To Be Precise: Regression Aware Debugging

Rohan Bavishi Awanish Pandey Subhajit Roy

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India
{rbavishi, awpandey, subhajit}@iitk.ac.in

Abstract

Bounded model checking based debugging solutions search for mutations of program expressions that produce the expected output for a currently failing test. However, the current localization tools are not *regression aware*: they do not use information from the passing tests in their localization formula. On the other hand, the current repair tools attempt to guarantee *regression freedom*: when provided with a set of passing tests, they guarantee that none of these tests can break due to the suggested repair patch, thereby constructing a large repair formula.

In this paper, we propose *regression awareness* as a means to improve the quality of localization and to scale repair. To enable regression awareness, we summarize the proof of correctness of each passing test by computing Craig Interpolants over a symbolic encoding of the passing execution, and use these summaries as additional soft constraints while synthesizing altered executions corresponding to failing tests. Intuitively, these additional constraints act as roadblocks, thereby discouraging executions that may "damage" the proof of a passing test. We use a partial MAXSAT solver to relax the proofs in a systematic way, and use a ranking function that penalizes mutations that damage the existing proofs.

We have implemented our algorithms into a tool, TINTIN, that enables *regression aware* localization and repair. For localizations, our strategy is effective in extracting a superior ranking of suspicious locations: on a set of 52 different versions across 12 different programs spanning three benchmark suites, TINTIN achieves a saving of developer effort by almost 45% (in terms of the locations that must be examined by a developer to reach the ground-truth repair) in the worst case and 27% in the average case over existing techniques. For automated repairs, on our set of benchmarks, TINTIN achieves a $2.3\times$ speedup over existing techniques without sacrificing much on the ranking of the repair patches: the ground-truth repair appears as the *topmost* suggestion in more than 70% of our benchmarks.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Debugging, fault localization, automated repair, interpolation

1. Introduction

Automatic fault localization techniques have become imperative in this modern era of huge software code-bases. Given a large code-base, fault localization techniques help the programmer narrow-down to a fault by providing suggestions about buggy locations and possible repairs. The research community has provided multiple bug localization and repair techniques that use symbolic [10, 23, 36, 40], statistical [26, 27, 38], fault injection [49], and genetic algorithm based techniques [24, 41, 47, 48].

Symbolic techniques search for expressions whose evaluations can be replaced by some "angelic" values [10] to correct a given faulty execution; such expressions are flagged as potentially buggy expressions. This search for angelic values can be attempted either using a bounded model checker (BMC) [23, 36] or via a symbolic execution engine [10, 40]. As a symbolic execution engine models only a few paths in its symbolic execution tree (construction of the complete tree being prohibitively expensive), symbolic execution based tools are necessitated to focus their localization/repair suggestions to a set of pre-selected highly *suspicious* statements; such *suspicious* statements are usually identified by statistical bug isolation techniques. On the other hand, BMC-based tools use a compact *trace formula (TF)* to model the complete program, and hence, are capable of searching program-wide for a viable fix; however, most of the BMC-based tools also employ statistical localization techniques to restrict the search to smaller regions of the program.

Recently, there have been some interesting proposals [23, 36] that are based on the hypothesis that a buggy program can often be corrected with a "small" change. Such bug localization tools (like BugAssist[23]) rank their suggestions by the number of locations they would need to alter; automated repair tools (like DirectFix [36]) search for the "smallest" (simplest) possible alteration that would correct the buggy execution. Such tools often differ in their ability to handle regression failures—to the extremes; for example, BugAssist [23] completely disregards the possibility of regression failures; any location that on altering can correct the failing execution is provided as a valid suggestion. On the other hand, DirectFix[36] attempts to provide provably correct guarantees that the repairs cannot cause regression failures; they do so by fusing regression and repair, thereby also including constraints that model the entire execution space of passing tests.[1] This causes a substantial blow-up in its repair formula—almost to a point where the technique remains practical only for tiny regions of the program. Also, tools like DirectFix[36] have to be necessarily used in the repair mode, and do not allow for a "cheaper" localization mode. Hence, overall, balancing scalability while providing regression-awareness has been the bane of the otherwise encouraging direction of BMC-based debugging efforts.

In this paper, we propose a new algorithm that attempts to prevent both the extremes by constraining the system so that regression failure is *mostly*, though not always, prevented. Instead of embedding the complete model of the test-suite, we extract summaries of the *correctness proofs* from the passing tests, and use them as *soft* roadblocks while generating localization and repair suggestions to achieve *regression awareness* instead of *regression freedom*. Any suggestion that can correct a faulty execution while "breaking" only a small number of proofs is ranked higher. Note that breaking a proof constraint does not necessarily imply regression failure; it simply implies that the earlier proof does not stand—it is often possible (as is in many cases in our experiments) that a new proof can be constructed for the (possibly altered) execution of the passing test whose proof had to be relaxed to allow a certain mutation (we discuss it in detail in section 4.6).

We extract the summaries of the *correctness proofs* by computing the *Craig Interpolants* [14] over a symbolic encoding of the passing executions. As such proofs are small, the blowup in the size of the overall model is not large. On our set of benchmarks, regression awareness increases the size of the localization formula by only about 1.42× (on an average).

We demonstrate the effectiveness of our algorithm by building our ideas into our debugging tool, TINTIN, and compare it with our reimplementations of a couple of popular tools [23, 36].

To the best of our knowledge, TINTIN is the first attempt at incorporating *regression-awareness* to BMC-based bug localization without requiring to transform the localization problem to a repair problem. Localization solutions are often preferred over repair tools as they are more scalable and are applicable more widely as they are not restricted by a *repair grammar* (set of permissible mutations). We achieve superior rankings than ReBugAssist (our reimplementation of BugAssist [23]): in terms of developer effort, our algorithm requires the developer to examine 45% lesser locations than ReBugAssist in the worst case and by 27% on average, to reach the ground-truth repair on our set of benchmarks.

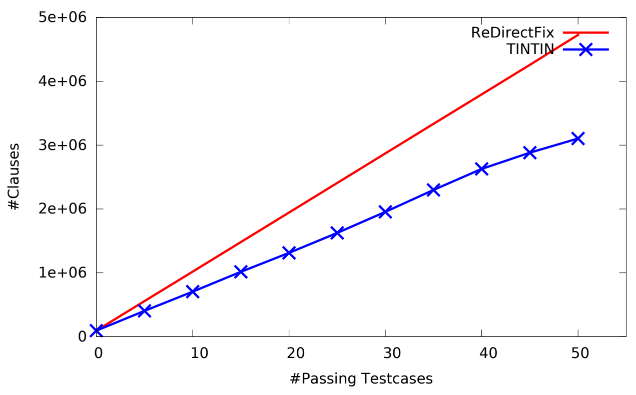

Figure 1: Size of the repair formula (in number of clauses) for TINTIN grows slower than ReDirectFix (our reimplementation of DirectFix [36]) with increasing size of the test-suite

Our algorithm also allows for a repair "mode" that uses our regression-aware ranking function to assign a low rank to suggestion that have a high plausibility of inducing regression failures. Our solution uses a smaller *repair formula* than existing techniques (like DirectFix[36]) without sacrificing much on the quality of the solution. Figure 1 shows how our algorithm scales with respect to our reimplementation of DirectFix (that we refer to as ReDirectFix) in terms of the size of the repair formula. In terms of time, TINTIN achieves a speedup of 2.3× over ReDirectFix (on an average) our benchmarks, while ranking the ground-truth repair patch as the topmost suggestion for more than 70% of our benchmarks.

Our contributions in this paper are as follows:

- We propose a new algorithm that embeds the *correctness proofs* from the passing tests to provide **regression awareness** for localization and repair.
- For localization, to the best of our knowledge, it is the first effort at embedding regression awareness for symbolic model checking-based fault localization. Our implementation substantially beats a regression-oblivious localization tool [23] on the quality of the rankings.
- For automated repair, we propose *regression awareness* in lieu of regression freedom. Our algorithm produces a much smaller repair formula than existing techniques while mostly retaining the ground-truth repair as the topmost suggestion.

[1] In this paper we constrain the scope of regression tests to only within a reduced representative test-suite (often selected using code-coverage metrics) that is employed for the localization/repair activity and not the complete "universal" test-suite.

2. Preliminaries

2.1 Craig Interpolation

Definition 1 Given formulas A and B such that $A \wedge B$ is *unsatisfiable*, a formula *I* is an *Interpolant* [14] for *A* and *B*, if the following conditions hold:

1. $A \Rightarrow I$ is *valid*;
2. $I \wedge B$ is *unsatisfiable*;
3. Free variables occurring in *I* occur free in both *A* and *B*.

Due these properties, interpolants often given concise explanations for the unsatisfiability of a formula.

2.2 Partial Maximum Satisfiability (pMax-SAT)

Given a boolean formula in *conjunctive normal form (CNF)*, the *Maximum satisfiability (Max-SAT)* problem deals with the question of the maximum number of clauses that can be satisfied by any assignment to the variables [16, 25, 30].

The *partial maximum satisfiability* problem additionally allows for marking the input clauses as either hard or soft; it returns the maximum number of soft clauses that can be satisfied by an assignment with the additional constraint that *all* hard clauses must be satisfied.

2.3 Regression Bugs

Given a program with a faulty execution, a developer strives to eliminate the error. However, in the process, new errors may get inadvertently introduced, thereby failing tests that were earlier successful; we refer to such bugs as *regression bugs*. An automated debugging tool is typically provided with a reduced "representative" test-suite (often selected using code-coverage metrics) rather than the complete "universal" test-suite. In this paper we constrain the scope of the term "regression" to only within this reduced test-suite and not the complete "universal" test-suite. As the universal test-suite is not exposed, a suggested repair from our tool (as for any other repair tool) may introduce regression bugs in the universal test-suite.

3. Overview

Consider the code excerpt from the TCAS program [1] in Figure 5. TCAS implements an air traffic collision avoidance system; the authors have created 41 different versions (referred to as v1 – v41) with seeded faults. Our example is inspired from v6; we have eliminated all functions that were not relevant to this bug to improve clarity.

The bug lies in line 32: the operator "<=" must be replaced by "<" to repair the fault. The `main()` function sets up the test-bench by reading in the inputs (via the `TakeInput()` method) and asserts the corresponding expected output (via the `AssertOutput()` method). The procedure `main()` calls the `Non_Crossing_Biased_Climb()` procedure whose result becomes the program output. Table 2(a) shows the inputs to the program, the actual and desired outputs for each test in our sample test-suite: T_1, T_2 and T_3 are the tests that produce the correct output (passing tests) while T_F produces an incorrect result (failing tests). In Table 2(b), a tick (cross) indicates that a program line is suggested (not suggested) by the respective schemes.

Primarily, we attempt to learn a *proof of correctness* for each passing test-case. This *proof* answers the following question: *why did the test produce the correct result?* Given a passing test T_i, we learn such a proof by computing the interpolants on an unsatisfiable formula constructed on the symbolic encoding of the execution of the passing test, along with the program input and the negation of the obtained output. Table 1 shows some of the sample interpolants (modified for readability) obtained at the program-points after lines 5, 32 and 45 for the tests T_1, T_3 and T_2 respectively. We now show how we apply these interpolants for localization and repair.

