2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

relifix: Automated Repair of Software Regressions

Shin Hwei Tan and Abhik Roychoudhury
National University of Singapore
{shinhwei,abhik}@comp.nus.edu.sg

***Abstract*—Regression occurs when code changes introduce failures in previously passing test cases. As software evolves, regressions may be introduced. Fixing regression errors manually is time-consuming and error-prone. We propose an approach of automated repair of software regressions, called *relifix*, that considers the regression repair problem as a problem of reconciling problematic changes. Specifically, we derive a set of code transformations obtained from our manual inspection of 73 real software regressions; this set of code transformations uses syntactical information from changed statements. Regression repair is then accomplished via a search over the code transformation operators – which operator to apply, and where. Our evaluation compares the repairability of *relifix* with GenProg on 35 real regression errors. *relifix* repairs 23 bugs, while GenProg only fixes five bugs. We also measure the likelihood of both approaches in introducing new regressions given a reduced test suite. Our experimental results shows that our approach is less likely to introduce new regressions than GenProg.**

I. Introduction

Regression captures the scenario where failures occur in previously passing tests. As software evolves due to changes in software requirement and bug fixes, regression bugs may be introduced. Even worse still, fixing a regression bug is likely to introduce another regression bug due to low-quality patches and inadequate testing.

Prior studies on regression errors primarily focus on techniques for localizing and understanding of regressions. The delta debugging approach searches for failure-inducing circumstances contributing to test failures (i.e., the set of code changes and the state differences between passing and failing tests) using a divide-and-conquer algorithm [55]. Given a reference program, a buggy program, and an input that fails on the buggy program, the Darwin approach generates alternative input that fails on the buggy program, then compare the executions of the two inputs to pinpoint the root cause of the error [46]. Previous studies show promising results in locating the cause of regression errors. However, after locating the cause of regression errors, how do we utilize the availability of a previous working version to automatically repair such errors? This is addressed in the current paper.

Fixing regression errors manually is time-consuming and error-prone. Recent study stated that some regression errors could take up to 8.5 years before they are detected and fixed by developers [29]. Recently, several automated program repair techniques have been introduced. Arcuri and Yao suggested adapting evolutionary algorithms for automatic program generation [26]. Weimer et al. utilized genetic programming for automated program repair [33], [37]. Wei et al. leverages software contract to automatically fix faulty Eiffel classes [52]. Nguyen et al. employed symbolic execution and component-based program synthesis for discovering the code required for fixing the buggy program [44]. Kim et al. proposed an automated patch generation approach (i.e., PAR) that utilizes common fix patterns learned from manual inspection of human patches [35]. Recent study shows that statements or expressions required for fixing exist in previous commits of the programs [28], [41]. However, existing automated program repair techniques have not fully exploited information from the software change history for automated repair of regressions. In this paper, we verify the possibility of using syntactical information between program versions and test execution history to repair problematic changes that causes regressions.

The key challenge in repairing regression errors is to retain as much of the new functionalities introduced along with the new version as possible while reproducing the regression tests' behavior in the previous version.

Criteria 1: We want our automated repair of regression error to follow the following criteria:

C1: Introduces small changes Retains as much of the code of the new version as possible as more changes may lead to more regression errors.

C2: Produces readable code Generates source code that developers can understand and verify easily.

C3: Passes progression tests Progression tests that pass in the new version and fail in the previous version must remain passing after the repair.

C4: Passes previously failed regression tests Regression tests that fails in the previous version and pass in the current version must be made passing in the new version.

C5: Only change if no regression will be introduced If changes caused other tests in the test suite to fail, then leave the source code unchanged. The repaired version should not introduce further regression error.

We present a novel approach, called *relifix*, for automated repair of software regressions. In particular, our contributions can be summarized as follows:

New Domain: We focus on program fixing on a new domain, specifically on repairing software regression errors. This domain was not studied in prior work in automated program repair, but various researches on fault localization [27], [46], [54], [55] and regression testing [49], [50] showed that this domain is important and widely represented in software development activities.

New Perspective: We formulate the software regression repair problem as a problem of reconciling problematic

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.65

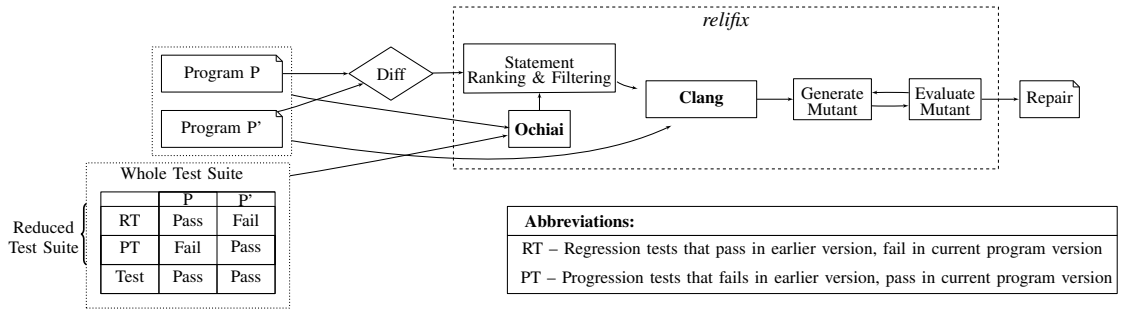

Fig. 1. *relifix*'s Overall Workflow

changes. We hypothesize that the fixes for regression bugs can be crafted using code from both the previous version (specifically, the preceeding version before the regression error occurs) and the current program version. We justify this formulation further in section II. This formulation allows us to introduce fixes only to the changed lines.

Program Repair using previous version : Our approach leverages different program versions and code changes for guiding automated repair of regression bugs.

New contextual operators: We manually inspected 73 real regression bugs from the CoREBench benchmarks [29]. Our manual inspection produces a set of operators that uses information from two program versions, including changed statements and program location of the changed statements.

