Precise Condition Synthesis for Program Repair

Yingfei Xiong*†, Jie Wang*†, Runfa Yan‡, Jiachen Zhang*†, Shi Han§, Gang Huang(✉)*†, Lu Zhang*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing 100871, China
†EECS, Peking University, Beijing 100871, China, {xiongyf, 1401214303, 1300012792, hg, zhanglucs}@pku.edu.cn
‡SISE, University of Electronic Science and Technology of China, Chengdu 610054, China, yanrunfa@outlook.com
§Microsoft Research, Beijing 100080, China, shihan@microsoft.com

***Abstract*—Due to the difficulty of repairing defect, many research efforts have been devoted into automatic defect repair. Given a buggy program that fails some test cases, a typical automatic repair technique tries to modify the program to make all tests pass. However, since the test suites in real world projects are usually insufficient, aiming at passing the test suites often leads to incorrect patches. This problem is known as weak test suites or overfitting.**

In this paper we aim to produce precise patches, that is, any patch we produce has a relatively high probability to be correct. More concretely, we focus on condition synthesis, which was shown to be able to repair more than half of the defects in existing approaches. Our key insight is threefold. First, it is important to know what variables in a local context should be used in an "if" condition, and we propose a sorting method based on the dependency relations between variables. Second, we observe that the API document can be used to guide the repair process, and propose document analysis technique to further filter the variables. Third, it is important to know what predicates should be performed on the set of variables, and we propose to mine a set of frequently used predicates in similar contexts from existing projects.

Based on the insight, we develop a novel program repair system, ACS, that could generate precise conditions at faulty locations. Furthermore, given the generated conditions are very precise, we can perform a repair operation that is previously deemed to be too overfitting: directly returning the test oracle to repair the defect. Using our approach, we successfully repaired 18 defects on four projects of Defects4J, which is the largest number of fully automatically repaired defects reported on the dataset so far. More importantly, the precision of our approach in the evaluation is 78.3%, which is significantly higher than previous approaches, which are usually less than 40%.

I. Introduction

Motivation. Given the difficulty of fixing defects, recently a lot of research efforts have been devoted into automatic program repair techniques [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. Most techniques generate a patch for a defect aiming at satisfying a specification. A frequently used specification is a test suite. Given a test suite and a program that fails to pass some tests in the test suite, a typical repair technique modifies the program until the program passes all tests.

We acknowledge participates of Dagstuhl 17022, especially Julia Lawall, Yuriy Brun, Aditya Kanade, and Martin Monperrrus, and anonymous reviewers for their comments on the paper, and Xuan-Bach D. Le, David Lo, and Claire Le Goues for discussion on their experiment data. This work is supported by the National Basic Research Program of China under Grant No. 2014CB347701, the National Natural Science Foundation of China under Grant No. 61421091, 61225007, 61672045, and Microsoft Research Asia Collaborative Research Program 2015, project ID FY15-RES-OPP-025.

However, the tests in real world projects are usually insufficient, and passing all tests does not necessarily mean that the program is correct. A patch that passes all tests is known as a *plausible* patch [11], and we use *precision* to refer the proportion of defects correctly fixed by the first plausible patch among all plausibly fixed defects. We argue that precision is a key quality attribute of program repair systems. If the precision of a repair system is similar to or higher than human developers, we can trust the patch generated by the system and deploy them immediately. On the other hand, if the precision of a repair system is low, the developers still need to manually review the patches, and it is not clear whether this process is easier than manual fix. As an existing study [12] shows, if the developers are provided with low-quality patches, the performance of the developers is even lower than those who are provided with no patches.

However, the precisions of existing testing-based repair techniques are not high [11], [13], [14]. As studied by Qi et al. [11], GenProg, one of the most well-known program repair techniques, produced plausible patches for 55 defects, but only two were correct, giving a precision of 4%.

The reason for the low precision, as studied by Long et al. [15], is that correct patches are sparse in the spaces of repair systems, while plausible ones are relatively much more abundant. In their experiments, often hundreds or thousands of plausible patches exist for a defect, among which only one or two are correct. It is difficult for the repair system to identify a correct patch from the large number of plausible patches.

A fundamental way to address this problem is to rank the patches by their probabilities of being correct, and return the plausible patch with the highest probability, or report failure when the highest probability is not satisfactory. This is known as *preference bias* in inductive program synthesis [16]. Research efforts have been made toward this direction. Prophet [17] and HistoricalFix [18] rank the patches using models learned from existing patches. DirectFix [5], Angelix [19] and Qlose [20] rank the patches by their distance from the original program. MintHint [9] ranks the patches by their statistical correlation with the expected results. However, the precisions of these approaches are not yet satisfactory. For example, Prophet [17] and Angelix [19] have precisions of 38.5% and 35.7% on the GenProg Benchmark [21].

An important reason for this imprecision, as we conjecture, is that the existing ranking approaches are too coarse-grained. As will be shown later, existing approaches cannot distinguish

many common plausible "if" conditions from the correct condition, and will give them the same rank.

To overcome this problem, in this paper we aim to provide more fine-grained ranking criteria for condition synthesis. Condition synthesis tries to insert or modify an "if" condition to repair a defect. Condition synthesis has been used in several existing repair systems [6], [19], [22] and is shown to be one of the most effective techniques. For example, among all defects correctly repaired by SPR [6], more than half of them are fixed by condition synthesis.

Our approach combines three heuristic ranking techniques that exploit the structure of the buggy program, the document of the buggy program, and the conditional expressions in existing projects. More concretely, we view the condition synthesis process as two steps. (1) *variable selection:* deciding what variables should be used in the conditional expression, and (2) *predicate selection:* deciding what predicate should be performed on the variables. For example, to synthesize a condition `if(a>10)`, we need to first select the variable "`a`" and then select the predicate "`>10`". Based on this decomposition, we propose the following three techniques for ranking variables and predicates, respectively.

- *Dependency-based ordering.* We observe that the principle of locality holds on variable uses: the more recent a variable in a topological ordering of dependency is, more likely it will be used in a conditional expression. We use this order to rank variables.
- *Document analysis.* We analyze the javadoc comments embedded in the source code. When variables are mentioned for a particular class of conditional expressions, we restrict the selection to only the mentioned variables when repairing the conditional expressions of the class.
- *Predicate mining.* We mine predicates from existing projects, and sort them based on their frequencies in contexts similar to the target condition.

To our knowledge, the three techniques are all new to program repair. We are the first to propose the locality of variable uses and apply it to program repair. We also are the first to utilize the documentation of programs to increase precision. Finally, predicate mining is the first technique that automatically mines components of patches from the source code (in contrast to those mining from patches [17], [18], [23]) of existing projects.

