Automatic Patch Generation Learned from Human-Written Patches

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim
The Hong Kong University of Science and Technology, China
{darkrsw,jcnam,jsongab,hunkim}@cse.ust.hk

***Abstract*—Patch generation is an essential software maintenance task because most software systems inevitably have bugs that need to be fixed. Unfortunately, human resources are often insufficient to fix all reported and known bugs. To address this issue, several automated patch generation techniques have been proposed. In particular, a genetic-programming-based patch generation technique, GenProg, proposed by Weimer et al., has shown promising results. However, these techniques can generate nonsensical patches due to the randomness of their mutation operations.**

To address this limitation, we propose a novel patch generation approach, Pattern-based Automatic program Repair (PAR), using fix patterns learned from existing human-written patches. We manually inspected more than 60,000 human-written patches and found there are several common fix patterns. Our approach leverages these fix patterns to generate program patches automatically. We experimentally evaluated PAR on 119 real bugs. In addition, a user study involving 89 students and 164 developers confirmed that patches generated by our approach are more acceptable than those generated by GenProg. PAR successfully generated patches for 27 out of 119 bugs, while GenProg was successful for only 16 bugs.

I. Introduction

Patch generation is an essential software maintenance task, since most software systems inevitably have bugs that need to be fixed [1], [2]. Unfortunately, human resources are often insufficient to generate patches [3], even for known bugs. For example, Windows 2000 was shipped with more than 63,000 known bugs, largely due to limited resources [4].

To reduce manual effort, several automatic patch generation techniques have been proposed. Arcuri and Yao introduced the idea of applying evolutionary algorithms to automatic patch generation [5]. Dallmeier et al. proposed an approach leveraging object behavior model and applied this approach to real bugs from open source projects [6]. Weimer et al. proposed a population-based technique [7], [8] leveraging genetic programming [9]. Wei et al. provided a contract-based technique to automate patch generation and showed its usefulness by applying it to bugs in Eiffel classes [10].

Among these, the award-winning patch generation technique, GenProg [7], and its extension [8] showed the most promising results. To fix a bug in a given program, this technique generates variants of the program by using crossover operators and mutation operators such as statement addition, replacement, and removal [9]. Then, it runs test cases to evaluate each variant. GenProg iterates these steps until one of the variants passes all test cases. Any program variant passing all test cases is regarded as a successful patch. They experimentally showed that this technique can create successful patches for 55 out of 105 real bugs [8].

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920    lhs = strings[getShort(iCode, pc + 1)];
1921  }
1922  Scriptable calleeScope = scope;
```

(a) Buggy program. Line 1920 throws an *Array Index Out of Bound* exception when `getShort(iCode, pc + 1)` is equal to or larger than `strings.length` or smaller than 0.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   lhs = ((Scriptable)lhs).getDefaultValue(null);
1921  }
1922  Scriptable calleeScope = scope;
```

(b) Patch generated by GenProg.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   i = getShort(iCode, pc + 1);
1921+   if (i != -1)
1922+     lhs = strings[i];
1923  }
1924  Scriptable calleeScope = scope;
```

(c) Human-written patch.

```
1918  if (lhs == DBL_MRK) lhs = ...;
1919  if (lhs == undefined) {
1920+   if (getShort(iCode, pc + 1) < strings.length &&
          getShort(iCode, pc + 1) >= 0)
1921+    {
1922       lhs = strings[getShort(iCode, pc + 1)];
1923+    }
1924  }
1925  Scriptable calleeScope = scope;
```

(d) Patch generated by PAR.

Fig. 1: Patches created by GenProg, a human developer, and PAR for Mozilla Bug #114493.

However, GenProg has an inherent limitation: since this technique basically relies on random program mutations such as statement addition, replacement, and removal, it is possible to generate nonsensical patches. Figure 1(b) shows an example of nonsensical patches generated by GenProg, for the bug shown in Figure 1(a). Compared to the human-written patch in Figure 1(c), GenProg's patch completely removed the "`strings[]`" variable from the program. Note that the program would assign an element of `strings` to `lhs` as long as the given index is valid, while the patch generated by GenProg does not do this. Although GenProg's patch can actually pass all the given test cases, developers would not accept the patch, as shown in Section IV-C.

978-1-4673-3076-3/13/$31.00 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

