

**AE**. Weimer et al. [49] report an adaptive repair method based on program equivalence, called AE. This method can fix 54 out of the same 105 bugs as in the work [26] while evaluating fewer test cases than GenProg.

**PAR**. Kim et al. [25] propose PAR, a repair approach using fix patterns representing common ways of fixing bugs in Java. These fix patterns can avoid nonsensical patches, which are caused by the randomness of some operators in genetic programming. Based on the fix patterns, 119 bugs are examined for patch generation.

In this work, the evaluation of patches is contributed by 253 human subjects, including 89 students and 164 developers.

**RSRepair**. Qi et al. [44] design RSRepair, a random search based technique for navigating the search space. This work indicates that random search performs more efficiently than genetic programming in GenProg [28]. RSRepair can fix 24 bugs, which are derived from a subset of 55 fixed bugs by GenProg [26]. Another work by Qi et al. [43] reduces the time cost of patch generation via test case prioritization.

**SemFix**. Nguyen et al. [38] propose SemFix, a constraint based repair approach. This approach generates patches for assignments and conditions by semantic analysis via SMT encoding. Program components are synthesized into one patch via translating the solution of the SMT instance. Our proposed approach, NOPOL, is motivated by the design of SemFix. The major differences between NOPOL and SemFix were discussed in Section 5.1.

**Mutation-based repair**. Debroy & Wong [13] develop a mutation-based repair method, which is inspired by the concept of mutation testing. Their method integrates program mutants with fault localization to explore the search space of patches.

**DirectFix**. Mechtaev et al. [35] propose DirectFix, a repair method for simplifying patch generation. Potential program components in patches are encoded into a Maximum Satisfiability (MaxSAT) problem, i.e. an optimization problem; the solution to the MaxSAT instance is converted into the final concise patch.

**SearchRepair**. Ke et al. [24] develop SearchRepair, a repair method with semantic code search, which encodes human-written code fragments as SMT constraints on input-output behavior. This method reveals 20% newly repaired defects, comparing with GenProg, AE, or RSRepair.

**SPR**. After the original publication presenting NOPOL [14], Long & Rinard [30] have proposed a repair technique called SPR using condition synthesis. SPR addresses repairing conditional bugs, as well as other types of bugs, like missing non-IF statements. The differences are as follows. First, a major difference between NOPOL and SPR is that NOPOL synthesizes a condition via component-based program synthesis while SPR is based on multiple trials of pre-defined program transformation schemas. For instance, in SPR, a transformation schema for conditional bugs is called *condition refinement*, which updates an existing condition in an IF via tightening or loosening the condition. To repair a bug, SPR tries a potential patch with the transformation schemas one by one and validates the patch with the test suite; the technique of NOPOL is entirely different, based on runtime data collection during test execution. Second, another difference is that SPR is for repairing C programs. Patches by SPR only contain primitive values while patches by NOPOL contain both primitive values and object-oriented expressions (e.g., fields and unary method calls). Third, in SPR, the technique of collecting angelic values is based on NOPOL's, yet extends it. It finds sequences of values rather than one simplified trace during collecting angelic values in NOPOL (Section 3.2). As mentioned in Section 3.2.3, the simplified trace in NOPOL reduces the search space of patch synthesis, but may result in failed repair attempts for specific bugs, where a condition is executed more than once by a test case. Examples of these bugs can be found in the SPR evaluation [30]. The simplification in NOPOL can be viewed as a trade-off between repairability and time cost.

**Prophet**. Also by Long & Rinard, Prophet is an extension of SPR that uses a probability model for prioritizing candidate patches. Based on historical patches, Prophet learns model parameters via maximum likelihood estimation. Experiments show that this method can generate correct patches for 15 out of 69 real-world defects of the GenProg benchmark. We have also noticed that in NOPOL, it is possible to synthesize more than one patch with our SMT-based synthesis implementation. Hence, the probability model in Prophet can be leveraged to direct the synthesis of more correct patches by NOPOL.

### 7.2 Other Kinds of Repair

Besides test-suite based repair, other approaches are designed for fixing software bugs and improving software quality. Dallmeier et al. [11] propose Pachika, a fix generation approach via object behavior anomaly detection. This approach identifies the difference between program behaviors by the execution of passing and failing test cases; then fixes are generated by inserting or deleting method calls. Carzaniga et al. [8] develop an automatic technique to avoid failures by a faulty web application. This technique is referred as an automatic workaround, which aims to find and execute a correct program variant. AutoFix by Pei et al. [41], employs a contract-based strategy to generate fixes. This approach requires simple specifications in contracts, e.g., pre-conditions and post-conditions of a function, to enhance the debugging and fixing process. Experiments on Eiffel programs show that this approach can fix 42% of over 200 faults.