3.1 TINTIN in Bug Localization

When we run TINTIN for localization, we construct a symbolic formula for the whole program (trace formula) that summarizes all possible executions through the program. On adding assertions for the program input and expected output for the buggy execution, the formula clearly becomes unsatisfiable as no execution can be found with the given input that can lead to the expected output (the precise reason why the test is buggy!). Now, TINTIN attempts to *relax* program expressions (i.e. replace the expressions with fresh variables that are free to take any value) in an attempt to construct a "corrected execution" for the failing test. The expressions, that on relaxing, can yield a successful execution are potential buggy locations (or *suspicious* locations). This relaxation is done systematically via a MAXSAT solver so that a corrected execution can be synthesized with the minimal change to the program.

However, the above procedure is not *regression aware*: although a change to the expression at a suspicious location may correct the faulty execution, it may break one (or more) of the passing tests. To prevent the same, TINTIN uses the *proofs* from the passing test as additional (soft) constraints to act as "roadblocks": *any potential corrected execution that attempts to break a proof is discouraged!*

Please note that breaking a proof term does not imply that the respective test would necessarily fail: it only implies that now the passing test *may* fail (we discuss this in detail in section 4.6).

TINTIN computes a *suspiciousness score* for each suggestion depending on the number of entities (program expressions or proof terms) that it had to relax: all suggestions that have the same suspiciousness score lie in the same *suspicious class*. The order in which the solver enumerates the suggestions within a class is completely dependent on its search heuristics and random seed.

Figure 5 shows all the locations that would be ranked in the *highest* suspicious class in gray background if our tool

was not regression aware. Our regression aware TINTIN, however, ranks only the lines enclosed within ## in its highest suspicious class. Table 2(b) summarizes how our results improve by employing regression awareness: out of 11 suggestions provided, more than 50% of the "incorrect" candidates (i.e. 6 suggestions) are eliminated. Now, let us see how TINTIN is able to use regression awareness to eliminate some of the "incorrect" candidates from its highest suspicious class.
The failing test T_F follows the following path:

- **Line 54**: the function `alt_sep()` called;
- **Line 42**: the function `Non_Crossing_Biased_Climb()` is called (as `need_downward` needs to be computed).
- **Line 12** the function `Inhibit_Biased_Climb()` is called, which returns `545`; the value of `Down_Sep` is `350`. Hence `upward_preferred` evaluates to `true`.
- **Line 16** : the variable `result` is evaluated, yielding a value `1`; `Own_Below_Threat()` returns **true** and `ALIM()` returns `500` as `Alt_Layer_Value` (which is `1`) is greater than `Down_Sep`.
- **Line 42** : Since `Own_Below_Threat()` returns `1`, `need_downward` also evaluates to `1`.
- **Line 45** : Finally, `1` is returned as the output.

TINTIN reasons about the program in the following way:

- **Line 45**: The output of the program depends on the value returned by `alt_sep()`, that, in turn, depends on lines 45 and 47. Since the failing test has a desired output of `0`, changing line 45 is a trivial but valid fix. If TINTIN was not *regression aware*, it would have simply ranked this fix in its highest suspicious class. However, note that any mutation to this location would break the test T_2 as it requires a return value of 1 to succeed. This is where the proof of T_2 becomes useful: Interpolant 3 (see Table 1) extracted from the proof of T_2 precisely captures this, thus penalizing this candidate location by ranking it lower. Hence, the regression aware TINTIN does not rank this fix in its highest class.
- **Line 5**: This line relates to the branch condition at line 14: changing it can fix the faulty test by inverting the branch direction. However, this fix is not reported in the highest suspicious class as Interpolant 1 (in Table 1) extracted from the proof of T_3 discourages the change at line 5 as a change at this location can break the proof for test T_3.

All other lines that are eliminated from the highest suspicious class due to regression-awareness (lines 12, 14, 2 & 1) also relate to the branch condition at line 14, and are blocked by some interpolants in a similar fashion.

Table 1: Sample Interpolants obtained for Figure 5

ITP	T_i	Interpolant
1	**1**	(Climb_Inhibit $\implies$ **return** Up_Sep + 100;)
2	**3**	(Own_Tracked_Alt >Other_Tracked_Alt) $\implies$ **return** 0;
3	**2**	(need_downward) $\implies$ **return** 1;

Table 2: The test-bench and the localization summary for Figure 5. For Table (b), only the suggestions with the highest rank are reported. B+ is the regression aware version while B- is the regression oblivious version of TINTIN

(a) Test-Bench for Figure 5

Variable	T_1	T_2	T_3	T_F
Cur_Ver_Sep	867	907	765	634
Climb_Inhibit	1	0	1	1
Up_Sep	594	961	401	445
Down_Sep	693	399	500	350
Own_Tracked_Alt	1774	560	500	433
Other_Tracked_Alt	2204	601	424	433
Alt_Layer_Value	0	3	4	1
Desired Output	0	1	0	0
Actual Output	0	1	0	1

(b) Results Summary for Localization

Ln	B+	B-
1	✗	✓
2	✗	✓
5	✗	✓
12	✗	✓
14	✗	✓
45	✗	✓
16	✓	✓
27	✓	✓
32	✓	✓
42	✓	✓
44	✓	✓

Table 3: Repair Results, with (PT) & without (No PT) passing tests, for Figure 5. Only the suggestions with the highest rank are reported

Ln	Repair	PT	No PT
32	Own_Tracked_Alt <Other_Tracked_Alt	✓	✓
32	Own_Tracked_Alt != Other_Tracked_Alt	✗	✓
32	Own_Tracked_Alt >Other_Tracked_Alt	✗	✓
27	**return** result xor 1	✗	✓
16	Change || to &&	✗	✓
16	Change >= to <=	✗	✓

3.2 TINTIN in Automated Repair

When employed for automated repair, TINTIN again uses the *proofs of correctness* from the passing tests, such that, these existing proof terms are "preserved" by the altered statement for each passing test. As was in localization, the assertion on the proof terms is *soft*; however, any repair suggestion that breaks a proof term pays a penalty in terms of the ranking of the suggestion.

Table 3 shows the repair suggestions obtained in the highest suspiciousness class if TINTIN was not *regression aware* (Figure 5 marks all these locations with an "(R)" against them); TINTIN, however, is able to rank the correct repair (highlighted with a gray background in table 3) in its **highest suspicious class**.

Let us now look at TINTIN 's operation in more detail.

- **Line 32** : A total of three repairs are suggested at line 32 and all of them are valid for the given failing test case.

```
    int Pos_Alt_Thresh[4] = {400, 500, 640, 740};
    int ALIM() { return Pos_Alt_Thresh[Alt_Layer_Value]; }

    int Inhibit_Climb () {
        return (Climb_Inhibit ? Up_Sep + 100 : Up_Sep);
            ----------(Interpolant 1)---------
    }

    int Non_Crossing_Biased_Climb() {
        upward_preferred = Inhibit_Biased_Climb() > Down_Sep;

        if (upward_preferred != 0) {

(R)      ## result = !(Own_Below_Threat()) ||
                     ((Own_Below_Threat()) &&
                     !(Down_Sep >= ALIM()); ##

        }
        else {
            result = Own_Above_Threat() &&
                     (Cur_Ver_Sep >= 600) &&
                     (Up_Sep >= ALIM()));
        }
(R)    ## return result; ##
    }

    int Own_Below_Threat() {
        // Bug Fix : Own_Tracked_Alt < Other_Tracked_Alt
(R)    ## return (Own_Tracked_Alt <= Other_Tracked_Alt); ##
            ----------(Interpolant 2)---------
    }

    int Own_Above_Threat()
        return (Other_Tracked_Alt < Own_Tracked_Alt);

    int alt_sep() {
        ## int need_downward = Non_Crossing_Biased_Climb() &&
                                   Own_Below_Threat(); ##
        ## if (need_downward) ##
            return 1;
            ----------(Interpolant 3)---------
        else
            return 0;
    }

    int main() {
        TakeInput(Cur_Ver_Sep, Climb_Inhibit, Up_Sep,
                  Down_Sep, Own_Tracked_Alt,
                  Other_Tracked_Alt, Alt_Layer_Value);
        output = alt_sep();
        AssertOutput(output);
    }
```

Figure 2: Code excerpt from TCAS. The program points at which the interpolants 1, 2 and 3 are computed are marked and the respective interpolants are shown in Table 1

```
int Pos_Alt_Thresh[4] = {400, 500, 640, 740};
int ALIM() { return Pos_Alt_Thresh[Alt_Layer_Value]; }

int Inhibit_Climb () {
    return (Climb_Inhibit ? Up_Sep + 100 : Up_Sep);

}

int Non_Crossing_Biased_Climb() {
    upward_preferred = Inhibit_Biased_Climb() > Down_Sep;

    if (upward_preferred != 0) {

        result = !(Own_Below_Threat()) ||
                 ((Own_Below_Threat()) &&
                  !(Down_Sep >= ALIM()));

    }
    else {
        result = Own_Above_Threat() &&
                 (Cur_Ver_Sep >= 600) &&
                 (Up_Sep >= ALIM()));
    }
    return result;
}

int Own_Below_Threat() {
    // Bug Fix : Own_Tracked_Alt < Other_Tracked_Alt
    return (Own_Tracked_Alt <= Other_Tracked_Alt);
}

int Own_Above_Threat()
    return (Other_Tracked_Alt < Own_Tracked_Alt);

int alt_sep() {
    int need_downward = Non_Crossing_Biased_Climb() &&
                        Own_Below_Threat();
    if (need_downward)
        return 1;
    else
        return 0;
}

int main() {
    TakeInput(Cur_Ver_Sep, Climb_Inhibit, Up_Sep,
              Down_Sep, Own_Tracked_Alt,
              Other_Tracked_Alt, Alt_Layer_Value);
    output = alt_sep();
    AssertOutput(output);
}
```

Figure 3: Code excerpt from TCAS. The program points at which the interpolants 1, 2 and 3 are computed are marked and the respective interpolants are shown in Table 1

```
int Pos_Alt_Thresh[4] = {400, 500, 640, 740};
int ALIM() { return Pos_Alt_Thresh[Alt_Layer_Value]; }

int Inhibit_Climb () {
    return (Climb_Inhibit ? Up_Sep + 100 : Up_Sep);
}

int Non_Crossing_Biased_Climb() {
    upward_preferred = Inhibit_Biased_Climb() > Down_Sep;

    if (upward_preferred != 0) {

      result = !(Own_Below_Threat()) ||
                 ((Own_Below_Threat()) &&
                 !(Down_Sep >= ALIM());

    }
    else {
        result = Own_Above_Threat() &&
                 (Cur_Ver_Sep >= 600) &&
                 (Up_Sep >= ALIM()));
    }
    return result;
}

int Own_Below_Threat() {
    // Bug Fix : Own_Tracked_Alt < Other_Tracked_Alt
   return (Own_Tracked_Alt <= Other_Tracked_Alt);
}

int Own_Above_Threat()
    return (Other_Tracked_Alt < Own_Tracked_Alt);

int alt_sep() {
     int need_downward = Non_Crossing_Biased_Climb() &&
                           Own_Below_Threat();
     if (need_downward)
        return 1;
    else
        return 0;
}

int main() {
    TakeInput(Cur_Ver_Sep, Climb_Inhibit, Up_Sep,
              Down_Sep, Own_Tracked_Alt,
              Other_Tracked_Alt, Alt_Layer_Value);
    output = alt_sep();
    AssertOutput(output);
}
```

