Constructing Regression Dataset from Code Evolution History

Xuezhi Song  
Fudan University  
China  
songxuezhi@fudan.edu.cn

Yun Lin*  
National University of Singapore  
Singapore  
dcslyn@nus.edu.sg

Siang Hwee Ng  
National University of Singapore  
Singapore  
sianghwee@u.nus.edu

Ping Yu  
Fudan University  
China  
20110240078@fudan.edu.cn

Xuezhi Song, Yun Lin*, Siang Hwee Ng, Ping Yu, Xin Peng, and Jin Song Dong. 2021. Constructing Regression Dataset from Code Evolution History. In Proceedings of The 44th International Conference on Software Engineering (ICSE 2022). ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Bug repositories are fundamental infrastructure to support various software engineering research such as software testing [10, 24, 25, 42, 44], fault localization [14, 22, 34, 39], and bug repair [9, 18, 36, 47, 55]. The SE/PL community has taken decades to construct bug dataset such as SIR [17], BugBench [41], CodeFails [56], and QuixBugs [38], and finally gravitate to the state-of-the-art repository, Defects4j [31], taking seven years to collect over 800 bugs.

While bug repositories like Defects4j [31] and CoREBench [8] have made significant contribution to the community, there are still concerns that their sizes are relatively small (and, thus arguably less representative [53]), regarding (1) the diversified bugs in practice (e.g., project-independent, concurrent, IO-related, API misuse, etc.) and (2) the needs of applying the emerging data-driven and AI techniques in SE tasks [21, 28, 40, 51, 52]. However, comparing to the cost of labelling an image or a sentence in the AI community, manually collecting and labelling a bug is far more expensive. It requires to (1) prepare at least two versions of code project (i.e., a correct version and a buggy version), (2) set up at least one test cases passing the correct version and failing the buggy version, and (3) isolate an environment where the bug can be well replicated.

Existing bug repositories [8, 17, 31, 38, 41, 56] are usually constructed manually, which naturally limits its scalability. A recent work, BugBuilder [29], is proposed to isolate bug-fixing changes from a bug-fixing revision based on a test case passing the bug-fixing revision and failing the revision before. Applied on many code repositories, BugBuilder can generate a bug repository similar to Defects4j, but with a much larger size. In such a bug repository, the buggy code are labelled with their fixes, potentially a useful dataset facilitating various data-driven bug-related learning tasks.

In this work, we further propose an approach, RegMiner, to harvest regression bugs from the code repository. Regression bugs are bugs which manifest their existence from a working function fail. Comparing to a large non-regression bug repository, an ultra-large regression repository has the following two benefits.

From the perspective of the support for learning tasks, regression dataset suffers less from the specification-missing problem,
Figure 1: The distribution of the number of regressions used in 15 regression localization work from 1999 to 2021

Comparing to a normal bug dataset, software bugs are essentially the inconsistencies between the implementation and its specification. Thus, a bug-related solution such as debugging and repair can be less reliable without sufficient specification as the input. Nevertheless, normal bug datasets, however large they are, embed very limited specification information. Note that, they only have their buggy code annotated with the fixes and there are limited number of test cases prepared for each normal bug. Therefore, it is arguably effective to design a bug-related learning task (e.g., fix prediction) from even a very large dataset. In contrast, regression bugs are equipped with their past passing version as a much stronger referenced specification, providing a more promising motivation for their data-driven solutions for tasks like debugging and repair.

From the perspective of benchmark construction, a large regression dataset lays foundation for various regression analysis. Existing regression analysis work (e.g., regression localization) are usually evaluated with limited number of regressions, given that their collection is highly laborious. Figure 1 shows the distribution of the number of real-world regressions used in 15 research works from the year of 1999 to 2021 [6, 11–13, 15, 32, 49, 55, 59–64, 66]. Overall, the mean number of regressions is 12.9, and the median is 10. Even worse, different benchmark are used in different work, making it difficult to evaluate their performance. A large-scale benchmark can not only mitigate this issue, but support various systematic empirical studies on regressions as well.

In this work, we design RegMiner to automate the regression harvesting process with zero human intervention. RegMiner can continuously harvest a large number of replicable regression bugs from code repositories (e.g., from Github). Technically, we address an information retrieval problem in the context of software evolutionary history, i.e., retrieving runnable regression on a code repository. Our approach takes a set of code repositories (e.g., Git) as input, and isolates their regressions with its running/replicable environment as output. Intuitively, we construct a regression by searching in code repository for a regression-fixing commit denoted as rfc, a regression-inducing commit denoted as ric, a working commit denoted as wc, and a test case denoted as test so that test can pass rfc and wc, and fail ric. To this end, we first design a measurement to select those bug-fixing commits with more regression potential. Next, for each such potential regression-fixing commit, rfc, we further search for a test case test which can pass rfc, fail the commit before (denoted as rfc − 1), and pass a commit before rfc − 1. Our search process addresses the technical challenges of (1) identifying relevant code changes in rfc to migrate through the code evolution history, (2) adopting the library upgrades with the history, and (3) minimizing the compilation overhead and handling incompilable revisions.