Evaluation: We applied *relifix* on seven open-source C projects (`Make`, `Find`, `Vim`, `Tar`, `Indent`, `Python` and `Perl`), which have well-developed and well-tested code. We compare the repairability of *relifix* with GenProg on 35 real regression errors. *relifix* successfully repaired 23 bugs, while GenProg only fixes five bugs. To compare the likelihood of both approaches in introducing new regressions, we also evaluated the regression rate of both approaches given the reduced test suite (test suite that contains the tests with different test behaviors in the two program versions). Our experimental results show that our approach is less likely to introduce new regressions compared to GenProg.

II. Repairing Regression as Reconciling Problematic Changes

When a developer fixes a regression error, he or she needs to execute the failing regression test, locate the cause of the test failure, and fix the current version of the program by referring to the previous version. This suggests that while repairing regression we probably try to find a fix that replicates the regression tests' behavior in the previous version. In fact, trivial fixes exist in the context of regression – execute the previous version for the failing regression test and run the current version for the remaining test cases. Such fixes are quick to issue and they pass all tests in the test suite. However, they are costly to maintain and difficult to understand. Worse still, the number of program versions will double when another regression bug emerges. Thus, a good repair must be able to reproduce the regression tests's behavior in the previous working version while maintaining the working functionalities for the current version.

There are generally three types of software regressions.

Local A code modification breaks existing functionality in the changed program element.

Unmask A code modification unmasks an existing bug that had no effect to some test's behavior before the modification.

Remote A code modification introduces a bug in another unchanged program element.

Intuitively, if a functionality works in the previous version, the **Local** regression error can be fixed by rolling back to its old implementation in previous version. This intuition is supported by the *Revert to previous statement* operator derived from the Corebench benchmarks. Our hypothesis that fixes for regression errors exist in the immediate version before a regression error occurs, is in line with this intuition. Recent studies that speculate on the probabilities of locating fixes from multiple program versions [28], [41] further validates this hypothesis.

In contrast, if some source code in previous version had successfully hidden an existing bug, the **Unmask** regression error may be fixed by re-masking the problematic changes. In this case, fixing regression involves searching for a condition under which the problematic code modifications have no effect to the tests' behavior. This intuition is supported by the *Add condition* operator derived from the Corebench benchmarks. We hypothesize that the condition for hiding the problematic changes can be found among the program expressions in the current version. Some of the existing automated program repair techniques [35], [37] share similar hypothesis.

From these two observations, we formulate the software regression repair problem as a problem of reconciling problematic changes.

III. Experience about real-life regressions

We manually examined 73 benchmarks obtained from four subject programs to understand how software evolves during real-world regressions. We used the CoREBench benchmarks[1] for our manual investigation. This set of benchmarks is derived from regression errors that were systematically

[1]http://www.comp.nus.edu.sg/~release/corebench/

deduced from version control repositories and bug reports of four open source GNU projects (i.e., `Make`[2], `Grep`[3], `Findutils`[4],`Coreutils`[5]).

For each of the 73 benchmarks, we examined two set of code changes: (1) changes that occur between the version before the regression is introduced (i.e., version P1) and the version immediately after P1 (i.e., version P2); and (2) code changes that occur between the version before the regression is fixed (i.e., version P3) and the version after the regression is fixed (i.e., version P4). We refer to each program version as P1, P2, P3 and P4 to denote the corresponding program version for the rest of the paper. We then derived a set of general code transformations by comparing version P3 with P4. This set of code transformations form the operators that can be applied to repair the regression bugs. Table I shows the code transformation derived from the benchmarks, together with the number of benchmarks that uses the corresponding operator in regression bug fixing. The table is sorted with the most commonly used operators at the top of the table. The last row in the table shows that 43 repairs that involves two code transformation operators (the other 73-43 = 30 repairs involve only one code transformation operator). Overall, our manual inspection shows that information given by code changes between program versions are often included in human patches.

A. Contextual Operators that use information from different program versions

Our manual inspection produces a set of operators that utilize information obtained from the previous version and from the code changes that occur between two consecutive program versions. We refer to this set of operators as *contextual operators* due to their references to different versions. We next provide examples of our contextual operators. Example code with a leading "-" denotes the statement removed from P4, while code with leading "+" marks the statement added from P3 to P4. Code without any leading symbol denotes the unchanged statement.

Below are the details of each contextual operator:

Use changed expression as input for other operator This operator uses the program expressions that change (i.e., modify, add or remove) between two versions as input to other non-contextual operators (e.g., Add condition). The example below shows a patch in regression bug-fixing for `Coreutils`. The expression `max_range_endpoint < eol_range_start` was removed in the evolution from version P1 to version P2.

```
if (output_delimiter_specified && !complement &&
     eol_range_start && max_range_endpoint
- && !is_printable_field(eol_range_start))
+(max_range_endpoint < eol_range_start
+|| !is_printable_field(eol_range_start)))
```

Listing 1. Example for use changed expression as input for other operator

Revert to previous statement This operator replaces newly added statements with the corresponding statements from old version, essentially reverting back some statements to the old version. The example in the following show a loop that was removed when `Make` evolves from version P1 to version P2. The developer added back the same loop to fix the regression in version P4.

```
- while(out > line && isblank((unsigned char)out[-1]))
- --out;
```

Listing 2. Revert to previous statement example

Remove incorrectly added statement This operator deletes program statements that were incorrectly added by the developer due to wrong bug fix.

Swap changed statement with neighbouring statement This operator exchanges a changed statement with another consecutive statement. The changed statement serves as the pivot position for the exchange. Listing 3 shows the code modifications from version P3 to version P4. The `FindUtils` developer added the statment `our_pred->est_success_rate = estimate_pattern_match_rate(...);` in version P2, and later changed the statement order relative to the statement `(*arg_ptr)++;`.

```
- (*arg_ptr)++;
our_pred->est_success_rate =
     estimate_pattern_match_rate(argv[*arg_ptr], 1);
+(*arg_ptr)++;
```

Listing 3. Example for Swap changed statement with neighbouring statement

Negate added condition This operator negates a branch condition that was previously added by the developer. For example, the `Grep` developer added the condition `included_patterns && !excluded_file_name(...)` in version P2. The bug is fixed by changing the added condition to `included_patterns && excluded_file_name(...)` in version P4.