Figure 4: Code excerpt from TCAS. The program points at which the interpolants 1, 2 and 3 are computed are marked and the respective interpolants are shown in Table 1

```
int Pos_Alt_Thresh[4] = {400, 500, 640, 740};
int ALIM() { return Pos_Alt_Thresh[Alt_Layer_Value]; }

int Inhibit_Climb () {
    return (Climb_Inhibit ? Up_Sep + 100 : Up_Sep);

}

int Non_Crossing_Biased_Climb() {
    upward_preferred = Inhibit_Biased_Climb() > Down_Sep;

    if (upward_preferred != 0) {

     ## result = !(Own_Below_Threat()) ||
                 ((Own_Below_Threat()) &&
                 !(Down_Sep >= ALIM()); ##

    }
    else {
        result = Own_Above_Threat() &&
                 (Cur_Ver_Sep >= 600) &&
                 (Up_Sep >= ALIM()));
    }
   ## return result; ##
}

int Own_Below_Threat() {
    // Bug Fix : Own_Tracked_Alt < Other_Tracked_Alt
   ## return (Own_Tracked_Alt <= Other_Tracked_Alt); ##
        ----------(Interpolant)---------
}

int Own_Above_Threat()
    return (Other_Tracked_Alt < Own_Tracked_Alt);

int alt_sep() {
    ## int need_downward = Non_Crossing_Biased_Climb() &&
                           Own_Below_Threat(); ##
    ## if (need_downward) ##
        return 1;
    else
        return 0;
}

int main() {
    TakeInput(Cur_Ver_Sep, Climb_Inhibit, Up_Sep,
              Down_Sep, Own_Tracked_Alt,
              Other_Tracked_Alt, Alt_Layer_Value);
    output = alt_sep();
    AssertOutput(output);
}
```

Figure 5: Code excerpt from TCAS. The program points at which the interpolants 1, 2 and 3 are computed are marked and the respective interpolants are shown in Table 1

But two of them, namely the ones involving the operators != and > cause regression for the passing test T_3. This is precisely captured in Interpolant 2 (see Table 1): any repair must return 0 if *Own_Tracked_Alt* is greater than *Other_Tracked_Alt*. As TINTIN would need to break this additional constraint (the proof term) to allow this fix, it ranks this repair low.

- **Line 16** : One of the repairs changes the `||` operator to &&. This would always make the variable `result` evaluate to `0`. This clearly breaks the passing test T_2, where the required value is 1. Hence it causes regression errors and is correctly ranked lower due to a corresponding interpolant from the correctness proof of T_2 (not shown) being violated.

3.3 Discussion

In the above example, we have only shown the locations/repairs that appear in the highest suspicious class. In general, TINTIN produces all localizations and repairs along with its ranking that suggests the plausibility that a change/repair in that location would cause regression errors in the given set of passing tests.

As seen above, in localization and repair, regression awareness helps reduce the false positives by a substantial margin. In the above example, localization improves by more than 50% while the correct repair appears at the highest rank among a total of six possible suggestions.

For repairs, while there have been proposals at providing *regression freedom* by conjoining the model of all passing tests, thereby providing a provable guarantee that the tests cannot be violated, such schemes are non-scalable. We advocate *regression awareness* in lieu of *regression freedom*; Figure 1 compares the size of the repair formula for our regression-aware TINTIN with the reimplementation (indicated as ReDirectFix) of an existing tool DirectFix [36] that guarantees *regression freedom*.

As debugging is a hard activity, TINTIN encourages the use of the developer's experience in debugging rather than decoupling the tool from human expertise. In this direction, we propose a methodology (section 5.3) on how TINTIN can be used by a developer in her debugging effort.

4. Algorithm

4.1 Preliminaries

We describe a program as a transition system $(\mathcal{L}, \Gamma, l_0, \mathcal{V})$ over a set of program locations $\mathcal{L}$ with the computations described over a set of variables $\mathcal{V}$. The location $l_0 \in \mathcal{L}$ is the entry point. The computations in the program are described by a set of transitions Γ, where each transition $\gamma \in \Gamma$ is described by a tuple (l_i, ρ, l_j), where ρ is a *computation constraint* over variables $\mathcal{V} \cup \mathcal{V}'$ that models the computation between the program points l_i and l_j; $\mathcal{V}$ are the current state variables while $\mathcal{V}'$ are the next-state variables capturing the program state at the next program point, i.e. at location l_j.

```
int a = 0;              || (a#1 == 0)
int c = InputVar();     ||
if (c == 1)             || (guard#1 == (c#1 == 1))
    a = a + 1;          || (a#2 == a#1 + 1)
else                    ||
    a = a - 1;          || (a#3 == a#1 + 1)
                        || (a#4 == (guard#1 ? a#2 : a#3))
assert (a != 1)         || (a#4 != 1)
```

Figure 6: Example of Trace Formula Generation

We assume that all loops and recursive function calls are unrolled a bounded number of times and all function calls are inlined.

4.2 Trace Formula Generation

Given a program $\mathcal{P}$, we proceed by computing a *trace formula* for the program that captures all possible executions in the given program. Construction of the trace formula involves encoding each assignment as an equality constraint, with fresh symbols for each defined variable. The conditional expressions dictating the direction of branch statements are computed into special *guard* variables and used to select the respective definitions preserving the program semantics. Figure 6 shows how a C-program is translated into a trace formula (each statement is translated into the boolean expression displayed on the right); the final trace formula is the conjunction of these expressions. See Clarke et al. [12] for further details.

4.3 Proof Constraints from Passing Tests

For each passing test, we compute its *proof of correctness*; this proof essentially answers the question: *why was this execution successful?* This proof is efficiently summarized by a set of interpolants over the *symbolic path formula* (SPF) for each passing test. With abuse of terminology, we refer to the sequence of interpolants as the *proof* (instead of summaries of the proof) in the rest of the paper.

The SPF is a conjunction of all conditions along the passing test and captures the execution of the passing test. Given the passing execution as a sequence of transitions $[(l_0, \rho_0, l_1), (l_1, \rho_1, l_2), \ldots, (l_{n-1}, \rho_{n-1}, l_n)]$, we compute the *proof* of correctness of the test as an indexed set of interpolants $\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n$ that are computed as follows:

$$\mathcal{I}_i = \textsc{ComputeInterpolant}(\mathcal{A}_i, \mathcal{B}_i), where \tag{1a}$$

$$\mathcal{A}_i = \bigwedge_{k \in [0,\ldots,i]} \rho_k, \tag{1b}$$

$$\mathcal{B}_i = \bigwedge_{k \in [i,\ldots,n-1]} \rho_k \wedge \Omega_{in} \wedge ((\bigwedge_p g_p) \implies \neg\Omega_{out}) \tag{1c}$$

Here, Ω_{in} and Ω_{out} refer to the binding of the input and output variables to their respective values of input and expected output from the test-cases. The conjunct $\bigwedge \rho_k$ corresponds to the program statements and $\bigwedge_p g_p$ corresponds to the path taken by the test-case (in the form of true/false expressions corresponding to each branch instruction), where g_p is a guard variable. The interpolants summarize the proof that answers to why the given input satisfies the respective output or, in other words, *why the passing test indeed produces the valid result*?
We provide a deeper intuition about the above encoding in Section 4.6, but, for the moment, we leave the reader with the following observations (*for all passing tests*):

- The conjunct $\bigwedge_p g_p$ is true as it is the exact path taken by the passing test;
- The formula $\mathcal{A}_i \wedge \mathcal{B}_i$ is UNSAT as Ω_{in} does produce the expected output Ω_{out} (as it is a passing test).

Intuitively, the interpolant $\mathcal{I}_i$ essentially captures the *property* satisfied by the program location l_i that allows the passing test to produce the desired output. We define the conjunction of the interpolants as a *proof constraint* for the test *t* in the test-suite:

$$\Phi^t = \bigwedge_{i \in 0,\ldots,st(t)} \mathcal{I}_i^t \tag{2}$$

where $\mathcal{I}_i^t$ refers to the interpolant at the i^{th} location for the t^{th} passing test in the test-suite. The function *st(t)* provides the number of transition steps for the t^{th} passing test in the test-suite.

Any change to the program that does not satisfy a proof constraint of a passing test *t may* cause the test *t* to fail. Please note that this is not an if-and-only-if clause: if a proof constraint is broken, it does not imply that the test now definitely fails; it only implies that the test *can* fail. We discuss this in detail in section 4.6

4.4 Regression-Aware Localization

Given a failing test, we are interested in finding a repair location that, if altered, can fix the faulty test. At the same time, a suggested repair location is *prudent*, if altering the location would not cause any regressions on the rest of the test suite. In this section, we describe how we use our proof constraints to construct a *regression-aware* model to identify such *prudent* locations. At the same time, as our *proof constraints* are small, *regression awareness* causes only a small increase in the size of the encoding.

We proceed as follows: if a test *f* fails, the trace formula $\mathcal{TF} \wedge \Omega_{in}^f \wedge \Omega_{out}^f$ is unsatisfiable (where Ω_{in}^f and Ω_{out}^f are the inputs and expected outputs of the failing test). We use a partial MAXSAT solver to get a model (that is equivalent to constructing a corrected execution for the failing test) by allowing it to relax some of the computation constraints ρ_i. Note that relaxing a computation is equivalent to searching for an *angelic value* [10] for the computed expression. A statement ρ_i that, when relaxed, produces a valid execution, is marked as a *suspicious* statement.

The above formula is not regression-aware: it will simply relax any statement that allows the failing test to succeed. To enable regression awareness, we must rank a suspicious statement low if it is highly plausible that it induces regression.

We add regression awareness to the above localization formulation by ranking suspicious locations that *preserve the proofs* of the passing tests higher on our list. However, if no solution can be found, we subsequently allow the proofs to be broken by relaxing their respective clauses gradually, while ranking them lower. Overall, all suspect locations that cause the minimum change to the program (i.e. propose to alter the fewest program statements) and break the minimum terms from the proof constraints of the passing tests are ranked highest.

We now propose an extended encoding that engineers this idea into the formula. We introduce fresh variables λ_e for each expression $e \in \mathcal{E}$ (where $\mathcal{E}$ is the set of all program expressions) and fresh variables δ_i^t for each interpolant (proof term) $\mathcal{I}_i^t$ at the i^{th} location for the t^{th} passing test. The same expression would have multiple instances due to loop unrolling: understandably, all such instances share the same instance of λ_e. These variables act as "switches" to control the relaxation of the computation constraint for e.