We evaluate our approach with a close-world and an open-world experiment. In the close-world experiment, we manually collect a small dataset consisting of 18 regressions. Each regression are manually verified real-world regressions with its regression-fixing, regression-inducing, and working commits specified. RegMiner recovers 61.1% regressions on the benchmark. In the open-world experiment, we run RegMiner on 66 code repositories within 3 weeks. RegMiner reports 537 regressions which constructs the largest Java regression dataset to the best of our knowledge. Moreover, we further conduct an empirical study on our regression dataset, which reveals (1) a gap between regression localization/fix and theoretically perfect delta-debugging technique; (2) a number of research opportunities on regression analysis (see Section 6).

In conclusion, we summarize our contributions as follows:

- We propose a fully automated regression mining technique, which allows us to continuously harvest regressions from a set of code repositories with zero human intervention.
- We build our RegMiner tool with extensive experiments evaluating its completeness to mine regressions. The results show that RegMiner is accurate, effective, and efficient to mine regressions from code repositories.
- We construct a regression dataset with RegMiner within 3 weeks. We foresee that the size of our regression dataset can keep growing with time.
- We conduct an empirical study on the regression dataset, showing the limitation of delta-debugging techniques for regression analysis, and revealing a number of research opportunities which can be facilitated by data-driven solutions in the future.

2 PROBLEM DEFINITION

We first clarify the denotation and terminologies in this paper, followed by the problem definition.

Commit and Revision. A code commit can correspond to two revisions in the code repository, i.e., the revision before and after (or caused by) the commit. In this work, we use the terminology commit and its caused revision interchangeably. Thus, given a commit denoted by c, we also use c to denote the revision caused by c. Moreover, we use c − 1 to denote the commit before c and the revision caused by the commit c − 1.

Regression. Given a fixed regression in the code repository, we denote it as reg = ⟨rfc, ric, test⟩, which indicates that a mined regression consists of a regression-fixing commit rfc, a regression-inducing commit ric, and a test case test. Moreover, rfc, ric, and test satisfy that test passes rfc and ric − 1, and fails rfc − 1 and ric. In addition, we also call the commit ric − 1 as the working commit.

Problem Definition. Formally, given a code repository C as the set of commits, we aim to maximize the size of regression set REG = {reg | reg = ⟨rfc, ric, test⟩} where ∀reg ∈ REG, ∀test, s.t. rfc ∈ C, ric ∈ C, rfc > ric, test passes rfc and ric − 1, and fails rfc − 1 and ric. Here, > indicates the chronological partial order between two
Algorithm 1: Regression Dataset Construction

Input: A code repository set, repositories
Output: A regression dataset, regressions; a threshold of regression potential, \( t_{\text{rep}} \)

// initialize the regression set
regressions = \( \emptyset \)

for repo \( \in \) repositories do
  // initialize the regression set
  commits = search_commits_with_test_addition(repo)
  for \( c \in \) commits do
    // fixing commit confirmation
    is_fix = confirm_fix(c.tests, c, c - 1)
    if is_fix then
      // RFP prediction
      estimate_regression_potential(c)
    else
      commits = commits \( \setminus \) \( \{c\}\)
    commits’ = rank_and_filter(commits, \( t_{\text{rep}} \))
  for \( c \in \) commits’ do
    // search regression with test migration
    ric, test = search_regression(c.tests, repo)
    if ric \( \neq \) null then
      rfc = c
      reg = (rfc, ric, test)
      regressions = regressions \( \cup \) \( \{\text{reg}\}\)
  return regressions

The retrieval process of a fixed regression reg is essentially the process of identifying rfc, ric, and test in the code commit history.