### 7.3 Empirical Foundations of Repair

Applying automatic repair to real-world programs is limited by complex program structures and semantics. We list existing work on the investigation of empirical foundations of test-suite based repair.

Martinez & Monperrus [32] mine historical repair actions to reason about future actions with a probabilistic model. Based on a fine granularity of ASTs, this work analyzes over 62 thousands versioning transactions in 14 repositories of open-source Java projects to collect probabilistic distributions of repair actions.

Such distributions can be used as prior knowledge to guide program repairing.

Fry et al. [17] design a human study of patch maintainability with 150 participants and 32 real-world defects. This work indicates that machine-generated patches are slightly less maintainable than human-written ones; hence, patches by automatic repair could be used as the patches written by humans. Another case study is conducted by Tao et al. [48]. They investigate the possibility of leveraging patches by automatic repair to assist the process of debugging by humans.

Barr et al. [5] address the "plastic surgery hypothesis" of genetic-programming based repair, such as GenProg. Their work presents evidences of patches based on reusable code, which make patch reconstitution from existing code possible. Martinez et al. [33] conduct empirical investigation to the redundancy assumption of automatic repair; this work indicates that code extracted from buggy programs could form a patch that passes the test suite.

Monperrus [36] details the problem statement and the evaluation of automatic software repair. This work systematically describes the pitfalls in software repair research and the importance of explicit defect classes; meanwhile, this paper identifies the evaluation criteria in the field: understandability, correctness, and completeness. Zhong & Su [56] examine over 9,000 real-world patches and summarize 15 findings in two key ingredients of automatic repair: fault localization and faulty code fix. This work provides empirical foundations for localization and patch generation of buggy statements.

Qi et al. [45] propose Kali, an efficient repair approach based on simple actions, such as statement removal. Their work presents the repair results via simple methods; meanwhile, their work checks previous empirical results by GenProg [28], AE [49], and RSRepair [43]. Empirical studies show that only two bugs by GenProg, three bugs by AE, and two bugs by RSRepair are correctly patched. All the reported patches for the other bugs are incorrect due to improper experimental configurations or semantic issues; an incorrect patch either fails to produce expected outputs for the inputs in the test suite, or fails to implement functionality that is expected by developers. As the latest result in test-suite based repair, the work by Qi et al. [45] shows that repairing real-world bugs is complex and difficult. Hence, it is worth investigating the empirical results on fixing real bugs.

Recent work by Smith et al. [46] investigates the overfitting patches on test cases in automatic repair. They report a controlled experiment on a set of programs written by novice developers with bugs and patches; two typical repair methods, GenProg [28] and RSRepair [44], are evaluated to explore the factors that affect the output quality of automatic repair.

Recent work by Le Goues et al. [27] presents two datasets of bugs in C programs to support comparative evaluation of automatic repair algorithms. The detailed description of these datasets is introduced and a quantified empirical study is conducted on the datasets. Defects4J by Just et al. [23] is a bug database that consists of 357 real-world bugs from five widely-used open-source Java projects. It has recently been shown [15] that NOPOL is capable of fixing 35 bugs of this benchmark.

### 7.4 Related Techniques: Program Synthesis and Fault Localization

Our approach, NOPOL, relies on two important techniques, program synthesis and fault localization.

Program synthesis aims to form a new program by synthesizing existing program components. Jha et al. [20] mine program oracles based on examples and employ SMT solvers to synthesize constraints. In this work, manual or formal specifications are replaced by input-output oracles. They evaluate this work on 25 benchmark examples in program deobfuscation. Their follow-up work [18] addresses the same problem by encoding the synthesis constraint with a first-order logic formula. In general, any advance in program synthesis can benefit program repair by enabling either more complex or bigger expressions to be synthesized.

In our work, fault localization is used as a step of ranking suspicious statements to find out locations of bugs. A general framework of fault localization is to collect program spectra (a matrix of testing results based on a given test suite) and to sort statements in the spectra with specific metrics (e.g., Tarantula [22] and Ochiai [4]). Among existing metrics in fault localization, Ochiai [4] has been evaluated as one of the most effective ones. In Ochiai, statements are ranked according to their suspiciousness scores, which are values of the Ochiai index between the number of failed test cases and the number of covered test cases. Fault localization techniques are further improved recently, for example, the diagnosis model by Naish et al. [37], the localization prioritization by Yoo et al. [54], and the test purification by Xuan & Monperrus [53].