Convert statement to condition variable statement This operator convert a statement with `Boolean` return type to a condition variable statement. Listing 4 presents the changes between version P3 and P4. The Coreutils developer forget to check for the condition when the set statement returns 0. The fix requires converting the set statement to a condition variable statement.

```
- set_acl(dst_name, dest_desc, 0666 &~cached_umask());
+ if(set_acl(dst_name, dest_desc, 0666 & ~cached_umask
     ()) != 0)
```

Listing 4. Code changes between version P3 and P4 to illustrate convert statement to condition variable statement

IV. Example

We illustrates how *relifix* can be used by showing three examples of fixes generated by *relifix* in three projects. The first two examples illustrate various operators involved in the fixes while the last example compares our generated fixes with the patches issued by developers. Consider first the `Vim`

[2]http://www.gnu.org/software/make/
[3]http://www.gnu.org/software/grep/
[4]http://www.gnu.org/software/findutils/
[5]http://www.gnu.org/software/coreutils/

TABLE I
THIS TABLE SUMMARIZES THE NUMBER OF CODE TRANSFORMATION OPERATORS THAT ARE USED FOR FIXING REGRESSION BUGS.

Operator	Operator Type	Implemented	Count
Add condition	Non-contextual	✓	27
Add statement	Non-contextual	✓	21
Use changed expression as input for other operator	Contextual	✓	13
Revert to previous statement	Contextual	✓	11
Replace with new expression	Non-contextual		13
Remove incorrectly added statement	Contextual	✓	9
Change type	Non-contextual		5
Add method	Non-contextual		5
Add parameter	Non-contextual		4
Add local variable	Non-contextual		3
Swap changed statement with neighbouring statement	Contextual	✓	2
Negate added condition	Contextual	✓	1
Convert statement to condition variable statement	Contextual	✓	1
Add field	Non-contextual		1
Total	6 Contextuals	8	116
Total cases requiring 2 operators			43

project [6], a popular editor that supports efficient text editing. A regression is introduced in version 7.2.50 of `Vim`, causing failures in two tests in `Vim`'s existing test suite. Listing 5 shows the repair generated by *relifix* with an application of the *Revert to previous statement* operator. This example demonstrates how *relifix* repairs a **Local** regression error. Note that there are approximately 18 *change hunks*[7] in the faulty source files between the two program versions, while the produced repair only modifies one hunk. Instead of reverting the entire source files to the previous version, the produced repair only reverts the faulty lines. This shows that our repair satisfies the criterion **C1**.

```
- fwrite(p, 1, (size_t)1, fd);
+ fwv &= fwrite(p, 1, (size_t)1, fd);
```

Listing 5. Example patch generated by *relifix* using the "Revert to previous statement" operator

Consider next the GNU `Indent` project [8], a utility that formats C source files according to specific indent style. An **Unmask** regression occurs in version 2.2.10 of Indent, causing the buggy version to append too many newlines between variable declarations of a C source files. Listing 6 shows the repair generated by *relifix* using the "Add inverted condition" operator and "Use changed expression as input for other" operator. *relifix* first generates the condition `!(parser_state_tos->decl_on_line)` by negating an existing `Boolean` expression. This intermediate patch passes the reduced test suite that contains one failing test, but it introduces new regression in other tests from the whole test suite. *relifix* then repairs the regression that it introduced by modifying the changed lines (i.e., the added condition). The final patch that passes all tests in the entire test suite is formed by appending another condition `parser_state_tos->procname != "\0"` (i.e., condition obtained by converting the assignment statement `parser_state_tos->procname = "\0"` to disequality) to the intermediate patch. This example illustrates the two-phase patch evaluation performed by *relifix*.

We next discuss one example that illustrates the differences between the patch generated by our approach and the patches issued by the developer. For this example, consider the GNU `Make` project[9], a tool that builds executables for a program from its source files. Two regression bugs (i.e., bug #12202 and bug #12267) are introduced with version 73e7767 of `Make`. Listing 7 shows the two patches generated in two different commits by the `Make` developer to fix the bugs. Listing 8 presents the code changes that causes the regression, while listing 9 shows the single patch generated by *relifix* that repairs both regression errors. To fix both regression bugs, *relifix* appends the condition `isintermed_ok` to one of the code change hunks. In this case, we consider that the code changes in listing 8 unmask an latent regression error and the added condition `isintermed_ok` has successfully masked both regression errors. While the fix generated by *relifix* is significantly different from the developer's patches, it may be preferable because (1) it satisfies all criteria in Criteria 1 , and (2) it fixes both regression errors using only one patch.

V. ALGORITHM

Figure 1 shows the overall work-flow of our approach. Our *relifix* approach follows a three-step process. The first step takes as input the source code of the two program versions and the whole test suite with at least one failing test case that captures the regression error, and generates a ranked list of suspicious statements. The second step modifies the source code for the buggy version at the program location according to the list generated at previous step to produce a candidate repair. The last step builds the modified source code and re-execute the test suite to check if the generated repair passes all test cases. The main novelty in our work is in coming up

[6]http://www.vim.org/

[7]A change hunk is a single sequence of contiguous source codes which has been modified from one version to another [39], [43], [45].

[8]http://www.gnu.org/software/indent/

[9]http://www.gnu.org/software/make/

```
// Patch that repairs the reduced test suite
+if (/* added */(!(parser_state_tos->decl_on_line)){
  ...
+}

// Patch that repairs all tests in the test suite
if (/* added */(!(parser_state_tos->decl_on_line)
+|| parser_state_tos->procname != "\0")}){
```

Listing 6. Example patches generated by *relifix* using two operators

```
// Developer fixes for regression bug #12202
+ f->is_target = 1;
...
 +file->is_target = 1;

// Developer fixes for regression bug #12267
- register struct file *f = enter_file (imf->name);
+ register struct file *f = lookup_file (imf->name);
+ if (f != 0)
+   f->precious = 1;
+ else
+   f = enter_file (imf->name);
...
+if (!f->precious)
```

Listing 7. Example patches issued by the developer

```
//In file.c
+ f2->is_target = 1;
// In implicit.c
+ struct file *f;
...
- if (lookup_file (p) != 0
+ if (((f = lookup_file (p)) != 0 && f->is_target)
```

Listing 8. Example code change hunks between version 73e7767 and its preceeding versions

```
// Patches generated by relifix
- if (((f = lookup_file (p)) != 0 && f->is_target)
+ if (((f = lookup_file (p)) != 0 && (f->is_target ||
     isintermed_ok))
```

Listing 9. Example patch generated by *relifix* using one operator

with the contextual operators, and then applying them at the "right" places.

