On the Interaction between Test-Suite Reduction and Regression-Test Selection Strategies

Sebastian Ruland and Malte Lochau

1Technical University Darmstadt, Darmstadt, Germany
2University of Siegen, Siegen, Germany

July 27, 2022

Abstract

Unit testing is one of the most established quality-assurance techniques for software development. One major advantage of unit testing is the adjustable trade-off between efficiency (i.e., testing effort) and effectiveness (i.e., fault-detection probability). To this end, various strategies have been proposed to exploit this trade-off. In particular, test-suite reduction (TSR) reduces the number of (presumably redundant) test cases while testing a single program version. Regression-test selection (RTS) selects test cases for testing consecutive program revisions. However, both TSR and RTS may influence—or even obstruct—each others’ performance when used in combination. For instance, test cases discarded during TSR for a particular program version may become relevant again for RTS. However, finding a combination of both strategies leading to a reasonable trade-off throughout the version history of a program is an open question. The goal of this paper is to gain a better understanding of the interactions between TSR and RTS with respect to efficiency and effectiveness. To this end, we present a configurable framework called REGRETS for automated unit-testing of C programs. The framework comprises different strategies for TSR and RTS and possible combinations thereof. We apply this framework to a collection of subject systems, delivering several crucial insights. First, TSR has almost always a negative impact on the effectiveness of RTS, yet a positive impact on efficiency. Second, test cases revealing to testers the effect of program modifications between consecutive program versions are far more effective than test cases simply covering modified code parts, yet causing much more testing effort.

1 Introduction

Background and Motivation. Software testing is concerned with revealing as many bugs as possible in a program within a—usually strictly limited—amount of time [1]. In particular, unit testing is one of the most important innovations in the recent past for pro-actively ensuring software quality in an effective, yet tractable and agile manner. Bugs revealed by unit tests include program crashes caused by programming errors as well as faulty input-output value pairs contradicting a given specification of the expected behavior (e.g., functionally incorrect program logics or ambiguous requirements). To this end, test cases are defined in terms of exemplary input values and expected output values for experimentally executing the program unit under test in a systematic manner.

One advantage of software testing as compared to other quality-assurance techniques is the—more or less—freely adjustable trade-off between two (generally conflicting) optimization goals, namely effectiveness (i.e., maximizing fault-detection probability) and efficiency (minimizing testing effort) [2][4]. In an idealistic setting, a perfect trade-off between both goals would
be a test suite that finds all bugs with minimum effort (e.g., requiring a minimum number of test cases). In reality, however, the number and exact location of bugs are, unfortunately, a-priori unknown: and even if all bugs would be known in advance (which would make testing obsolete), then finding a minimum set of test cases for revealing them is NP-hard [5]. Various heuristics have been proposed to control the selection of sufficiently effective, yet efficient sets of test cases for a program unit under test. Heuristics for measuring effectiveness often rely on structural code-coverage metrics (e.g., branch coverage) to be optimized by a selected test suite [6], whereas heuristics for efficiency apply test-suite reduction (TSR) techniques to decrease the number of redundant test cases, again, with respect to that given code-coverage criterion [2].

Moreover, modern software development is faced with the challenge of ever-shortened release cycles leading to increasing frequencies of consecutive program revisions. The goal of regression testing is to reveal emerging bugs introduced by erroneous program modifications between subsequent program versions [2]. RTS strategies define criteria for updating an existing test suite of a previous program version, by removing outdated test cases, by keeping still relevant test cases, and by adding further test cases for newly added functionality. Many RTS and regression-test generation strategies are further concerned with prioritizing test cases to apply the presumably more effective test cases first. The aforementioned trade-off between efficiency and effectiveness is therefore also the primary goal of regression testing, but now starting from an existing test suite inherited from previous program revisions.

**Problem Statement and Research Challenges.** Both TSR strategies for testing a single program version as well as regression-testing strategies for testing consecutive program revisions are concerned with the same problem:

*How to achieve a reasonable trade-off between efficiency and effectiveness?*

Nevertheless, both kinds of strategies may potentially influence—or even obstruct—each other in various ways. For instance, a test case being considered redundant for one program version (thus being removed during TSR) may become relevant, again, for later program revisions. As a result, expensive re-selections of eventually “lost” test cases may become necessary during regression testing. Hence, excessive TSR, although potentially improving testing efficiency for one program version, may, eventually, have a negative impact on regression testing. On the other hand, too reluctant TSR may lead to an ever-growing test suite thus also increasing test-selection effort during regression testing. Those subtle interactions between TSR and regression testing are neither obvious nor fully predictable in advance. Finding a feasible combination of both strategies yielding a suitable efficiency/effectiveness trade-off throughout the entire version history of a program is an open challenge involving further fine-grained details to be taken into account (e.g., code-coverage criteria, TSR heuristics, RTS criteria etc.).

**Contributions.** We present a conceptual framework for an in-depth investigation of interactions between strategies for TSR and RTS and the resulting impact on (mutually contradicting) efficiency and effectiveness measures. We focus on unit testing of C programs and use practically established control-flow coverage criteria as effectiveness measure for RTS and TSR [1]. As effectiveness heuristics for RTS, we consider modification-traversing as well as modification-revealing test cases [2].

In our experimental comparison of the different strategies, we additionally consider fault-detection rate as a-posteriori effectiveness measure. Concerning testing efficiency, we measure sizes of test suites (i.e., number of test cases) as well as the computational effort (i.e., CPU-time) for (regression) test-case generation and/or selection as well as TSR.

To summarize, we make the following contributions.
• **Configurable framework** for unit-test generation and selection, integrating recent TSR strategies, as well as existing and novel RTS strategies.

• **Tool support** for automatically applying different strategies to a version history of C program units.

• **Experimental evaluation results** gained from applying our tool to a collection of C program units with available version history. The evaluation results show that:
  
  1. TSR based on a coverage-based notion of test-case redundancy almost always decreases effectiveness of RTS, yet obviously having a positive impact on testing efficiency,
  2. modification-revealing test cases are far more effective than modification-traversing test cases for RTS, yet modification-revealing test-case generation requires much more computational effort and
  3. the number of test cases and the number of previous program versions considered for RTS only has a low impact on the effectiveness of regression testing.

**Outline.** The remainder of this paper is structured as follows.

**Section 2** contains an overview on the necessary background and terminology used in the remainder of the paper. We first introduce an illustrative example to introduce essential notions and concepts and also use this example to derive the motivation for the methodology proposed in the main part of the paper.

**Section 3** contains a conceptual description of the proposed evaluation framework for investigating the interaction between TSR and RTS strategies. After a general overview, the different possible strategies are then described in more detail using the illustrative example.

**Section 4** provides a detailed description of our tool support for the envisioned methodology together with experimental evaluation results gained from applying the tool to a collection of subject systems.

**Section 5** provides an overview about related work from the fields of regression testing strategies, automated test-case generation, as well as related approaches for regression verification.

**Section 6** concludes the paper and provides a brief outlook on possible future work.

**Verifiability.** To make our results reproducible, we provide the tool implementation and all experimental results as well as raw data on a supplementary web page.

## 2 Background and Motivation

We first describe the necessary background for the rest of this paper. We introduce an illustrative running example by means of a small, evolving C program unit with corresponding unit test cases. Based on this example, we describe basic notions of unit testing, test-suite reduction and regression testing. We conclude by summarizing the research challenges addressed in the remainder of this paper.

**Program Units.** Let us consider the sample C program unit in Fig. 1a. This source code constitutes the initial version $P_0$ of an evolving program developed and (re-)tested in an incremental manner. Function **find_last** receives as inputs an integer-array $x[]$ and an integer value $y$ and is supposed to return the index of the last occurrence of the value of $y$ in $x[]$. The first element of the array is supposed to carry as meta-information the length (i.e., the number of elements) of the array. For this reason, the first element should be ignored during the search.
for the value of \( y \). The search itself is supposed to be performed by iterating over the array and by keeping track of the latest occurrence of \( y \). Additionally, the function should return predefined error codes:

- If the size of the array (i.e., the value of the first element) is less or equal to zero, the returned error-code is \(-1\).
- If the value of \( y \) does not occur in the array, the returned error-code is \(-2\).

**Program Bugs.** The initial program version \( P_0 \) in Fig. 1a does, however, not satisfy the specified functionality as it contains the following three bugs.

1. **Bug 1:** The search index starts at 0, thus incorrectly including the value of the meta-information into the search (see line 5).

2. **Bug 2:** The search index stops at \( x[0] - 2 \) thus incorrectly excluding the last element from the search (see line 5).

3. **Bug 3:** The search incorrectly matches all values smaller than, or equal to, \( y \) instead of solely considering values equal to \( y \) (see line 6).

To fix those bugs, assume the developer to consecutively creates *program revisions* by correcting erroneous code parts.

**Program Revisions.** Let us assume that three consecutive revisions, \( P_1, P_2 \) and \( P_3 \), of the initial program version \( P_0 \) have been created, where \( P_3 \) finally has all three bugs fixed as described above. A *program revision* denotes a new program version as a result of modifications made to some parts of the previous program version. We represent program revisions using the established diff-syntax frequently used in patch files. Line numbers marked with post-fix `--` refer to the respective lines in the previous program version being removed in the new version, and those with post-fix `++` refer to lines in the new version being added to the previous version:

- **Patch 1** in Fig. 1b provides a fix for Bug 1.
- **Patch 2** in Fig. 1d provides a fix for Bug 2.
- **Patch 3** in Fig. 1f provides a fix for Bug 3.

After the first bug fix, we obtain the improved (yet still faulty) program version \( P_1 \) (cf. Fig. 1c). The second bug fix to \( P_1 \) (yielding program version \( P_2 \), cf. Fig. 1e) and the third bug fix to \( P_2 \) finally yield the bug-free version \( P_3 \) (cf. Fig. 1g). Such an incremental process is often interleaved by consecutive unit-testing steps to assure correctness of new versions and/or to reveal further bugs potentially emerging during revisions.

**Unit Testing.** Test cases for unit testing by means of program inputs are usually selected with respect to (structural) code coverage criteria. For instance, we require at least two different test cases for branch coverage of program version \( P_0 \). First, we require a test case for covering (reaching) the true-branch of the if-statement in line 2 (i.e., the input-array is empty). Additionally, we require at least one further test case for covering the remaining branches as follows.

---

Of course, a more reasonable implementation may perform a reversed iterative search and return the first occurrence. The purpose of the example is to demonstrate essential testing concepts in a condensed way thus requiring some simplifications which do, however, not threaten the validity of the overall approach.
int find_last (int x[], int y) {
    if (x[0] <= 0)
        return -1;
    int last = -2;
    for (int i=0; i <= x[0]-2; i++)
        if (x[i] <= y)
            last = i;
    return last;
}

(a) Initial Version $P_0$ of a Program Unit

5--: for (int i=0; i <= x[0]-2; i++)
5++: for (int i=1; i <= x[0]-2; i++)

(b) First Bug Fix

int find_last (int x[], int y) {
    if (x[0] <= 0)
        return -1;
    int last = -2;
    for (int i=1; i <= x[0]-2; i++)
        if (x[i] <= y)
            last = i;
    return last;
}

(c) New Program Version $P_1$ After Applying the First Bug Fix

5--: for (int i=1; i <= x[0]-2; i++)
5++: for (int i=1; i <= x[0]-1; i++)

(d) Second Bug Fix

int find_last (int x[], int y) {
    if (x[0] <= 0)
        return -1;
    int last = -2;
    for (int i=1; i <= x[0]-1; i++)
        if (x[i] <= y)
            last = i;
    return last;
}

(e) New Program Version $P_2$ After Applying the Second Bug Fix

6--: if (x[i] <= y)
6++: if (x[i] == y)

(f) Third Bug Fix

int find_last (int x[], int y) {
    if (x[0] <= 0)
        return -1;
    int last = -2;
    for (int i=1; i <= x[0]-1; i++)
        if (x[i] == y)
            last = i;
    return last;
}

(g) New Program Version $P_3$ After Applying the Third Bug Fix

Figure 1: Program Versions $P_0$, $P_1$, $P_2$ and $P_3$ with their corresponding Bug Fixes
• The **false**-branch of the if-statement in line 2 (i.e., requiring the input-array to have at least one element).

• The **true**-branch of the for-loop in line 5 (i.e., requiring the input-array to have at least two elements).