## 8 CONCLUSION

In this paper, we have proposed NOPOL, a test-suite based repair approach using SMT. NOPOL targets two kinds of bugs: buggy IF conditions and missing preconditions. Given a buggy program and its test suite, NOPOL employs angelic fix localization to identify potential locations of patches and expected values of IF conditions. For each identified location, NOPOL collects test execution traces of the program. Those traces are then encoded as an SMT problem and the solution to this SMT is converted into a patch for the buggy program. We conduct an empirical evaluation

on 22 real-world programs with buggy IF conditions and missing preconditions. We have presented four case studies to show the benefits of generating patches with NOPOL as well as the limitations.

NOPOL is publicly-available to support further replication and research on automatic software repair: http://github.com/SpoonLabs/nopol/.

In future work, we plan to evaluate our approach on more real-world bugs. Our future work also includes addressing the current limitations, e.g., designing better strategy for angelic fix localization, collecting more method calls, and improving the SMT encoding.

## ACKNOWLEDGMENT

The authors would like to thank David Cok for giving us full access to jSMTLIB. This work is partly supported by the INRIA Internship program, the INRIA postdoctoral research fellowship, the CNRS delegation program, and the National Natural Science Foundation of China (under grant 61502345).

## REFERENCES

[1] Nopol dataset website. http://sachaproject.gforge.inria.fr/nopol/. Accessed: 2015-06-01.

[2] Nopol source code (gpl license). http://github.com/SpoonLabs/nopol/. Accessed: 2015-06-01.

[3] On contributing patches of apache commons. http://commons.apache.org/patches.html#Test_Cases. Accessed: 2015-06-01.

[4] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007*, pages 89–98. IEEE, 2007.

[5] E. T. Barr, Y. Brun, P. T. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317, 2014.

[6] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an Eclipse plug-in for testing and debugging. In *Proceedings of Automated Software Engineering*, 2012.

[7] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezze. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791. IEEE Press, 2013.

[8] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 237–246. ACM, 2010.

[9] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceeding of the International Conference on Software Engineering*, pages 121–130. IEEE, 2011.

[10] D. R. Cok. jSMTLIB: Tutorial, validation and adapter tools for SMT-LIBv2. In *NASA Formal Methods*, pages 480–486. Springer, 2011.

[11] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. IEEE Computer Society, 2009.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[13] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74, 2010.

[14] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automaticr epair 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.

[15] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. Technical Report 1505.07002, Arxiv, 2015.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, 2014.

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

[18] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[19] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 167–178. ACM, 2008.

[20] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[21] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification*, pages 226–238, 2005.

[22] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[23] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, July 23–25 2014.

[24] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, USA, November 2015.

[25] 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.

[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[27] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering (TSE), in press*, 2015.

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

[29] Y. Li, S. Cheung, X. Zhang, and Y. Liu. Scaling up symbolic analysis by removing z-equivalent states. *ACM Trans. Softw. Eng. Methodol.*, 23(4):34:1–34:32, 2014.

[30] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 166–178, 2015.

[31] S. L. Marcote and M. Monperrus. Automatic repair of infinite loops. Technical Report 1504.05078, Arxiv, 2015.

[32] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[33] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Proceedings of the 36th International Conference on Software Engineering*, pages 492–495, 2014.

[34] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[35] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

[36] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[37] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):11, 2011.

[38] 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. IEEE Press, 2013.

[39] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[40] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, page na, 2015.

[41] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[42] J. H. Perkins, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou. Automatically Patching Errors in Deployed Software. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[43] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189, 2013.

[44] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[45] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of ISSTA*. ACM, 2015.

[46] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[47] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 314–324. ACM, 2013.

[48] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74, 2014.

[49] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 356–366, 2013.

[50] J. Xu, Z. Zhang, W. Chan, T. Tse, and S. Li. A general noise-reduction framework for fault localization of java programs. *Information and Software Technology*, 55(5):880–896, 2013.

[51] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus. Dynamic Analysis can be Improved with Automatic Test Suite Refactoring. Technical Report 1506.01883, Arxiv, 2015.

[52] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 2014*, pages 191–200, 2014.

[53] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 52–63. ACM, 2014.

[54] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19, 2013.

[55] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM, 2006.

[56] H. Zhong and Z. Su. An empirical study on fixing real bugs. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 913–923. IEEE, 2015.