```
3561   public ITextHover getCurrentTextHover() {
3562+    if (fTextHoverManager== null)
3563+     return null;
3564     return fTextHoverManager.getCurrentTextHover();
3565   }
```

Fig. 2: Example of "Null Checker", a bug-fix pattern for Null Pointer Exception bugs. This patch fixes a bug of **TextViewer.java** described in Eclipse JDT Bug #26028. It inserts an `if` statement to avoid calling `getCurrentTextHover()` when `fTextHoverManager` is **`null`**.

To address this limitation, we propose a novel patch generation technique: Pattern-based Automatic program Repair (PAR). This approach leverages knowledge of human-written patches. We first carefully inspected 62,656 human-written patches of open source projects. Interestingly, we found that there were several common fix patterns. Based on our observations, we created 10 fix templates, which are automatic program editing scripts based on the identified fix patterns. PAR uses these fix templates to generate program patches. Although creating fix templates requires manual effort, this is only a one-time cost and these templates are highly reusable in different contexts after they are created. Figure 1(d) shows a patch generated by our approach that is similar to the human-written patch (Figure 1(c)).

To evaluate PAR, we applied it to 119 actual bugs collected from open source projects including Apache log4j[1], Rhino[2], and AspectJ[3].

We asked 253 human subjects (89 students and 164 developers) to compare patches that they would accept, if they were code reviewers of the anonymized patches generated by PAR and GenProg. The results of this study clearly showed that patches generated by PAR are much more acceptable than patches generated by GenProg. In addition, our approach generated more successful patches than GenProg: PAR successfully generated 27 patches out of 119 bugs, while GenProg was successful for 16 bugs.

Overall, this paper makes the following contributions:

- **Manual observations on human-written patches:** Our investigation of human-written patches reveals that there are common fix patterns in patches.
- **PAR, an automatic patch generation technique leveraging fix patterns**: We propose a novel automatic patch generation technique using fix templates derived from common fix patterns.
- **Empirical evaluation**: We present the empirical evaluation results by applying PAR to 119 real bugs.

The remainder of this paper is organized as follows. After presenting common fix patterns identified from human-written patches in Section II, we propose our approach, PAR, in Section III. Section IV empirically evaluates our approach, and Section V discusses its limitations. After surveying the related work in Section VI, we conclude with directions for future research in Section VII.

[1]http://logging.apache.org/log4j/

[2]https://developer.mozilla.org/en-US/docs/Rhino

[3]http://www.eclipse.org/aspectj/

TABLE I: Common fix patterns identified from Eclipse JDT's patches.

Fix Patterns
Altering method parameters
Calling another method with the same parameters
Calling another overloaded method with one more parameter
Changing a branch condition
Adding a null checker
Initializing an object
Adding an array bound checker
Adding a class-cast checker

II. Common Fix Patterns

This section presents common fix patterns identified from our manual investigation of human-written patches. We first describe how we collected and examined a large number of human-written patches. Then, we report a list of common fix patterns.

A. Patch Collection

For our investigation, 62,656 human-written patches were collected from Eclipse JDT[4]. We used Eclipse JDT, because it has a long revision history (more than 10 years), and is widely used in the literature [11], [12]. We used the Kenyon framework [13] to retrieve bug patches [14].

B. Mining Common Patches

Since our goal is to explore human knowledge in patch generation, we focused on semantic rather than syntactic changes [15]. First, we examined whether any semantics are added in or removed from the patches. Second, we identified the root cause of each bug and the resolution of the corresponding patch. Lastly, similar patches were grouped into common patterns.

To reduce manual inspection time, we first gathered similar patches using *groums* [16]. A groum is a graph-based model for representing object usage. Although not designed for patch analysis, groums can help detect semantic rather than syntactic differences. For each patch, we built two groums from two consecutive program versions: before and after applying the patch. Then, the differences of nodes and edges between the two groums were computed automatically. We could gather patches having the same differences into a group. Although a patch group is not necessarily a fix pattern, doing this can substantially reduce manual inspection time.

To identify fix patterns, we first classified patches as additive, subtractive, or altering patches. Additive patches insert new semantic features such as new control flows, while subtractive patches remove semantic features. Altering patches just change control flows by replacing semantic features.

We then examined the root causes of bugs and how the corresponding patches specifically resolved the bugs. For example, the patch shown in Figure 2 inserts a new `if` statement to avoid a crash when `fTextHoverManager` is `null`. In this example, the root cause is "*null value*" and the patch resolves it by adding a new control flow.

Some patches address multiple causes and these are called composite patches [17]. We divided a composite patch into multiple independent patches and analyzed them individually.

[4]http://www.eclipse.org/jdt

C. Fix Patterns

After inspecting the patches, we identify many recurring similar patches, i.e., *fix patterns*. Table I shows common patterns identified by our investigation. These top eight patterns cover almost 30% of all patches we observed. The following paragraphs describe the details of each pattern:

- **Pattern**: Altering method parameters.
  Example: `obj.method(v1,v2)` → `obj.method(v1,v3)`
  Description: This pattern can fix a bug since it makes the caller give appropriate parameters to the method.

- **Pattern**: Calling another method with the same parameters.
  Example: `obj.method1(param)` → `obj.method2(param)`
  Description: This pattern changes the callee in a method call statement to fix an inappropriate method invocation.

- **Pattern**: Calling another overloaded method with one more parameter.
  Example: `obj.method(v1)` → `obj.method(v1,v2)`
  Description: This pattern adds one more parameter to the existing method call, but it actually replaces the callee by another overloaded method.

- **Pattern**: Changing a branch condition.
  Example: `if(a == b)` → `if(a == b && c != 0)`
  Description: This pattern modifies a branch condition in conditional statements or in ternary operators. Patches in this pattern often just add a term to a predicate or remove a term from a predicate.

- **Pattern**: Initializing an object.
  Example: `Type obj;` → `Type obj = new Type()`
  Description: This pattern inserts an additional initialization for an object. This prevents an object being `null`.

- **Pattern**: Adding a "null", "array-out-of-bound", and "class-cast" checker.
  Example: `obj.m1()` → `if(obj!=null){obj.m1()}`
  Description: These three patterns insert a new control flow in a program. They often add a new "`if(...)`" statement to avoid throwing exceptions due to an unexpected state of the program. Figure 2 shows an example of these patterns.

Overall, we found that there are common fix patterns in human-written patches. Since these major patterns are used in many real patches (almost 30%) to fix bugs, we may generate more successful patches by leveraging them in automatic patch generation.

III. PAR: Pattern-based Automatic Program Repair

PAR generates bug-fixing patches automatically by using fix patterns described in Section II-C. Figure 3 illustrates an overview of our approach. When a bug is reported, (a) PAR first identifies fault locations, i.e., suspicious statements, by using existing fault localization techniques [8]. These fault locations and their adjacent locations are modified to fix the bug. (b) PAR uses fix templates to generate program variants (patch candidates) by editing the source code around the fault locations. (c) Program variants are evaluated by a fitness function that computes the number of passing test cases of a patch candidate. If a candidate passes all given test cases, our approach assumes that it is a successful patch [7]. Otherwise, our approach repeats the patch candidate generation and evaluation steps.

Algorithm 1: Patch generation using fix templates in PAR.