• The **false**-branch of the for-loop in line 5 (i.e., requiring the input-array to have at least one element).

• The **true**-branch of the if-statement in line 6 (i.e., requiring the input-array to have at least one element being less or equal to \( y \)).

• The **false**-branch of the if-statement in line 6 (i.e., requiring the input-array to have at least one element not being less or equal to \( y \)).

To satisfy branch coverage on \( P_0 \), a developer/tester may select a *test suite* consisting of the following two *test cases*:\footnote{More precisely, this test suite is only able to reach Line 8 due to Bug 3}

- \( t_1 = (x=[0], \ y=0) \),
- \( t_2 = (x=[3,5,5,3], \ y=4) \).

We denote test cases \( t \) as collections of input-value assignments (i.e., an array \( x \) and a value for \( y \)). Test-case specifications are often further equipped with the expected output values (i.e., last=-1 for \( t_1 \) and last=-2 for \( t_2 \)). If applied to \( P_0 \), \( t_1 \) would *pass*, whereas \( t_2 \) would indeed *fail* as it produces the erroneous return value 0 instead of the expected value -2. Hence, this test suite, although satisfying branch coverage, only reveals Bug 1 and Bug 3, but not Bug 2. In contrast, a test suite containing as test cases:

- \( t_1 = (x=[0], \ y=0) \),
- \( t_3 = (x=[1,1,1], \ y=2) \),
- \( t_4 = (x=[1,2,2], \ y=0) \).

also satisfies branch coverage and reveals Bug 1 and Bug 2, but not Bug 3. Hence, this test suite is similarly *effective* as the first (revealing two out of three bugs), yet being less *efficient* as it causes more testing effort due to the additional test-case execution. The test suite:

- \( t_1 = (x=[0], \ y=0) \),
- \( t_2 = (x=[3,5,5,3], \ y=4) \),
- \( t_3 = (x=[1,1,1], \ y=2) \).

satisfies branch coverage and reveals all three bugs, since \( t_2 \) detects Bug 1 and Bug 3, and \( t_3 \) detects Bug 1 and Bug 2. In contrast, the test suite

- \( t_1 = (x=[0], \ y=0) \),
- \( t_5 = (x=[3,1,1,2], \ y=1) \).

also satisfies branch coverage, but reveals none of the three bugs, since \( t_1 \) is unable to reach any bug and the execution of \( t_5 \) (accidentally) produces a correct output for all versions. These examples illustrate the well-known dilemma of unit-test selection that (1) adherence to (purely syntactic) code-coverage criteria does not guarantee effective test-suite selections\footnote{More precisely, this test suite is only able to reach Line 8 due to Bug 3}, and (2) using more test cases does not necessary increase effectiveness, yet obviously decreases testing efficiency. Concerning (2), *TSR* aims at removing redundant test cases from a test suite without decreasing code coverage\footnote{More precisely, this test suite is only able to reach Line 8 due to Bug 3}.
Test-Suite Reduction. Let us now assume that a developer/tester first selects the test cases $t_1$ and $t_2$ for testing program version $P_0$, where $t_2$ fails due to Bugs 1 and 3. After applying the first bug fix to remove Bug 1, the initial test suite consisting of $t_1$ and $t_2$ does no more satisfy branch coverage. This is due to the fact, that after the first bug fix is applied, the first element (i.e., the meta-information) of the input array is no more included into the search. As the last element is also not included (due to Bug 2), execution of $t_2$ will not enter the true-branch of the if-statement in line 6 and is, therefore, unable to reach 100% branch coverage. Hence, the developer/tester has to select a further test case, for instance:

- $t_6 = (x=[3,0,1,0], y=0)$

to cover the missing branch. Thus, the existing test case $t_2$ becomes redundant and might be removed from the test suite as $t_1$ and $t_6$ are sufficient for 100% branch coverage.

More generally, TSR is concerned with selecting from an existing set of test cases a sufficient, yet preferably small number of test cases for a program under test. Corresponding strategies for TSR have to address several challenges:

- Finding a minimal set of test cases from an existing test suite satisfying a given coverage criterion is NP-hard, being reducible to the minimum set-cover problem [5].
- As illustrated by the running example, defining redundancy of test cases only with respect to a code-coverage criterion might be misleading thus obstructing testing effectiveness [7, 8].

A further challenge arises in the context of evolving programs. Let us next assume that bug fix 2 is applied to remove Bug 2. After that, the new test case $t_6$ is no more able to reveal the remaining Bug 3, as the last element of the array is equal to $y$ and, therefore, the test output (accidentally) satisfies the specification. In contrast, the previously removed test case $t_2$ would have revealed Bug 3, since the last element is smaller than $y$, thus leading to the output of 3 instead of $-2$. This example shows that TSR solely based on structural coverage criteria defined on a current program version might be problematic in case of evolving programs. Test cases being redundant for a current program version might become relevant again after a program revision. In particular, regression testing is concerned with selecting a suitable set of test cases after program revisions [2, 9].

Regression Testing. As illustrated by the previous example, after revealing a bug due to a failed test-case execution, a developer/tester consecutively creates program revisions to fix the bugs [6]. After creating a program revision, the current test suite is re-executed to assure modifications made during the revision (1) successfully fix a discovered bug (i.e., test cases previously failing now pass) and (2) do not introduce new bugs (i.e., test cases previously passing still pass). Those test cases from the existing test suite should be re-executed investigating program parts affected by modifications made during the revision. New test cases might be required to investigate existing and/or newly introduced program parts not yet and/or no more covered by existing test cases. In both cases, one distinguishes between modification-traversing test cases and modification-revealing test cases [2]. Executions of modification-traversing test cases at least reach a program modification, but may, however, not reveal the modification to a tester (i.e., the program versions before and after the revision may produce the same output)

---

5 The term regression testing often summarizes a wide range of different disciplines of testing evolving programs not only including RTS, but also test-case prioritization, test-history analysis, test-artifact storage etc. We will focus on to the core problem of selecting test cases for regression testing.

6 For the sake of clarity, we assume in the following that the granularity of each individual modification made during a revision is limited to one particular line of code. However, this assumption does not possess any threat to the validity of our approach.
values for the same test inputs). In contrast, executions of modification-revealing test cases not only reach a program modification, but also yield different output values when applied before and after the revision.

For instance, recall the case in which the initial test suite consisting of \( t_1 \) and \( t_2 \) is applied to \( P_0 \). As \( t_2 \) fails on \( P_0 \), a developer might perform the first bug fix, leading to \( P_1 \). Next assume that \( t_1 \) and \( t_2 \) are simply reused for testing \( P_1 \). Both test cases would pass now although the second and third bug are still present in \( P_1 \). This is because \( t_1 \) is unable to reach the bug and \( t_2 \) is unable to detect the bug as the last element which would reveal Bug 3 is not included in the search due to Bug 2. The output yielded by \( t_2 \), therefore, also conforms to the specification. In contrast, selecting \( t_3 \) in addition to, or instead of, \( t_2 \) would also reveal the second bug on \( P_1 \), thus enabling the developer to perform a further bug fix leading to \( P_2 \). However, after this bug fix, \( t_3 \) becomes unable to detect the last bug as there are no values contained in \( x \) having a smaller value than \( y \). In contrast, \( t_2 \) is now able to detect the last bug.

The previous example shows that effectiveness of a test suite not only depends on the particular test cases, but also on the current program version under test. Hence, fault-detection probability of test cases may both decay as well as improve over time.

To generalize, RTS is concerned with selecting a sufficient, yet preferably small number of existing/new test cases to be (re-)executed on a revised program. Strategies for RTS have to address several challenges:

- Finding a \textit{minimal} modification-traversing set of test cases from an existing test suite is NP-hard and does, by definition, not guarantee effective assurance of program revisions.
- Finding \textit{any} modification-revealing test case for a program modification corresponds to the program-equivalence problem which is undecidable.
- In a practical setting, the aforementioned idealized assumptions on program revisions do usually not hold: units may contain several bugs which influence or even obfuscate each other and not all modifications applied during a program revision are actually bug fixes.

### Test-Suite Reduction vs. Regression-Test Selection

Both techniques aim at improving \textit{testing efficiency} by reducing the number of test cases to be executed without presumably harming effectiveness:

- TSR strategies aim at selecting from an existing test suite of one program version a presumably small subset being sufficient to satisfy the given code-coverage criterion.
- RTS strategies aim at selecting from an existing test suite of previous program versions and/or new test cases a presumably small subset being sufficient to traverse/reveal critical program modifications in the current program revision.

The previous examples illustrate the subtle interplay between both strategies affecting efficiency and effectiveness in non-obvious ways:

- Removing too few test cases during TSR might obstruct the gain in efficiency of subsequent RTS as well as TSR steps due to critically increasing overhead required for those strategies in case of ever-growing test suites.
- Removing too many test cases during TSR might obstruct effectiveness of subsequent RTS as well as TSR steps as currently redundant test cases might become relevant again.

We next present a novel evaluation framework to systematically investigate these interactions.
3 Evaluation Methodology

We present a configurable framework for systematically addressing the challenges described in the previous section. The methodology allows us to evaluate interactions between different strategies for TSR and RTS with respect to practical efficiency and effectiveness measures. We first present a conceptual overview and then describe the provided parameters for adjusting the strategies.

3.1 Overview

Figure 2 provides an overview of our methodology using the running example introduced in the previous section.

Starting from the initial program version $P_0$, the subsequent program versions, $P_1, P_2, P_3, \ldots$, result from applying consecutive revisions, given as patches Patch1, Patch2, Patch3. For each program version $P_i$, we consider a corresponding test suite $T_i$ containing the test cases selected for this particular program version. Based on the initial test suite $T_0$ created for $P_0$ (e.g., using either a coverage criterion like branch coverage, or randomly generated test cases, or other criteria), the regression test suites $T_i$ for testing subsequent program versions $P_i, i > 0$, result from applying a RTS strategy. As a complementing step, a TSR strategy may be applied to test suites $T_i$ of program versions $P_i$ to remove redundant test cases.

Our methodology comprises strategies for both TSR as well as RTS. The parameters for fine-tuning the strategies are denoted as circled numbers in Fig. 2. We briefly describe these parameters which will be explained in more detail below.

1 Reduction Strategy (RS): Technique used for TSR. Possible strategies: None, ILP,
FAST++, and DIFF.

2 **Regression Test-Case Selection Criterion (RTC):** New regression test cases for a program version \( P_i \) may be added to test suite \( T_i \) either by means of modification-traversing test cases (i.e., at least reaching the lines of code modified from \( P_{i-1} \) to \( P_i \)) or by modification-revealing test cases (i.e., yielding different outputs if applied to \( P_i \) and \( P_{i-1} \)).

3 **Number of Regression Test Cases (NRT):** The number of different regression test cases added into \( T_i \) satisfying RTC for each previous program version \( T_j \), \( 0 \leq j < i \).

4 **Number of Previous Program Revisions (NPR):** The (maximum) number of previous program versions \( P_j \) (\( i - \text{NPR} \leq j < i \)) for all of which NRT different test cases satisfying RTC are added to \( T_i \).

5 **Continuous Reduction (CR):** Controls whether the non-reduced test suite \( T_{i-1} \) (No-CR) or the reduced test suite \( T'_{i-1} \) (CR) of the previous program \( P_{i-1} \) version is (re-)used for the next program version \( P_i \) or if the previous test cases are ignored (None). This test suite is extended by new test cases according to the previously described parameters.

Note that parameter CR can be applied for all test suites of all versions. However, for clarity and space reasons, it is only depicted from program version \( P_2 \) on in Fig. 2.

### 3.2 Test-Suite Reduction Strategies

We now describe the different TSR strategies supported by our framework. We, again, use an illustrative example to explain the impact of the strategies on the trade-off between precision and computational effort. We first give a general characterization of the test-suite minimization problem for one single program \( P \), a given test suite \( T \) and a code-coverage criterion on \( P \) [4].

**Input:** Program \( P \), Test Suite \( T \), where

- \( P \) contains a set of test goals \( G = \{ g_1, g_2, \ldots, g_n \} \) according to the given code-coverage criterion and

- test suite \( T = t_1, t_2, \ldots, t_n \) consists of a set of test cases, where each \( t_j \in T \) covers a subset \( G_j \subseteq G \) of test goals on \( P \) such that for each \( g_i \in G \) there exists at least one test case \( t_j \in T \) with \( g_i \in G_j \).