A. Fault Localization

Our goal is to find a faulty program location that leads to the regression error. We first compute a suspiciousness score for each statement in the buggy program using the Ochiai formula [23] given below:

$$\text{suspiciousness}(s) = \frac{\text{failed}(s)^{10}}{\sqrt{\text{total failed}^{13} \times (\text{failed}(s) + \text{passed}(s)^{14})}}$$

We choose the Ochiai formula due to its effectiveness demonstrated by previous studies [21], [22]. To obtain the code changes between the two versions, we use the open-source GNU Diffutils [19]. Diffutils perform plain text comparisons to find the differences between two text files. After sorting the suspiciousness score for each statement, we remove the statements that do not lie within the set of modified statements from the list of suspicious statements. This step allows us to (1) reduce the inspection cost for the location to apply the fix and (2) increase the probability of applying our contextual operators. We share assumptions that are commonly used to evaluate testing and debugging techniques [32], that the error is among the changed statements. Note that this may lead to loss of residual latent error that are not manifested by the current test suite.

[18]failed(s): Number of failing tests that executes statement s
[18]total failed: Total number of failing tests
[18]passed(s): Number of passing tests that executes statement s
[19]http://www.gnu.org/software/diffutils/

B. Mutant Generation and evaluation

1) Algorithm: After *relifix* generates the list of suspicious statement, it uses our mutation generation component to generate a mutant. Our definition of program mutant is similar to prior work on mutation testing [31]: each mutant is defined as a program that are modified through some applications of mutation operators at some faulty locations.

We collect a set of program expressions of type `Boolean` to be combined with all the parameterizable operators at later step. This set contains (1) all `Boolean` program expressions (that are within the program scope at the faulty location), and (2) expressions formed by converting the assignment operators in all the assignment statements (that are within the program scope at the faulty location) to the equality operators.

Algorithm 1 shows the pseudo-code of the *relifix* mutant generation and evaluation algorithm. Our *relifix* approach applies a randomly chosen contextual operator at a faulty location, evaluates each mutant against the current test suite, and iteratively repeats these steps until all the tests pass, no contextual operators apply, or the time limit is reached. Before applying each operator, we check whether the statement at the faulty location matches the given context for the operator and gather the required contextual information from both program versions. For example, if the faulty location has integer return type, the *Convert statement to conditional statement* operator cannot be applied.

We implement two optimizations for our random search algorithm: (1 – highlighted in red) we store the index for the program expressions that do not compile in a *tabu* list, which helps us to avoid reusing program expressions that are not compilable. (2 – highlighted in blue) we enumerate the number of *well-formed mutants* (Definition 1). As each *well-*

Input: List of suspicious statements $RankList$
Input: Set of test suite T, Reduced test suite $T_r \subseteq T$
Input: List of contextual operators O
Input: Set of program expression E
Input: Period P – the number of iterations for each location before considering next location
Output: Program mutant that passes all test cases

```
iter ← 0; currO ← Shuffle(O); currTS ← T_r;
Tabu ← {}; currL ← 0; currC ← original program;
while repair not found do
    currL ← next top ranked location ∈ RankList;
    changedCount ← 0;
    while iter ≤ P ∧ changedCount < size(CurrO) − 1
    do
        op ← Dequeue(CurrO);
        if op is parameterizable then
            /* select expression that are not in tabu */
            repeat
                currE ← randomly chosen expr ∈ E;
            until currE ∉ Tabu;
            /* apply operator op with currE as parameter to
            candidate currC at location currL */
            c ← currC.apply(op, currL, currE);
        else
            /* apply operator op to candidate currC at
            location currL */
            c ← currC.apply(op, currL);
        end
        Result ← Evaluate(c, currTS);
        /* two-phrase mutant evaluation */
        if ∀r ∈ Result, r = passes then
            currTS ← T;
            AResult ← Evaluate(c, currTS);
            //check if repair is found
            if ∀a ∈ AResult, a = passes then
                break;
            else
                /* c causes new regressions, repair c with the
                whole test suite */
                currC ← c;
                /* reset and re-shuffle O */
                currO ←Shuffle(O);
            end
        else
            /* check if the operator op can be applied at
            location currL and if candidate c is compilable */
            if canBeApplied(op, currL) ∧ isCompiled(c)
            then
                /* reuse operator used in candidate c if it
                induces any change in the test execution
                results for any test in the test suite */
                if op is parameterizable ∧ ∃r ∈
                Result,diffResult(r) then
                    Enqueue(CurrO, op);
                end
                changedCount ← changedCount + 1;
                iter ← iter + 1;
            else
                Tabu ← Tabu ∪ currE;
            end
        end
    end
end
```

Algorithm 1: *relifix* Mutant Generation and Evaluation Algorithm

formed mutant indicates progress in generating the final repair, the operator involved in generating that mutant can be reused in generating the next set of mutants.

Definition 1: Well-formed mutants are mutants that satisfy the following conditions:

Compilable Mutants generated should not generate any compilation errors

Match Given Context The program location and the structural type of the program element must match the context for the chosen operator used in generating the repair.

Induce Change in Test Execution Results Mutants generated should induce changes that affect the test behavior of some tests within the test suite.

2) Implementation: We modify the *clang-mutate* tool [20] to implement our mutant generation component. *clang-mutate* is built on top of the Clang [21] LibTooling library that offers utilities for parsing C programs and performing source-to-source transformations. Our mutant generation component satisfies the criterion **C2** as it modifies C source files directly to produce understandable code annotated with code comments (see Section IV for examples of our generated code).