```
Input  : fitness function Fit: Program → ℝ
Input  : T: a set of fix templates
Input  : PopSize: population size
Output : Patch: a program variant that passes all test cases
1 let Pop ← initialPopulation(PopSize);
2 repeat
3 |   let Pop ← apply(Pop,T);
4 |   let Pop ← select(Pop,PopSize,Fit);
5 until ∃ Patch in Pop that passes all test cases;
6 return Patch
```

Our approach leverages evolutionary computing techniques [9] to generate program patches. Evolutionary computing is an iterative process in which a population is reproduced, evaluated, and selected. One cycle of these three steps is called a *generation*.

Algorithm 1 shows our approach following this evolutionary computing process. Our approach first takes a fitness function, fix templates, and population size as input. After creating an initial population of program variants equal in number to the given population size (Line 1), it iterates two tasks: generating new program variants by using fix templates (Line 3, reproduction) and selecting top variants based on the given fitness function (Line 4, evaluation and selection). This iteration stops when any program variant passes all given test cases (Line 5) or when it meets predefined termination conditions (see Section IV-A). Algorithm 1 returns a program variant that passes all test cases as a successful patch for the given bug (Line 6).

We adopt this evolutionary computing process, because it is effective in automatic patch generation [8] by efficiently exploring a large number of program variants. Applying an evolutionary computing process to program repair was pioneered by Weimer et al. [7] and Arcuri et al. [5].

The remainder of this section describes the details of fault localization, fix templates, and fitness function used in PAR.

A. Fault Localization

To determine fault locations, PAR uses statistical fault localization based on test cases [8]. This technique assumes that a statement visited by failing test cases is more likely to be a defect than other statements. Specifically, this technique assigns a value to each statement in a program. This value represents a degree of suspiciousness. Our approach uses that value to decide whether a statement should be modified.

This localization technique first executes two groups of test cases: passing and failing. Then, the technique records the statement-based coverage of both test case groups. Comparing the coverage of each statement results in one of the following four outcomes: 1) covered by both groups, 2) covered only by the passing group, 3) covered only by the failing group, and

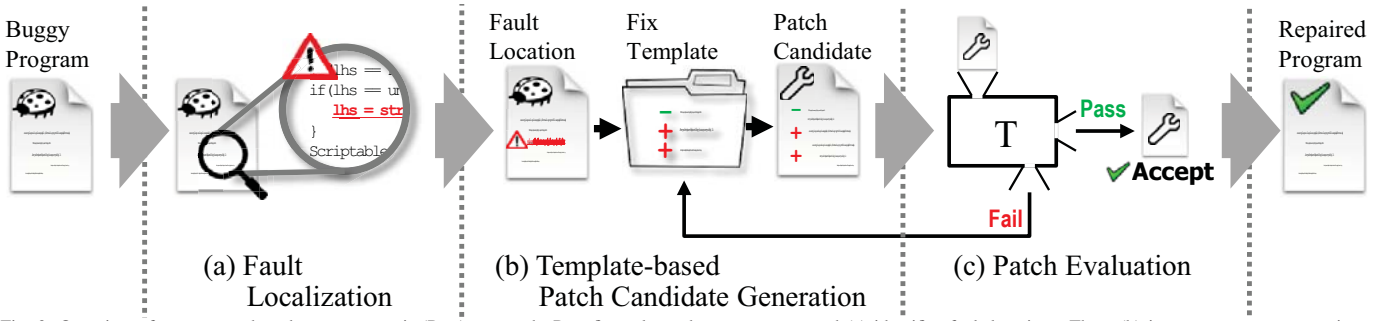


Fig. 3: Overview of our pattern-based program repair (PAR) approach. PAR first takes a buggy program and (a) identifies fault locations. Then, (b) it generates program variants by using fix templates, which are program editing scripts derived from common patch patterns (Section II-C). The templates modify source code around the fault locations. These variants are patch candidates. Finally, (c) patch candidates are evaluated by using test cases. If a candidate passes all test cases, we assume it is a patch for the bug. Otherwise, our approach repeats Steps (b) and (c) to generate another patch.