**Output:** Minimal Test Suite \( T' \subseteq T \), where

- for each \( g_i \in G \) there exists at least one test case \( t_j \in T' \) with \( g_i \in G_j \) and

- for each \( T'' \subseteq T \) also satisfying the first property, it holds that \( |T'| \leq |T''| \).

The test-suite minimization problem is NP-hard being reducible to the minimum set cover problem [4] such that finding exact minimal solutions is computational infeasible for realistic programs. Various TSR heuristics have been proposed for approximating minimal test suites constituting different trade-offs between precision (deviation from exact solutions) and computational effort for performing the reduction.

To illustrate the different approaches in our framework, \( P_1 \) from our running example together with \( T_3 \) containing four test cases selected for branch coverage:

- \( t_1 = (x=0, y=0) \),

Note that parameter CR can be applied for all test suites of all versions. However, for clarity and space reasons, it is only depicted from program version \( P_2 \) on in Fig. 2.
Figure 3: Comparision of Test-Suite Reduction Strategies

- $t_2 = ([x=3,5,5,3], y=4),$
- $t_3 = ([x=1,1,1], y=2),$ and
- $t_4 = ([x=1,2,2], y=0).$

Figure 3 (on the left) illustrates the test-suite minimization problem. Program version $P_1$ contains three conditional branches (i.e., if-statement in line 2, loop-head in line 6 and if-statement in line 7), leading to six goals $G = \{g_1, g_2, \ldots, g_6\}$. A line between test case $t_i$ and test goal $g_j$ indicates that $g_j \in G_i$ (i.e., $t_i$ satisfies $g_j$). Test suite $T_3$ indeed satisfies branch coverage on $P_1$ as for each $g \in G$, there is at least one test case in $T_3$. However, $T_3$ is not minimal as, for instance, $t_2$ and $t_4$ cover exactly the same test goals, $g_2, g_3, g_4$ and $g_6$, and this set is further subsumed by the test goals satisfied by $t_3$. In fact, $t_3$ is the only test case covering $g_5$ and $t_1$ is also indispensable as it is the only test case satisfying $g_1$. Hence, $T_3 = \{t_1, t_3\}$ is the (unique) minimal test suite as shown on the left.

Computing the minimal test suite requires, in the worst case, to enumerate all $2^{|T|}$ subsets of $T$ which is infeasible in case of realistic programs. Practical approaches usually compute reduced test suites approximating the minimal solution. We now describe the strategies considered in our framework.

**ILP Strategy.** The test-suite minimization problem can be encoded as Integer linear optimization problem which can be precisely solved using Integer Linear Programming (ILP) solvers [5]. Our ILP encoding uses for each test case $t_i \in T$ a decision variable $x_i$ either having value 1 if $t_i$ is selected, or value 0 if $t_i$ is not selected. The ILP formula contains for each test goal $g_j \in G$ a clause building the sum of the decision variables $x_i$ of all test cases $t_i \in T$ for which $g_j \in G_i$ holds. By requiring the value of each such sum to be greater than 0, we ensure each test goal to be covered by the minimal test suite. To enforce minimality, the overall optimization objective is to minimize the sum over all values of decision variables $x_i$.

Applied to our example, this encoding introduces the variables $x_1, x_2, x_3, x_3$ for the test cases in $T$ and adds the following clauses for the test goals:

\[
\begin{align*}
x_1 &\geq 1 \\
x_2 + x_3 + x_4 &\geq 1 \\
x_2 + x_3 + x_4 &\geq 1 \\
x_2 + x_3 + x_4 &\geq 1 \\
x_3 &\geq 1 \\
x_2 + x_3 + x_4 &\geq 1
\end{align*}
\]
| Test Case \ Value | 0 | 1 | 2 | 3 | 5 | 6 |
|------------------|---|---|---|---|---|---|
| (t₁)             | 2 | 0 | 0 | 0 | 0 | 0 |
| (t₂)             | 0 | 0 | 0 | 2 | 1 | 2 |
| (t₃)             | 0 | 3 | 1 | 0 | 0 | 0 |
| (t₄)             | 1 | 1 | 2 | 0 | 0 | 0 |

Table 2: Vector Encoding of Test Cases after Random Projection

| Test Case \ Value | x  | y  | z  |
|------------------|----|----|----|
| (t₁)             | 0  | 0  | 0  |
| (t₂)             | 0  | 0  | 5  |
| (t₃)             | 0  | 3  | 2  |
| (t₄)             | 0  | 2  | 0  |

The optimization objectives is defined as:

\[
min(x₁ + x₂ + x₃ + x₄).
\]

As illustrated in Fig.3, recent ILP solvers are able to deliver an exact minimal solution. However, large amount of test cases naturally lead to high computational effort. We discuss two TSR heuristics to avoid intractable computational effort in case of larger-scaled problems, yet providing an acceptable approximation of the exact (minimal) solution.

**FAST++ Strategy.** The FAST++ approach encodes the selection of test cases into a (reduced) test suite using the Euclidean distance of a reduced vector-space model computed by random projection \[10\]. We also illustrate this approach using our example. Test cases are encoded as vectors, where the number of elements corresponds to the number of different input values of all test cases. The first element of each vector denotes the number of occurrences of the lowest input value (e.g., for our running example the number '0'). The next element of each vector is the next lowest input value (e.g., for our running example the number '1'). The encoding of all test cases is shown in Tab.1 and the result of reducing the dimensions of the vector-space model by random projection is shown in Tab.2.

Based on this encoding, TSR works as follows. First, a random test case (e.g., \(t₂\)) is selected into the (initially empty) reduced test suite. For the selected test case, the Euclidean distances to all other remaining test cases is computed based on the reduced vector-space model. The probability of selecting a particular remaining test case next into the reduced test suite increases with the distance value (thus preferring test cases covering test goals being dissimilar to those covered by previously selected ones). In our example, this would be, for instance, test case \(t₁\). This iterative step is repeated until all test goals are covered. For instance, the next test case might be \(t₃\) which suffices to achieve full branch coverage thus leading to termination. This technique is, on average, more efficient than ILP, yet potentially leading to less precise results as demonstrated by the example. There is not even a guarantee to find local optima as random projection may obfuscate necessary information. Additionally, the ordering of input values is ignored.

**DIFF Strategy.** This strategy also incrementally selects test cases until all test goals are covered \[11\]. In contrast to FAST++, DIFF is a purely greedy-based approach which always selects as next test case one that covers a maximum number of uncovered test goals. Applied to our example, DIFF would, for instance, perform the following selections:

1. \(t₃\) (covering 5 uncovered test goals),
2. \( t_1 \) (covering 1 uncovered test goal).

This technique is, in general, similarly efficient and (im-)precise as FAST++, where this particular example is well-suited for DIFF as the local optimum is the same as the global one. However, if local and global optima differ, there is no guarantee about the optimality of the result.

The different TSR technique might lead to different results in terms of the number of test cases of the reduced test suite. Although, this clearly leads to different results in terms of efficiency (in terms of test-suite size) it also might affect effectiveness. For example, test case \( t_2 \) does not cover additional test goals and also does not detect any bugs in program version \( P_1 \). However, in program version \( P_2 \) it will again detect Bug 3. We chose those three TSR strategies as they provide different trade-offs between the effectiveness of reduction (i.e., how small the resulting test-suite will be) and efficiency in terms of CPU time needed for reduction. Since ILP provides an optimal solution, the resulting test-suite will be minimal, however, leading to much more computational effort. The greedy approach (DIFF) is more efficient, however, the resulting test-suite might be larger compared to ILP. FAST++ is even more efficient compared to the greedy approach, however, leading to to even larger test-suites.

3.3 Regression-Test Selection Strategies

The TSR strategies considered so far are concerned with selecting from an existing test suite of one single program version a reduced subset preserving test coverage on that program version. In contrast, RTS strategies are concerned with selecting from an existing test suite of a previous program version a sufficient set of test cases to investigate modifications applied to that program version leading to a subsequent version. If no such test case(s) can be found among the existing ones, new test cases must be created and added to the test suite of the subsequent program version. We next describe different RTS strategies supported by our framework. We use an illustrative example to explain the impact of the strategies on the trade-offs between precision and computational effort. We start with a general characterization of the regression-test selection problem, consisting of a program version \( P \) with existing test suite \( T \) and subsequent version \( P' \) for which a regression test suite \( T' \) is created.

\[ \text{Input:} \text{ Program version } P_{i-1} \text{ with Test Suite } T_{i-1} \text{ and subsequent program version } P_i. \]

\[ \text{Output:} \text{ Regression Test Suite } T_i = T'_{i-1} \cup T'_i, \text{ where } T'_{i-1} \subseteq T_{i-1} \text{ are existing test cases selected for reuse from } T_{i-1} \text{ and } T'_i \text{ are newly added test cases.} \]

Consider \( P_3 \) from our running example together with test suite \( T_3 \). Figure 4 provides an illustration of the RTS problem. For \( P_3 \), the selected test suite contains three modification-traversing test cases (i.e., \( t_2 \), \( t_3 \) and \( t_4 \)) being able to at least traverse the modified line of code. However, only \( t_2 \) is modification revealing as it not only reaches the modification, but also disposes a difference in the output behavior of \( P_3 \) as compared to \( P_2 \). Hence, depending on the RTS strategy, also new test cases have to be generated for \( T'_i \).

Regression-Test-Case Generation. We now describe how we automatically generate regression test cases utilizing off-the-shelf test-generation tools originally designed for covering test goals encoded as reachability properties. We utilize a software model-checker for C programs to automatically derive test cases from counterexamples for (non-) reachability queries of test goals. We encode the problem of generating regression test cases for different versions of a program in a generic comparator-function. The shape of the comparator-function \( P_{i,j} \) for the program versions \( P_i \) and \( P_j \) depends on parameter \( \text{RTC} \). We illustrate this using the comparator-function \( P_{2,3} \) for our running example.
Figure 4: Comparison of Regression-Test Selection Strategies

(a) Modification-Traversing Comparator-Function for \( P_2 \) and \( P_3 \)

```c
int find_last_p2_3(int x[], int y) {
    if(x[0] <= 0)
        return -1;
    int last = -2;
    for (int i=1; i <= x[0]-1; i++) {
        test_goal:
        if (x[i] == y)
            last = i;
    }
    return last;
}
```

(b) Modification-Revealing Comparator-Function for \( P_2 \) and \( P_3 \)

```c
void find_last_p2_3(int x[], int y){
    if(find_last_p2(x,y) != find_last_p3(x,y)
        test_goal:printf("differencing test case found");
}
```

Figure 5: Comparator-Functions
\begin{figure}[h]
  \centering
  \begin{verbatim}
    int find_last_p3 (int x[], int y) {...}
    struct intStruct find_last_p4 (int x[], int y) {...}

    void find_last_p3_4(int x[], int y){
      if(find_last_p3(x,y) != find_last_p4(x,y)
        test_goal:printf("differencing␣test␣case␣found");
    }
  \end{verbatim}
  \caption{Invalid Comparator Program}
  \label{fig:invalidComparator}
\end{figure}

- **RTC = MT**: The comparator-function $P_{2,3}$ for generating modification-traversing test cases between $P_2$ and $P_3$ is shown in Figure 5a. The program location affected by the program modification leading from $P_2$ to $P_3$ is marked by a special program label `test_goal`. A test-case generator can be used to generate a program input reaching this program label (and therefore at least traversing this modification).

- **RTC = MR**: The comparator-program $P_{2,3}$ for generating modification-revealing test cases between $P_2$ and $P_3$ is shown in Fig. 5b. Here, the program input is delegated to $P_2$ and $P_3$ and the output is compared by an `equals`-function (or `!=` in case of basic types) based on the return-type. Hence, the special program label `test_goal` in line 3 is only reachable by a test case if the test input generated by a test-case generator yields different output values for $P_2$ and $P_3$ (thus not only reaching, but also revealing this modification).