As will be shown later, the combination of the three techniques gives us high precision in program repair. Based on the high precision, we further employ a new repair method that was considered to be too overfitting: directly returning the test oracle of the failed test to repair a faulty method. This is considered overfitting because a test oracle is usually designed for a specific test input, and it is not clear whether this test oracle can be and should be generalized to other inputs. However, this overfitting is likely not to exist when the test input belongs to a boundary case. A boundary case is a case that cannot be captured by the main program logic, and for such a case usually a value or an exception is directly returned. For example, a patch we generated in our experiment, `if (marker==null) return false`, directly return the oracle value `false` when the input is `null`. Since our condition synthesis is precise, if we can successfully synthesize a condition that checks for a boundary case, it is probably safe to directly return the oracle.

We have implemented our approach as a Java program repair system, ACS (standing for Accurate Condition Synthesis), and evaluated ACS on four projects from the Defects4J benchmark. ACS successfully generated 23 patches, where 18 are correct, giving a precision of 78.3% and a recall of 8.0%. Both the precision and the recall are the best results reported on Defects4J so far within our knowledge. Most importantly, the precision is significantly higher than previous testing-based approaches, which are usually less than 40%. Furthermore, we evaluated the three ranking techniques on the top five projects from GitHub besides the four projects. The result suggests that our approach may achieve a similar precision across a wide range of projects.

II. Motivating Example

```
1    int lcm=Math.abs(mulAndCheck(a/gdc(a,b),b));
2   +if (lcm == Integer.MIN_VALUE) {
3   +   throw new ArithmeticException();
4   +}
5    return lcm;
```

Fig. 1. Motivating Example

Example. Figure 1 shows the Math99 defect in the Defects4J [24] benchmark fixed by ACS. This method calculates the least common multiple and is expected to return a non-negative value, but it fails to handle the case where the method `abs` returns `Integer.MIN_VALUE` at input `Integer.MIN_VALUE` due to the imbalance between positives and negatives in integer representation. Two tests cover this piece of code. Test 1 has input `a=1`, `b=50`, and expects `lcm=50`. Test 2 has input `a=Integer.MIN_VALUE`, `b=1`, and expects `ArithmeticException`. Test 1 passes and test 2 fails.

To fix the defect, ACS tries to directly return the test oracle, and adds lines 2-4 to the program. In particular, ACS synthesizes the condition at line 2. Since we have only two tests, a condition is plausible if it evaluates to `false` on test 1 and evaluates to `true` on test 2. Thus, there exists many plausible conditions that can make the two tests pass, such as `b==1` or `lcm != 50`. As a result, we need to select the correct condition from the large space of plausible conditions.

Existing approaches are not good at this kind of fine-grained ranking. For example, Prophet [17] always assigns the same priority to `lcm!=50` and `lcm==Integer.MIN_VALUE` because, due to efficiency reasons, Prophet can only consider variables in a condition. For another example, Qlose [20] always assigns the same priority to `b==1`, `lcm!=50`, and `lcm==Integer.MIN_VALUE` because the three conditions exhibit the same behavior in testing executions, which Qlose uses to rank different patches.

To implement fine-grained ranking, ACS decomposes condition ranking into variable ranking and predicate ranking, and using three techniques to rank variables and predicates.

Dependency-based ordering. Our observation is that the principle of locality also holds on variable uses. If variable x is assigned from an expression depending on y, i.e., x depends on y, x has a higher chance than y to be used in a following conditional expression. The intuition is that y is more likely to be a temporary variable whose purpose is to compute the value of x. In our example, variable `lcm` depends on variables `a` and `b`, and thus `lcm` is more likely to be used in the "if" condition. Based on the dependency relations, we can perform a topological sort of the variables, and try to use most dependent variables in the synthesis first.

Document Analysis. We further observe that many Java methods come with javadoc comments, documenting the functionality of the method. If we extract information from the javadoc comment, we could further assist condition synthesis. As the first step of exploiting comments, in this paper we consider one type of javadoc tags: `@throws` tag. Tag `@throws` describes an exception that may be thrown from the method, such as the following one from Math73 in Defects4J [24].

```
/** ...
 * @throws IllegalArgumentException if initial is not between
 * min and max (even if it <em>is</em> a root)
**/
```

According to this comment, an exception should be thrown when the value of `initial` is outside of a range. To exploit this information for variable selection, we analyze the subject of the sentence. If the subject mentions any variable in the method, we associate the variable with the exception. When we try to generate a guard condition for such an exception, we consider only the variables mentioned in the document. Since programmers may not refer to the variable name precisely, we use fuzzy matching: we separate variable names by capitalization, i.e., `elitismRate` becomes "elitism" and "rate", and determine that a variable is mentioned in the document if the last word (usually the center word) is mentioned.

Note that our current approach only makes the above lightweight use of javadoc comments, but more sophisticated document analysis techniques may be used to obtain more information, or even directly generate the condition [25]. This is future work to be explored.

Predicate Mining. After we have an ordered list of variables, we select predicates for the variables one by one. Based on our observations, we have the insight that the predicates used in conditional expressions have highly sparse and skewed conditional distribution given the contexts of the expressions. We currently use variable type, variable name, and/or the surrounding method name as context. For example, given an integer variable `hour`, predicates such as `<=12` or `>24` are often used. In our running example, `lcm` indicates the least common multiple, on which `==Integer.MIN_VALUE` is more frequently used than a large number of observed alternatives such as `!= 50`. For another example, in methods whose names contain "factorial", predicates such as `<21` is often used, because 20! is the largest factorial that a 64bit integer can represent.

Based on such insights, we prioritize and prune predicates based on their conditional distributions of surrounding contexts. We approximate such conditional distributions based on the statistics against a large scale repository of existing projects. Concretely, we search predicates under similar contexts in a large repository, and rank them by their occurrences.

Combining the three techniques, we could successfully synthesize `lcm==Integer.MIN_VALUE` at line 2. Since this condition is only valid for one value of `lcm`, it is likely to be a boundary case and thus we can safely generate the patch.

III. Approach

In this section we explain the details of our approach. The input of our approach consists of a program, a test failed by the program, a set of tests passed by the program. The output is a patch on the program.

A. Overview

We use two types of templates to fix defects. The first type is to directly return the oracle as mentioned in the introduction. We first identify the last executed statement s in the failed test, and then insert one of the following statement before s to prevent the failure.

- *Value-Returning*. `if (`c`) return` v`;`
- *Oracle-Throwing*. `if (`c`) throw` e`;`

When the failed test expects a return value, value-returning is used, otherwise oracle-throwing is used. Here v or e is the expected return value or exception, and c is the synthesized condition. We also use heuristic rules to check whether the synthesized c is a boundary check, and discard the patch if it is not a boundary check.

We always insert before the last executed statement because, if the defect leads to crash, for example, the program fails to check a null pointer, we usually need to place the guarded return statement right before the crashed statement.

The second type is the modification of an existing condition. We first locate a potentially faulty "if" condition c', and then apply one of the following modifications based on the result of predicate switching.