```
1  [Null Pointer Checker]
2  P = program
3  B = fault location
4
5  <AST Analysis>
6  C ← collect object references (method invocations,
     field accesses, and qualified names) of B in P
7
8  <Context Check>
9  if there is any object references in C ⇒ continue
10 otherwise ⇒ stop
11
12 <Program Editing>
13 insert an if() statement before B
14
15 loop for all objects in C {
16   insert a conditional expression that checks whether a
       given object is null
17 }
18 concatenate conditions by using AND
19
20 if B includes return statement {
21   negate the concatenated conditional expression
22   insert a return statement that returns a default value
       into THEN section of the if() statement
23   insert B after the if() statement
24 } else {
25   insert B into THEN section of the if() statement
26 }
```

Fig. 4: Null pointer checker fix template. This template inserts an `if()` statement checking whether objects are `null`.

4) not covered by either group. We assign 0.1 to statements with the first outcome and 1.0 to statements with the third outcome. Otherwise, 0.0 is assigned.

Our approach uses the assigned values to determine the probability of each statement to be edited. For example, 1.0 implies that the statement is always edited while 0.1 implies that it is edited once in 10 generations.

We adopted the simple fault localization technique used in [8], but other fault localization techniques can be used to determine which statements to mutate.

B. Fix Templates

Fix templates are program editing scripts that rewrite a program's Abstract Syntax Tree (AST). Each fix template defines three steps to edit a program: 1) AST analysis, 2) context check, and 3) program editing. The AST analysis step scans a given program's AST and analyzes the given fault location and its adjacent locations. The context check step examines whether the given program can be edited by a template by inspecting the analyzed AST. If it is editable, our approach rewrites the given program's AST based on the predefined editing script in the template (the program editing step).

In this section, we first describe how to create and apply the fix templates. Then, we provide the list of the fix templates used in PAR.

1) Creating Fix Templates: We first carefully inspect patches in each pattern (Table I). Since program patches change a program's AST, we can compute AST differences between before and after applying a patch. We transform these differences into editing scripts in a fix template. Note that patches in the same pattern may have various ASTs even though their semantic changes are identical. We try to generalize these changes in fix templates by identifying the most common change set.

For the context check step, we extract context information such as the existence of an array access from patches, which is necessary to check whether our approach can edit a program by using a fix template. Then, we add the AST analysis step to the template, which scans a program's AST at the fault location and extracts AST elements, such as variable names and types. We identify these elements by looking up fix patterns. These elements are used in the context check and the program editing steps.

For example, a fix template, *Null Pointer Checker*, is shown in Figure 4. This template is derived from the "Adding a null checker" pattern shown in Section II-C. To create this template, we first generalize how patches in the pattern change the ASTs of programs. Commonly, they insert an `if()` statement containing the fault location. The detailed program editing script is shown in Lines 13 – 26 (the program editing step). Then, we identify common context information necessary for program editing, which will be added to the context check step. For this template, the context check step verifies that the fault location must have at least one object reference (Lines 9–10). Finally, we add the AST analysis step (Line 6), which collects all object references in the given fault location.

In the same manner, we have created 10 fix templates as shown in Table II. Although creating fix templates requires manual effort, this is only a one-time cost, and the templates can be reused to fix other similar bugs. Our evaluation in Section IV confirms that the templates created from fix patterns in Eclipse JDT can be successfully applied to fix bugs in other programs such as Rhino.

TABLE II: Fix templates derived from patterns in Section II-C.

Template Name	Description
Parameter Replacer	For a method call, this template seeks variables or expressions whose type is compatible with a method parameter within the same scope. Then, it replaces the selected parameter by a compatible variable or expression.
Method Replacer	For a method call, this template replaces it to another method with compatible parameters and return type.
Parameter Adder and Remover	For a method call, this template adds or removes parameters if the method has overloaded methods. When it adds a parameter, this template search for compatible variables and expressions in the same scope. Then, it adds one of them to the place of the new parameter.
Expression Replacer	For a conditional branch such as `if()` or ternary operator, this template replaces its predicate by another expression collected in the same scope.
Expression Adder and Remover	For a conditional branch, this template inserts or removes a term of its predicate. When adding a term, the template collects predicates from the same scope.
Null Pointer Checker	For a statement in a program, this template adds `if()` statements checking whether an object is `null` only if the statement has any object reference.
Object Initializer	For a variable in a method call, this template inserts an initialization statement before the call. The statement uses the basic constructor which has no parameter.
Range Checker	For a statement with array references, this template adds `if()` statements that check whether an array index variable exceeds upper and lower bounds before executing statements that access the array.
Collection Size Checker	For a collection type variable, this template adds `if()` statements that check whether an index variable exceeds the size of a given collection object.
Class Cast Checker	For a class-casting statement, this template inserts an `if()` statement checking that the castee is an object of the casting type (using `instanceof` operator).

2) Applying Fix Templates: PAR applies a template to a fault location in each individual generation. As described in Section III-A, each fault location has a selection probability, and PAR uses that probability to determine which locations will be modified by a fix template. Since our approach follows the evolutionary computing process, each location can be edited by multiple templates over several generations.

When applying a fix template, PAR first takes a program and a fault location as input values. PAR then executes the AST analysis step in the template to collect necessary information such as variable types and method parameters. By using the collected information, PAR runs the context check step to figure out if the program has appropriate context to apply the given template. If the context check passes, our approach executes the template's program editing step to rewrite the program's AST of the given fault location. AST rewriting includes node addition, parameter replacement, and predicate removal. The modified program is a new program variant that PAR regards as a patch candidate.

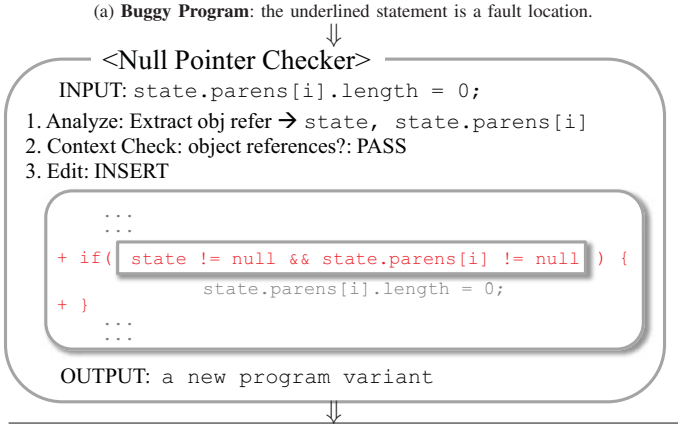
It is possible that several templates pass the context checking for given fault locations. In this case, PAR randomly selects one of the passing templates.

Figure 5 shows an example of applying the null pointer checker template (see Figure 4) to generate a new program variant from `NativeRegExp.java` for fixing Rhino Bug #76683. In this example, Line 4 in Figure 5(a) is a fault location. Our approach checks whether the location contains any object reference to ensure the Null Pointer Checker template is applicable. Since the location has two references, the template is applicable. Then, our approach generates a new program variant by applying the editing script in the template: inserting an `if` statement containing the fault location as described in Figure 4. As a result, a new program variant (i.e., patch candidate) has a predicate that checks whether the two objects are not `null`.

3) List of Fix Templates: Table II lists all 10 fix templates used in our approach. The *Parameter Replacer* template is derived from the "Altering method parameters" pattern in Table I. Its editing script can change parameters for a method call. The candidates of a substitutive parameter are collected from the same scope of the target method call at the fault location, and they must have compatible types. These parameter candidates

```
01  if (kidMatch != -1) return kidMatch;
02  for (int i = num; i < state.parenCount; i++)
03  {
04      state.parens[i].length = 0;
05  }
06  state.parenCount = num;
```

(a) **Buggy Program**: the underlined statement is a fault location.