**Remarks.** The problem of generating modification-revealing test cases obviously subsumes the generation of modification-traversing test cases and therefore, presumably, causes (much) more computational effort. In fact, both problems are undecidable as reachability of program locations is reducible to the halting problem. Hence, the test-generation approach described above is inherently incomplete.\footnote{This fact, however, is already true for the problem of generating a test case simply reaching a test goal in one version of a program in a non-regression scenario.} Our framework further comprises the parameter \texttt{NRT} to control the number of multiple different test cases distinguishing the output behavior of two consecutive program versions (which may, however, fail due to the theoretical limitations described before). Lastly, the technique based on the comparator-function suffers from some technical limitations. For example, if the data type of the return value changes the comparison of the return values is, in general, no more possible (see Fig. 6, where the return types of function `find_last_p3` and `find_last_p4` differ such that the comparator-program would not compile). Additionally, as both program versions need to be compiled into one single program, merging of those versions can become technical challenging (e.g., if the data type of global variables or functions changes or the members of structs are modified).

**Multiple Regression Test-Case Generation.** Most test-case generators transform the input program into an intermediate representation like a control-flow automaton (CFA). To keep the following examples graspable, we use the basic example shown in Figure 7a instead of our running example. This CFA corresponds to a small program with input value $x$ returning 0 if $x$ is smaller than zero and the value of $x$, otherwise. A CFA can be used to identify all program paths leading to test goals (e.g., all CFA-edges in case of branch coverage) and to control automated test-suite generation by performing reachability queries for uncovered test goal \footnote{\cite{14}}. To support parameter \texttt{NRT}, we have to generate multiple different test cases reaching the same test goal (i.e., a modified program location). We start by generating the first test case based on the original CFA. For instance, to generate a test case reaching the
**Figure 7: On-the-fly CFA Instrumentation**

The return-edge in Fig. 7a can be any input is feasible (e.g., input -5 would traverse the left path). For further test cases reaching the return-edge, we modify the CFA as shown in Fig. 7b to exclude already covered paths. For this, we add for each path for which we already created a test case a fresh variable with initial value 0 (i.e., variable $w1$ introduced by the new edge from node 2 to $e1$). We assign value 1 to this variable after traversing the associated branch of the path (i.e., the new edge from node 4 to $e2$). By further requiring the value of at least one of the fresh variables before reaching the test goal must not be 1 (i.e., the new edge from node 6 to $e3$), we exclude all previous test cases reaching that test goal. In the example, assumption $w1! = 1$ enforces a further test case to assign a positive value to $x$, whereas a third run of the test-case generator will find no further path reaching this goal.

For example, to generate two test cases reaching line 7 in program version $P_3$, we first generate a test case as usual (e.g., $t_7 = (x=[1,0], y=1$)). For the second test case, we introduce a fresh variable into the CFA which is incremented when traversing a branch of the path already taken by the previous test case (i.e., the true-branch of the if-statement, the true-branch of the for-loop, the false-branch of the for-loop and the false-branch of the last if-statement). As there are four branches, the final check will ascertain that the fresh variable is not equal to 4. When iterating through the for-loop more than once (as opposed to the first test case) or taking the true-branch of the last if-statement, the value will differ from 4 and, therefore, the path can be taken for a further test case being different from all previous ones (e.g., $t_8 = (x=[2,0,0], y=1$)).

**Remarks.** Again, this methodology is inherently incomplete as it is unknown whether further test cases exist if no more test cases are found by the test-case generator (e.g., due to time-outs). In addition, each incrementally added fresh variable presumably increases program complexity thus potentially increasing the computational effort required for every further test case.

**Regression Test-Suite Generation.** We conclude by summarizing the description of our framework. Algorithm 1 depicts a generic procedure for generating regression test suites $T_i$ for program version $P_i$ meeting all possible instantiations of the parameters RTC, NRT, NPR, RS and CR. Parameter RTC is either set to modification-traversing (MT) or modification-revealing (MR), whereas NRT and NPR can be any number greater than zero (NPR is naturally restricted to be at most $i$). Besides the current version $P_i$, the algorithm receives as inputs the sequence of patches applied since version $P_0$ as well as the test suite $T_{i-1}$ of the preceding version $P_{i-1}$ based on parameter CR. Lastly, RS is set to ILP, DIFF, FAST++,
Algorithm 1 Regression Test-Suite Generation

Input: \( \text{RTC} \in \{MT, MR\} \), \( \text{NRT} \in \mathbb{N}^+ \), \( \text{NPR} \in \{1, \ldots, i\} \)

Input: \( P_i, \text{Patch}_1, \ldots, \text{Patch}_i \)

Input: \( T_{i-1} \)

Output: \( T_i \)

1: function \( \text{Main} \)
2: \( T_i \leftarrow T_{i-1}(\text{CR}) \)
3: \( T_{i,j} \leftarrow \emptyset \)
4: \( P_j \leftarrow P_i \)
5: for \( k = 0 \) to \( \text{NPR} - 1 \) do
6: \( P_j \leftarrow \text{apply Patch}_{i-k} \) to \( P_j \)
7: \( P_{i,j} \leftarrow \text{Comparator}(P_i, P_j)(\text{RTC}) \)
8: \( l \leftarrow 0 \)
9: while \( l < \text{NRT} \) do
10: \( t \leftarrow \text{NewTestGen}(P_{i,j}, T_{i,j}) \)
11: if \( t = \text{null} \) then
12: continue in line 5
13: end if
14: \( T_{i,j} \leftarrow T_{i,j} \cup \{t\} \)
15: \( l \leftarrow l + 1 \)
16: end while
17: end for
18: \( T_i \leftarrow T_i \cup T_{i,j} \)
19: \( T_i \leftarrow \text{Reduction}(T_i)(\text{RS}) \)

or \( \text{None} \).

Test suite \( T_i \) is initialized with the test cases from the existing test suite \( T_{i-1} \) (line 2), and \( P_j \) (initialized in line 4 by \( P_i \)) refers to the previous version for which differentiating test cases are generated in the next iteration. The outer loop from line 5 to 15 performs a descending traversal over \( \text{NPR} \) previous versions by reversely applying the corresponding patches to \( P_j \) (i.e., applying Patch\(_{i-k}^\text{-1} \) to \( P_j \) yields version \( P_{j-1} \)). In line 7 the \( \text{comparator-function template} \) is instantiated with \( P_i \) and \( P_j \) to synthesize a \( \text{comparator-function} \) \( P_{i,j} \) as input for a coverage-based test-case generator in line 10 [13]. The inner loop (lines 9–15) repeatedly generates test cases for \( P_{i,j} \) until at most \( \text{NRT} \) different test cases have been generated (cf. Sect. 3.3). The test-case generator receives as additional input the set \( T_{i,j} \) of already generated differentiating test cases. If the test-case generator fails to find further test cases (either due to a time-out or due to exhaustive search), the current iteration of the inner loop is terminated and the next iteration of the outer loop starts (lines 11–12). Otherwise, the new test case is added to \( T_{i,j} \) (line 14) and finally to \( T_i \) (line 18). Finally, the test-suite is reduced depending on parameter \( \text{RS} \) in line 19.

Remarks. For generating multiple \textit{different} test cases for the \textit{same} modification, only the set \( T_{i,j} \) of already generated differentiating test cases for the current pair \( P_i, P_j \) of program versions is taken into account. However, the approach may be generalized by blocking the whole set \( T_i \) of already generated test cases in subsequent runs of the test-case generator. This may, however, lead to drastically increased computational effort and is, therefore, currently not supported.

3.4 Integration and Discussion of Strategies

The parameters \( \text{RS}, \text{RTC}, \text{NRT}, \text{NPR} \) and \( \text{CR} \) allow for adjustments of efficiency and effectiveness achievable by a regression-testing strategy concerning the detection of regression bugs.
Efficiency. Efficiency of regression-testing strategies may be measured in terms of (1) the computational effort (e.g., CPU time) for selecting/generating regression test cases, together with (2) the number of regression test cases being selected. Concerning (1), strategies with \( RTC = MT \) are presumably more efficient than those with \( RTC = MR \) as finding modification-revealing test cases is, on average, computationally more complicated than only reaching a modified program location. Concerning (2), the number of regression test cases to be selected for each new program version is (at most) \( NRT \cdot NPR \) thus presumably growing with increasing values of \( NRT \) and \( NPR \). Additionally, strategies with \( RS \neq None \) are presumably less efficient in terms of CPU time, but presumably much more efficient in terms of the number of test cases. Finally, \( CR \) also presumably decreases efficiency in terms of CPU time, however presumably increases efficiency in terms of number of test cases.

Effectiveness. Effectiveness of regression-testing strategies may be measured in terms of the number of regression bugs detected by the selected regression test cases. Test cases selected for \( RTC = MR \) are presumably more effective than those for \( RTC = MT \). Similarly, fault-detection rates of regression test suites presumably increase with higher values for \( NRT \) (i.e., leading to more different ways of testing modifications between program versions) as well as \( NPR \) (i.e., taking more previous program versions into account). For instance, concerning the revision from \( P_2 \) to \( P_3 \) in our example, revealing bug 3 in \( P_2 \) requires a test case such as \( t_2 \). This can be ensured by strategies with \( RTC = MR, NRT > 0, \) and \( NPR > 0 \) which are, however, presumably less efficient than strategies with \( RTC = MT \). Finally, \( RS \) and \( CR \) might decrease effectiveness as reducing the test suite might remove test cases that would succeed in finding the bug (see Sect. 3.2).

In the next section, we empirically investigate these assumptions by presenting the results of an experimental evaluation gained from applying a tool for our methodology to a collection of subject systems.

4 Experimental Evaluation

The configurable framework presented in the previous section allows us to investigate the impact of the parameters \( 1-5 \) (see Sect. 3) on efficiency and effectiveness of regression testing. In our experimental evaluation, we consider version histories of C program units (i.e., function level). We use real-world systems obtained from GitHub and the change history provided by git. To systematically compare effectiveness of different regression-testing strategies, we further employ simulated bugs throughout program-version histories and measure the corresponding fault-detection ratio of the selected test suites. We do this by repeatedly applying standard mutation operators for C programs. Correspondingly, we measure efficiency in terms of the computational effort for generating regression test cases as well as in terms of the number of test cases to be (re-)executed throughout the version history.

4.1 Research Questions

We consider the following research questions.

(RQ1) How does the regression-testing strategy impact testing effectiveness?

(RQ2) How does the regression-testing strategy impact testing efficiency?

(RQ3) Which regression-testing strategy constitutes the best trade-off between effectiveness and efficiency?
4.2 Experimental Setup

We next describe the evaluation methods and experimental design used in our experiments to address these research questions.

Methods and Experimental Design  To compare different strategies for regression-testing, we instantiate the five parameters \(\text{RTC}, \text{NRT}, \text{NPR}, \text{RS} \text{ and } \text{CR} \) of our methodology as described in Sect. 3 as follows:

- \(\text{RTC} \in \{\text{MT, MR}\} \) (RTS Generation Criterion).
- \(\text{NRT} \in \{1, 2, 3\} \) (Number of Regression Test Cases per Revision).
- \(\text{NPR} \in \{1, 2, 3\} \) (Number of Previous Program Revisions).
- \(\text{RS} \in \{\text{None, ILP, FAST++}, \text{DIFF}\} \) (TSR Strategy).
- \(\text{CR} \in \{\text{CR, No-CR, None}\} \) (Continuous Reduction).

To answer RQ1-RQ3, each instantiation of all five parameters, therefore, corresponds to one particular regression-testing strategy under consideration, where we denote a strategy by \([\text{RTC, NRT, NPR, RS, CR}]\) for short. We thus obtain 144 reasonable regression-testing strategies (since the combination of \(\text{RS} = \text{None} \text{ and } \text{CR} = \text{CR}\) is meaningless and we do not take into account strategies having \(\text{CR} = \text{None} \text{ and } \text{RS}! = \text{None}\) as reduction of non-accumulated test-suites is also meaningless). Two of these strategies may be considered as proper baselines:

- Baseline 1 (basic regression testing strategy): \([\text{MT, 1, None, No-CR}]\).
- Baseline 2 (basic regression testing without initial test suite): \([\text{MT, 1, None, None}]\).

We limit our considerations to 3 as maximum value for NRT and NPR as the increase of effectiveness diminishes for higher values of NRT and NPR. Additionally, we do not consider 0 as value for NRT and NPR, as this would obviously result in an empty test-suite.

We further divide RQ2 into the following sub-questions.

- (RQ2.1) How does the regression-testing strategy impact efficiency in terms of CPU time?
- (RQ2.2) How does the regression-testing strategy impact efficiency in terms of number of test-cases?