The variables λ_e are introduced in the respective clauses that they are supposed to control; for example, if λ_e was supposed to control an expression *e*, the expression *e* would be replaced by $([|e|] \vee \neg\lambda_e)$ in the $\mathcal{TF}$, where $[|e|]$ represents the symbolic encoding of *e*. Therefore, the expression *e* is enforced only when the value of λ_e is *true*. Similarly, the variables δ_i^t controls the interpolant constraints (proof terms) in the same way.

Note the role of the λ_e and the δ variables: the δ variables attempt to prefer localizations that construct a corrected execution for the failing test that breaks a minimum number of proof terms from passing tests; λ_e attempts to construct a corrected execution that would cause the smallest change to the program (change the minimum number of expressions). Unlike prior tools [23, 36] that only attempt to minimize the change to the program (thereby optimize on only λ_e), our ranking function is two dimensional.

Also, controlling the usage of the δ variables allows us to have multiple configurations for our algorithm:

- Using a single instance of δ^t per passing test t allows us to prioritize localizations that break minimum proofs;
- Using a single instance of δ_i per location i allows us to prioritize localizations that cause minimum damage to the proof term at location i;

- Using different δ_i^t gives equal weight to both proofs and expressions.

The final regression-aware localization formula is:

$$\overbrace{(\Omega_{in}^f) \wedge \bigwedge_t \hat{\Phi}^t \wedge \hat{\mathcal{TF}} \wedge (\Omega_{out}^f))}^{\zeta_H} \wedge \tag{3a}$$

$$\underbrace{\bigwedge_{e\in\mathcal{E}} \lambda_e \wedge \bigwedge_{t\in TS} \bigwedge_{i\in steps(t)} \delta_i^t}_{\zeta_S} \tag{3b}$$

where

$$\hat{\Phi}^t = \bigwedge_{i\in[0,\ldots,st(t)]} (\mathcal{I}_i^t \vee \neg\delta_i^t) \tag{4a}$$

$$\hat{\mathcal{TF}} = \forall_{e\in\mathcal{TF}}\ TF[(e \vee \neg\lambda_e)/e] \tag{4b}$$

In the above formula, Ω_{in}^f and Ω_{out}^f refer to the mapping of the input and output variables to their (expected) values from the failing test. $\bigwedge_t \hat{\Phi}^t$ shows the proof constraints, extended with δ variables for each passing test in the test-suite while $\hat{\mathcal{TF}}$ denotes the trace formula for the program, extended with λ variables.

4.5 Detailed Algorithm

We are now in shape to discuss the complete algorithm: the main procedure kicks off by computing the proof constraints for the passing tests. For the program $\mathcal{P}$, for each passing test $t \in \mathcal{TS}_P$, we extract its execution as a sequence of transitions (line 3). We, then, pass this execution to the INTERPOLANTS() method to compute a list of interpolants using equation (1) (line 4). The (extended) proof constraint Φ^t is computed (line 5) as the conjunction of all the interpolants (using the 'switch' variable δ_i^t to enable/disable this term).

Next, we compute the extended trace formula by replacing each expression e with a *switchable* expression $e \vee \neg\lambda_e$ (lines 10-13).

The core localization procedure commences with constructing the hard (ζ_H) and soft (ζ_S) constraints as per equations (4) (lines 19-20). The partial MAXSAT query produces a set of *relaxed* clauses B (line 22). If any variable δ_i^t (the switch variable for test t) is contained in the set, it implies that the proof term i has been violated by the corrected execution of the failing test, implying that the passing test t may have broken. For all expressions e that were relaxed (indicated by the λ_e variables belonging to B), we report it as a suspicious statement with a rank defined by a scoring function that assigns a score depending on the number of λ_e and δ variables in the set B (lines 23-32).

We, finally, make the disjunction of the selector variables λ_e in B as hard constraints (lines 33-35) and repeat the process.

Algorithm 1 Localization Algorithm

```
 1: procedure PROOFCONSTRAINTS($\mathcal{P}, TS_P$)
 2:     for all $t \in TS_P$ do
 3:         $[(l_0, \rho_0, l_1), \ldots, (l_n, \rho_n, l_{n+1})] \leftarrow \mathcal{P}(t)$
 4:         $[\mathcal{I}_0^t, \ldots, \mathcal{I}_n^t] \leftarrow$ INTERPOLANTS($\mathcal{P}, t, \rho_0, \rho_1, \ldots, \rho_n$)
 5:         $\hat{\Phi}^t \leftarrow \bigwedge_{i\in[0,\ldots,n]} (\mathcal{I}_i^t \vee \neg\delta_i^t)$
 6:     end for
 7: end procedure
 8:
 9: procedure TRACEFORMULA($\mathcal{P}$)
10:     $\hat{\mathcal{TF}} \leftarrow$ CONSTRUCTTF($\mathcal{P}$)
11:     for all $e \in \hat{\mathcal{TF}}$ do
12:         $\hat{\mathcal{TF}} \leftarrow \hat{\mathcal{TF}}[(e \vee \neg\lambda_e)/e]$
13:     end for
14: end procedure
15:
16: procedure MAIN($\mathcal{P}, TS_P, F$)
17:     $[\hat{\Phi}^0, \hat{\Phi}^1, \ldots] \leftarrow$ PROOFCONSTRAINTS($\mathcal{P}, TS_P$)
18:     $\mathcal{TF} \leftarrow$ TRACEFORMULA($\mathcal{P}$)
19:     $\zeta_H \leftarrow Input[F] \wedge Output[F] \wedge \bigwedge_{t\in TS_P} \hat{\Phi}^t \wedge \mathcal{TF}$
20:     $\zeta_S \leftarrow \bigwedge_{e\in\mathcal{E}} \lambda_e \wedge \bigwedge_t \bigwedge_i \delta_i^t$
21:     while TRUE do
22:         $B$ = PMAXSAT($\zeta_H, \zeta_S$)
23:         if $B = \emptyset$ then
24:             exit
25:         else
26:             for all $\delta^t \in B$ do
27:                 print "Passing test t may have broken"
28:             end for
29:             for all $\lambda_e \in B$ do
30:                 $bugRank \leftarrow score(B)$
31:                 print "[bugRank] Expr e is suspicious"
32:             end for
33:             $b \leftarrow \bigvee\{s \mid s \in B \wedge$ s is a $\lambda_e$ selector variable$\}$
34:             $\zeta_H \leftarrow \zeta_H \cup b$
35:             $\zeta_S \leftarrow \zeta_S \setminus b$
36:         end if
37:     end while
38: end procedure
```

4.6 Discussion

Let us now dissect our localization formula to understand why it is constructed likewise. We derive an interpolant (to act as a proof term) at a program point l_i to provide an overapproximation of the transitions till l_i with respect to the specification of the program, i.e. $(\Omega_{in} \wedge (\bigwedge_p g_p \implies \neg\Omega_{out}))$. Here, $\bigwedge_p g_p$ corresponds to the path taken by the test-case in terms of branch conditions. The above term can be simplified as $((\Omega_{in} \wedge \neg \bigwedge g_p) \vee (\Omega_{in} \wedge \neg\Omega_{out}))$. With a disjunction in place, this requires two separate correctness proofs (thus forcing the interpolants to capture enough information to construct both the proofs):

1. **Control-Flow Proof**: Reasoning about why the current trace is inconsistent with $(\Omega_{in} \wedge \neg \bigwedge g_p)$ essentially captures the control-flow of the buggy execution: *why does*

the program, when executed with the given input, produce nothing but the correct execution path?

2. **Data-Flow Proof**: Reasoning about (input $\wedge$ $\neg$output) evokes a question about the flow of data: *why did the flow of the input values produce nothing but the desired output?*

As capturing the control-flow behavior and the data-flow behavior well summarizes the behavior of a (sequential) program, our proof, and hence, our interpolants capture enough information about *why the passing test actually passed!*

Finally, forcing the concrete test input/output in $\mathcal{B}$ in equation (1) allows our interpolants to be more general, rather than specializing on the given input/output pair.

Our localization formula essentially attempts to prioritize localizations such that the corrected execution of the buggy program would obey most of the terms from the proof constraints. The intuition behind the same is as follows: if the buggy execution does need to break some of the proof constraints, the respective tests may fail if the computation at the given location is altered.

Note that if a proof constraint is broken, it does not imply that the passing test would necessarily fail. It may happen that a proof constraint is broken but the proof is intact due to several reasons:

- The proof used was not the weakest proof: this is common as the interpolants produced by the solvers are not the weakest possible interpolants. Consider the program in Figure 7(a); the solver produces a stronger interpolant ($c = a + 100$) instead a possibly weaker ($c > a$). Due to that, the weaker fix will need to break the interpolant to allow the required mutation; note that if the solver had produced the weaker interpolant, the interpolant would have still been satisfied by the mutation.
- The program path for the passing test now changes, but it can successfully produce a proof along the new path. See the program in Figure 7(b): the branch condition needs to be changed to allow the passing test to take the *else* part. This interpolant will have to be violated to allow for the repair; however, the required mutation to correct the program does not cause the existing passing test to fail.

Also, if a single proof constraint is violated, it does not mean that the rest of the proof is rendered useless as the new proof corresponding to the altered execution may differ only slightly with the original proof. Moreover, attempting to contain the *damage* to a proof allows for a more fine-grained ranking of the bug locations and the locations causing minimal *damage* to proof is more likely to produce fewer regression errors.

5. Synthesizing Repairs

In this section, we discuss how TINTIN operates in the automated repair mode. Our automated repair algorithm ranks

```
// Pass : (20, 119)       #  // Pass : (2, 4)
// Fail : (0, 199)        #  // Fail : (2, 3)
1  Input(a, b);           #  1  Input(a, b);
                          #
// Should be a + 200      #  // Should be a < (b - 1)
2  c = a + 100;           #  2  if (a < b)
// Obtained ITP :         #  // Obtained ITP :
   c == a + 100           #     (a < b) => (branch = true)
// A Weaker ITP : c > a   #
3                         #  3      c = 1;
4                         #  4  else
5  out = (c > b);         #  5      c = 0;
6                         #  6
7  Assert(out == 1);      #  7  Assert(c == desiredOutput)

(a) Weakest interpolants  (b) Path-changing repair
    not computed
```

Figure 7: Reasons for proof breakage even in absence of regression

repairs that break a small number of proofs of the passing tests higher. We transform a buggy program to enable repairs by introducing "holes" in the program for each suspect expression; we also add the computation constraint obtained from the current expression (as *soft* constraints) to provide higher ranking to repairs that are syntactically close to the existing program. TINTIN targets a certain *class* of repairs like off-by-one errors, incorrect relational operator etc. We utilize a MAXSAT solver in an attempt to fill most of the holes with the current expressions from the buggy program while only breaking a small number of proofs for the passing tests. Though TINTIN can produce multi-line repairs of multiple buggy expressions spanning over many source lines, we begin our discussion to repairing a single expression.