```
01  if (kidMatch != -1) return kidMatch;
02  for ( ... )
03  {
04+     if( state != null && state.parens[i] != null)
05          state.parens[i].length = 0;
06  }
07  state.parenCount = num;
```

(b) **After applying a fix template**: a patch generated by PAR. As shown in the fix template, corresponding statements have been edited.

Fig. 5: Real example of applying a fix template to `NativeRegExp.java` to fix Rhino Bug #76683.

are sorted according to the distance (the number of nodes) from the given fault location in the given program's abstract syntax tree. Our approach selects one of them based on the distance and replaces method parameters of the statement at the given fault location. Parameters with a shorter distance are more likely to be selected.

The *Method Replacer* template replaces the name of the callee in method call statements. This template includes a context check step which scans other method calls having the same parameters and return type in the same program scope. If there are several candidates, our approach randomly selects one of them to replace the method name in the given fault location. The template is derived from "Calling another method with the same parameters" in Table I.

We created the *Parameter Adder and Remover* template to make a method call have more or fewer parameters. This

TABLE III: Data set used in our experiments. "LOC" (Lines of code) and "# statements" represent the size of each subject. "# test cases" is the number of test cases used for evaluating patch candidates generated by PAR.

Subject	# bugs	LOC	# statements	# test cases	Description
Rhino	17	51,001	35,161	5,578	interpreter
AspectJ	18	180,394	139,777	1,602	compiler
log4j	15	27,855	19,933	705	logger
Math	29	121,168	80,764	3,538	math utils
Lang	20	54,537	40,436	2,051	helper utils
Collections	20	48,049	35,335	11,577	data utils
Total	119	483,004	351,406	25,051	

template is applicable only if there are overloaded methods for the target method. Our approach selects one of the available overloaded methods randomly. When adding parameters, the template's script specifies how to scan the same scope of the given fault location for variables compatible with the new place. When removing parameters, our approach just filters out parameters not included in the selected overloaded method. This template is derived from the "Calling another overloaded method with one more parameter" pattern in Table I.

Expression Replacer and *Expression Adder and Remover* are derived from patches modifying predicates in conditional or loop statements such as `if()` or `while()`. Many patches we collected change or introduce new predicates to fix a bug as described by the "Changing a branch condition" pattern in Table I. To replace or add a predicate, our approach first scans predicates in the same scope of the given fault location. These are sorted according to their distances from the fault location in the given program's AST. Our approach selects one of them based on the distance and replaces or adds the predicate into the target statements at the fault location. When removing a term from a given predicate, our approach randomly selects a term to remove.

We created *Object Initializer* which inserts an initialization statement into the fault location. The initializer of this statement is the basic constructor without parameter. This statement prevents the variable from being `null`. This template is derived from the "Initializing an object" pattern in Table I.

Null Pointer Checker, *Range Checker*, *Collection Size Checker*, and *Class Cast Checker* are derived from corresponding patterns in Table I. By using these patterns, our approach can insert a new `if()` statement to check if there is any abnormal state such as null pointer, index out of bound, or wrong class-casting.

C. Fitness Evaluation

The fitness function of our approach takes a program variant and test cases, and then computes a value representing the number of passing test cases of the variant. This fitness function is adapted from [7], [8]. All the test cases are collected from the corresponding code repository for a given program. The resulting fitness value is used for evaluating and comparing program variants in a population: "*which variant is better than others?*" Based on fitness values of program variants, PAR chooses program variants by using a tournament selection scheme [18] (Line 4 in Algorithm 1).

IV. EVALUATION

We present the experimental evaluation of our approach in this section. Specifically, our experiments are designed to address the following research questions:

- **RQ1 (Fixability)**: How many bugs are fixed successfully?
- **RQ2 (Acceptability)**: Which approach can generate more acceptable bug patches?

A. Experimental Design

To evaluate PAR, we collected 119 real bugs from open source projects as shown in Table III. For each bug, we applied both PAR and GenProg [8] to generate patches. Then, we examined how many bugs are successfully fixed by each approach (RQ1). We also conducted a user study to compare the patch quality of the two approaches (RQ2).

Six projects in Table III are written in Java. We chose those projects for two main reasons. First, Java is one of the most popular programming languages[5], so there are many Java projects. Second, thanks to JUnit[6] and other Java-based testing frameworks, Java projects often include many test cases.

Many open source projects maintain their own issue tracking systems such as Bugzilla and JIRA. For our experiment, six open source projects including Mozilla Rhino, Eclipse AspectJ, Apache Log4j, and Apache Commons (Math, Lang, Collections) were selected, since they are commonly used in the literature [6], [19], [20] and have well-maintained bug reports. We tried to search their corresponding issue trackers for reproducible bugs. Among them, we randomly selected 15 to 29 bugs per project, since some projects had too many bugs. Although we invested our best effort in bug collection, the collected bugs did not represent the entire bugs. However, to our best knowledge, 119 was the largest number in automatic patch generation evaluation to date.

For each bug, we collected all available test cases from their corresponding code repositories including failing test cases that reproduce the bugs. Projects often include many test cases since developers continuously write and maintain test cases [21]. In our experiment, we used all test cases, as shown in Table III, to validate a candidate patch [6], [7], [10].

When applying PAR and GenProg to each bug, we conducted 100 runs. Arcuri and Briand recommended at least 1,000 runs for evaluating randomized algorithms [22], but we conducted 100 runs due to time limitation. Our experiment totaled 23,800 runs (100×119 bugs $\times$ 2 approaches).

Each run stopped when it took more than 10 generations or eight hours (wall-clock time), at which time we assumed that the run failed to generate a successful patch. We used exactly the same termination condition as used in [8] for fair comparison.

In addition, we used the same population size (=40) as used in GenProg. We also used the same parameters, such as the mutation probability suggested in [8] for running GenProg.

[5]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[6]http://www.junit.org/

TABLE IV: Patch generation results. Among 119 bugs, PAR successfully fixed 27 bugs while GenProg was successful for only 16 bugs. Note that 5 bugs were fixed by both approaches. We used these 5 bugs in our comparative study for acceptability evaluation.

Subject	# bugs	# bugs fixed by GenProg	# bugs fixed by PAR	# bugs fixed by both
Rhino	17	7	6	4
AspectJ	18	0	9	0
log4j	15	0	5	0
Math	29	5	3	1
Lang	20	1	0	0
Collections	20	3	4	0