- *Widening*. `if (`c'`)` $\Rightarrow$ `if (`c' `||` c`)`
- *Narrowing*. `if (`c'`)` $\Rightarrow$ `if (`c' `&& !`c`)`

We locate the potentially faulty condition by combining spectrum-based fault localization (SBFL) [26] and predicate switching [27]. Both approaches be used to detect potentially faulty conditions. SBFL scales better while predicate switching is more precise. In our approach, we first use SBFL to localize a list of potentially faulty methods, and then use predicate switching on demand within each method. In this way we achieve a balance between precision and scalability. The SBFL formula we used in our implementation is Ochiai [28], which is shown to be among the most effective spectrum-based fault localization algorithms [29], [30], [31].

If predicate switching negates the original condition from `true` to `false`, we apply the narrowing template, otherwise

we apply the widening template. Here c is the synthesized condition.

In both types of templates, we need to synthesize condition c. We require c to capture the failed test execution, i.e., evaluating to `true` at the failed test execution. We first produce a ranked list of variables to be used in the condition. Then for each variable x, we produce a sorted list of predicates to be applied on the variable. For each predicate p, we validate whether they can form a condition $p(x)$ that capture the failed test execution, i.e., evaluates to `true` on target condition evaluation. If so, we synthesize a condition $p(x)$ and run all tests to validate the plausibility of the patch. In this paper we only consider synthesizing conditions containing one variable, but note that our idea is general and may be extended to conditions containing multiple variables.

We always apply the oracle-returning templates first and the condition-modification templates second. We return the first plausible patch if it is found within the time limit, otherwise we report a failure.

Some of the above steps require more detailed explanations, and we explain them one by one in the rest of the section.

B. Returning the Oracle

Extracting the Oracle. When applying the oracle returning template, we need to copy the test oracle from the test code to the body of the generated conditional statement. There are several different cases. (1) The oracle is a constant. In this case we only need to directly copy the text of the constant. (2) The oracle is specified via `(expected=XXXException.class)` annotation. In this case we throws an instance of `XXXException` by calling its default constructor. (3) The oracle is a function mapping the test input to the output. This case is more complex. For example, the following piece of code is a test method for Math3 defect in Defects4J [24].

```
1  public void testArray() {
2    final double[] a = { 1.23456789 };
3    final double[] b = { 98765432.1 };
4    Assert.assertEquals(a[0] * b[0],
5      MathArrays.linearCombination(a, b), 0d);
6  }
```

Here the oracle is `a[0]*b[0]`, which is a function mapping input a and b to the expected result. ACS inserts the following statement into method `linearCombination` to fix the defect.

```
+ if (len == 1) {return a[0]*b[0];}
```

The condition `len==1` is generated using our condition synthesis component and is not our concern here. The key is how to generate `a[0]*b[0]`.

To extract a functional oracle, we first identify the related pieces of code by performing slicing. We first perform backward slicing from the oracle expression (the oracle slice), then perform backward slicing from the test input arguments (the input slices), and subtract the input slices from the oracle slice. In this way we get all code necessarily to be copied but not the code used to initialize the test inputs. In this example, the oracle slice contains line 2, line 3 and the expression `a[0]*b[0]`, the input slices contain line 2, line 3, and the expression `a` and `b`. By subtracting the latter slices from the former, we get only the expression `a[0]*b[0]`, which should be copied to the generated "if" statement. Finally, we rename the variables representing the test input arguments (in this case, a and b) to the formal parameter names of the target method (which happens to be still a and b in this example).

Determining Boundary Checks. We consider a statement `if(`c`)` s as a boundary check if one of the following rules are satisfied. We use x to denote a variable and v to denote a constant.

Rule 1: Condition c takes the form of $x == v$, $x.equals(v)$, or their negations.

The intuition is that s is special logic to compute the result for the boundary case where x equals v.

Rule 2: Condition c takes the form of $x > v$, $x < v$, or their negations, and s takes the form of `throw` e.

The intuition is that the input is outside of the input domain, and an exception should be thrown.

C. Variable Ranking

Given a target location for condition synthesis, we first try to select variables that can be used in the expression. Our variable ranking consists of the following steps: (1) preparing candidate variables, (2) filtering variables by document analysis, and (3) sorting the variables using dependency-based ordering. The second step is already illustrated in Section II. In the following we explain the first and the third steps.

Preparing Candidate Variables. We consider four types of variables: (1) local variables, (2) method parameters, (3) `this` pointer, and (4) expressions used in other "if" conditions in the current method. Strictly speaking, the last type is not a variable, we nevertheless include it because many defects require a complex expressions to repair. To treat the last type unified as variables, we assume a temporary variable is introduced for each expression, i.e., given an expression e, we assume there exists a statement `v=`e before the target conditional expression, where `v` is a fresh variable.

Not all the above three types of variables can be used to synthesize a plausible condition. If a variable takes the same value in two test executions, but the condition are expected to evaluate to two different values, it is impossible to synthesize a plausible condition with the variable. Therefore, we first filter out all such variables. The remaining variables form the *candidate variables* to be used in condition synthesis.

Sorting by Dependency. Given a set of candidate variables, we sort them using dependency-based ordering.

To sort the variables, we create a dependency graph between variables. The nodes are variables, the edges are dependency relations between variables. Concretely, we consider the following types of intra-procedural dependencies.

- *Data Dependency*. Given an assignment statement x=exp, x depends on all variables in exp.
- *Control Dependency*. Given an a conditional statement such as `if(`$cond$`)` {$stmts$} `else` {$stmts'$}, or `while(`$cond$`)`

$\{stmts\}$, all variables assigned in $stmts$ and $stmts'$ depend on variables in $cond$.

Note that the dependency relations here are incomplete. For example, if exp contains a method call, extra dependencies may be caused by the method. However, implementing a complete dependency analysis is difficult and the current relations are enough for ranking the variables.

For example, Fig. 2 shows the dependency graph of the example in Fig. 1. The dependency edges are created by the assignment statement at lines 1 and 2.

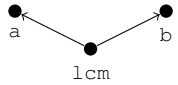

Fig. 2. Dependency Graph of Fig. 1

Then we perform a topological sorting to ensure the most dependent variables are sorted first. First we identify all circles on the graph, and collapse them as one node containing several variables. Now the graph is acyclic and we can perform topological sorting. We first identify the nodes with no incoming edges (`lcm` in Fig. 2), and give them priority 0. Then we remove these variables and their outgoing edges from the graph. Next we identify the nodes with no incoming edges in the new graph (`a` and `b` in the example), give them priority 1, and remove these nodes. This process stops when there is no variable in the graph.

Note there may be multiple variables having the same priority. We further sort the variables by the distance between the potentially faulty condition and the assignment statement that initializes the variable. In the rest of the paper we would use *(priority) level* to refer to the priority assigned to variables by the partial order and use *rank* to refer to the rank in final total order.

To ensure the precision of the synthesized condition, we use only variables at the first n priority levels in condition synthesis. In our current implementation we set n as 2. As will be shown later in our evaluation, the first two levels contain a vast majority of the variables that would be used in a conditional expression.