Note that we consider test-case generation time for CPU time and not test-case execution time. This is due to the fact, that test-case execution time was negligible during our evaluation.

Subject Systems.  We consider program units as our subject systems in terms of testable functions extracted from real-world C programs for which we further require available version history. We therefore focus on open-source projects from GitHub. We selected program units in terms of preprocessed c-files consisting of an entry-function (i.e., the function-under-test) as well as all functions within the same compilation unit having (direct or indirect) callee-dependencies to the function-under-test. To sum up, our subject systems have to fulfill the following requirements to be useable for our evaluation purposes:

- The unit must be processable by the test-case generator used in our framework (e.g., syntax must be ANSI C, no multi-threading, etc.).
• The functions-under-test (as well as all callees) must have undergone a version-history of at least 3 modifications.

• The signature of the functions-under-test must provide input parameters for which different values will yield different return values (or affect the values of at least one global variable). However, this needs not to be known in advance as the existence of such parameters is actually checked during test generation (which may timeout if this requirement is not met).

• Calling the function-under-test multiple times with the same parameter values produces the same return value or global-variable values (i.e., no non-deterministic behavior or external system-function calls are considered).

The resulting collection of subject systems comprises program units from open-source projects published in GitHub:

• betaflight\(^9\) contains six program units from the flight controller software betaflight.

• netdata\(^{10}\) contains six program units from the infrastructure monitoring and troubleshooting software netdata.

• wrk\(^{11}\) contains one program unit from the HTTP benchmarking tool wrk.

The size of subject systems ranges from 270 to 950 lines of code (after removing unnecessary code). The number of changes (i.e., commits) for each subject system is between 4 and 18 (only counting commits changing code inside the unit).

**Simulating Bugs.** Although software evolution and regression testing become more and more important, properly documented and practically usable histories of program versions combined with information about real bugs in software projects are still barely available. Hence, for in-depth investigations of the interactions between program revisions, program bugs and their detection by regression test cases as required for answering our research questions, we have to rely on synthetically generated bugs. Mutation testing is a well-established approach for evaluating effectiveness of testing techniques by simulating common program faults [15]. In particular, mutation-testing tools provide collections of syntactic program-transformation operations such that the resulting modified program potentially shows different (i.e., erroneous) output behaviors.

Fortunately, it has been recently shown that mutations provide a reliable substitute for realistic bugs in testing experiments [15]. We, therefore, utilize concepts from mutation testing to simulate bugs in our subject systems. Figure 8 provides an overview of our approach. For each program version other than program version \(P_0\), we create mutants containing simulated bugs. The regression-test generation is then executed on this bugged version and effectiveness of the resulting test suite is measured by executing the generated/selected regression-test suite on both the bugged and the bug-fixed (i.e., the original) version and by comparing the return values (see Sect. 3). If the return values differ, the bug is successfully detected by the test-suite. We used 62 mutation operators in total (the names of the mutation operators are provided on our supplementary web page \(^{12}\), which can be clustered into three different types. The first group consists of mutation operators replacing constants and variables (e.g., replace variable \(a\) by variable \(b\), replace constant 5 by constant 6 etc.). The second group consists of mutation operators replacing operators (e.g., replacing + by \(-\)). The last group consists of mutation operators replacing pointers and array references (e.g., replacing pointer \(pt1\) by pointer \(pt2\)).

---

\(^{9}\)https://github.com/betaflight/betaflight

\(^{10}\)https://github.com/netdata/netdata

\(^{11}\)https://github.com/wg/wrk

\(^{12}\)https://www.es.tu-darmstadt.de/regrets
Data Collection. For comparing different regression-testing strategies, we first generate a faulty revisions $B_i$ from all revisions $P_i$ of each subject system by simulating bugs as described before. Next, we introduce program labels into each faulty revision $B_i$ marking each line of code which has been modified since the last version $P_{i-1}$. These labels are used as test goals to generate modification-traversing test cases (similarly for $P_{i-2}$ and $P_{i-3}$ depending on parameter NPR). In addition, we generate the comparator-programs described in Sect. 3 for generating modification-revealing test-cases. The comparator-program is further used for comparing test-case executions on the faulty version $B_i$ and the previous versions $P_{i-1}$, $P_{i-2}$ and $P_{i-3}$. If no comparator-program can be created (e.g., due to the limitations explained in Sect. 3), no test suite will be generated for all strategies targeting this specific comparator-program.

We applied Algorithm 1 to generate regression-test suites $T_i^s$ for every bugged revision $B_i$ using all 144 strategies.

To answer RQ1, we measure effectiveness of the generated test suites such that test suite $T_i^s$ detects the bug in $B_i$ if it contains at least one test case $t$ whose execution yields different output values when executed on $P_i$ and $B_i$ (denoted $P_i(tc) \neq B_i(tc)$). Hence, effectiveness with respect to a test suite $T_i^s$ is calculated as

$$\text{effectiveness}(T_i^s) = \text{detects}(T_i^s, i)$$

where $\text{detects}(T_i^s, i) = 1$ if $\exists t \in T_i^s : P_i(t) \neq B_i(t)$, or 0 otherwise, and $B_i$ is the faulty revisions generated for $P_i$. Effectiveness of a strategy $s$ is calculated as

$$\text{effectiveness}(s) = \frac{\sum_{i=0}^{n} \text{effectiveness}(T_i^s)}{n}$$

where $n$ is the number of revisions.

To answer RQ2, we measure efficiency in two ways. First, efficiency of regression test suites generated by strategy $s$ for program unit $P_i$ is calculated as the average size of test suites $T_i^s$ in terms of the number of test cases

$$\text{efficiency}_{size}(T_i^s) = |T_i^s|$$

such that the overall efficiency of strategy $s$ is calculated as

$$\text{efficiency}_{size}(s) = \frac{\sum_{i=0}^{n} \text{efficiency}_{size}(T_i^s)}{n}.$$
equation for calculating $\text{efficiency}_{\text{CPU}}$ of the different strategies is
\[
\text{efficiency}_{\text{CPU}}(s) = \frac{\sum_{i=0}^{n} \text{efficiency}_{\text{CPU}}(T_{s}^{i})}{n}.
\] (12)

Finally, to answer RQ3, we calculate the mean values of each strategy for each program unit in our subject system for effectiveness and efficiency and, thereupon, calculate the trade-off as
\[
\text{trade-off}(s) = \frac{\text{effectiveness}(s)}{\text{efficiency}_{\text{size}}(s)}.
\] (13)

**Tool Support.** We implemented Algorithm 1 in a tool, called REGRETS (Regression Testing Strategies). REGRETS extends the software model-checker CPAchecker for C programs to generate regression test cases. This is achieved by performing reachability-analysis runs for program locations marked as test goals (see Sect. 3). Additionally, we use MUSIC\textsuperscript{[16]}, a mutation tool for C programs, to generate synthetic bugs as described above. For measuring test coverage, we utilize TestCov\textsuperscript{[13]} a test-suite executor for C programs. TestCov further supports TSR with a greedy algorithm, as explained in Sect. 3, called DIFF. Furthermore, we extended TestCov to also support FAST++ and ILP. All tools are available on our website\textsuperscript{[14]}

**Measurement Setup.** REGRETS as well as MUSIC and TestCov have been executed on an Ubuntu 18.04 machine, equipped with an Intel Core i7-7700k CPU and 64 GB of RAM. The CPU time for test-suite generation is limited to 900 s per revision, and the CPU time for the execution of the test cases to detect bugs is limited to 30 s per test case. The TSR is not limited by CPU time as the CPU time needed for reduction was negligible. We executed our evaluation with Java 1.8.0-171 and limited the Java heap to 15 GB.

4.3 Results

The measurements for RQ1 are depicted in Fig. 9a for all strategies with $\text{RTC} = \text{MR}$ and in Fig. 9b for all strategies with $\text{RTC} = \text{MT}$. Results for RQ2 are shown in Figs. 10a and 11a for all strategies with $\text{RTC} = \text{MR}$ and Figs. 10b and 11b for all strategies with $\text{RTC} = \text{MT}$. The results for RQ3 are shown in Fig. 12a for all strategies with $\text{RTC} = \text{MR}$ and Fig. 12b for all strategies with $\text{RTC} = \text{MT}$. The box plots in Figs. 10a, 10b, 11a and 11b aggregate the results after applying the formulas 11 and 12 (see above) for all subject systems. The boxes depict the range of results while the black dashes depict the mean values. The whiskers show the minimum and maximum values (excluding outliers).

RQ1 (Effectiveness). The best effectiveness for our subject systems is achieved by the strategies $\{\text{MR}, 3, 3, \text{None}, \text{No-CR}\}$, $\{\text{MR}, 2, 3, \text{None}, \text{No-CR}\}$, $\{\text{MR}, 2, 3, \text{None}, \text{No-CR}\}$, $\{\text{MR}, 3, 3, \text{FAST++}, \text{CR}\}$ and $\{\text{MR}, 3, 3, \text{FAST++}, \text{No-CR}\}$ with an average bug detection rate of 0.3659. Compared to the baseline 1 $\{\text{MT}, 1, 1, \text{None}, \text{No-CR}\}$ with an average detection rate of 0.306 and baseline 2 $\{\text{MT}, 1, 1, \text{None}, \text{None}\}$ with an average detection rate of 0.224. The worst performing strategy is $\{\text{MT}, 3, 3, \text{FAST++}, \text{CR}\}$ with an average detection rate of 0.119.

RQ2 (Efficiency). The best efficiency measure concerning CPU time is obtained by strategy $\{\text{MT}, 1, 1, \text{None}, \text{No-CR}\}$ and $\{\text{MT}, 1, 1, \text{None}, \text{None}\}$ with an average amount of 6.058s. The worst performing strategy, $\{\text{MR}, 3, 3, \text{ILP}, \text{No-CR}\}$, requires 993.229s on average. Concerning test-suite sizes, $\{\text{MT}, 1, 1, \text{ILP}, \text{CR}\}$, $\{\text{MT}, 2, 1, \text{ILP}, \text{CR}\}$, and $\{\text{MT}, 3, 1, \text{ILP}, \text{CR}\}$

\textsuperscript{[13]}https://gitlab.com/sosy-lab/software/test-suite-validator
\textsuperscript{[14]}https://www.es.tu-darmstadt.de/regrets
perform best with an average measure of 0.225 test cases, whereas strategy \([\text{MT, 3, 3, None, No-CR}]\) leads to the largest test-suite sizes with an average measure of 115.44 test cases.

**RQ3 (Trade-Off).** The base trade-off between effectiveness and CPU time is obtained by strategy \([\text{MT, 1, 1, None, No-CR}]\) with an average of 0.0503 bugs found per second. The worst performing strategy is \([\text{MT, 3, 3, FAST++, CR}]\) with an average of 0.0015 bugs found per second.

### 4.4 Discussion and Summary

**RQ1 (Effectiveness).** Setting parameter \(\text{RTC}\) to \(\text{MR}\) increases effectiveness for all strategies. In fact, almost all strategies using \(\text{MR}\) are more effective than even the most effective strategy using \(\text{MT}\). Indeed, parameter \(\text{RTC}\) has the highest impact on effectiveness. While the impact of other parameters is smaller, it is nonetheless also observable. Choosing \(\text{None}\) for parameter \(\text{RS}\) increases effectiveness, as the other parameter value reduces effectiveness by reducing the number of test cases. Choosing \(\text{CR}\) or \(\text{No-CR}\) for parameter \(\text{CR}\) has nearly no impact on effectiveness. Strategies with \(\text{MR}\) and \(\text{ILP}\) cause a significant loss in effectiveness if \(\text{CR}\) is selected. Choosing \(\text{None}\) for parameter \(\text{CR}\) has a negative impact on strategies with \(\text{MR}\) selected, however, only having a small impact if \(\text{MT}\) is selected. Lastly, parameters \(\text{NRT}\) and \(\text{NPR}\) also increase effectiveness with increasing values (even in case of small increases).

| Answer RQ1 |
|----------------|
| The best effectiveness measure is reached by strategy \([\text{MR, 3, 3, None, No-CR}]\) (i.e., modification revealing test cases, three test cases per test goal, up to three previous revisions and no test-suite reduction) which improves effectiveness compared to the baseline \([\text{MT, 1, 1, None, No-CR}]\) by 19% and compared to the baseline \([\text{MT, 1, 1, None, None}]\) by 61%. The highest impact on effectiveness is caused by parameter \(\text{RTC}\). |

**RQ2.1 (CPU Time).** Parameter \(\text{RTC}\) has by far the highest impact on CPU time. When choosing \(\text{MT}\), CPU time increases nearly 20 fold. As expected, parameter \(\text{NPR}\) increases the CPU time nearly linearly to its respective value. Unexpectedly, parameter \(\text{NRT}\) does not affect CPU time by a large margin. This is most likely due to the fact, that reachability information computed during the first run of the test generator can be re-used for the next test cases. Parameters \(\text{CR}\) and \(\text{RS}\) are negligible in terms of CPU time if \(\text{MR}\) is selected, since the number of test-cases remains small. If \(\text{MT}\) is selected, both parameters impact CPU time, however still only by a small margin.

| Answer RQ2.1 |
|----------------|
| The best efficiency measure in terms of CPU time is reached by the baseline strategies \([\text{MT, 1, 1, None, No-CR}]\) and \([\text{MT, 1, 1, None, None}]\) (i.e., modification traversing test cases, one test case per test goal, one previous revision and no test-suite reduction, either ignoring or using the previous test-suite (as this makes no different in CPU time)). The highest impact on CPU time is caused by parameter \(\text{RTC}\). |

**RQ2.2 (Test-Suite Size).** The highest impact on efficiency in terms of test-suite size is caused by parameter \(\text{RS}\). If \(\text{None}\) is selected, no reduction is enabled, and therefore, the test-suite grows with each version. However, the choice of the technique used for test-suite reduction only has a small impact on the test-suite size.