Total	119	16	27	5

This experiment was conducted on several machines with two hexa-core 3GHz CPUs and 16GB RAM. When running test cases, we executed them in parallel to accelerate the experiment. In addition, we memoized the fitness value of a program variant to prevent re-evaluation of the same variant again, as described in [7].

B. RQ1: Fixability

Table IV shows patch generation results of PAR and GenProg. PAR successfully fixed 27 out of 119 bugs, while GenProg successfully generated patches for only 16 bugs.

Among fixed bugs, 5 (4 of Rhino + 1 of Math) were fixed by both PAR and GenProg. Note that the two approaches generated different patches for these five bugs. However, these patches passed all the given test cases successfully. These patches will be used in our comparative study for patch acceptability (Section IV-C).

Using fix templates is effective for fixing bugs. The Range Checker template generated a patch for Rhino bug #114493 by inserting an `if()` statement. This patch is shown in Figure 1(d). AspectJ bug #131933 could be fixed by the Class Cast Checker template because its buggy statements used an invalid caster. The "Expression Replacer" and "Expression Adder and Remover" templates could generate successful patches for two Rhino bugs (#192226 and #222635) that had invalid conditional expressions in buggy `if()` statements.

Note that we identified common fix patterns from Eclipse JDT and used them to create fix templates. PAR leverages these templates to generate patches for bugs of other projects such as Mozilla Rhino and Apache Commons Math. This indicates that fix templates learned from one project are reusable for other projects.

Although we have introduced only a small number of fix templates in this paper, those templates obviously expanded the fixability of patch generation. Identifying more fix templates from existing patches might improve fixability further. This remains as future work.

GenProg fixed about 10% of the bugs in our evaluation, while a recent systematic study [8] reported that it can fix almost 50% of the bugs in their subjects written in C language. Perhaps, this is because GenProg's mutation operators might be less effective in Java programs. Most Java programs tend to decompose their functionality into small-sized classes and methods. This limits the number of statements that can be used in GenProg's mutation operators since it collects and uses statements in the same scope of the given fault location. On the other hand, C programs usually have many global variables and larger methods. This may provide more chances to use bug-fixing statements in mutation (the authors of GenProg showed a patch for global variable accessor crashes as a representative example of successful cases in [8]). However, this does not imply that our subject selection was biased against GenProg. In our experiments, PAR had the same constraints as GenProg. In addition, our approach is not limited to Java programs.

> PAR *generated patches for* 27 *bugs, whereas GenProg resolved 16 bugs.*

C. RQ2: Acceptability

In this section, we measure the acceptability of patches generated by PAR and GenProg. Since all the successful patches pass the provided test cases, it is challenging to select more or less acceptable patches systematically. Instead, we presented anonymized patches and asked human subjects to select more or less acceptable patches.

To answer **RQ2**, we formulated the following two null hypotheses.

- $H1_0$: Patches generated by PAR and GenProg have no acceptability difference from each other.
- $H2_0$: Patches generated by PAR have no acceptability difference from human-written patches.

The corresponding alternative hypotheses are:

- $H1_a$: PAR generates more acceptable patches than GenProg.
- $H2_a$: Patches generated by PAR are more acceptable than human-written patches.

1) Subjects: For this study, we recruited two different participant groups: computer science (CS) students and developers. The student group consisted of 17 software engineering graduate students who have two to five-year Java programming experience. For the developer group, 68 developers participated in our study. We recruited these developers from both online developer communities such as "stackoverflow.com" and "coderanch.com", and software companies such as Daum, a leading Internet software company in Korea. They were asked to participate in this study only if they had Java programming experience.

2) Study Design: Our user study had five sessions. In each session, we showed one of the five bugs fixed by both PAR and GenProg, shown in Table IV. Each session explained in detail why a bug is problematic and gave a link to the corresponding bug report. Then, the session listed three anonymized patches: a human-written patch and patches generated by PAR and GenProg. Each participant was asked to compare them as a patch reviewer and to report their rankings according to acceptability.

For this study, we built a web-based online survey engine that shows five sessions in a random sequence. We gave the engine's hyperlink to both the student and the developer groups. At the beginning of our survey, we emphasized that the presented patches can pass all the given test cases collected from the corresponding project. There was no time limit, so

TABLE V: Average rankings evaluated by 17 students (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	**1.33** (0.62)	2.27 (0.59)	2.40 (0.83)
Rhino #114493	2.00 (0.54)	**1.33** (0.62)	2.67 (0.72)
Rhino #192226	**1.47** (0.64)	1.67 (0.62)	2.67 (0.72)
Rhino #217379	1.69 (0.70)	**1.50** (0.63)	2.81 (0.40)
Rhino #76683	2.13 (0.51)	**1.07** (0.26)	2.80 (0.41)
Average	1.72 (0.67)	1.57 (0.68)	2.67 (0.64)

TABLE VI: Average rankings evaluated by 68 developers (standard deviation is shown in parentheses). The lower values indicate that the patch obtained higher rankings on average by the evaluators.

Bugs	Human	PAR	GenProg
Math #280	**1.92** (0.76)	2.00 (0.82)	2.08 (0.95)
Rhino #114493	**1.60** (0.63)	2.40 (0.74)	2.00 (0.93)
Rhino #192226	2.00 (0.68)	**1.79** (0.98)	2.21 (0.80)
Rhino #217379	**1.62** (0.77)	1.69 (0.63)	2.69 (0.63)
Rhino #76683	1.92 (0.64)	**1.23** (0.43)	2.85 (0.38)
Average	1.81 (0.70)	1.82 (0.80)	2.36 (0.90)

that the participants had spend enough time inspecting and ranking the patches.

3) Result — Students: Average patch acceptability rankings assigned by 17 student participants are shown in Table V. Consistently, patches generated by PAR are ranked higher than those by GenProg for all five bugs. The average rankings of patches generated by PAR is 1.57, and its standard deviation is 0.68. The average ranking and standard deviation of patches generated by GenProg are 2.67 and 0.64, respectively. Ranking differences between PAR and GenProg are statistically significant (p-value =0.000 $<$ 0.05). The Wilcoxon signed-rank test [23] was used for this statistical test since we compared two related samples and these samples are non-parametric. Based on the results, we can reject the null hypothesis $H1_0$ for the student group.

Some students ranked patches generated by PAR as good as or even higher than human-written patches. However, this may not be necessarily indicate that patches generated by PAR are better than human-written patches (average: 1.72, standard deviation: 0.67). Their ranking differences are not statistically significant (p-value =0.257 $>$ 0.05). Thus, we cannot reject the null hypothesis $H2_0$ for the student group.

4) Result — Developers: The survey results of 68 developers are shown in Table VI. Similar to Table V, patches generated by PAR are ranked higher than those by GenProg except for Rhino Bug #114493. For this bug, the developers might think the patch generated by GenProg was more acceptable than the patch generated by PAR, since it assigned a default value when the `lhs` variable was `undefined` (see Figure 1(b)). However, this does not represent the overall results.