D. Predicate Ranking

Mining Related Conditions. We select an ordered list of predicates by predicate mining. Given a repository of software projects, we first collect the conditional expressions that are in a similar context to the conditional expression being synthesized. Currently we use variable type, variable name, and method name to determine a context. We say two variable or method names are similar if we decompose the names by capitalization into two sets of words, the intersection of the two sets are not empty. We say a variable name is meaningful if its length is longer than two. Assuming we are synthesizing a condition c with variable x in the method m. A conditional expression c' is considered to be in a similar context of c, if (1) it contains one variable x', (2) x' has the same type as x, (3) the name of x' is similar to x when the name of x is meaningful, or the name of the method surrounding c' is similar to m when the name of x is not meaningful.

$$
\begin{array}{lcl}
preds & := & p\_equal \mid o\_equal \mid io \mid lt \mid le \mid gt \mid ge \\
p\_equal & := & \texttt{==}\ prim\_const \\
o\_equal & := & \texttt{.equals(}obj\_const\texttt{)} \\
io & := & \texttt{instanceof}\ class\_name \\
lt & := & \texttt{<}\ \ numeric\_const \\
le & := & \texttt{<=}\ float\_const \\
gt & := & \texttt{>}\ \ numeric\_const \\
ge & := & \texttt{>=}\ float\_const \\
prim\_const & := & numeric\_const \mid boolean\_const \\
numeric\_const & := & integer\_const \mid float\_const
\end{array}
$$

Here obj_const represents a constant expression evaluating to an object (which can be determined conservatively by static analysis), $class\_name$ represents a class name, and $integer\_const$, $float\_const$, and $boolean\_const$ represents constant values of integer, float, boolean type, respectively.

Fig. 3. The Syntax of Predicates

In our current implementation, we utilize the search engine of GitHub[1] so that we can use all open source projects in GitHub as the repository. Each time we invoke the search engine for returning Java source files relating to three keywords: `if`, t_x, and n_x, where t_x is the type of variable x, and n_x is the name of x if the name is meaningful, otherwise n_x is the name of the surrounding method.

Counting Predicates. Given a conditional expression in a similar context, we extract the predicates used in the conditional expressions. While we can extract any predicate syntactically from the collected conditions, we choose to consider a predefined space of predicates due to the following two reasons. (1) As shown by an existing study [15], arbitrarily expanding the search space often leads to more incorrect patches and even fewer correct patches. (2) Syntactically different predicates may semantically be the same. For example, > 1 and $>= 2$ is semantically the same for an integer variable. If we deal with the predicates syntactically, we may incorrectly calculate their frequencies.

Fig. 3 shows the syntax definitions of the predicates we considered. Basically, there are predicates comparing a primitive variable with a constant ($==, <, >, <=, >=$), testing equality of an object with a constant (`.equals`) and testing the class of an object (`instanceof`). Note operators $<=$ and $>=$ only apply to floats. Since $x >= v$ is equivalent to $x > v-1$ for integers, we normalize the predicates on integers by considering only $<$ and $>$. We also normalize symmetric expressions such as $x > v$ and $v < x$ by considering only the former. We deliberately exclude `!=` operator because the synthesized condition represents cases that are ignored by the developers, and it is unlikely the developers ignore a large space such as `!=1`. More discussion can be found in Section V.

We use a function $pred$ to extract a multiset of predicates from a conditional expression. Fig. 4 shows part of the definition of $pred$. We recursively traverse to the primitive

[1]https://github.com/search

$$
\begin{array}{lcl}
pred[\![e_1 \texttt{ \&\& } e_2]\!] & = & pred[\![e_1]\!] \cup pred[\![e_2]\!] \\
pred[\![e_1 \texttt{ || } e_2]\!] & = & pred[\![e_1]\!] \cup pred[\![e_2]\!] \\
pred[\![\texttt{!}\, e]\!] & = & pred[\![e]\!] \\
pred[\![x\texttt{.equals(}v\texttt{)}]\!] & = & \{\texttt{.equals(}eval[\![v]\!]\texttt{)}\} \\
pred[\![x == v]\!] & = & \{==eval[\![v]\!]\} \\
pred[\![x \texttt{ instanceof } l]\!] & = & \{\texttt{instanceof } l\} \\
pred[\![x^i <= v]\!] & = & \{<eval[\![v{+}1]\!], >eval[\![v]\!]\} \\
pred[\![x^i < v]\!] & = & \{<eval[\![v]\!], >eval[\![v{-}1]\!]\} \\
pred[\![x^i >= v]\!] & = & pred[\![x^i < v]\!] \\
pred[\![x^i > v]\!] & = & pred[\![x^i <= v]\!] \\
pred[\![x^f <= v]\!] & = & \{<=eval[\![v]\!], >eval[\![v]\!]\} \\
 & \ldots & \\
pred[\![v == x]\!] & = & pred[\![x ==v]\!] \\
pred[\![x^i <= v]\!] & = & pred[\![v >= x^i]\!] \\
 & \ldots & \\
pred[\![\_]\!] & = & \{\}
\end{array}
$$

Here e_1, e_2, e are arbitrary expressions, x is a variable, v is a constant expression, $eval$ is a function evaluates a constant expression to a constant value, x^i is an integer variable, x^f is float variable, and "$\_$" denotes an arbitrary expression not captured by previous patterns.

Fig. 4. The $pred$ Function

predicates and return them. Here we normalize the expressions such as x^{i}<=v and v==x. Furthermore, since a predicate $p(x)$ can be used as $p(x)$ or $\neg p(x)$, we also consider the negation of a predicate if it is also in our predicate space. As an example, from $x^i <= v$ we get two predicates $\{<= v, > v\}$, and then we normalize the former and get $\{< v + 1, > v\}$. We omit the definitions for floats and the symmetric forms in Fig. 4 as they can be easily derived.

Given a set of conditions in similar contexts, we apply the $pred$ function to each condition, and sort the predicates by their frequencies in the conditions. To ensure the precision of our approach, we will only consider the top k predicates for condition synthesis. Currently we set k heuristically as 20. As will be shown in the evaluation later, 20 is enough to cover the correct predicate in most cases.

We also use a predefined set of predicates for cases that are often ignored by developers. The current set includes tests for Boolean true and false, such as `==true`, tests for minimum and maximum values, such as `==Integer.MIN_VALUE`, and tests for frequently-used JDK interfaces, such as `instanceof Comparable`.

IV. Evaluation

A. Research Questions

Our evaluation intends to answer the following research questions.

- RQ1: how do the three ranking techniques perform on ranking variables and predicates?
- RQ2: how does our approach perform on real world defects?
- RQ3: how does our approach compare with existing approaches?
- RQ4: to what extent does each component of our approach contribute to the overall performance?

B. Implementation

We have implemented our approach as a Java program repair tool. Our implementation is based on the source code of Nopol [22] and the fault localization library GZoltar [32]. Natural language processing is implemented using Apache OpenNLP. Our open-source implementation and the detailed results of the experiments are available online[2].

C. Data Set

Our evaluation is performed on two datasets. The first dataset consists of the top five most starred Java projects on GitHub as of Jul 15th, 2016. The second dataset consists of four projects from Defects4J [22], a bug database widely used for evaluating Java program repair systems [14], [18]. We use both datasets to answer the first question and use the defects in Defects4J to answer the rest three questions.