Parameter \(\text{RTC}\) also has a high impact on the test-suite size. This is due to the fact, that a patch might contain multiple modifications. Therefore, the number of test cases is higher in case of \(\text{MT}\) (i.e., requiring one test-case per modified line) as compared to \(\text{MR}\), where only one test case is required. Parameter \(\text{CR}\) also affects test-suite sizes, but only by a small amount.
Figure 9: Results Effectiveness
Figure 10: Results Generation Time

(a) Strategies with \(RTC = MR\)

(b) Strategies with \(RTC = MT\)
Figure 11: Results Test-Suite Size

(a) Strategies with $RTC = MR$

(b) Strategies with $RTC = MT$
(a) Strategies with RTC = MR

(b) Strategies with RTC = MT

Figure 12: Results Bugs per Second
Lastly, parameters NRT and NPR affect the test-suite size almost linearly to their respective values.

| Answer RQ2.2 |
| ------------ |
| The best efficiency measure in terms of test-suite size is reached by the strategy [MR, 1, 1, ILP, CR] (i.e., modification revealing test cases, one test case per test goal, one previous revision, ILP as test-suite reduction strategy and using the reduced test-suite of the previous revision) which improves effectiveness compared to the baseline [MT, 1, 1, None, No-CR] by 6200% and compared to the baseline [MT, 1, 1, None, None] by 970%. The highest impact on the test-suite size is caused by parameter RS. |

| Answer RQ3 |
| ---------- |
| The strategy yielding the best trade-off is [MT, 1, 1, None, No-CR] (i.e., modification traversing test cases, one test case per test goal, one previous revisions, no test-suite reduction and using the test-suite of the previous revision as well) for which the fault-detection capability is acceptable, but the efficiency in terms of CPU time is very high. Compared to the second baseline [MT, 1, 1, None, None] the trade-off is increased by 36%. |

Remarks. Strategy [MT, 3, 3, None, No-CR] is less effective than strategy [MT, 3, 2, None, No-CR] which, by definition, is counter-intuitive. This is due to the fact, that some comparator-programs for comparing a faulty version \( B_i \) to a program revision \( P_{i-2} \) are invalid (e.g., due to different return types), whereas the comparator-program for \( B_i \) and program revisions \( P_{i-3} \) are actually valid. However, no test cases could be found by the test-case generator revealing the bug. Hence, the number of test suites for [MT, 3, 3, None, No-CR] and [MT, 3, 2, None, No-CR] differ slightly, which is the reason for the discrepancy between the factual results and the theoretical specification of the technique. In our subject systems, the probability that no valid comparator-program can be generated is approximately 8%. Therefore, while this technical limitation is present, it should not affect the results by a large margin.

4.5 Threats to Validity

Internal Validity. Our regression-testing methodology relies on the assumption that when different program versions are tested with the same test case, all factors (e.g., platform, environment) that might influence the output except for the source code itself remain constant. This so-called controlled-regression-testing assumption is commonly used in regression-testing experiments and does, therefore, not harm validity of the results [2].

Concerning the soundness of our methodology, we tested the test-generation loop by manually checking results for selected subject systems. However, due to undecidability of reachability of program locations, if no more test cases can be found (e.g., due to time-outs or imprecise counter-examples), it is unknown if further test cases exist. Nevertheless, we expect precision improvements to not substantially obstruct the (relative) results of our evaluation.

Another threat to validity might arise from our selection of mutation operators and their applications to our subject systems. However, our selection comprises those mutations leading to useful results w.r.t. our experimental setting, namely affecting one line of code, performing no code deletions and producing a compilable result.

Another threat to validity might arise from our selection of mutation operators and their applications to our subject systems. However, our selection comprises those mutations leading to useful results w.r.t. our experimental setting, namely affecting one line of code, performing no code deletions and producing a compilable result.

Limiting our considerations to (functional) unit testing may also threaten internal validity. As unit testing constitutes the most established and relevant testing technique in practice, it is particularly interesting and relevant to investigate our methodology at this level first. In addition, the proposed concepts might be likewise applicable at integration- and system-level.
Additionally, our methodology does not incorporate systematic reusability-checks of existing test cases for revealing modifications also in later revisions which may affect efficiency measures. We plan to extend our approach, accordingly, in a future work but we expect similar results as in our current setting.

Finally, we expect our current focus on C programs to also not seriously harm validity as we expect similar results for other programming languages, at least for those relying on an imperative core (e.g., most OO languages).

**External Validity.** We are not aware of any competitive tools with similar functionality as REGRETS, especially concerning the generation of a configurable number of modification-revealing test cases. Surprisingly, it was not possible for us to use other recent test-case generators for strategies with \( RTC = MR \), which, by design, should have been possible. This might be due to the fact, that the subject systems are real-world programs, which where explicitly selected to be processable by our test-case generator. However, test-case generators are (usually) limited in supporting certain constructs of the C language. We tried to use other test-case generators (i.e., KLEE, FuSEBMC, SYMBIOTIC and PRTEST) from the international testing competition 2021 [17]. However, these test-case generators were barely able to generate any test cases at all. In addition, successfully generated test cases were actually unable to reveal modifications between different program version and were thus immediately removed from the test-suites during TSR (leading to empty test suites). Only PRTEST was able to generate some meaningful test-cases but was, however, also not able to generate any test case for more than half of the subject systems thus being unusable for a proper comparison. However, the main focus of this paper is to compare different regression-testing strategies, and not to compare different test-case generators for regression-test generation. We thus assume the results to be very similar for other test-case generation techniques (even though effectiveness in terms of CPU time might change, the ratio of the CPU time of different strategy presumably stay similar).

Another threat might arise from the selection of subject systems and the usage of simulated bugs. Unfortunately, real-world systems with sufficient information about revisions and bugs as required for our experiments are barely available. We evaluated three prominent candidates for potential candidates. First, CoreBench only provides a very short version history (often only 1–2 versions) and incorporates many bugs being undetectable at unit level (e.g., involving files and global errors like overflows) [18]. Second, the regression-verification tasks from the SV-Benchmarks [19] also have a small version history and the different tasks cannot be executed in a self-sustained manner as needed to reveal those bugs. However, we spend a lot of time in searching for other freely available community benchmarks including version history and known bugs. However, suitable benchmarks are still very rare. Amongst others, we had a look into further subject systems from the SIR Repository including programs like gzip and make [20], but either our test-case generator was not able to handle those programs for mostly technical reasons, or the programs were not suitable for our purposes (see descriptions above). As also already discussed above, mutation testing is a reliable fault-injection technique for measuring effectiveness in testing experiments [15].

Finally, our tool relies on third-party software, namely CPAchecker, a software model-checker, and the mutation tool MUSIC for C programs. However, both tools are established and have been extensively used for other experiments in the recent past, so we expect them to produce sound results.
5 Related Work

5.1 Regression-Testing

A comprehensive overview about regression testing is provided by [2], describing three categories: (1) minimization of test suites as well as (2) selection and (3) prioritization of test cases for regression testing.

**Test-suite minimization** is concerned with selecting from an existing test suite a subset of test cases to reduce the number of redundant test executions during regression testing. Many works propose heuristics for approximating near-optimal solutions [4, 21–23] for this NP-complete optimization problem, requiring as inputs an existing test suite and a-priori defined metrics for measuring effectiveness of test cases. The approaches used in this paper for TSR have been proposed before. The greedy algorithm as used by DIFF has been introduced in [11]. To use ILP solving for TSR was initially proposed by [6] and FAST++ has been introduced by [10]. However, none of these works investigate the interactions between RTS and TSR techniques as done in this paper.

[24] evaluate existing test-suite-reduction techniques on real-world projects based on their failed builds. [25] proposed a new technique to reduce test suites based on assertions instead of structural code coverage to improve effectiveness of the resulting test suite. In an earlier work, [26] compare and combine TSR and test-case selection to further increase efficiency of regression testing. Our methodology goes beyond their approach as we consider further strategic parameters which turned out to be very relevant.

**Regression-test selection** is concerned with selecting from an existing test suite of an evolving program a subset of test cases to be (re-)executed on a new program version. A variety of different techniques has been applied (e.g., control-flow analysis [27–29] and/or data-flow analysis [30–33]). Other works take behavior-preserving modifications (e.g., refactorings) into account [34], apply RTS to highly-configurable software [35], and try to find pareto-optimal solutions for multi-objective RTS [36]. However, none of these works aim at generating new modification-revealing test cases to enhance effectiveness of regression testing as done in our work. In particular, most recent works only guarantee test cases to be modification-traversing.

**Test-case prioritization** is concerned with selecting from an existing test suite a (re-)test-execution order such that effectiveness of testing increases as quickly as possible over time (e.g., to find as many faults as fast as possible) [37]. The underlying problem is very similar to minimization/selection problems. Most existing approaches consider code coverage as effectiveness criterion to statically compute an (a-priori) ordering among test cases [38, 39]. In a recent work, Wang and Zeng propose a dynamic prioritization technique based on fault-detection history and other properties [44]. In contrast, in our methodology, prioritization is currently out of scope, but may be easily incorporated during test-case generation using recent approaches.

5.2 Test-Case Generation

A wide range of technique exist for (automatically) generating test cases which we will describe in the following grouped by the test-case generation technique applied. However, we are not aware of related works in terms of the multiple test-cases per test goal to increase effectiveness of the resulting test suite.
5.2.1 Coverage-Based Test-Case Generation

**Fuzzing.** Fuzzing is currently very popular both in research and practice. The idea is to quickly generate a large number of test cases by generating (semi-)random input values. In some approaches, the input values of existing test cases are reused and modified to generate new test cases. \[45\] Recent fuzzing techniques are based on evolutionary algorithms \[46\] or context-free grammars of the input data \[47\]. In addition, grey-box fuzzing \[48\] and whitebox fuzzing \[47, 49\] have been proposed (i.e., fuzzers also considering the source code of the program under test). However, the primary goal of fuzzing is not to generate test cases for regression testing systematically traversing / revealing particular program modifications through different possible paths and/or program versions as done in our approach.

**Plain Random.** A test-case generation technique that is similar to, yet simpler than, is plain random test-case generation. \[50\]. Test cases are randomly generated, and afterwards, the achieved coverage is measured. This approach is clearly more efficient than test-goal guided techniques. On the other hand, more complicated test goals are often not reached, as the chance to generate valid input values reaching those goals is small (e.g., to generate a test case for input value \(x\) which evaluates true for \(x == 1\) is \(\frac{1}{2^{32}}\) for a 32-bit system). Therefore, generating (multiple different) modification-revealing test cases is usually extremely expensive and ineffective using purely random approaches.

**Symbolic Execution.** Symbolic execution employs a symbolic reachability graph cope with the reachable state space of input programs during test-case generation. One prominent example is Klee \[51\]. Based on symbolic execution, it might be also possible to generate multiple test cases covering the same goal through different paths as done in our work. However, we are not aware of any recent work going into this direction.