The average rankings of PAR and GenProg are 1.82 and 2.36, respectively. Their standard deviation values are 0.80 and 0.90. Since ranking differences between PAR and GenProg are statistically significant (p-value =0.016 $<$ 0.05), we can reject the null hypothesis $H1_0$ for the developer group.

The average ranking and standard deviation of human-written patches are 1.81 and 0.70, respectively. Ranking differences between patches generated by PAR and human-written patches are not statistically significant (p-value =0.411 $>$ 0.05). Thus, the null hypothesis $H2_0$ cannot be rejected.

TABLE VII: Indirect patch comparison results.

Selection	# response
PAR	130 (21%)
Both	175 (28%)
Human	229 (37%)
Not Sure	87 (14%)
Total	621 (100%)

(a) PAR comparison results.

Selection	# response
GenProg	68 (20%)
Both	40 (12%)
Human	176 (51%)
Not Sure	60 (17%)
Total	344 (100%)

(b) GenProg comparison results.

Our two comparative studies (student and developer) consistently show that patches generated by PAR have higher rankings than those generated by GenProg on average. In addition, this result is statistically significant. Between PAR and human-written patches, both studies show different results but ranking differences are not statistically significant. This implies that our approach can generate more acceptable patches than GenProg. In addition, patches generated by PAR are comparable to human-written patches.

5) Indirect Patch Comparison: The user studies in Sections IV-C3 and IV-C4 showed the direct patch comparison results, but the results are limited to five bugs fixed by both approaches.

To address this issue, we conducted a user study to indirectly compare the acceptability of all 43 patches generated by PAR (27 patches) and GenProg (16 patches), by comparing them to the corresponding human-written patches. We built a web-based online survey engine for this study. Each survey session showed a pair of anonymized patches (one from human and the other from PAR or GenProg for the same bug) along with corresponding bug information. Participants were asked to select more acceptable patches if they were patch reviewers. In addition, participants were given the choice of *both* are acceptable or *not sure* if they could not determine acceptable patches. We randomly presented all 43 sessions to participants, and they could answer as many sessions as they wanted.

For this study, we recruited participants by posting online survey invitations in software developer communities and personal twitters. We also sent invitation emails to CS undergraduate students who took the Software Engineering class in Spring 2012 at the Hong Kong University of Science and Technology, since they have Java programming knowledge. In all survey invitations, we clearly stated that only developers/students who have Java experience are invited.

The survey results are shown in Table VII. Total 168 (72 students and 96 developers) participants answered 965 sessions. The session response rate (PAR:GenProg = 621:344) is similar to the rate of successful patches generated by each approach (PAR:GenProg = 27:16). This implies that our survey sessions were randomly presented and answered.

As shown in Table VII(a), participants chose patches generated by PAR as more acceptable in 130 (21%) out of 621 sessions and selected "*both were acceptable*" in 175 (28%) sessions. In total, participants chose patches generated by PAR as acceptable patches in 305 (49%) sessions (PAR: 21% + both: 28%). On the other hand, participants selected patches generated by GenProg as acceptable patches in 108 (32%) out of 344 sessions (GenProg: 20% + both: 12%), as shown in Table VII(b).

This indirect comparison result also shows that patches generated by PAR are more acceptable.

> PAR *generates more acceptable patches than GenProg does.*

V. DISCUSSION

This section discusses unsuccessful patches generated by PAR, and identifies threats to validity of our experiments.

A. Unsuccessful Patches

In Table IV, 92 out of 119 bugs were not fixed by our approach. We examined the main cause of patch generation failures and identified two main challenges: *branch conditions* and *no matching patterns*, as shown in Table VIII. We discuss each challenge in detail.

Branch conditions indicates that PAR cannot generate predicates to satisfy branch conditions at the fault location by using fix templates. PAR could not fix 26 (28%) bugs due to this reason. For example, Rhino bug #181834 occurs when the `scope` variable is assigned to a `NativeCall` or `NativeWith` type object. To fix this bug, an appropriate control statement must be inserted before the fault location. This control statement has a predicate checking the type of `scope`. Generating this predicate from scratch is challenging.

No matching pattern indicates that PAR cannot generate a successful patch for a bug since no fix template has appropriate editing scripts. PAR could not fix 66 (72%) bugs due to this reason. For example, to fix AspectJ Bug #109614, its human-written patch added a control statement before the fault location: "`if(sources[i] instanceof ExceptionRange)...`". However, `ExceptionRange` cannot be inferred from the fault location. For PAR to fix this bug, matching fix templates must be created.

We inspected 11 bugs fixed by GenProg but failed to be fixed by PAR. Our approach could not fix 7 out of 11 bugs, since there are no matching fix patterns for these bugs. This implies if we add more fix patterns, these bugs might be fixable by our approach. Adding more fix patterns remains as future work. The other 4 bugs belong to the *branch conditions* category.

B. Threats to Validity

We identify the following threats to the validity of our experiments.

- **Systems are all open source projects:** We collected bugs only from open source projects to examine our approach. Therefore, these projects might not be representative of closed-source projects. Patches of closed-source projects may have different patterns.
- **Some participants of our user studies may not be thoroughly qualified.** In our survey invitations for the developer group, we clarified that only developers can participate in the survey. However, we could not fully verify the qualifications of the survey participants.

TABLE VIII: Causes of unsuccessful patches.

Cause	# of bugs
Branch condition	26 (28%)
No matching pattern	66 (72%)

VI. RELATED WORK

Weimer et al. [7] proposed GenProg, an automatic patch generation technique based on genetic programming. This approach randomly mutates buggy programs to generate several program variants that are possible patch candidates. The variants are verified by running both passing and failing test cases. If a variant passes all test cases, it is regarded as a successful patch of a given bug. In 2012, the authors extended their previous work by adding a new mutation operation, *replacement* and removing the *switch* operation [8]. In addition, they provided systematic evaluation results with 105 real bugs [8]. Although the evaluation showed that GenProg fixed 55 out of 105 bugs, GenProg can generate nonsensical patches that may not be accepted by developers as shown Section IV-C.