TABLE I
Statistics of the Defects4J Database

Project	KLoc	Test Cases	Defects	ID
GitHub Projects				
Google I/O App	62	269	-	IO
OkHttp	68	1734	-	Http
Universal Image Loader	13	13	-	Image
Retrofit	18	405	-	Retrofit
Elasticsearch	879	8120	-	Search
Defects4J				
JFreeChart	146	2205	26	Chart
Apache Commons Math	104	3602	106	Math
Joda-Time	81	4130	27	Time
Apache Common Lang	28	2245	65	Lang
Total	1388	22723	224	-

Table I shows basic metrics of the two datasets. Among them, IO is an Android app for Google I/O conference. HTTP is an efficient HTTP client. Image is an android library for image loading. Retrofit is a type-safe HTTP client. Elasticsearch is a distributed search engine. Chart is a library for displaying charts. Math is a library for scientific computation. Time is a library for date/time processing. Lang is a set of extra methods for manipulate JDK classes.

Note that Defects4J contains five projects in total. We did not use the fifth project, Closure, because GZoltar does not support this project due to its customized testing format. This is consistent with an existing study [14], which also dropped Closure due to its incompatibility with GZoltar.

Since our predicate mining was implemented on a search engine of GitHub, we may happen to locate code pieces from the subject projects, in which the defects were already fixed. To prevent such a bias, we used the following two techniques: (1) our implementation automatically excludes the files from the same project and from all known forked projects; (2) we manually reviewed the search results for all correctly repaired defects, and deleted all results that may be a clone of the project code.

[2]https://github.com/Adobee/ACS

D. RQ1: Performance of the Three Techniques

To answer the first question, we took the nine projects in our datasets, used our three techniques to rank variables and predicates in the conditional expressions in these projects.

TABLE II
DEPENDENCY-BASED ORDERING PERFORMANCE

Project	Variables	Level 1	Level 2	Avg.Rank	p-value
IO	996	91.2%	7.2%	40.5%	4.3e-10
Http	773	92.1%	5.9%	38.1%	4.5e-16
Image	246	89.8%	6.5%	33.7%	1.9e-11
Retrofit	276	79.3%	10.9%	43.8%	1.7e-2
Search	7714	92.7%	6.1%	39.5%	5.8e-65
Chart	3979	84.6%	9.9%	43.0%	1.2e-24
Math	2937	86.9%	9.7%	41.4%	5.1e-26
Time	1686	90.2%	7.9%	35.4%	1.2e-49
Lang	1997	95.5%	3.6%	40.4%	7.9e-23
Total	20604	89.9%	7.4%	40.3%	2.0e-214

"Variables" shows the number of variables in the conditional expressions. "Level 1/2" shows how many correct variables are ranked in the corresponding priority level. "Avg.Rank" shows the average rank of the correct variable in the normalized form. "p-value" shows the result of the wilcoxon signed-rank test.

We first evaluated dependency-based ordering. We took each conditional expression in the subjects and checked how the variables used in the conditions are ranked in dependency-based ordering. Table II shows the results. We first consider the priority levels of the variables. As we can see, 89.9% of the variables are in the first priority level, and 97.3% of the variables are in the first two levels, and there is no significant difference between different types of projects. To further understand how the correct variable is ranked in the final total order, we normalized the rank into [0,1] by this formula $\frac{rank-1}{total-1}$, where $rank$ is the rank of the variable starting from 1 and $total$ is the total number of variables. The result is shown in the column "Avg.Rank". As we can see, the average rankings of all projects are significantly smaller than 50%. We further performed a wilcoxon signed-rank test to determine whether our ranking results are significantly different from random ranking, and the last column shows that the results on all projects are significant (< 0.05). Further considering many of the variables may not be repair candidates, the correct variable would be among the earliest variables selected for synthesis.

We then evaluated document analysis. Since a comment usually mentions only a few variables, it is clear that document analysis is effective at filtering out variables, but it is not clear (1) whether it filters out wrong variables, and (2) how many conditions can benefit from document analysis. To answer the two questions, we took all conditions guarding an exception, i.e., conditional statement of the form `if (c) throw X`, and ran document analysis on those containing one variable. Table III shows the result. As we can see from the table, in a vast majority of the cases (97.8%) the exception is undocumented, and among those documented cases, 72.6% of the documents do not mention any variable. This result shows that document analysis can only benefit a small number of conditions. On the other hand, the false positives are significantly lower than the true positives, indicating that the positive effect of document analysis would outperform the negative effect. We also manually inspected some false positives, and found a main cause is that a word may take multiple meanings. For example, a comment mentions a word "value", which actually refers to the value of variable "fieldType", but in the method there happens to have a variable "value", causing a false positive.

TABLE III
DOCUMENT ANALYSIS PERFORMANCE

Project	Conds	NoDoc	NoVars	Correct	Incorrect
IO	1291	99.3%	33.3%	33.3%	33.3%
Http	988	100.0%	0.0%	0.0%	0.0%
Image	248	93.9%	43.8%	33.3%	18.8%
Retrofit	150	100.0%	0.0%	0.0%	0.0%
Search	10880	99.9%	62.0%	22.0%	16.0%
Char	36490	98.6%	93.3%	0.0%	6.7%
Math	4342	84.9%	86.1%	11.6%	2.2%
Time	1472	85.3%	38.2%	46.5%	15.2%
Lang	3569	89.5%	70.8%	23.6%	5.6%
Total	59430	97.8%	72.6%	21.1%	6.3%

"Conds" shows the number of exception-guarding conditions. "NoDoc" shows the proportion of conditions that do not have a JavaDoc comment. The last three columns show the proportions within those having a document. "NoVars" shows the proportion of conditions that have a JavaDoc comment that does not mention any variable. "Correct" shows the proportion of conditions where the JavaDoc comment mentions its variable. "Incorrect" shows the proportion of conditions where the set of mentioned variables by the JavaDoc comment is not empty but does not include the correct one.

TABLE IV
PREDICATE MINING PERFORMANCE

Project	Preds	Included	First	wef≤0	wef≤4
IO	594	87.4%	96.3%	94.3%	98.3%
Http	433	80.4%	95.7%	90.1%	97.7%
Image	153	85.0%	92.3%	85.0%	94.1%
Retrofit	187	54.5%	100.0%	99.5%	99.5%
Search	5780	73.3%	96.8%	94.9%	98.6%
Chart	3535	50.2%	90.5%	90.5%	96.4%
Math	1371	52.0%	76.9%	59.1%	76.2%
Time	1068	73.4%	85.8%	76.9%	89.9%
Lang	1139	79.4%	89.8%	87.4%	95.6%
Total	14270	66.7%	92.5%	88.2%	94.9%

"Preds" shows the number of predicates. "Included" shows the percentage of predicates that is included in the returned list. "First" shows the percentages of predicates are ranked first among all predicates included. "wef≤k" shows the percentages of cases where the wasted effort is smaller than or equal to k.