We implement all the contextual operators listed in section III-A, including five non-parameterizable operators: (1) *Revert to previous statement*, (2) *Remove incorrectly added statement*, (3) *Swap changed statement with neighbouring statement*, (4) *Negate added condition* , and (5) *Convert statement to condition variable statement*. We also implement four parameterizable operators (i.e, operators that needs to be with program expression), including (1) *Add condition to changed expression* (this operator combines the operator *Use changed expression as input for other operator* and the operator *Add condition*), (2) *Add condition* , and (3) *Add statement*. The first four parameterizable operators aim to find the condition for hiding an **Unmask** regression error.

Before applying contextual operators, we collect contextual information (program location, changed expression and type of changes) required to support the defined operations.

C. Test case Prioritization and Reduced Test Suite

We evaluates each generated patch using a two-phase approach. We first execute the resulting patch against the reduced test suite. The reduced test suite (Figure 1) consists of tests with different execution results in both versions, namely the progression tests (*i.e.*, tests that fail in the previous version but pass in current version) and the failing regression tests (*i.e.*, tests that fail in the previous version but pass in current version). When a patch that passes both set of tests is found, we then check if the patch introduces any new regression by re-executes all tests in the test suite.

We prioritize test cases using the reduced test suite based on the assumption that test cases that evolve across the two versions are more likely to fail in future execution [50]. Our goal is to save the time spent in evaluating each patch against

[20] https://github.com/eschulte/clang-mutate
[21] http://clang.llvm.org/

TABLE II
SUBJECT PROGRAMS AND THEIR BASIC STATISTICS

Subject	Description	Size in kLOC	Bug Introducing Commit	Bug Report	PT Size/Test Suite Size
Vim	Text Editor	150	f80e67 [18]	[17]	1/74
			509890 [16]	[15]	2/73
			a3552c [20]	[19]	1/71
			220906 [14]	[13]	1/72
CPython	Programming language	407	b878df [4]	[3]	1/268
			5b0fda [2]	[1]	1/286
Perl	Programming language	271	dca606 [10]	[9]	1/159
			bb9ee97 [8]	[7]	1/159
Indent	Source code re-format utilities	15	2.2.10 [6]	[5]	1/159
Tar	Archives manipulation utilities	21	1.14 [12]	[11]	1/15
Findutils	Directory searching utilities	18	6 versions	10 bugs	[1,10]/1054
Make	Program executable generation utilities	35.3	12 versions	15 bugs	[1,2]/528

TABLE III
OPERATORS USED IN FIXES GENERATED BY *relifix* FOR THE SUBJECT PROGRAMS

Subject	Operators Used	Number of Operators	Change Hunks
Vim-f80e67 [18]	Swap	1	1
Vim-509890 [16]	Revert	1	1
Vim-a3552c [20]	AddIf	1	1
Vim-220906 [14]	-	-	-
Cpython-b878df [4]	Revert	1	1
Cpython-5b0fda [2]	AddIf	1	1
Perl-dca606 [10]	Revert	1	3
Perl-bb9ee97 [8]	-	-	-
Indent-2.2.10 [6]	AddIf & AddOld	2	1
Tar-1.14 [12]	Revert	1	2
Findutils	4 AddIf, 3 Revert, 1 Insert	8/8	10/8
Make	3 AddIf, 3 AddNegated, 1 Revert	7/7	7/7
Total/Mean	10 AddIf, 8 Revert, 3 AddNegated, 1 AddOld, 1 Swap, 1 Insert	24/23=1.04	28/23=1.22

the entire test suite, and to allow more candidate mutants to be generated within the time limit.

VI. EXPERIMENTAL EVALUATION

We perform an evaluation on real regressions by comparing the effectiveness of our approach with GenProg [37]. To evaluate the effectiveness of our approach, we aim to address the following research questions:

RQ1 How many regression errors can our approach repair compared to GenProg?

RQ2 Given only the test cases that evolves across the two versions, how likely is our approach to introduce new regressions, as compared to GenProg?

RQ3 Are our produced fixes suitable for patching latent regression errors or for patching errors due to code changes?

RQ4 Can we fix regression errors by making only small code changes without introducing new regressions?

The first question (RQ1) assesses the repairability of both approaches in the context of regression error, given the whole test suite. The second question (RQ2) evaluates the likelihood of both approaches in producing other test failures after repairing a regression error based on a reduced test suite (see Subsection V-C for definition of reduced test suite). The third question (RQ3) asks if our approach is more effective in fixing existing errors (i.e., latent errors) compared to new errors that are introduced due to the code modification. Lastly, the fourth question (RQ4) validates our hypothesis that mutant with small code changes (according to our below definition of small code changes) are less likely to introduce new regressions.

At the beginning of the paper, we presented Criteria 1 which guides our regression repair. We now present Criteria 2 which checks whether our approach produces repairs with small code changes, and further clarifies the first property mentioned in Criteria 1.

Criteria 2: Our repair should introduce small code changes, such that each repair should satisfy the following criteria:

Least number of change hunks Our repair should introduce the least number of change hunks. A change hunk is a single sequence of contiguous source codes which has been modified from one version to another [39], [43], [45].

Least number of applied operators Our repair should apply the least number of operators to the original program.

A. Experimental Setup

We evaluate *relifix* on 35 real regression errors collected from seven open-source C projects. Table II lists information about these projects. The last column in Table II shows the number of progression tests(PT) and the total number of tests in the whole test suite. For each regression error, we run both *relifix* and GenProg [36] to generate repair. GenProg provides several options that control the fault localization scheme used (e.g., path-based and line-based). We use the line-based fault localization scheme and provide the changed lines for the faulty locations to simulate a specialized version of GenProg for fixing regression errors (we call this *rGenProg*). We then

TABLE IV
OVERALL REPAIRABILITY (I.E., RP) AND REGRESSION RATE (I.E. RR) FOR *relifix* AND GENPROG ON THE NEW SUBJECT PROGRAMS