**Bounded Model Checking.** Another technique to scale test-case generation to larger programs is bounded model checking \[52\]. A bounded model checkers also computes the reachable state space of programs (either in an abstract or concrete representation), where loops are only explored up to maximum number of iterations \(k\) (bound). This enables the model checker to prove program properties with certainty only within that bound. Such a tool can be also used for test cases generation similar to symbolic model checking (i.e., by encoding test goals as reachability problems, see below). Again, we are not aware of any works using bounded model checking to generate multiple test cases for the same test goal or program modification, respectively.

**Symbolic Model Checking.** Another approach to handle larger input programs is symbolic model checking as applied, for instance, by the CPAchecker framework, which is also used by our approach \[13\]. CoVeriTest is another recent test-case generator based on the CPAchecker framework \[53\]. Again, these and other tools do currently not support generation of multiple test-cases per test goal or program modification as required in our approach. However, encoding the underlying problem as reachability query as done in our approach would also enable the usage of these other tools for regression-test generation.

5.2.2 Regression-Test-Case Generation.

We next discuss related work on generating test cases for systematically investigating semantic differences between similar programs. **Differential testing** \[54\] is concerned with the following problem: Given two comparable programs and an set of differential test cases, the systems can be checked for bugs by running the test cases. If the outputs differ or the test loops indefinitely
or crashes, the test case is a candidate for a bug-revealing test. Thereupon, Evans and Savioa proposed an approach for detecting program changes by comparing test-execution results of two different program versions [55]. The tool CSMITH combines differential testing with fuzzing (i.e., C programs) to find bugs in C-compiler implementations [56]. The work being presumably most closely related to our methodology is DiffGen [57] for generating test cases comparing two versions of a Java program. This is achieved by instrumenting programs with equality-assert-statements and generating test suites for statement coverage on (failed) assertions. This work differs from ours as it does not support multiple test cases finding differences between program versions, and also does not take multiple prior versions into account. Additionally, they do not take TSR into account to increase efficiency of regression testing.

The goal of mutation testing is to measure effectiveness of test suites or test-generation techniques, respectively, by deriving from an original program a set of syntactically slightly changed mutants (simulating faults) [58]. A test case detects a mutant if its test-execution results for the original program differ from those for the mutant. Based on this principle, [59] pursue mutation-driven generation of test cases for Java programs by using genetic algorithms to find test cases that detect mutants. In contrast, [60] use a combination of symbolic execution and a search-based heuristic to identify test cases that are likely to reveal mutants. [61] also propose a heuristic for generating strong mutation-detecting test cases using hill climbing. As these works are mainly based on heuristics, the generated test cases do not guarantee to traverse/reveal program modifications. Additionally, those approaches do not allow to configure the number of test cases or the number of different programs as our parameters NPR and NRT, nor do they consider TSR.

Automatically generating test cases for differentiating two program versions for regression testing has been initially proposed by [62]. Their white-box testing tool automatically compares output values of two given program versions to derive input values leading to different outputs. Their approach is applicable to Pascal programs only and does not support multiple test cases and/or program versions as in our work.

### 5.3 Regression Verification

The goal of regression verification is to analyze different versions of a system or program to check whether the specification is still fulfilled after modifications. To this end, intermediate verification results are re-used between versions to increase efficiency [63,65]. For instance, intermediate results (so-called abstraction precisions) of verification runs enable reuse for later version [66], and regression verification may be applied to restrict the set of program inputs manually [67] or by syntactic checks [68]. Furthermore, there is work on lifting principles of regression verification to multi-threaded programs [69] and re-checking evolving software of automated production systems [70]. Moreover, efficiency of regression verification may be improved, for instance, by applying state-space partitioning-techniques [71] and by improved encodings of reuse-information [72]. Other works in this area reuse final verification results in case of a subsequent change in the program [14,73,76]. However, none of these approaches further utilizes the information collected for regression analysis to derive modification-revealing test cases.

Similarly, conditional model checking aims at reusing results of verification runs to perform collaborative verification [77,78]. To this end, there are exchange formats for verification witnesses for property violations [79] as well as for correctness proofs [80]. Furthermore, there is work on keeping track of unverified parts of a program to apply test-case generation for these parts [81,82] and reusing verification results for hybrid systems [83]. However, none of these approaches aim at finding differences between versions of the same program as done in our approach.
6 Conclusion and Future Work

We presented a configurable regression-testing methodology for automating the selection of regression test-cases with a particular focus on revealing regression bugs in evolving programs. REGRETS currently supports regression testing of C programs at unit level. Our experimental results show that effectiveness and the efficiency highly depend on the selected regression test-case generation strategy, where the parameter RTC (i.e., either generating modification-traversing or modification-revealing test cases) has the strongest impact on the effectiveness and efficiency. To conclude, our experimental results show that for obtaining the best efficiency in terms of CPU time and the best trade-off modification traversing test cases should be used without test-suite reduction. However, for the best effectiveness, modification revealing test cases should be used with multiple test cases per test goal and multiple previous revisions taken into account. Additionally, test-suite reduction obstructs effectiveness, therefore, for optimal effectiveness test-suite reduction should be disabled.

As a future work, we plan to extend our approach. First, we plan to further improve our test-generation technique to support additional subject systems as well as in utilizing alternative test-case generation techniques within our methodology (e.g., symbolic execution) and to compare the outcome with our current results. Furthermore, we plan to investigate other kinds of regression errors and to identify suitable regression strategies for effectively revealing those bugs. Finally, we plan to adapt REGRETS to other testing scenarios including, for instance, other input languages besides C and other testing levels beyond unit testing which enables us to conduct experiments on a richer set of subject systems.

Acknowledgments This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

References

[1] Paul Ammann and Jeff Offutt. Introduction to Software Testing. Cambridge University Press, 2016. ISBN: 978-1-107-17201-2.
[2] Shin Yoo and Mark Harman. “Regression Testing Minimization, Selection and Prioritization: A Survey”. In: Softw. Test. Verif. Reliab. 22.2 (2012), pp. 67–120. ISSN: 0960-0833. DOI: [10.1002/stv.430](https://doi.org/10.1002/stv.430).
[3] Emelie Engström, Mats Skoglund, and Per Runeson. “Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review”. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ACM, 2008, pp. 22–31. ISBN: 978-1-59593-971-5. DOI: [10.1145/1414004.1414011](https://doi.org/10.1145/1414004.1414011).
[4] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. “A Methodology for Controlling the Size of a Test Suite”. In: ACM Trans. Softw. Eng. Methodol. 2.3 (1993), pp. 270–285. ISSN: 1049-331X. DOI: [10.1145/152388.152391](https://doi.org/10.1145/152388.152391).
[5] Alireza Khalilian and Saeed Parsa. “Bi-criteria Test Suite Reduction by Cluster Analysis of Execution Profiles”. In: Advances in Software Engineering Techniques. Springer Berlin Heidelberg, 2012, pp. 243–256. ISBN: 978-3-642-28038-2.
[6] Saswat Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: Journal of Systems and Software 86.8 (2013), pp. 1978–2001. ISSN: 0164-1212. DOI: [https://doi.org/10.1016/j.jss.2013.02.061](https://doi.org/10.1016/j.jss.2013.02.061).
[7] Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. “The Relation of Test-Related Factors to Software Quality: A Case Study on Apache Systems”. In: Empirical Software Engineering 26.2 (2021). DOI: [10.1007/s10664-020-09891-y](https://doi.org/10.1007/s10664-020-09891-y).
[8] Laura Inozemtseva and Reid Holmes. “Coverage is Not Strongly Correlated with Test Suite Effectiveness”. In: Proceedings of the 36th International Conference on Software Engineering. Association for Computing Machinery, 2014, pp. 435–445. ISBN: 9781450327565. DOI: 10.1145/2568225.2568271.

[9] Jung-Min Kim and A. Porter. “A history-based test prioritization technique for regression testing in resource constrained environments”. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. Association for Computing Machinery, 2002, pp. 119–129. ISBN: 158113472X. DOI: 10.1109/ICSE.2002.1007961.

[10] Emilio Cruciani, Breno Miranda, Roberto Veredicchia, and Antonia Bertolino. “Scalable Approaches for Test Suite Reduction”. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE Press, 2019, pp. 419–429. DOI: 10.1109/ICSE.2019.00055.

[11] Dirk Beyer and Thomas Lemberger. “TestCov: Robust Test-Suite Execution and Coverage Measurement”. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2019, pp. 1074–1077. ISBN: 9781728125084. DOI: 10.1109/ASE.2019.00105.

[12] Zhenyu Chen, Xiaofang Zhang, and Baowen Xu. “A Degraded ILP Approach for Test Suite Reduction.” In: Jan. 2008, pp. 494–499.

[13] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. “Generating Tests from Counterexamples”. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, 2004, pp. 326–335. ISBN: 978-0769521633. DOI: 10.5555/998675.999437.

[14] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. “Information Reuse for Multi-goal Reachability Analyses”. In: ESOP '13. Vol. 7792. Springer, 2013, pp. 472–491. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_26.

[15] James H. Andrews, Lionel C. Briand, and Yvan Labiche. “Is mutation an appropriate tool for testing experiments? [software testing]”. In: Proceedings. 27th International Conference on Software Testing, Verification and Validation Workshops. IEEE, 2018, pp. 40–46. DOI: 10.1109/ICSTW.2018.00026.

[16] Marcel Böhme and Abhik Roychoudhury. “CoREBench: Studying Complexity of Regression Errors”. In: Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2014, pp. 105–115.

[17] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. “Supporting Controlled Experimentation with Testing Techniques”. In: Empirical Software Engineering: An International Journal 10.4 (2005), pp. 405–435.

[18] Tsong Yueh Chen and Man Fai Lau. “Dividing Strategies for the Optimization of a Test Suite”. In: Inf. Process. Lett. 60.3 (1996), pp. 135–141. ISSN: 0020-0190. DOI: 10.1016/0020-0190(96)00135-4.
[22] Joseph R. Horgan and Saul London. “A data flow coverage testing tool for C”. In: [1992] Proceedings of the Second Symposium on Assessment of Quality Software Development Tools. IEEE, 1992, pp. 2–10. DOI: 10.1109/AQSDT.1992.205829.

[23] A. Jefferson Offutt, Jie Pan, and Jeffrey M. Voas. “Procedures for reducing the size of coverage-based test sets”. In: Twelfth International Conference on Testing Computer Software. 1995, pp. 111–123.

[24] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. “Evaluating Test-Suite Reduction in Real Software Evolution”. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, 2018, pp. 84–94. ISBN: 9781450356992. DOI: 10.1145/3213846.3213875.

[25] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. “How Do Assertions Impact Coverage-Based Test-Suite Reduction?” In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2017. DOI: 10.1109/icst.2017.45.

[26] August Shi, Tiffany Yung, Alex Gyori, and Darko Marinov. “Comparing and Combining Test-Suite Reduction and Regression Test Selection”. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. Association for Computing Machinery, 2015, pp. 237–247. ISBN: 9781450336758. DOI: 10.1145/2786805.2786878.

[27] J. Hartmann and David J. Robson. “Revalidation during the software maintenance phase”. In: Proceedings. Conference on Software Maintenance 1989. IEEE, 1989, pp. 70–80. DOI: 10.1109/ICSM.1989.65195.

[28] J. Hartmann and David J. Robson. “RETEST-development of a selective revalidation prototype environment for use in software maintenance”. In: Twenty-Third Annual Hawaii International Conference on System Sciences. Vol. 2. IEEE, 1990, 92–101 vol.2. DOI: 10.1109/HICSS.1990.205179.

[29] J. Hartmann and David J. Robson. “Techniques for selective revalidation”. In: IEEE Software 7.1 (1990), pp. 31–36. ISSN: 0740-7459. DOI: 10.1109/52.43047.

[30] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. “An approach to regression testing using slicing”. In: Proceedings Conference on Software Maintenance 1992. IEEE Computer Society, 1992, pp. 299–308. DOI: 10.1109/ICSM.1992.242531.

[31] Mary Jean Harrold and M. L. Souffia. “An incremental approach to unit testing during maintenance”. In: Proceedings. Conference on Software Maintenance, 1988. IEEE, 1988, pp. 362–367. DOI: 10.1109/ICSM.1988.10188.