Fry et al. conducted a human study to indirectly measure the quality of patches generated by GenProg by measuring patch maintainability [24]. They presented patches to participants and asked maintainability related questions developed by Sillito, Murphy, and Volder [25]. In addition, they presented machine-generated change documents [26] along with patches to participants. They found that machine-generated patches [8] with machine-generated documents [26] are comparable to human-written patches in terms of maintainability. We also compared machine-generated patches with human-written patches. However, instead of asking the maintainability related questions, we asked participants which patch is more/less acceptable for direct comparison. In addition, we compared patches generated by two different patch generation approaches, GenProg and PAR.

Demsky et al. focused on avoiding data structure inconsistency [27], [28]. Their approach checks data structure consistency by using formal specifications and inserts runtime monitoring code to avoid inconsistent states. However, this technique provides workarounds rather than actual patches since it does not modify source code directly.

Arcuri et al. introduced an automatic patch generation technique [5]. Although they also used genetic programming, their evaluation was limited to small programs such as bubble sorting and triangle classification, while our evaluation includes real bugs in open source software. Their approach relies on formal specifications, which our approach does not require.

Wei et al. proposed a contract-based patch generation technique [10]. This technique also relies on specifications (i.e., contracts). In addition, this technique can generate only four kinds of program variants. These variants check only contract violations, whereas our approach generalizes human-written patches and generates various program variants.

PACHIKA [6] leverages object behavior models. PACHIKA is evaluated on 26 bugs from Mozilla and Eclipse. However, PACHIKA created successful patches for only three out of 26 bugs in their evaluation since it generates only a limited

number of program variants (i.e., recorded object behavior). Our evaluation also includes these 26 bugs (see Table III), and PAR successfully fixed 15 bugs including the three bugs fixed by PACHIKA.

Martinez and Monperrus identified common program repair actions (patterns) from around 90,000 fixes [15]. They claimed that most common repair actions are semantic change patterns such as "Additional functionality." However, their identified patterns are too abstract and coarse-grained to be used in automatic patch generation. Our fix templates described in Section III-B3 are concrete enough to be used in automatic patch generation.

SYDIT [29] automatically extracts an edit script from a program change. Its limitation is that a user must specify a program change to extract an edit script and a target program to apply it. In addition, SYDIT cannot take multiple program changes to extract a (generalized) edit script. SYDIT's authors proposed an improved technique called LASE [30]. This technique can take multiple changes to extract more general edit scripts and automatically find a target program. PAR can leverage these techniques to automatically create fix templates.

VII. CONCLUSION

In this paper, we have proposed a novel patch generation approach, PAR, learned from human-written patches to address a limitation of existing techniques such as GenProg [7], [8]: generating nonsensical patches. We first manually inspected human-written patches and identified common fix patterns, which we then used in automatic patch generation. Our experimental results on 119 real bugs showed that our approach successfully generated patches for 27 bugs, while GenProg was successful for 16 bugs. To evaluate whether the generated patches were acceptable for fixing bugs, we conducted a user study of 253 participants (89 students and 164 developers). This study showed that our approach generated more acceptable patches than GenProg, and our patches were comparable to human-written patches.

Our pattern-based approach might be useful in improving other automatic patch generation techniques. For example, contract-based techniques [10] alone can generate only four kinds of variants for each bug, but fix templates could be used to generate more program variants for such techniques. PAR could also be used to generate more program variants in model-based techniques [6].

Our future work includes *automatic fix template mining* and *balanced test case generation*. First, although the proposed fix templates are successfully used for generating patches, more fix templates are desirable to fix more bugs efficiently. We plan to investigate more human-written patches and develop automatic algorithms to extract fix templates. Second, our approach requires test cases to evaluate program variants, but often few failing test cases are available for bugs. Since the imbalance in test cases may lead to inaccurate patch evaluation, we will develop failing-test generation techniques by leveraging existing automatic testing techniques [31], [32], [20].

All materials used in this paper and detailed patch generation results are publicly available at

`https://sites.google.com/site/autofixhkust/`

REFERENCES

[1] The Guardian, "Why we all sell code with bugs," Aug 2006.
[2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.
[3] Technology Review, "So many bugs, so little time," Jul 2010.
[4] CNN, "Will bugs scare off users of new Windows 2000," Feb 2000.
[5] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *CEC '08*.
[6] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *ASE '09*.
[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE '09*.
[8] 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 '12*.
[9] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, 1st ed. The MIT Press, Dec. 1992.
[10] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA '10*.
[11] D. Babich, P. J. Clarke, J. F. Power, and B. M. G. Kibria, "Using a class abstraction technique to predict faults in OO classes: a case study through six releases of the eclipse JDT," in *SAC '11*.
[12] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *ICSE '08*.
[13] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with kenyon," in *ESEC/FSE '05*.
[14] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *ICSE '12*.
[15] M. Martinez and M. Monperrus, "Mining repair actions for guiding automated program fixing," INRIA, Tech. Rep., 2012.
[16] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE '09*.
[17] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do developers understand code changes? an exploratory study in industry," in *FSE '12*.
[18] B. L. Miller and D. E. Goldberg, "Genetic algorithms, selection schemes, and the varying effects of noise," *Evol. Comput.*, vol. 4, no. 2.
[19] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code," in *ICSE '12*.
[20] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: object capture-based automated testing," in *ISSTA '10*.
[21] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *FSE '12*.
[22] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *ICSE '11*.
[23] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6.
[24] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *ISSTA '12*.
[25] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *FSE '06*.
[26] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *ASE '10*.
[27] B. Demsky and M. Rinard, "Data structure repair using goal-directed reasoning," in *ICSE '05*.
[28] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *ISSTA '06*.
[29] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," in *PLDI '11*.
[30] ——, "Lase: Locating and applying systematic edits by learning from examples," in *ICSE '13*.
[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed random test generation," in *ICSE '07*.
[32] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Trans. on Softw. Eng.*, vol. 37, no. 1, pp. 80–94, Feb. 2011.