5.1 Repair Grammar

TINTIN targets a set of well-directed repair actions. At present, we include three classes of repairs:

- **Off-by-One Repairs**: this repair strategy targets each constant c in the program in an attempt to synthesize a repair by replacing c by $c + i$, where $i \in \{-1, 0, 1\}$;
- **Incorrect Relational Operator**: this repair strategy attempts to replace relational operators to allow an alternate operation from $\{\leq, \geq, <, >, =, \neq\}$;
- **Incorrect Logical Combinator**: this repair strategy focuses on replacing "&&" operations by "||" and vice-versa.

Use of such well-defined *repair classes* has been advocated in [39]. Such strategies are useful as it is convenient for the developer to easily understand the repair patch and so, has been quite recent among popular repair tools (like [29]).

For each repair class, we introduce a *repair grammar* to enable a possible repair. Figure 8 shows our repair grammar for *incorrect relational operator*: the grammar is designed to pick a relational operator based on the value of *f*.

$$\begin{aligned}
&repair(f,h) = f \in \{f_{\leq}, f_{\geq}, f_{<}, f_{>}, f_{=}, f_{\neq}\} \wedge \\
&(\neg(f = f_{\leq}) \vee (h = a \leq b)) \ \wedge (\neg(f = f_{\geq}) \vee (h = a \geq b)) \wedge \\
&(\neg(f = f_{<}) \vee (h = a < b)) \ \wedge (\neg(f = f_{>}) \vee (h = a > b)) \wedge \\
&(\neg(f = f_{=}) \vee (h = (a == b))) \wedge \\
&(\neg(f = f_{\neq}) \vee (h = (a \neq b)))
\end{aligned}$$

Figure 8: Repair Grammar for relational operators. Inputs a and b are assumed to be available

5.2 Core Algorithm

To enable this repair, for every expression e that matches an expression grammar, we replace e with a fresh variable h_e in $\mathcal{TF}$, allowing the expression e to take any possible value. Hence, now this trace formula ($\mathcal{TF}_h$) has "holes" (Equation (5)). We bind the holes to the repair grammar, constraining the values that the holes can accept to the ones allowed by the *repair grammar* (Equation (6)). The variable λ_e acts like a "switch" that disables the relaxation by forcing h_e to necessarily be the original value of the expression in the current (buggy) program; the choice variable f_o corresponds to the current expression in the program.

$$\mathcal{TF}_h = \forall_e \ \mathcal{TF}[e \to h_e] \tag{5}$$

$$\begin{aligned}
\psi_f = \ & \mathcal{TF}_h \wedge \Omega_{in} \wedge \Omega_{out} \wedge \\
& \bigwedge_e (repair(f_e, h_e) \wedge ((f_e = f_o) \vee \neg\lambda_e))
\end{aligned} \tag{6}$$

For each passing test, instead of using the complete trace formula, we use an approximated constraint: we compute interpolants $\mathcal{I}_{pre}$ and $\mathcal{I}_{post}$ at the program points just before and after the program points enclosing each of the suspicious expressions. These interpolants effectively act as summaries of the pre-condition and post-condition of their correctness proofs: if these conditions hold for an altered expression, the correctness proof would also survive (Equation (7)).

$$\begin{aligned}
\psi_p^t = & ((\mathcal{I}_{pre} \wedge \mathcal{I}_{post}) \vee \neg\delta_t) \wedge \\
& \Omega_{in}^t \wedge \bigwedge_p g_p^t \wedge \bigwedge_e repair(f_e, h_e)
\end{aligned} \tag{7}$$

Here $\bigwedge g_p^t$ corresponds to the path taken by the passing test defined by the values of the guards at different branch locations.

The overall repair formula is constructed as a conjunction of all the failing and passing tests as hard constraints, with the "switch" variables as soft constraints.

$$\psi = \underbrace{\psi_f \wedge \bigwedge_t \psi_p^t}_{\zeta_H} \wedge \underbrace{\bigwedge_e \lambda_e \wedge \bigwedge_t \delta_t}_{\zeta_S} \tag{8}$$

Algorithm 2 shows our overall algorithm: we generate the interpolants (similar to section 4.5) in the procedure PROOFCONSTRAINTS() (lines 3-4). In contrast to section 4.5, given the suspicious transition ρ_s, we now only compute the interpolants before and after the suspicious transition; for simplicity, in this algorithm we assume that only a single suspicious transition is provided; in case there are multiple suspicious transitions, we compute an $(\mathcal{I}_{pre}, \mathcal{I}_{post})$ pair corresponding to each suspicious transition. Then, we create the extended constraint ψ_p^t for passing test t using the "switch" variable δ_t according to equation 7 (line 5); this term is used to compute the repair formula (line 6) with the hard and soft constraints as per Equation (8).

The main computation commences with performing the partial MAXSAT query (line 13) that attempts to disable the minimum number of the δ_t variables (for regression awareness) and λ_e variable (for prioritizing small repairs) so as to compute a ranking on repairs: small repair patches that break only a few proofs (of the passing tests) are ranked higher. The MAXSAT query returns the instances of δ_t and λ_e variables that were relaxed in B_δ and B_λ respectively. To enable the heuristic that the corrected program is syntactically "close" to the original (buggy) program, we invoke the partial MAXSAT query with the λ_e variables as part of the soft constraints (Equation (8)). This encourages the solver to provide a solution that has the minimal changes over the original program. We, then, disable this suggestion by making the disjunction over the relaxed λ_e variables hard and continue the process till no more repairs can be found.

Let us reiterate the role of the λ_e and δ_t variables: the δ_t variables attempt to attain a repair that breaks the minimum number of proofs from passing tests; the λ_e variables attempt to construct the simplest repairs (change the minimum number of expressions). Hence, our ranking function is two dimensional unlike other tools [23, 36] that optimize only on λ_e.

The inclusion of Ω_{in}^t in Equation 7 has an important consequence. If a suggested repair does not violate any interpolants ($\mathcal{I}_{pre} \wedge \mathcal{I}_{post}$), we are guaranteed regression freedom. This is because the inputs of the passing tests are used. In localization however, the interpolants are assigned values using the inputs of the failing test, and hence, the aforementioned guarantee is not provided.

5.3 Repair Validation

To ensure soundness, the suggested repair patches must be verified against potential regression errors. We propose the following software engineering methodology that a developer (say Alice) would use for verifying repairs:

1. TINTIN proposes a ranked list of suggested repairs, $\mathcal{R}_i$;
2. Alice removes the first suggestion $\mathcal{R}_i$ from the list; Alice may outright reject the patch $\mathcal{R}_i$ on a (somewhat casual) examination; in that case, she selects the next repair for examination till she gets a plausible repair $\mathcal{R}_k$;

Algorithm 2 Repair Algorithm

```
1:  procedure ProofConstraints($\mathcal{P}, TS_P$)
2:      forall $t \in TS_P$
3:      $[\overbrace{(l_0, \rho_0, l_1), .., (l_s, \rho_s, l_{s+1})}^{\rho_{pre}}, \overbrace{.., (l_n, \rho_n, l_{n+1})}^{\rho_{post}}] \leftarrow \mathcal{P}(t)$
4:      $[\mathcal{I}^t_{pre}, \mathcal{I}^t_{post}] \leftarrow$ Interpolants($\rho^t_{pre}$, $\rho^t_{post}$, $\mathcal{P}(t)$)
5:      $\psi^t_p \leftarrow$ ConstructFormulaPT($\delta_t$, $\mathcal{I}^t_{pre}$, $\mathcal{I}^t_{post}$, $\mathcal{P}(t)$)
6:      $(\zeta_H, \zeta_S) \leftarrow$ ConstructRepairModel($\psi^t_p$)
7:      endfor
8:  end procedure
9:
10: procedure Main($\mathcal{P}, TS_P, F$)
11:     $(\zeta_H, \zeta_S) \leftarrow$ ProofConstraints($\mathcal{P}, TS_p$)
12:     while TRUE do
13:         $(B_\lambda, B_\delta) \leftarrow$ pMaxSat($\zeta_H, \zeta_S$)
14:         if $B = \emptyset$ then
15:             exit
16:         else
17:             for all $\delta_t \in B_\delta$ do
18:                 print ”Passing test t may have broken”
19:             end for
20:             for all $\lambda_e \in B_\lambda$ do
21:                 $repairRank \leftarrow score(B_\lambda, B_\delta)$
22:                 print ”[repairRank] Repair[s] is fix for expr e”
23:             end for
24:             $b \leftarrow \bigvee\{s \mid s \in B_\lambda\}$
25:             $\zeta_H \leftarrow \zeta_H \cup b$
26:             $\zeta_S \leftarrow \zeta_S \setminus b$
27:         end if
28:     end while
29: end procedure
```

3. Alice runs all tests to check if $\mathcal{R}_k$ indeed causes regression;
    - If it causes regression, she rejects $\mathcal{R}_k$ and repeats the process with the next repair in the ranked list $\mathcal{R}_{k+1}$;
    - It it does not cause any regression, she either accepts the repair or unleashes a larger test-suite on the repaired program.
4. The process continues till either Alice is satisfied with a repair, or the repair list exhausts. If Alice is not satisfied with any of the suggested repairs, it implies that the repair is beyond the current repair grammar of TINTIN . Alice may consider re-executing TINTIN with a different repair grammar.

6. Experiments

We start this section by describing our reimplementations of two popular tools, BugAssist[23] for bug localization and DirectFix[36] for repairs; then, we compare TINTIN against these two tools.

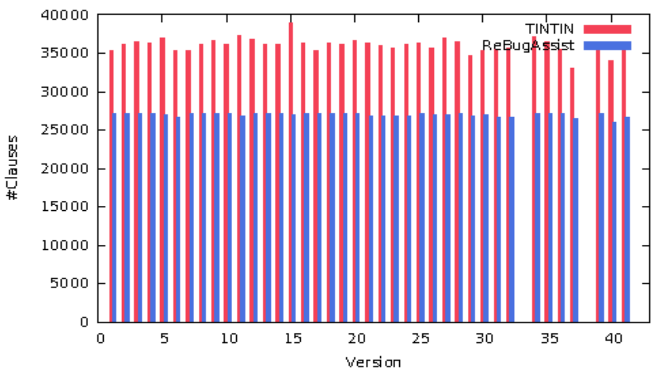

Figure 9: Number of Clauses in localization in TCAS

6.1 Reimplementation of Existing Tools

6.1.1 BugAssist

BugAssist [23] is a popular BMC-based tool that employs the hypothesis that the correct program usually differs from a buggy program at a few lines in the source program. Given a failing test, it gives a set of locations a high suspiciousness score if changing a few lines in the source code can pass the failing test. In its search for the smallest set of locations that can fix an error, it employs a MAXSAT solver to get the smallest set of locations whose constraints can be relaxed to achieve satisfiability of the trace formula. However, the localizations produced by BugAssist are not *regression aware* as it does not use information from any of the passing tests: making changes to the given locations can break the passing tests as their encoding of the localization formula does not capture the behavior of these tests.

For localization, our reimplementation of BugAssist (we refer to it as ReBugAssist) is almost identical to the original paper. However, it is an implementation of their core algorithm and lacks the side-tricks of using weighted MAXSAT queries to extract small counterexamples. This keeps our comparison with BugAssist fair lest we put BugAssist at a disadvantage on the running time as the weighted MAXSAT queries are more expensive. This does not otherwise affect the quality of their results on the dimensions that we evaluate. Our implementation looks similar to the original BugAssist as we were able to closely reproduce their results in terms of the number of candidate suggestions generated (see Table 1 in [23]).