Subject	*relifix*		*relifix*	GenProg		GenProg	*rGenProg*		*rGenProg*
	Reduced test suite		Whole test suite	Reduced test suite		Whole test suite	Reduced test suite		Whole test suite
	RP	RR		RP	RR		RP	RR	
`Vim-f80e67 [18]`	1	0	1	0	0	0	0	0	0
`Vim-509890 [16]`	1	0	1	0	0	0	0	0	0
`Vim-a3552c [20]`	1	0	1	0	0	0	0	0	0
`Vim-220906 [14]`	0	0	0	0	0	0	0	0	0
`Cpython-b878df [4]`	1	0	1	0	0	0	0	0	0
`Cpython-5b0fda [2]`	1	0	1	0	0	0	0	0	0
`Perl-dca606 [10]`	1	0	1	0	0	0	0	0	0
`Perl-bb9ee97 [8]`	0	0	0	0	0	0	0	0	0
`Indent-2.2.10 [6]`	1	1	1	0	0	0	0	0	0
`Tar-1.14 [12]`	1	0	1	0	0	0	0	0	0
`Findutils`	8/10	0/8	8/10	2/10	2/2	5/10	0	0	5/10
`Make`	8/15	1/8	7/15	0/15	0/15	0/15	0/15	0/15	0/15
Total	24/35	2/24	23/35	2/35	2/2	5/35	0/35	0/35	5/35

compare the repairability (RQ1) of all the three approaches: *relifix*, GenProg and *rGenProg*. We also compares how likely each approach introduces new regressions.

All subject programs in Table II are utilities or libraries that are commonly used. As we perform our evaluation only on real regression errors, we select two subjects (i.e., `Findutils` and `Make`) from the CoREBench benchmarks [29], (2) two subjects (i.e., `Indent` and `Tar`) from [54] and one subject used in GenProg experiments. We also add two additional subjects (i.e., `Vim` and `Perl`). We choose these regression errors because (S1) they contain detailed bug report that specifies the bug introducing commit, and (S2) all the regression errors are reproducable with at least one test that passes in previous version and fails in the faulty version. We exclude 8 bugs (i.e, 5 bugs from `Findutils` and 3 bugs from `Make`) from the CoReBench benchmarks as they violate (S2).

For running GenProg, we reuse the same parameters stated in [36]. One significant difference is that we switch to the deterministic adaptive search algorithm (AE) [53] to control potential randomness. Each run of relifix, GenProg and *rGenProg* is terminated after one hour or when a repair is found.

All experiments were performed on a machine with a dual-core Intel i5-2520M 2.50GHz processor and 4GB of memory.

B. Repairability (RQ1)

Table IV presents the repairability and the regression rate for the 10 individual regression errors (subjects *outside* CoREbench). For all tables, we denote x bugs out of a total of y bugs with x/y. The last row of Table IV shows the aggregated repairability (i.e., repairability for all bugs) and the aggregated regression rate for *relifix* and GenProg on the two CoReBench subjects. Given the entire test suite, *relifix* successfully repair 15 out of 25 regression errors (as stated in the "Whole test suite" column in Table IV) for the two CoReBench subjects, while GenProg only fixes 5 bugs. For the 10 regression errors in subjects outside CoRebench in Table IV, *relifix* fixes 8 out of 10 regression errors but GenProg fails to generate any repair for all the 10 bugs. Although GenProg is able to fix approximately half of the evaluated programs in their recent study in [36], GenProg can only fix 14.3% (i.e., 5 out of 35 bugs) of all the evaluated subjects. In comparison, *relifix* repairs 65.7% (i.e., 23 out of 35 bugs). We attribute the low repairability of GenProg to (1) the high complexity of the real regression errors as some regression errors in CoReBench has fairly high error complexity [29], (2) the lack of availability of fixes within the same program (i.e., the fixes may only exist in the previous version of the same program as argued in the recent paper [28]).

Next, we compare the repairability of both *relifix* and GenProg for the reduced test suites (read Subsection V-C for definition of reduced test suite). Given the reduced test suite, GenProg generates only 2 out of 35 repairs, whereas *relifix* produces 24 out of 35 repairs. In comparison, *rGenProg* that fixes only the changed lines fails to produce any repair with both set of test suites. The repairability of GenProg decreases (i.e., from generating 5 repairs to generating only 2 repairs) when provided with the reduced test suite compared to the whole test suite because the search space for the faulty locations increases significantly due the reduced test suite. *relifix* does not suffer from the same problem as (1) it reduces the search space for the faulty location by ignoring program location that are not within the set of code modifications, and (2) it further refines the fix location by applying fixes to each faulty location iteratively for a limited period of time.

On average, GenProg requires 44 patch evaluations (i.e., patch trials in [47]) before generating a repair while *relifix* takes 25 mutant evaluations for producing the final repair.

> *relifix* repairs 65.7% of all investigated bugs, while GenProg only fixes 14.3% of all evaluated bugs.

C. Regression Rate (RQ2)

The "RR" columns in Table IV represent the regression rate of each approach using only the reduced test suite, while the "RP" columns denotes the measurement given the whole test suite. We define regression rate as the likelihood of introducing new regression errors after fixing all tests in the reduced test

suite. We calculate the regression using the formula below:

$$RR = \frac{Number\ of\ Repairs\ that\ introduce\ new\ regression}{Number\ of\ All\ Generated\ Repairs} \quad (1)$$

In total, *relifix* introduces new regressions in 2 out of 24 repairs with the reduced test suite (see Subsection V-C for explanation of reduced test suite). In comparison, GenProg introduces regression in all repairs (i.e., 2 out of 2) generated with the reduced test suite, while *rGenProg* does not generate any repair with the reduced test suite.

We next discuss the regressions introduced by both approaches. *relifix* causes a regression in a test that check if parallel execution of `Make` works correctly when fixing the regression error for `Make-bug-#39203`[22]. This regression cannot be fixed when executing *relifix* on the entire test suite. *relifix* also introduces new regression when fixing the `Indent` program [5]. As discussed in section IV, this regression can be repaired given the entire test suite. GenProg causes 45 test failures out of 80 tests in the entire test suite when fixing the regression error for the `Findutils-bug-#18222`[23], while it makes 1 out of 81 tests fails in the whole test suite when repairing the `Findutils-bug-#19605`[24]. We classify the fixes for `Findutils-bug-#18222` as a bad fix because it causes more failures compared to the original buggy versions that has only one test failure. We think that the high regression rate of GenProg may be due to (1) the imprecise fault localization used, and (2) the massive number of modifications in the patches. The speculation regarding the problem with fault localization is supported by the fact that *rGenProg* , which fixes only the changed lines, does not share similar regression rate as the original GenProg.