[32] Mary Jean Harrold and Mary Lou Soffa. “Interprocedural Data Flow Testing”. In: Proceedings of the ACM SIGSOFT ’89 Third Symposium on Software Testing, Analysis, and Verification. ACM, 1989, pp. 158–167. ISBN: 0-89791-342-6. DOI: 10.1145/75308.75327.

[33] Abu-Bakr Taha, Stephen M. Thebault, and Sying-Syang Liu. “An approach to software fault localization and revalidation based on incremental data flow analysis”. In: [1989] Proceedings of the Thirteenth Annual International Computer Software Applications Conference. IEEE, 1989, pp. 527–534. DOI: 10.1109/CMSAC.1989.65142.

[34] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. “Towards Refactoring-aware Regression Test Selection”. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018, pp. 233–244. ISBN: 978-1-4503-5638-1. DOI: 10.1145/3180155.3180254.
[35] Dusica Marijan and Marius Liaaen. “Practical Selective Regression Testing with Effective Redundancy in Interleaved Tests”. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. Association for Computing Machinery, 2018, 153–162. ISBN: 9781450356596. DOI: 10.1145/3183519.3183532

[36] Ankur Choudhary, Arun Prakash Agrawal, and Arvinder Kaur. “An Effective Approach for Regression Test Case Selection Using Pareto Based Multi-objective Harmony Search”. In: Proceedings of the 11th International Workshop on Search-Based Software Testing. ACM, 2018, pp. 13–20. ISBN: 978-1-4503-5741-8. DOI: 10.1145/3194718.3194722

[37] Weichen E. Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. “Effect of Test Set Minimization on Fault Detection Effectiveness”. In: 1995 17th International Conference on Software Engineering. ACM, 1995, pp. 41–41. DOI: 10.1145/225014.225018

[38] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. “Test case prioritization: an empirical study”. In: Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). Software Maintenance for Business Change’ (Cat. No.99CB36360). IEEE, 1999, pp. 179–188. DOI: 10.1109/ICSM.1999.792604

[39] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary J. Harrold. “Prioritizing test cases for regression testing”. In: IEEE Transactions on Software Engineering 27.10 (2001), pp. 929–948. ISSN: 0098-5589. DOI: 10.1109/32.962562

[40] Sebastian Elbaum, David Gable, and Gregg Rothermel. “Understanding and measuring the sources of variation in the prioritization of regression test suites”. In: Proceedings Seventh International Software Metrics Symposium. IEEE Computer Society, 2001, pp. 169–179. DOI: 10.1109/METRIC.2001.915525

[41] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. “Incorporating varying test costs and fault severities into test case prioritization”. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001. IEEE Computer Society, 2001, pp. 329–338. DOI: 10.1109/ICSE.2001.919106

[42] Alexey Malishevsky, Gregg Rothermel, and Sebastian Elbaum. “Modeling the cost-benefits tradeoffs for regression testing techniques”. In: International Conference on Software Maintenance, 2002. Proceedings. IEEE, 2002, pp. 204–213. DOI: 10.1109/ICSM.2002.1167767

[43] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia. “The impact of test suite granularity on the cost-effectiveness of regression testing”. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. ACM, 2002, pp. 130–140. DOI: 10.1145/581356.581358

[44] Xiaolin Wang and Hongwei Zeng. “History-Based Dynamic Test Case Prioritization for Requirement Properties in Regression Testing”. In: 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED). IEEE, 2016, pp. 41–47. DOI: 10.1109/CSED..2016.016

[45] Jun Li, Bodong Zhao, and Chao Zhang. “Fuzzing: a survey”. In: Cybersecurity 1.1 (2018). DOI: 10.1186/s42400-018-0002-y

[46] Sanjay Rawat, Vivek Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. “VUzzer: Application-aware Evolutionary Fuzzing”. In: Proceedings 2017 Network and Distributed System Security Symposium. Internet Society, 2017. DOI: 10.14722/ndss.2017.23404
[47] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. “Grammar-Based Whitebox Fuzzing”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2008, 206–215. isbn: 9781595938602. doi: [10.1145/1375581.1375607](https://doi.org/10.1145/1375581.1375607)

[48] Michał Zalewski. *Technical "whitepaper" for afl-fuzz*. Tech. rep. 2006. url: [https://lcamtuf.coredump.cx/afl/technical_details.txt](https://lcamtuf.coredump.cx/afl/technical_details.txt)

[49] Patrice Godefroid, Michael Y. Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: *Network Distributed Security Symposium (NDSS)*. 2008.

[50] Thomas Lemberger. “Plain random test generation with PRTest”. In: *International Journal on Software Tools for Technology Transfer* (2020). doi: [10.1007/s10009-020-00568-x](https://doi.org/10.1007/s10009-020-00568-x)

[51] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. Vol. 8. USENIX Association, 2008, 209–224.

[52] Mikhail R. Gadelha, Rafael Menezes, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis Nicole. “ESBMC: Scalable and Precise Test Generation based on the Floating-Point Theory”. In: *Fundamental Approaches to Software Engineering*. Springer International Publishing, 2020, pp. 525–529. isbn: 978-3-030-45234-6. doi: [10.1007/978-3-030-45234-6_27](https://doi.org/10.1007/978-3-030-45234-6_27)

[53] Marie-Christine Jakobs. “CoVeriTest with Dynamic Partitioning of the Iteration Time Limit (Competition Contribution)”. In: *Fundamental Approaches to Software Engineering*. Vol. 12076. Springer International Publishing, 2020, pp. 540–544. isbn: 978-3-030-45234-6. doi: [10.1007/978-3-030-45234-6_30](https://doi.org/10.1007/978-3-030-45234-6_30)

[54] William M. McKeeman. “Differential Testing for Software”. In: *Digital Technical Journal* 10 (1998), pp. 100–107.

[55] Robert B. Evans and Alberto Savoia. “Differential Testing: A New Approach to Change Detection”. In: *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. ACM, 2007, pp. 549–552. isbn: 978-1-59593-812-1. doi: [10.1145/1295014.1295038](https://doi.org/10.1145/1295014.1295038)

[56] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, pp. 283–294. isbn: 978-1-4503-0663-8. doi: [10.1145/1993498.1993532](https://doi.org/10.1145/1993498.1993532)

[57] Kunal Taneja and Tao Xie. “DiffGen: Automated Regression Unit-Test Generation”. In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 407–410. doi: [10.1109/ASE.2008.60](https://doi.org/10.1109/ASE.2008.60)

[58] Yue Jia and Mark Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. issn: 0098-5589. doi: [10.1109/TSE.2010.62](https://doi.org/10.1109/TSE.2010.62)

[59] Gordon Fraser and Andreas Zeller. “Mutation-driven Generation of Unit Tests and Oracles”. In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ACM, 2010, pp. 147–158. isbn: 978-1-60558-823-0. doi: [10.1145/1831708.1831728](https://doi.org/10.1145/1831708.1831728)

[60] Mark Harman, Yue Jia, and William B. Langdon. “Strong Higher Order Mutation-based Test Data Generation”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ACM, 2011, pp. 212–222. isbn: 978-1-4503-0443-6. doi: [10.1145/2025113.2025144](https://doi.org/10.1145/2025113.2025144)
[61] Francisco Carlos M. Souza, Mike Papadakis, Yves Le Traon, and Márcio E. Delamaro. “Strong Mutation-based Test Data Generation Using Hill Climbing”. In: Proceedings of the 9th International Workshop on Search-Based Software Testing. ACM, 2016, pp. 45–54. ISBN: 978-1-4503-4166-0. DOI: 10.1145/2897010.2897012.

[62] Bogdan Korel and Ali M. Al-Yami. “Automated Regression Test Generation”. In: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 1998, pp. 143–152. ISBN: 0-89791-971-8. DOI: 10.1145/271771.271803.

[63] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. “Efficient regression verification”. In: WODES ’96. IEEE, 1996, pp. 157–150.

[64] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. “Extreme Model Checking”. In: Verification: Theory and Practice. Vol. 2772. LNCS. Springer, 2003, pp. 332–358. ISBN: 978-3-540-39910-0. DOI: 10.1007/978-3-540-39910-0_16.

[65] Ofer Strichman and Benny Godlin. “Regression Verification - A Practical Way to Verify Programs”. In: VSTTE ’05. Vol. 4171. LNCS. Springer, 2008, pp. 496–501. ISBN: 978-3-540-69149-5. DOI: 10.1007/978-3-540-69149-5_54.

[66] Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. “Precision Reuse for Efficient Regression Verification”. In: ESEC/FSE ’13. ACM, 2013, pp. 389–399. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491429.

[67] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhiik Roychoudhury. “Partition-based Regression Verification”. In: ICSE ’13. IEEE Press, 2013, pp. 302–311. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6737015.

[68] Benny Godlin and Ofer Strichman. “Regression verification: proving the equivalence of similar programs”. In: Software Testing, Verification and Reliability 23.3 (2013), pp. 241–258. DOI: 10.1020/stvr.1472.

[69] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. “Regression Verification for Multi-threaded Programs”. In: VMCAI ’12. Vol. 7148. Springer, 2012, pp. 119–135. ISBN: 978-3-642-27940-9. DOI: 10.1007/978-3-642-27940-9_9.

[70] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. “Regression Verification for Programmable Logic Controller Software”. In: ICFEM ’15. Vol. 9407. Springer International Publishing, 2015, pp. 234–251. ISBN: 978-3-319-25423-4. DOI: 10.1007/978-3-319-25423-4_15.

[71] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. “Regression Verification Using Impact Summaries”. In: SPIN ’13. Vol. 7976. Springer, 2013, pp. 99–116. ISBN: 978-3-642-39176-7. DOI: 10.1007/978-3-642-39176-7_7.

[72] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. “Automating Regression Verification”. In: ASE ’14. ACM, 2014, pp. 349–360. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642987.

[73] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. “Green: Reducing, Reusing and Recycling Constraints in Program Analysis”. In: FSE ’12. ACM, 2012, pp. 58:1–58:11. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393566.2393665.

[74] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. “Incremental Upgrade Checking by Means of Interpolation-based Function Summaries”. In: FMCAD ’12. IEEE, 2012, pp. 114–121.

[75] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. “Regression Model Checking”. In: ICSM ’19. IEEE, 2009, pp. 115–124. DOI: 10.1109/ICSM.2009.5306334.
[76] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. “Incremental State-Space Exploration for Programs with Dynamically Allocated Data”. In: *ICSE ’08*. ACM, 2008, pp. 291–300. DOI: [10.1145/1368088.1368128].

[77] Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. “Conditional Model Checking: A Technique to Pass Information between Verifiers”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2012. ISBN: 9781450316149. DOI: [10.1145/2393596.2393664].

[78] Maria Christakis, Peter Müller, and Valentin Wüstholz. “Collaborative Verification and Testing with Explicit Assumptions”. In: *FM ’12*. Vol. 7436. Springer, 2012, pp. 132–146. ISBN: 978-3-642-32759-9. DOI: [10.1007/978-3-642-32759-9_13]

[79] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. “Witness Validation and Stepwise Testification Across Software Verifiers”. In: *ESEC/FSE ’15*. ACM, 2015, pp. 721–733. ISBN: 978-1-4503-3675-8. DOI: [10.1145/2796805.2796867]

[80] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. “Correctness Witnesses: Exchanging Verification Results Between Verifiers”. In: *FSE ’16*. ACM, 2016, pp. 326–337. ISBN: 978-1-4503-4218-6. DOI: [10.1145/2950290.2950351]

[81] Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. “Just Test What You Cannot Verify!” In: *FASE ’15*. Vol. 9033. Springer, 2015, pp. 100–114. ISBN: 978-3-662-46675-9. DOI: [10.1007/978-3-662-46675-9_7]

[82] Maria Christakis, Peter Müller, and Valentin Wüstholz. “Guiding Dynamic Symbolic Execution Toward Unverified Program Executions”. In: *ICSE ’16*. ACM, 2016, pp. 144–155. ISBN: 978-1-4503-3900-1. DOI: [10.1145/2884781.2884843]

[83] Stefan Mitsch, Grant Olney Passmore, and André Platzer. “Collaborative Verification-Driven Engineering of Hybrid Systems”. In: *Mathematics in Computer Science* 8.1 (2014), pp. 71–97. ISSN: 1661-8289. DOI: [10.1007/s11786-014-0176-y].