6.1.2 DirectFix

DirectFix [36] is another BMC-based tool that addresses this problem by adding the trace formula corresponding to each passing test in the test-suite in its repair formula so that the suggested fixes cannot cause regression errors on the given test-suite. Motivated by the hypothesis that a buggy expression would need a small *syntactic* alteration for a corrected execution, DirectFix again employs a MAXSAT solver to search for the smallest change in all the potentially buggy ex-

Table 4: Time taken (in seconds) by ReBugAssist (ReBug) and TINTIN for localization in TCAS

V	ReBug	TINTIN	V	ReBug	TINTIN	V	ReBug	TINTIN
1	3.64	13.03	14	0.90	4.88	27	3.64	18.78
2	2.29	12.17	15	3.21	17.22	28	2.75	13.72
3	3.51	10.76	16	3.35	14.42	29	2.44	10.71
4	3.16	10.78	17	3.44	9.03	30	2.53	13.22
5	3.12	14.13	18	2.72	13.47	31	2.66	8.59
6	3.14	12.15	19	2.06	10.36	32	2.62	6.63
7	3.35	8.91	20	2.04	9.00	33	2.96	12.79
8	2.94	14.11	21	1.95	8.12	34	1.97	9.35
9	1.44	10.72	22	1.42	6.56	35	0.47	1.52
10	3.61	11.82	23	1.71	9.49	36	2.20	10.96
11	2.47	12.07	24	3.22	19.53	37	1.72	7.96
12	3.28	9.00	25	2.19	11.19	38	1.64	7.98
13	3.19	9.84	26	3.61	15.14	39	2.88	10.37

pressions (thereby combining localization and repair). However, the problem with DirectFix is its scalability: as it needs to include a copy of the trace formulas of all the passing tests, its repair formula is manyfold larger (proportional to the number of tests in the test-suite) than that of BugAssist. Moreover, as MAXSAT solving is expensive, the algorithm is not practical unless limited to small sections of a program. Further, DirectFix cannot be used in localization-mode: it must necessarily perform localization and repair together; hence, its debugging capabilities are restricted by its allowable set of mutations.

Our reimplementation of DirectFix (what we refer to as ReDirectFix) does not use component-based synthesis to synthesize the repair patch. Instead, we restrict our repair grammar to three well-understood class of repairs (off-by-one constants, incorrect relational operators and incorrect logical combinators).

As we intended to evaluate the core algorithms, we do not implement all the optimizations used in the tools (most of these are hinted at but not detailed well in the respective papers). It keeps our comparisons fair across the tools, and otherwise should not affect the *relative* performance of the tools.

6.2 Effectiveness of Our Algorithm

For showing the effectiveness of our algorithm, we run TINTIN on 52 buggy versions from 12 programs from the Siemens [15], SV-COMP [7] and Cascade [46] benchmark-suites. We tested all the benchmarks on a Intel Core i7-4770 3.40 GHz processor along with 15.4 GB RAM. For the SV-COMP benchmarks, we used an unroll factor of 6 for the loops. We built TINTIN on top of the CBMC model checker [12] to generate trace formulas from input programs. All complexities of real-world code (procedures, pointers, objects, virtual calls) is modelled by CBMC while our tool simply rides on the symbolic representations provided by CBMC. For the interpolation procedures, we used the C/C++ API of MathSAT5 [11]. We used an off-the-shelf Max-SAT solver MsUncore [31] for solving the pMax-SAT instances.

Our experiments attempt to answer the following research questions:

RQ1. Does regression awareness improve the results for bug localization?

RQ2. How much does regression awareness cost (in terms of number of clauses and time) for bug localization?

RQ3. For program repairs, does using only regression awareness instead of regression-freedom degrade the results?

RQ4. How much scalability can be achieved with regression awareness (instead of regression freedom) for program repairs?

6.3 Bug Localization

We performed detailed analysis of all the 39 versions for the TCAS program from the Siemens suite to understand the cost and improvement in the quality localization due to regression awareness. The versions 33 and 38 have faults (like incorrect array initializations) that we do not model (BugAssist [23] also did not model these faults). As all these versions could be repaired with change to a single location (for a single failing test), we constrain our search to changing only a single expression, both for ReBugAssist and for TINTIN.

6.3.1 Comparisons with respect to the Ground-Truth

For producing the worst-case ranking for TINTIN, we used the 1-3-3-4 scheme i.e. if there is a tie at the second position, we put both the candidates at rank 3; this essentially captures the number of locations that a developer would need to examine in the *worst* case to reach the ground-truth. For the average-case ranking for TINTIN, we used the 1-2.5-2.5-4 scheme i.e. if there is a tie at the second position, we put both the candidates at rank 2.5; this essentially captures the number of locations that a developer would need to examine on an average to reach the ground-truth.

Figure 10 shows the improvement in the worst-case rank of the ground truth of TINTIN with respect to ReBugAssist for the TCAS program; the figure shows the average over all failing tests. The height of the bars in the x-axis denotes the number of suggestions that a developer would need to examine before she discovers the ground truth repair for each tool in the worst-case. For example, for v1, out of a total of all possible suggestions of locations that can repair the program, ReBugAssist (red bar) provides the ground truth repair at rank 27 while TINTIN (green bar) ranks the ground truth repair at rank 10. The total number of suggestions is obtained by successive relaxations by the pMAXSAT solver. Figure 11 shows the comparison for the average-case ranks.

Overall, to reach the ground-truth repair, TINTIN reduces the developer effort — in terms of the number of locations that a developer needs to examine to zero on the ground-truth repair location — by an average (geometric mean) of 39% on

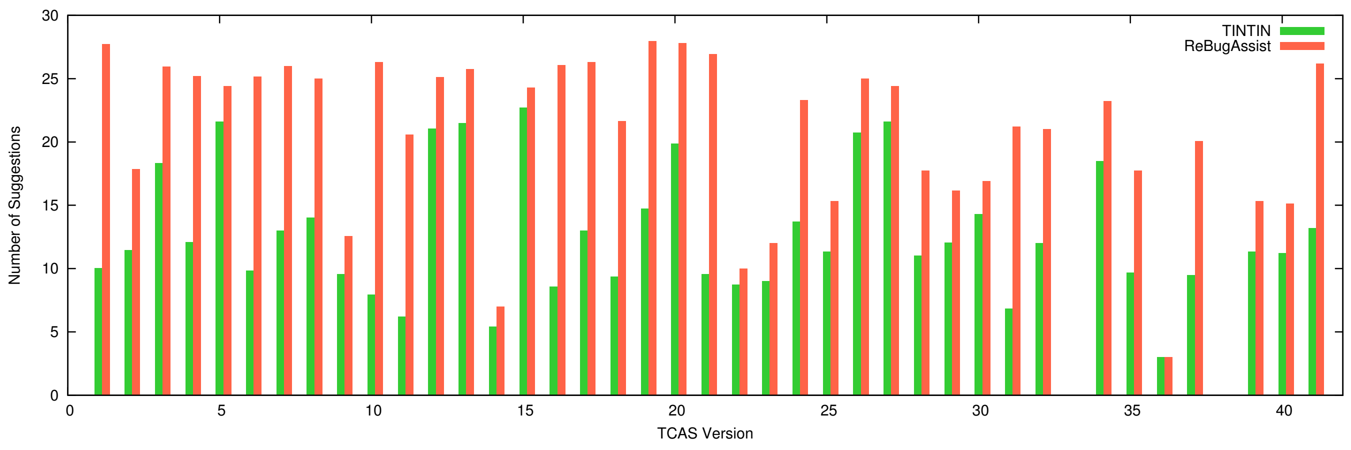

Figure 10: Number of Suggestions (Worst Case) in localization for TCAS from Siemens [15] benchmarks, requiring inspection to reach ground-truth repair

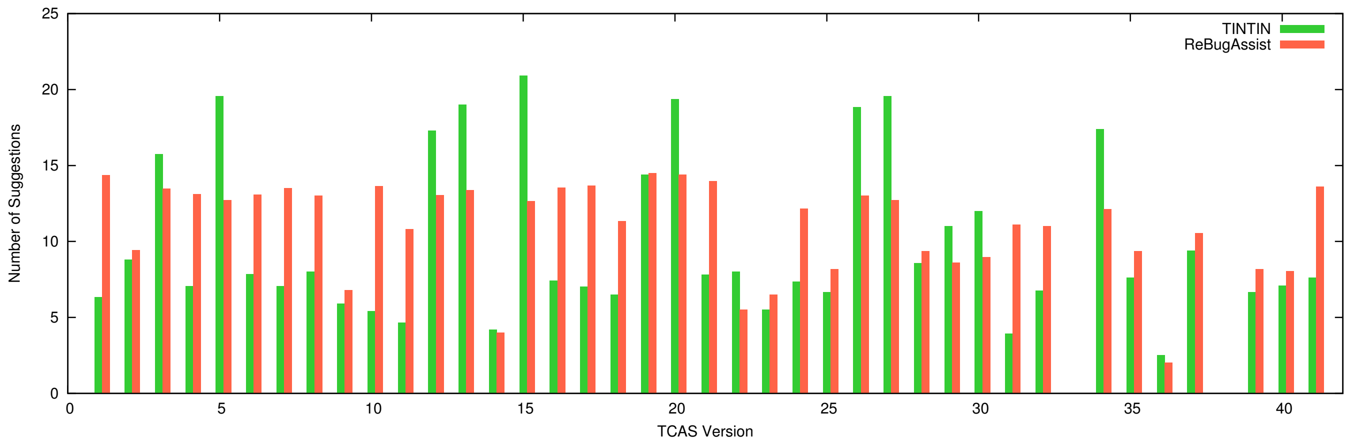

Figure 11: Number of Suggestions (Average Case) in localization for TCAS from Siemens [15] benchmarks, requiring inspection to reach ground-truth repair

these benchmark programs for the worst-case and by 15% in the average case. This proves that *regression awareness* is indeed a powerful asset for bug localization.

Figure 9 shows the cost of regression awareness in terms of the increase in the size of the localization formula (due to the interpolants): the number of clauses of our localization formula increases by about 1.33× on an average. In terms of time (Table 4), TINTIN is about 4.5× slower than ReBugAssist on an average.

All approaches based on bounded model checking have a limitation on their scalability. As suggested in other work [23, 36], TINTIN also needs model reduction techniques like concolic execution, abstract interpretation and delta debugging to handle larger programs. Also, using statistical bug localization approaches to identify buggy "regions" and restricting the trace formula to such regions has also been found effective. Table 5 shows the case when TINTIN is employed for larger programs with model reduction techniques: for these experiments, we restricted our model reduction techniques to employ concolic execution [17] and program slicing [45]. Even on such underapproximated models, TINTIN could consistently rank the ground truth location as the topmost suggestion in contrast to ReBugAssist that suggested two fixes of equal weights in each case.