Finally, we evaluated predicate mining. We first extracted the predicates in our space from the conditional expressions in the subject projects. For each predicate, we performed predicate mining to retrieve a list of predicates and checked how the original predicates were ranked. We did not use any pre-defined predicates in this process. If there is a tie including the original predicate, we consider the original predicate has the average rank of all predicates in the tie. Note here many predicates cannot lead to the generation of a patch. A necessary condition of generating patches from a predicate q is that q should capture (evaluates to "true") the case of the failed test execution. Since we know the original predicate p evaluates to "true" at the failed execution, we check the satisfiability of $p \wedge q$, and any q causing the formula

unsatisfiable cannot generate a patch. We removed all such predicates that cannot generate a patch.

The result is shown in Table IV. As we can see from the table, a large proportion (66.7%) of predicates are included in the rank result. This confirms our assumption: the predicates are heavily unevenly distributed. Furthermore, those included in the list are ranked high, with a 92.5% of the predicates are ranked first. To further understand how often an incorrect predicate is ranked higher than a correct one uniformly, we use the measurement "wasted effort" [33]. The wasted effort is defined as the number of incorrect predicates ranked higher than the correct one if the correct one is included in the returned list, otherwise is defined as the length of the returned list. As we can see from the table, the wasted efforts is in general small: in 88.2% of the cases there is no wasted effort and in 94.9% of the cases the wasted efforts are smaller than or equal to 4.

E. RQ2: Performance of ACS

To answer this research question, we executed ACS against all the bugs in Defects4J. Then we manually compared the generated patches with the user patches, and deem an ACS patch to be correct only when we can discover a sequence of semantically equivalent (basic) transformations that turn one patch into the other. Note that this criterion is conservative, and the reported correct patches are a subset of all correct patches. Our experiments were conducted on an Ubuntu virtual machine with i7 4790K 4.0GHz CPU and 8G memory. We use 30 minutes as the timeout of each defect.

TABLE V
REPAIR RESULTS ON DEFECTS4J

Project	Correct	Inc.	Precision	Recall	In Space	Recall (In Space)
Chart	2	0	100.0%	7.7%	2	100.0%
Math	12	4	75.0%	11.3%	12	100.0%
Time	1	0	100.0%	3.7%	2	50.0%
Lang	3	1	75.0%	4.6%	3	100.0%
Total	18	5	78.3%	8.0%	19	94.7%

"Inc." stands for incorrect patches. There are 224 defects in our dataset.

The results are shown in the first 5 columns of Table V. Our approach generated 23 patches in total, where 18 are correct, giving a precision of 78.3% and a recall of 8.0%. The current recall considers all defects, and many defects cannot be fixed by changing a condition or returning an oracle, i.e., not belonging to our defect class [34].

To understand the recall within our defect class, we further manually analyzed all Defects4J user patches and selected those that take a form that our approach can generate. A patch is in space if (1) it modifies an if condition, or returns a value or throws an exception with a guarded condition, and (2) the condition contains one variable or an expression used in other conditional statements, and the predicate on the variable/expression is defined in Fig. 3. Note that this process is also conservative, as other defects may also be fixed by our approach using a form different from user patches. The last two columns of Table V show the results. To our surprise, besides the 18 defects we fixed, we were only able to further identify one defect, Time19, whose patch is within the space of our approach, giving our approach a recall of 94.7%. Time19 was not repaired because the correct variable is in the 3rd level after ranking and was filtered out. This result suggests that our approach is able to fix most defects in our space.

The patch generation time is short. Our approach spent in maximum 28.0 minutes to generate a patch, with a median 5.5 minutes and a minimum 0.9 minutes. Note that since the web query time greatly depends on the network speed, we exclude the web query time. A more elaborated implementation could use a local repository to avoid most network query time.

We also qualitatively reviewed the defects and the generated patches. Our observation is that, although an ACS patch usually takes a simple form, it can fix challenging defects. Our running example in Fig. 1 requires advanced knowledge with the `Math` library to know that `abs` may return a negative value. Another example is Lang7 in method `createNumber` of class `NumberUtils`, which converts a string into a number. Our approach generates the following patch for the method.

```
1 + if (str.startsWith("--"))
2 +     throw new NumberFormatException();
3   return new BigDecimal(str);
```

The standard routine is to parse the string to the constructor of `BigDecimal`, and if the string cannot be parsed, an exception should be thrown by `BigDecimal`. However, though unstated in the specification, the `BigDecimal` implementation in Java JDK would accept a string with two minus sign, but parses it into a wrong value. Thus a guard must be added. This defect is difficult as we actually deal with a defect in JDK implementation.

F. RQ3: Comparison with Existing Approaches

We compare our results with four program repairs systems, jGenProg [14], Nopol [14], a reimplementation of PAR [18] (mentioned as xPAR), and HistoricalFix [18], which are all program repair systems that have been evaluated on Defects4J within our knowledge. Among them, jGenProg and Nopol were evaluated on the same four projects [14] as us while xPAR and HistoricalFix were evaluated on all the five projects [18]. For a fair comparison, we took only the results on the four subjects. Please note that in the experiments, the timeout for jGenProg and Nopol was set to three hours [14], the timeout for xPAR and HistoricalFix was set to 90 minutes [18], and the timeout for ACS was 30 minutes. Though the machines for executing the experiments were different, the results are unlikely to favor our approach because our timeout setting is much shorter than others.

Table VI shows the results from different systems. From the table we can see that our approach has the highest precision among all approaches, and is more than four times more precise than the second precise approach. Our recall is also the highest among the five approaches.

Some of the other state-of-the-art systems, such as Angelix [19] and Prophet [17], were designed for C and cannot be directly compared. Nevertheless, their evaluations on the

TABLE VI
PERFORMANCE OF RELATED APPROACHES

Approach	Correct	Incorrect	Precision	Recall
ACS	18	5	78.3%	8.0%
jGenProg	5	22	18.5%	2.2%
Nopol	5	30	14.3%	2.2%
xPAR	3	–[4]	–[4]	1.3%[2]
HistoricalFix[1]	10(16)[3]	–[4]	–[4]	4.5%(7.1%)[2,3]

[1]The evaluation was based on manually annotated faulty methods but not automatically located methods.
[2]HistoricalFix and PAR were tested on selected 90 defects from Defects4J, but the authors of that paper [18] believe all other defects cannot be fixed.
[3]HistoricalFix generated correct patches for 16 defects, but only 10 were ranked first.
[4]Not reported.

GenProg benchmark [21] shows that they have a precision of 35.7% and 38.5%, respectively. Since our precision is noticeably higher than them, it might be possible to combine our approach with them to increase their precisions in future.

We also determine whether the fixed defects by ACS were also fixed by any other approaches. We found only 2 defects were fixed by other approaches, and 16 were fixed for the first time. This result shows that our approach can effectively complement existing approaches on fixing more defects.