> Given the reduced test suite, *relifix* is less likely to introduce new regression errors compared to GenProg.

D. Repairability of latent error versus new errors (RQ3) and the simplicity of the generated repair (RQ4)

Table III lists the operators involved in the fixes generated by *relifix*. The table belows explains the abbreviation used to denote the name of the operator in Table III.

Revert	**Revert to previous statement**
Swap	**Swap changed statement with neighbouring statement**
AddIf	**Add condition**
AddOld	**Add condition to changed expression**
AddNegated	**Add negated condition**
Insert	**Add statement**

As the classification of a regression error as a latent error or a new error caused by code modifications requires deep understanding of the regression error, we cannot provide a precise answer for RQ3. However, since the *Revert to previous statement* operator are directly related to the **Local** regression error that are caused by a broken existing functionality, we can provide an rough estimate of latent errors repaired by *relifix* by calculating the number of fixes generated using the *Revert to previous statement* operators. Based on this estimation, 8 out of 24 generated fixes are latent errors. This suggest that *relifix* can fix both types of errors (i.e., latent errors and new errors due to code modification) equally well.

> *relifix* can fix both latent errors and new errors due to new code modifications.

We hypothesize that repair that makes only small code changes are less likely to introduce new regressions. Hence, we check if our repair satisfies all criteria in Criteria 2. The last two columns in Table III shows the number of operators used and the number of change hunks involved for each generated repair. As shown in the last row of the table, the mean value for the number of operator used in the final repair is 24/23 $\approx$ 1.04, while the mean value for the number of change hunks involved in the generated fixes is 28/23 $\approx$ 1.22. The two low mean values suggest that most fixes generated by *relifix* involve making only small code changes to the original program. While we do not have enough data to support the claim that the small code changes are less likely to introduce new regressions, we observe that all the new regressions produced by *relifix* and GenProg with the reduced test suite involve introducing more than one change hunks and applying more than one mutation operators in the generated fixes.

> *relifix* generates repairs with small code modifications to avoid introducing new regressions.

VII. Threats To Validity

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

Subjects While our evaluation uses subjects of various sizes and from various sources, we reuses two subjects, in which we obtained the set of contextual operators, for evaluation. This selection compensates for the lack of benchmarks with real regression errors but it may be biased towards *relifix* due the operators derived in our manual inspection. However, we note that the operators used in generating repair by *relifix* in those subjects are generally different from the original operators observed due to the gap between the error introducing commit and the bugfixing commit.

Contextual Operators We derived the set of contextual operators from a benchmark that contains only C programs. The same set of operators may not be generalized to other languages. As we investigated only open-source projects, the operator may not be generalized to closed-source projects.

Readability of Patches We claim that the code generated by *relifix* are understandable with some examples in Section IV. This claim relies on the intuition that source code annotated with comments are generally more readable than CIL (i.e., Common Intermediate Language) file produced by other tools (e.g., GenProg). We leave detailed evaluation of the readability of automatically generated patches as future work.

Time We restrict the time limit for evaluating *relifix* and GenProg to one hour due to limited resources. The repairability for both tools may increase given a longer timeout.

[22]http://savannah.gnu.org/bugs/?39203
[23]http://savannah.gnu.org/bugs/?18222
[24]http://savannah.gnu.org/bugs/?19605

VIII. Related Work

Fault Localization There are several fault localization techniques that utilizes multiple program versions [25], [27], [46], [48], [54], [55]. DARWIN uses the previous version to localize the regression bug using dynamic symbolic execution in both program versions [46]. Delta debugging isolates failure-inducing circumstances that are responsible to test failures using a divide-and-conqueer algorithm [55]. It can be used to repair regression errors by first isolating problematic changes and reverting these changes. This way of repairing regressions is part of our set of contextual operators.

Automatic Program Repair A few automated program repair techniques have been proposed to reduce the time and effort required to fix software bugs. Arcuri and Yao proposed adopting evolutionary algorithms for automatic program generation [26]. Weimer et al. proposed using genetic programming for automated program repair [37], [33]. GenProg generates fixes using statements that exist within the same program, while we utilize statements in the previous program version to repair regression errors. After generating a repair, GenProg requires a separate minization step to produce patches with simpler code changes. Our repair algorithm does not require this step as it generates repairs by applying only a small number of operators to some changed lines.

Kim et al. proposed a automated patch generation that utilizes common fix templates learned from manual inspection of human patches [35]. Their user study demonstrated that patches generated by PAR are more acceptable than patches produced by GenProg. While we also derive our contextual operators from human patches, our operators are more general as they are not designed to fix a particular defect class (i.e., null pointer exceptions and array out of bounds errors) [42].

Wei et al. leverages software contract to automatically repair faulty Eiffel classes [52]. Our approach does not require manually written program contract as it utilizes syntactical information from the previous program version that may serve as an implicit specification.

Nguyen et al. employs program synthesis for discovering the intent pieces of code required for fixing the buggy program [44]. While we employed random search for the condition to hide the **Unmask** regression errors due to scalability issues, we believe that program synthesis may be used to generate the required condition.

Repair that uses domain specific knowledge There are several program repair techniques that utilizes domain specific information. In particular, PACHIKA relies on differences between passing and failing runs to automatically infer object behavioral model from Java program and produce fixes by either inserting or deleting method calls [30]. BugFix is a tool that incorporates information gathered from several debugging sessions in order to increase precision for producing bug-fix suggestion [34]. R2Fix closes the loop between bug report submission and patch generation by automatically classify the type of bug discussed in bug report and extracting pattern parameters to generate fixes based on a several predefined fix patterns [38]. PHPRepair fixes malformed HTML generation errors by encoding the string output for each test case execution as a constraint over variables corresponding to constant prints in the program and uses a constraint solver to generate string modification [51]. Martinez and Monperrus mine semantic code modifications(which they referred to as repair actions) from human patches and attach a probability distribution to the mined repair actions [40]. None of these techniques focus on repairing regression errors.