We also studied the behaviour of TINTIN with automatically generated tests when test-suites were not available. Table 6 shows the performance of TINTIN on benchmarks from the Cascade and SV-Comp suites. These programs also form a different class of programs as these programs have well-defined specifications (provided by assertions) on shallower properties in contrast to the above programs which were debugged with the exact input-output specifications. We generate tests for these programs using the KLEE [9] symbolic execution engine. On an average, TINTIN reduces the size of the ground truth repair class by 65% as compared to ReBugAssist in the worst-case and by 61% in the average

Table 5: This table shows the number of lines in code (L), the time taken (in seconds) and the number of suggestions in average case (SA) and worst case (SW) to reach the ground truth location for each tool

Version	L	Redn	ReBugAssist			TINTIN		
			Time	SW	SA	Time	SW	SA
totinfo(v3)	565	CS	0.2	2	1.5	0.3	1	1
totinfo(v14)	565	CS	0.2	2	1.5	0.3	1	1
totinfo(v21)	565	CS	0.2	2	1.5	0.3	1	1
schedule(v9)	564	CS	0.2	2	1.5	0.3	1	1

case. TINTIN performs exceptionally well in eurekaunsafe and linearsearch where the developer effort is reduced by 86% and 80% respectively in the worst case, and by 77% and 80% in the average case.

6.3.2 Comparisons with respect to All Repairable Locations

The ground-truth repair is not the only possible way to fix a program; however, the set of all potential repairs is difficult to compute. Hence, we do a best-effort estimate by using the SemFix[40] synthesizer module of the Angelix[37] tool to approximate the set of all suggested repairs. We use Angelix as it has a large repair grammar facilitated by component-based synthesis. We feed each potential location suggested by TINTIN/ReBugAssist to Angelix to check if a repair is indeed possible at the given location. We synthesize the repairs by taking a union of the repairs suggested by Angelix in two following modes:

- Using the strongest possible mode (enable all repair classes) with a timeout of 20 minutes;
- Using the default mode (only a few common repair classes) with no timeout.

However, note that the set of repairs computed using the above methodology is still approximate as a potential repair may be missed because the repair is beyond the grammar of Angelix or Angelix times out before reaching the repair; at the same time, a reported repair may be faulty as it fails on some test that is absent in the universal test-suite.

We restrict this experiment to only one failing test for each version (we simply pick the first failing test); for the passing tests, we use all the passing tests from the universal test-suite for the respective version (about 1500 tests). We use the passing tests from the universal test-suite to demonstrate that the suggestions from TINTIN do generalize. Note that TINTIN uses interpolants from a smaller test-suite of representative tests and it is not exposed to the full universal test-suite.

Figure 12 shows a plot of our findings: the height of the green bar shows the number of suggestions from ReBugAssist in its topmost class; we mark the regression-free repairs by a × mark. We compute the average case via the 1-2.5-2.5-4 scheme, i.e. on a tie, we assign the same average rank to each of them assuming the set of all regression-free suggestions are distributed uniformly in the class. The blue dotted lines show the rank 60% mark for the number of suggestions for ReBugAssist. The experiment allows us to make the following observations:

- TINTIN ranks most of the regression-free suggestions at the higher ranks. For example, in version 1, eight of the nine regression-free suggestions appear at an average-case rank of 8.5 (the cluster of × show the number of regression-free suggestions at the specified average rank). Overall, one can see that most of the regression-free suggestions (about 75%) lie below the blue line.
- TINTIN ranks most of the regression causing suggestions low. This can be shown by showing the decreasing density of the regression-free suggestions as we move from the high to the lower ranks.
- TINTIN is able to generalize well from a small representative set of tests: though TINTIN is only exposed to a small subset of the universal set of tests, it generalizes well on the much bigger test-suite.

6.4 Repair Synthesis

For automated repairs, we compare our tool to ReDirectFix. We use about fifty passing tests in each case (similar to experiments described in [36]) that were selected to provide good code coverage, to serve as the representative tests. Our passing tests were sufficient to allow ReDirectFix to produce only one repair patch in each case. For producing the ranking for TINTIN, we used the 1-3-3-4 scheme i.e. if there is a tie at the second position, we put both the candidates at rank 3; this essentially captures the number of repair patches that a developer would need to examine in the *worst* case to reach the ground-truth (or semantically equivalent) repair. To enable a better interpretation of results in terms of the effect of *regression awareness*, we limit the number of statements that can be relaxed by the MAXSAT query to one. Hence, all the repairs suggested by ReDirectFix belong to the same suspicious class. However, the repairs suggested by TINTIN are ranked by the number of proof terms that break.

Table 7 shows the results of our experiments: we show only the versions of TCAS that could be repaired by one of our repair grammars consisting of off-by-one errors (OBO), incorrect relational operator (IRO) and incorrect logical combinator (IRC). The ranking of repairs by TINTIN is high: in 8 out of 11 cases, TINTIN ranks the ground truth repair as the *first* suggestion by the 1-3-3-4 ranking scheme. The version v16 produces two repairs at the topmost rank that includes the ground-truth repair.

In terms of the number of clauses, TINTIN constructs a repair formula which is about 43% smaller in size (on an average) of that of ReDirectFix. On time, TINTIN is 2.3× faster than ReDirectFix on an average. As a developer would examine each repair one-by-one till she gets a repair with

Table 6: This table shows the number of line in code (L), the number of clauses(#C), the time taken (in seconds) and the number of suggestions in average case (SA) and worst case (SW) to reach the ground truth location for each tool

Version	L	ReBugAssist				TINTIN			
		#C	Time	SW	SA	#C	Time	SW	SA
inf1	33	6208	0.18s	13	7	13144	2.9s	6	3.5
inf4	59	9293	0.2s	13	7	22694	3.5s	7	4
inf5	60	6584	0.28s	14	7.5	14044	3.7s	4	2.5
sum1unsafe	15	8414	0.16s	7	4	12597	0.3s	3	2
sum2unsafe	15	17209	0.5s	10	5.5	26143	2s	5	3
sum01unsafe	15	13961	0.27s	9	5	20951	1.3s	2	1.5
nec20unsafe	26	6959	0.14s	7	4	8418	0.23s	3	2
eurekaunsafe	57	82371	17s	29	15	182371	117s	4	3.5
linearsearch	25	4.3m	198s	15	8	4.3m	220s	3	2

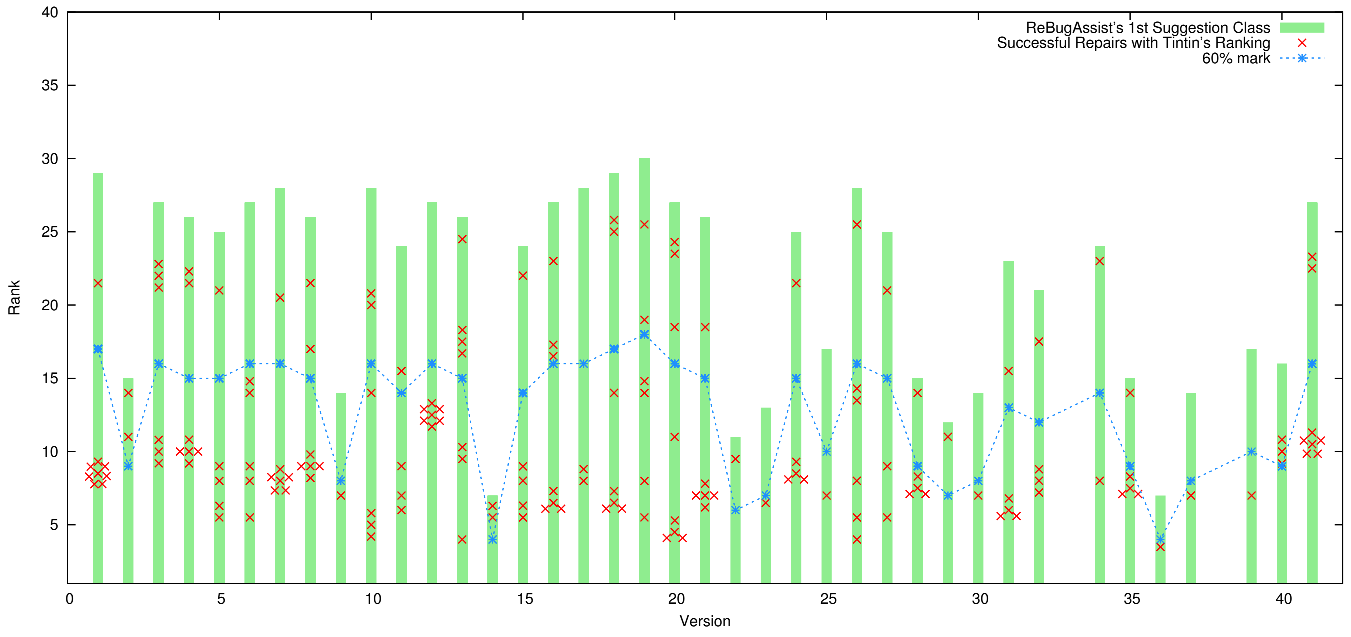

Figure 12: Performance of TINTIN on the set of all repairable locations (average case)

which she is satisfied (that we take as the ground truth repair), we show the time till we hit the ground truth repair (or a semantically equivalent patch). The time includes the time to enumerate all repairs of the same suspicious class that includes the ground-truth repair, including the time the solver declares that it is unable to generate any further such a suggestion.

To summarize, our experiments demonstrate that TINTIN can significantly scale bounded model checking based repair to larger programs and larger (representative) regression test-suites without much degradation of the ranking of the repair patches.

We close our discussion with a brief description of v3 and v12. These programs have a problem with incorrect logical combinator. The MaxSAT queries for TINTIN is much quicker, but it loses time in suggesting the more number of repairs at a higher rank.

7. Related Work

Symbolic methods [10, 23, 36, 40] of fault localization essentially search angelic values for expressions that can correct the faulty execution. Earlier tools [6, 18] attempted to flag an expression/statement buggy with respect to its “distance” from a correct execution. Ball et al. [6] explores the state-space of the program via a the SLAM model checker [5]. For a faulty execution, the cause of failure is identified as the transitions in the faulty trace that do not intersect with any transition on the correct traces. The approach had multiple shortcomings, as it needed the entire state-space of the program to be executed, and it necessarily needed to operate on an abstract model of the program; such overapproxi-

Table 7: In this table, #C is the number of clauses, T is the time (in seconds) required to reach the ground truth repair and #S is the number of repairs suggested by ReDirectFix and TINTIN till the ground truth repair is reached (in TCAS). RC is type of error in the respective version. In version v16, the ground-truth repair is tied at the top position with another repair; so its rank as per 1-3-3-4 scheme is 2, but as per 1-2-2-4 scheme is 1 (as shown in brackets).