G. RQ4: Detailed Analysis of the Components

In this section we evaluate the performance of the three ranking techniques based on the 19 defects in search space. Table VII shows the detailed data about the repair process of the 19 defects. From the table we analyze the performance of the three techniques as follows.

Dependency-based Ordering. As we can see from the table, dependency-based ordering performs a significant role to select the correct variables. As we can see from "Can." column, there may be a large number of candidate variables, up to 12 variables. After dependency-based ordering, the correct variable is usually in the first or the second to select, which greatly reduces the risk of producing incorrect patches and the time for repair.

Document Analysis. Document Analysis performs a relatively small but a useful role. From the table we can see that document analysis was only able to filter one variable, which is consistent with the result of RQ1 as many exceptions are undocumented. To understand how much document analysis contributed to our result, we further reran ACS without document analysis on all defects where document analysis filtered variables. The result showed that document analysis successfully prevented one incorrect patch to be generated and had not prevented any correct patch. Therefore document analysis did contribute to the overall performance of ACS.

Predicate Mining. Predicate mining also play a significant role of preventing incorrect fixes from being generated. In theory, each candidate variable at a wrong location or each wrong variable ranked higher than the correct variable at the correct location can produce incorrect patches. This is because we can always generate $x == v$ to capture the failed test case,

TABLE VII
DETAILED PERFORMANCE ANALYSIS

Bug ID	Can.	Flt.	Rank	Prv.	Blk.
Fixed					
Chart14	2	-	2	1	no
Chart19	2	-	2	1	no
Math3	3	-	3	2	no
Math4	1	-	1	0	no
Math5	1	-	1	1	no
Math25	12	0	2	6	no
Math35	3	1	2	0	no
Math61	3	-	1	0	no
Math82	8	0	1	19	no
Math85	12	-	1	2	no
Math89	1	0	1	0	no
Math90	2	-	1	0	no
Math93	1	-	1	1	no
Math99	3	0	1	0	no
Time15	1	0	1	0	no
Lang7	1	0	1	0	no
Lang24	3	0	1	2	no
Lang35	1	0	1	0	no
Unfixed					
Time19	4	-	3	3	no

"Can." shows the number of candidate variables in the correct location. "Flt." shows how many variables were filtered out by document analysis. "Ranks" shows how the correct variable was ranked by dependency-based ordering. "Prv." shows how many variables were prevented to generate incorrect patches by predicate mining. "Blocked" shows whether the correct patch was blocked by predicate mining.

where x is the candidate variable and v is the value of x in the failed test execution. As we can see from "Prv." column, many incorrect patches have been prevented by predicate mining on the 19 defects. Further considering the large number of defects that are not within the search space, we can assume that predicate mining must have prevented a large number of incorrect patches from being generated. Please note that "Prv." may be larger than "Rank" because fault localization may initially locate wrong locations. Furthermore, as we can see from the "Blk." column, no correct patch is blocked by predicate mining.

Since predicate mining uses both predefined predicates and mined predicates, we further inspect how many successful patches whose predicates are predefined and how many are mined from GitHub. We found that 12 (66.67%) out of the 18 patches use mined predicates, and 6 (33.33%) uses predefined patches. We further investigate whether it is possible to enlarge the set of predefined predicates to replace mined predicates. Among the 12 patches, we found four use predicate `==null`, which has the potential to be predefined. The predicates of the remaining eight patches seem to be difficult to be predefined. For example, below is the patch generated for Math35, where the predicate tests the range of a "rate", which is usually between 0 and 1. The two lines are added from two failed tests, respectively.

```
+ if(elitismRate<(double)0) throw ...
+ if(elitismRate>(double)1) throw ...
```

Another example is the patch for Math5, where the predicate compares with an instance of the `Complex` class.

```
+ if(this.equals(new Complex(0,0))) return INF;
```

Templates. We also study how many defects each template fixes. We found that 10 are fixed by exception-throwing, 6 are fixed by value-returning, and 2 are fixed by narrowing. This result suggests that our approach may be effective in fixing defects related to missing boundary checks.

V. Discussion

More Patches for a Defect. In this paper we focus on generating only one patch for a defect. This is because our long-term goal is to achieve fully automatic defect repair, and thus we need to improve the precision of the first generated patch. On the other hand, we may also view the generated patches as debugging aids, then we can generate more than one patches for a defect and aim to rank the correct patch as high as possible. The performance of our approach in the latter scenario is a future work to be explored.

Alternative Method in Condition Synthesis. There are two methods to synthesize a predicate and a variable. For example, below is a correct patch we synthesize for Math85.

```
1  -if (fa*fb>=0) {
2  +if (fa*fb>=0 && !(fa*fb==0)) {
```

In our approach, ACS mines a predicate `==0` to capture the failed execution where `fa*fb` is zero, and then negate the condition to get the expected result. An alternative is not to negate conditions, and rely on predicate mining to discover a predicate `!=0` that evaluates to `false` on failed test execution. We choose the former approach because it gives us more control over the predicate space to exclude the predicates that are unlikely to be ignored by human developers. Currently we exclude `!=v`, as the developers are unlikely to ignore such a large input space. If we resort to the latter solution, we have to at least include `!=0` into the predicate space, which unlikely leads to more correct patches than the former approach but nevertheless brings the risk of generating more incorrect patches.

Generalizability. A question is whether our repair results on Defects4J can be generalized to different types of projects. To answer the question, we designed RQ1 that tested our ranking techniques on different types of projects, and the results show that dependency-based ordering and predicate mining gives a consistent ranking among different projects. Document analysis heavily depends on how much documentation is provided. However, as RQ4 shows, document analysis plays a relatively minor role in affecting our results: only one variable is removed for 19 defects. Therefore, we believe our approach can achieve a similar precision across a wide range of projects.

VI. Related Work

Automatic defect repair is a hot research topic in recent years and many different approaches have been proposed. Some recent publications [22], [17], [35] have made thorough survey about the field, and we discuss only the approaches that are related to condition synthesis and patch ranking.

Condition Synthesis for Program Repair. Several program repair approaches adopt condition synthesis as part of the repair process. Typically, a condition synthesis problem is treated as a specialized search problem where the search space is the all valid expressions and the goal is to pass all tests. Nopol [22] reduces the search problem as a constraint solving problem and uses an SMT solver to solve the problem. Semfix [4] uses a similar approach to repair conditions and other expressions uniformly. SPR [6] uses a dedicated search algorithm to solve the search problem. DynaMoth [36] collects dynamic values of primitive expressions and combines them.

Several approaches try to fix a defect by mutating the program, which may also mutate conditions. For example, GenProg [37] and SearchRepair [38] mutate programs by copying statements from other places. PAR [2] mutates the program with a set of predefined templates.

However, since the goal of these approaches is to pass all tests, it is difficult for these approaches to achieve a high precision. As reported [14], jGenProg (the Java implementation of GenProg) and Nopol have a precision of 18.5% and 14.3% on Defects4J, respectively.