Utilizing Previous Version as a fix Various studies speculate on the possibility of locating fixes from various program versions [24], [28], [41]. Martinez et. al. demonstrates that statements or expressions that are required for fixing exist in previous commits of the programs [41]. Barr et al. analyzes commits from several open-source Java projects and they found that commits can be reconstructed from codes from the preceeding versions [28]. Alkhalaf et al. uses semantic differences between a reference function and a target function to synthesize a validation, a length, and a sanitization patch for repairing web-application code [24]. We share similar observation that the previous program version may be used for automatic repair generation, specifically in fixing **Local** regression bugs. The key difference is that some of our contextual operators use program location information from the previous version, while other operators utilize program expressions from the current version. Furthermore, to the best of our knowledge, ours is the first work to *develop* a repair method and tool specifically for patching regression bugs.

IX. Conclusions

In this paper, we proposed *relifix*– an approach of automated repair of software regression. This was achieved by considering the regression repair problem as a problem of reconciling problematic changes. We justified our claim using a set of contextual operators derived from our manual inspection of 73 real software regressions. Our evaluation on 35 real regression bugs shows that *relifix* can repair 23 bugs, while GenProg only fixes five bugs. Our experimental results with the reduced test suite suggests that our approach is less likely to introduce new regression compared to GenProg.

In future, we plan to investigate the usage of semantic differences between two program versions for repairing regressions. While we focus on two program versions in this study, we believe that extending the work to multiple versions poses unique challenges. Hence, we plan to perform further studies on multiple program versions. We are also interested in the integration of this approach with test generation frameworks and building an IDE plugin for the integration.

Acknowledgment

We thank Sergey Mechtaev and Marcel Böhme for discussions about this work. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] Cpython 5b0fda regression bug report. http://bugs.python.org/issue21233.

[2] Cpython 5b0fda source code. http://hg.python.org/cpython/rev/5b0f-da8f5718.

[3] Cpython b878df regression bug report. http://bugs.python.org/issue16012.

[4] Cpython b878df source code. http://hg.python.org/cpython/rev/b878-df1d23b1/.

[5] Indent version 2.2.10 regression bug report. http://savannah.gnu.org/bugs/?27036.

[6] Indent version 2.2.10 source code. http://ftp.gnu.org/gnu/indent/indent-2.2.10.tar.gz.

[7] Perl bb9ee9 regression bug report. http://perl5.git.perl.org/perl.git/commit/bb9ee97444732c84b33c2f2432-aa28e52e4651dc.

[8] Perl bb9ee9 source code. http://perl5.git.perl.org/perl.git/commit/bb9-ee97444732c84b33c2f2432aa28e52e4651dc.

[9] Perl dca606 regression bug report. https://rt.perl.org/Public/Bug/Display.html?id=74290.

[10] Perl dca606 source code. http://perl5.git.perl.org/perl.git/commitdiff/dca6062a863d0.

[11] Tar version 1.14 regression bug report. http://lists.gnu.org/archive/html/bug-tar/2004-10/msg00034.html.

[12] Tar version 1.14 source code. http://ftp.gnu.org/gnu/tar/tar-1.14.tar.gz.

[13] Vim commit 220906 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_-dev/DbW2gnNqj04/6KaQn2jsDvAJ.

[14] Vim commit 220906 source code. http://code.google.com/p/vim/source/detail?r=2209060c340d.

[15] Vim version 7.2.50 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_-dev/9hG4HrhQuhk/ogbOOYwfPPkJ.

[16] Vim version 7.2.50 source code. http://code.google.com/p/vim/source/-detail?name=v7-3-202.

[17] Vim version 7.3.202 regression bug report. http://code.google.com/p/vim/issues/detail?id=9.

[18] Vim version 7.3.202 source code. http://code.google.com/p/vim/source/detail?name=v7-3-202.

[19] Vim version 7.3.251 regression bug report. https://groups.google.com/forum/#!searchin/vim_dev/regression/vim_-dev/TUbaixgUilQ/YV38sAkof10J.

[20] Vim version 7.3.251 source code. http://code.google.com/p/vim/source/detail?name=v7-3-251.

[21] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[22] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 39–46. IEEE, 2006.

[23] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[24] M. Alkhalaf, A. Aydin, and T. Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 225–236, New York, NY, USA, 2014. ACM.

[25] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 520–523, Nov 2011.

[26] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 162–168. IEEE, 2008.

[27] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *FSE' 2010*, pages 177–186, New York, NY, USA, 2010. ACM.

[28] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis.

[29] M. Böhme and A. Roychoudhury. Corebench: Studying complexity of regression errors. ISSTA, 2014.

[30] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 550–554, Washington, DC, USA, 2009. IEEE Computer Society.

[31] R. DeMillo, R. J. Lipton, and F. Sayward. Program mutation: A new approach to program testing. *Infotech State of the Art Report, Software Testing*, 2:107–126, 1979.

[32] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[33] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[34] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 70–79, May 2009.

[35] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE' 2013*, pages 802–811. IEEE Press, 2013.

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

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

[38] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[39] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 388–391. IEEE, 2013.

[40] M. Martinez and M. Monperrus. Mining repair actions for guiding automated program fixing. Technical report, INRIA, Tech. Rep, 2012.

[41] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. *arXiv preprint arXiv:1403.6322*, 2014.

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

[43] S. K. Nath, R. Merkel, M. F. Lau, and T. K. Paul. Towards a better understanding of testing if conditionals. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 772–777. IEEE, 2012.

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

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

[46] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21(3):19:1–19:29, July 2012.

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

[48] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 432–448, New York, NY, USA, 2004. ACM.

[49] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[50] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[51] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[52] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[53] W. Weimer, Z. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366, Nov 2013.

[54] K. Yu, M. Lin, J. Chen, and X. Zhang. Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers' perspectives. *J. Syst. Softw.*, 85(10):2305–2317, Oct. 2012.

[55] A. Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, Oct. 1999.