Ver.	ReDirectFix			TINTIN			RC
	# C	T	# S	# C	T	# S	
v1	4.73m	50.78	1	3.08m	22.70	1	IRO
v3	2.18m	14.18	1	0.87m	19.15	3	ILC
v4	2.18m	16.13	1	0.87m	12.62	1	ILC
v6	4.75m	43.08	1	3.1m	31.30	1	IRO
v9	4.72m	159.24	1	3.11m	31.62	1	IRO
v10	4.75m	43.00	1	3.1m	30.91	1	IRO
v12	2.19m	15.48	1	0.88m	27.68	4	ILC
v16	2.85m	32.00	1	1.47m	23.40	2 (1)	OBO
v17	2.85m	28.31	1	1.48m	15.00	1	OBO
v20	4.72m	138.12	1	3.1m	52.57	1	IRO
v39	4.73m	129.46	1	3.1m	19.75	1	IRO

mations lead to incompleteness (i.e. an "angelic" transition exists in the concrete model but not in the abstract model). Secondly, this scheme fails to work if the program is provided an incomplete specification in terms of a test-suite. Groce et al. [18] used the CBMC model-checker to search for a successful execution that is *closest* to the given failing execution. The closeness metric is defined in terms of the values of the program-variables at each program point. The program variables whose values differ are marked as potentially faulty. Unlike the previous approach, this does not need the full-state space exploration of the program and thus, does not necessitate abstraction. Instead, CBMC constructs a compact trace formula for the program (without abstraction), and hence, assures completeness. Liu et al. [28] used CBMC to relax the set of branch predicates in the program in a search for a path that could pass a failing test. The path that deviates the faulty path the least for a corrected execution is more likely to be the culprit. BugAssist is a more recent approach to fault localization that again uses the CBMC model-checker to find the minimum number of program statements that would need to change to repair the faulty program. The BugAssist tool used a partial MAXSAT solver to search for such "suspicious" program statements. The authors demonstrate that their technique is effective.

However, the primary problem with all the above tools is that they are not "regression-aware": they do not attempt to use the potential "harm" that changing a given statement can have on the passing executions. In fact, none of the algorithms above take a set of passing tests for input. TINTIN embeds regression awareness for localization by prioritizing suspicious location on the amount of damage they can cause to a set of passing tests. On the other hand, DirectFix [36] solves this problem by fusing the localization and repair steps, thereby synthesizing a repair that can provably not cause regression errors on the test-suite. DirectFix does so by including the trace formula of all the passing tests such that any repair that fails on any of the tests renders the formula unsatisfiable. The problem with DirectFix is its scalability and inability to run in a localization-only mode. We provide a detailed analysis of TINTIN with DirectFix. Samimi et al. [43] propose an algorithm to repair a class of failures involving *constant prints* in PHP programs; they encode their problem as string constraints to synthesize repairs that make all tests pass.

The second category of symbolic techniques use a symbolic execution engine instead of a model-checker. Angelic Debugging [10] first uses statistical techniques to identify suspicious location and sets it output value to a fresh symbolic variable (thereby allowing it to latch on to a symbolic value). It, then, employs a symbolic execution engine to explore the symbolic execution tree of the given expression in search of a passing execution. SemFix [40] extends this idea by also synthesizing a repair automatically by using component-based synthesis [22] for the angelic values discovered. However, both the above tools could localize/repair only 1-fixable program, i.e. if the program can be repaired using only a single change. Moreover, a bigger problem is that symbolic execution techniques are highly dependent on the performance of statistical techniques to suggest good "suspicious" locations as each execution of the above tool targets a single location. Angelix [37] combines the ideas of SemFix and DirectFix by using DirectFix-style repairs on angelic values as was done for SemFix, thus allowing multi-line fixes.

Most of the statistical fault localization techniques [26, 27, 38] are based on indicators that provide a location that is "close" to the actual error site, often measured by the number of trips on the program dependence graph to reach the actual fault site [42]. Moreover, though the techniques are more scalable, the quality of repairs is much inferior to BMC based techniques.

There have been multiple other proposals for fault localization and repair: Delta Debugging [49] is a dynamic analysis technique that tracks the concrete program states in an attempt to simplify failing test cases to a minimal test case that still reproduces the problem. Genprog [24, 41, 47, 48] attempt to synthesize a repair by generating mutations of the program and use genetic programming to evolve the mutations that are likely bug patches. Though GenProg is highly scalable, the quality of the repair is low [36].

Interpolants have been used in many applications for model-checking. McMillan [32] proposed the idea of interpolation for bounded-model checking by using interpolants to overapproximate reachability in k-steps, and iteratively increasing k while searching for inductive invariants. The Impact algorithm [33] used interpolants to overapproximate the reachable states of program locations derived from infeasible paths that the model-checker encountered while traversing the program. These interpolants are iteratively weakened by successive traversals till a proof of correctness for the program is obtained. Interpolants have also found their use in summarizing functions [44] and Horn clause solvers [33] which have found applications in faster model checkers [8]. UFO [2, 3] uses a powerful combination of abstraction and interpolation for program verification: it starts off with with abstract interpretation [13] to prove the program, but uses interpolants to refine from counterexamples. Albarghouthi et al. [4] provide a compositional approach to interpolation that is capable of providing simpler interpolants than traditional SMT as a decision procedure. Both the above techniques can potentially improve our algorithm as they often produce simpler and weaker interpolants. Such weaker interpolants can help our cause by requiring TINTIN to break fewer proof terms for the correct repairs (see section 4.6).

Interpolation has also been used for symbolic execution for pruning paths [20, 21] by using interpolants to succinctly capture why the executions through a given program point cannot reach the error location. Jaffar et al. [19] extend the technique to applying interpolation for concolic execution that supports any given search strategy. So far, use of interpolants have primarily been restricted to using them to summarize the proof of unreachability to the error location, thereby gaining execution time by pruning the state-space exploration, both in model checking [34, 35] and in symbolic/concolic execution [19–21]. We believe that our use of interpolants in summarizing the proof of correctness of the passing tests to use them as *soft* "guards" for regression awareness is novel.

8. Discussion

We propose *regression awareness* as a solution to the pain points of two common debugging activities — bug localization and automated repair.

For bug localization, though tools like BugAssist [23] significantly reduce the number of program lines that need to be inspected (BugAssist reports about 8% reduction on the TCAS suite), which, however, in terms of the the total number of suspicious locations that a developer would have to examine, is still large. This calls for a good ranking scheme. We found that adding *regression awareness* by incorporating proofs (from the passing tests) in the localization formula) significantly improves the ranking for the ground truth repair; our implementation reduces developer effort by 45% in the worst-case and by 27% on average, i.e. the developer will need to examine fewer locations to reach the ground truth repair.

For automated repair, the primary pain point is in terms of its scalability: the symbolic encoding of the grammar of allowable mutations (repair grammar) significantly blows up the *repair formula*; hence, the size of the *repair formula* is significantly larger than the corresponding *localization formula*. We propose that if we can identify a small set of potential repairs quickly (using regression-awareness instead of regression-freedom), they can be validated easily by simply executing them on the test-suite without requiring any additional developer time over DirectFix. Also note that though DirectFix guarantees regression-freedom over the selected tests, it provides no guarantees over the universal test-suite, and hence, even the DirectFix repairs have to be validated on the universal test-suite.

For example for the version v20 of the TCAS program, the number of clauses for the repair formula for ReDirectFix is 4.72 million and it takes 138s for the repair activity; in contrast to that, ReBugAssist constructs a localization function with only about 0.027 million clauses for the complete program and reaches the ground truth repair in 2.27s. So, for the repair solution, we are able to achieve smaller repair formula and faster solving times with *regression awareness* (replacing regression freedom). We attain a speedup between 1.27× to 6.53× over most of our benchmark programs, with the ground-truth repair patch still appearing as the *first* suggestion in over 70% of our benchmarks.

References

[1] Software-artifact infrastructure repository (SIR). `http://sir.unl.edu/portal/index.php`.

[2] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with Interpolants and Abstract Interpretation. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 637–640, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: A Framework for Abstractionand Interpolation-based Software Verification. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 672–678, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Aws Albarghouthi and Kenneth L. McMillan. Beautiful Interpolants. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 313–329, Berlin, Heidelberg, 2013. Springer-Verlag.

[5] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[6] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 97–105, New York, NY, USA, 2003. ACM.

[7] Dirk Beyer. Software Verification and Verifiable Witnesses - (Report on SV-COMP 2016), 2016. (accessed in Jan 2016) `https://github.com/sosy-lab/sv-benchmarks`.

[8] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification.

In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[10] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.

[11] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 93–107, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[13] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[14] William Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(03):269–285, 1957.

[15] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[16] Zhaohui Fu and Sharad Malik. On Solving the Partial MAX-SAT Problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 252–265, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[18] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error Explanation with Distance Metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[19] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. Boosting Concolic Testing via Interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 48–58, New York, NY, USA, 2013. ACM.

[20] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 758–766, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Joxan Jaffar, Jorge A. Navas, and Andrew E. Santosa. Unbounded Symbolic Execution for Program Verification. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 396–411, Berlin, Heidelberg, 2012. Springer-Verlag.

[22] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.

[23] Manu Jose and Rupak Majumdar. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, January 2012.

[25] Chu Min Li and Felip Manya. MaxSAT, Hard and Soft Constraints. *Handbook of satisfiability*, 185:613–631, 2009.

[26] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[27] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 286–295, New York, NY, USA, 2005. ACM.

[28] Yongmei Liu and Bing Li. Automated Program Debugging via Multiple Predicate Switching. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, pages 327–332. AAAI Press, 2010.

[29] Fan Long and Martin Rinard. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[30] Joao Marques-Silva. Minimal Unsatisfiability: Models, Algorithms and Applications (Invited Paper). In *Proceedings of the 2010 40th IEEE International Symposium on Multiple-Valued Logic*, ISMVL '10, pages 9–14, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Joao Marques-Silva and Jordi Planes. Algorithms for Maximum Satisfiability Using Unsatisfiable Cores. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 408–413, New York, NY, USA, 2008. ACM.

[32] K. L. McMillan. Applications of Craig Interpolants in Model Checking. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Anal-*

ysis of Systems, TACAS'05, pages 1–12, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] Kenneth L. McMillan. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136, Berlin, Heidelberg, 2006. Springer-Verlag.

[34] Kenneth L. McMillan. Lazy Annotation for Program Testing and Verification. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 104–118, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Kenneth L. Mcmillan. Lazy Annotation Revisited. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 243–259, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[36] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015.

[37] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. ACM.

[38] Varun Modi, Subhajit Roy, and Sanjeev K. Aggarwal. Exploring Program Phases for Statistical Bug Localization. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 33–40, New York, NY, USA, 2013. ACM.

[39] Martin 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.

[40] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[41] ThanhVu Nguyen, Westley Weimer, Claire Le Goues, and Stephanie Forrest. Using execution paths to evolve software patches. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 152–153. IEEE, 2009.

[42] Manos Renieres and Steven P Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE, 2003.

[43] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[44] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. *FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization*, pages 203–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[45] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[46] Wei Wang and Clark Barrett. Cascade 2016 - (Competition Contribution), 2016. (accessed in Feb 2016) `http://cascade.cims.nyu.edu/vmcai.html`.

[47] Westley Weimer. Patches As Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 181–190, New York, NY, USA, 2006. ACM.

[48] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.