Patch Ranking Techniques. Many researchers have realized the problem of low precision and have proposed different approaches to rank the potential patches in the repair space, so that patches with higher probability of correctness will be ranked higher.

DirectFix [5], Angelix [19] and Qlose [20] try to generate patches that make minimal changes to the original program. DirectFix and Angelix use syntactic difference while Qlose uses semantic differences [39]. MintHint [9] uses the statistical correlation between the changed expression and the expected results to rank the patches. Prophet [17] learns from existing patches to prioritize patches in the search space of SPR. HistoricalFix [18] learns from existing patches to prioritize a set of mutation operations. AutoFix [3] ranks the patches by the invariants used to generate them.

Compared with these approaches, our approach uses more refined ranking techniques specifically designed for condition synthesis. Our approach is also the first to utilize the locality of variables, the program document, and the existing source code (in contrast to the existing patches) for ranking.

Here we use an example to analyze why our approach is more fine-grained than existing approaches on ranking conditions. Given two plausible patches: adding a guard condition $a > 1$ to a statement and adding a guard condition $a > 2$ to the same statement, none of the above approaches can distinguish them. DirectFix and Anglix would treat the two patches as having the same syntactic structure and thus the same distance from the original program. Qlose mainly uses the coverage information and variable values in test execution, and the two patches would not exhibit any difference since they are all plausible. The case of MintHint is similar to Qlose as the two conditions would evaluate to the same value in all tests. Prophet ranks partial patches, i.e., conditions are reduced to only variables and the predicates are ignored, and the two patches are the same in the partial form. Prophet cannot rank full patches because the time is unaffordable.

HistoricalFix ranks mutation operators, and it is not affordable to distinguish each integer as a different mutation operator. Similarly, AutoFix ranks patches by invariants, and it would not affordable to identify an invariant for each integer. On the other hand, our approach can still rank them by their frequencies in existing projects.

Another way to increase precision is the recently proposed anti-patterns [40]. An anti-pattern blocked a class of patches that are unlikely to be correct. Compared with anti-patterns, our approach aims to rank all patches, including those not blocked by anti-patterns.

DeepFix [41] uses deep learning to directly generate patches. The precision of this approach depends on the neutral network learned from the training set. However, so far this approach is only evaluated on syntactic errors from students' homework and its performance on more complex defects is yet unknown.

QACrashFix [42] is an approach that constructs patches by reusing existing patches on StackOverflow. QACrashFix achieves a precision of 80%. However, this approach is limited to crash fixes whose answers already exist on the QA site, which does not apply to most defects fixed by our approach on Defects4J.

Defect Classes with Precise Specification. Several approaches target at defect classes where the specification is complete, effectively avoiding the problem of weak test suites [11]. Typical defect classes include memory leaks [43], where the specification is semantically equivalent to the original program without leaks, concurrency bugs [44], [45], [46], where the specification is semantically equivalent to the original program without concurrency bugs, and configuration errors [47], where the specification can be interactively queried from the user. Though these approaches have a high precision, they target totally different defect classes compared with our work.

Fix with Natural Language Processing. Existing research has already brought natural language processing into program repair. R2Fix [8] generates patches directly from bug report by using natural language processing to analyze bug reports. Different from R2Fix, in our approach we utilize the document to enhance the precision of program repair, and we still require a failed test.

VII. Conclusion

In this paper we study refined ranking techniques for condition synthesis, and the new program repair system achieves a relatively high precision (78.3%) and a reasonable recall (8.0%) on Defects4J. The result indicates that studying refined ranking techniques for specific repair techniques is promising, and calls for more studies on more different types of repair technique.

REFERENCES

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE '09*, 2009, pp. 364–374.

[2] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE '13*, 2013, pp. 802–811.

[3] 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*, vol. 40, no. 5, pp. 427–449, 2014.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[5] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015.

[6] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *ESEC/FSE*, 2015.

[7] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[8] C. Liu, J. Yang, L. Tan, and M. Hafiz, "R2Fix: Automatically generating bug fixes from bug reports," in *ICST*, 2013.

[9] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated synthesis of repair hints," in *ICSE*, 2014, pp. 266–276.

[10] Y. Qi, X. Mao, Y. Wen, Z. Dai, and B. Gu, "More efficient automatic repair of large-scale programs using weak recompilation," *SCIENCE CHINA Information Sciences*, vol. 55, no. 12, pp. 2785–2799, 2012.

[11] Z. Qi, F. Long, S. Achour, and M. Rinard, "Efficient automatic patch generation and defect identification in Kali," in *ISSTA*, 2015, pp. 257–269.

[12] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: A human study," in *FSE*, 2014, pp. 64–74.

[13] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *FSE*, 2015, pp. 532–543.

[14] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[15] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *ICSE*, 2016.

[16] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *International Workshop on Approaches and Applications of Inductive Programming*. Springer, 2009, pp. 50–73.

[17] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016.

[18] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016.

[19] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

[20] L. DAntoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantiative objectives," in *CAV*, 2016.

[21] 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 *ICSE*, 2012, pp. 3–13.

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

[23] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms and search spaces for automatic patch generation systems," MIT, Tech. Rep. MIT-CSAIL-TR-2016-010, 2016.

[24] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[25] J. Zhai, J. Huang, S. Ma, X. Zhang, L. Tan, J. Zhao, and F. Qin, "Automatic model generation from documentation for java api functions," in *ICSE*, 2016, pp. 380–391.

[26] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.

[27] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006, pp. 272–281.

[28] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[29] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*, 2014, pp. 191–200.

[30] 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 *ISSTA*, 2013, pp. 314–324.

[31] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, 2013.

[32] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *ASE*, 2012, pp. 378–381.

[33] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*, 2016, pp. 177–188.

[34] 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 *ICSE*, 2014, pp. 234–242.

[35] ——, "Automatic software repair: a bibliography," University of Lille, Tech. Rep. #hal-01206501, 2015.

[36] T. Durieux and M. Monperrus, "Dynamoth: dynamic code synthesis for automatic program repair," in *AST*, 2016, pp. 85–91.

[37] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[38] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *ASE*, 2015, pp. 295–306.

[39] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ESEC/FSE*. Springer, 1997.

[40] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.

[41] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *AAAI*, 2017.

[42] Q. Gao, H. Zhang, J. Wang, and Y. Xiong, "Fixing recurring crash bugs via analyzing q&a sites," in *ASE*, 2015, pp. 307–318.

[43] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *ICSE*, 2015.

[44] D. Deng, G. Jin, M. de Kruijf, A. Li, B. Liblit, S. Lu, S. Qi, J. Ren, K. Sankaralingam, L. Song, Y. Wu, M. Zhang, W. Zhang, and W. Zheng, "Fixing, preventing, and recovering from concurrency bugs," *Science China Information Sciences*, vol. 58, no. 5, pp. 1–18, 2015.

[45] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *ISSTA*, 2014, pp. 237–247.

[46] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *ICSE*, 2016.

[47] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *Software Engineering, IEEE Transactions on*, vol. 41, no. 6, pp. 603–619, 2015.