3 OVERVIEW AND CHALLENGES

Algorithm 1 shows an overview of our approach to mine and construct a regression dataset, which takes as input a set of code repositories and a threshold to predict potential regression-fixing commits. When searching for the regressions, our approach first collects the commits including the code change of test case addition/ modification from the code repositories (line 3). We assume that the test cases are created or modified for guarding the correctness of a feature or function in the project. Next, for each of such commit c, we confirm whether it is a bug-fixing commit by running the added/modified test cases against c and c - 1 (line 5). If any tests can pass c and fail c - 1, we deem c as a bug-fixing commit (line 6). Then we estimate its potential as a regression-fixing commit (line 7). With their quantified regression potential, we can rank the bug-fixing commits and remove those with potential smaller than the predefined threshold (line 10). For each such potential regression-fixing commit, we search for its regression-inducing commit ric which satisfies that one of the tests test fails in ric and passes in ric - 1 (line 12). If we can find such a regression-fixing commit, we can confirm the bug-fixing commit c as a regression-fixing commit rfc. Then, we construct a regression reg consisting of rfc, ric, and the relevant test case test. The design of Algorithm 1 needs to overcome the following three challenges.

Challenge 1: Non-regression Fixing Commits Cause the Search Futile. It is obvious to see that searching for a regression across the history is time-consuming, which includes the overhead of project compilation, test case migration, and test case execution. Moreover, a bug-fixing commit is not necessary a regression-fixing commit, and starting with a non-regression fixing commit ends up with that the whole search futile. Even worse, we observe that non-regression fixing commits are very common, which makes lots of the computation power spent futile. In this work, we design a novel measurement to quantify the potential of a bug-fixing commit to be regression-fixing commit, which allows us to save lots of efforts.

Challenge 2: Test Dependency Migration Across the History. With a test case and the potential regression-fixing commit, it is also a non-trivial task to verify its regression-inducing commit. In practice, the regression can happens year ago. The project can undergo radical changes and depend on different versions of libraries. Without appropriate adaption, we can miss reporting a large number of real regressions.

Challenge 3: Large Regression Validating Overhead. Starting from a potential regression-fixing commit, there could be thousands of commits to check out, recompile, and run the test case, which incurs huge runtime overhead. Assume that (1) a sophisticated project has over 3000 commits, and (2) checking out, compiling, running a test on a revision takes 15 seconds (which is very optimistic), verifying an individual regression-fixing commit can require 12.5 hours. Even worse, some commits can be inherently incomparable after being checked out. Therefore, how can we search through the commit history in an efficient way?

4 APPROACH

4.1 Estimating Regression Potential

Conceptually, a regression revolves around a feature. The feature fails or works if the regression is introduced or fixed. Here, we call the feature of a regression as regression feature, and the set of code elements implementing the feature as feature code. Assume that we know that (1) a function is fixed in a revision bfc, and (2) the precise set of code elements (e.g., methods) set\(_f\) of feature code, we can estimate the probability of bfc to be a regression-fixing revision as

\[
P(rfc|bfc, set_f) = 1 - (1 - p)^N\tag{1}
\]

In Equation 1, \( N \) is the number of changes applied on set\(_f\) in the commits before bfc, \( p \) is the probability that a change introduces a bug on the feature \( f \), which we call as regression-inducing probability. Intuitively, the more changes applied on set\(_f\) in the history, the more likely the bfc is a regression-fixing revision.

However, it is non-trivial to infer the precise set of program elements set\(_p\) purely based on a test case. Introducing additional noise in set\(_f\) can make non-regression fixing revisions have a large probability. In contrast, missing important elements in set\(_p\) can make real regression-fixing revision have a lower probability. Either problem can make Equation 1 misguide the prediction of regression-fixing revision. We introduce our solution as follows.
4.1 Feature Code Identification. In practice, the feature code can distribute over a set of methods, and even relevant methods can have different level of relevance. Therefore, we design an approach to (1) identify a set of relevant methods to a specific feature and (2) assign each method with a relevance score. Then, we covert the relevance of each method into its specific regression-inducing probability. Intuitively, the invoked code of a test can be an estimation of the feature code being tested. However, such an estimation can introduce noise by utility code shared by other features.

Taking the granularity of method or function, we can have a set of covered method $M$ by a test case test, in a bug-fixing commit. We design the measurement of test uniqueness and textual similarity to distinguish the relevant methods from the others. Specifically, we first parse the set of methods $M$ called by test case test, then we assign a relevance score to each $m \in M$ via

$$ m.rel = \min(1.0, test\_uniq(m, test) \times (1 + \sum(m, test))) \quad (2) $$

In Equation 2, the relevance score of a method $m$ depends on (1) its test uniqueness, i.e., how unique $m$ is to test (2) and the textual similarity between $m$ and test. We design the measurement of $test\_uniq(m)$ and $sim(m, test)$ with a range from 0 and 1. The $\min(\cdot)$ function in Equation 2 ensure that the relevance score has an upper bound of 1.0.

Test Uniqueness. To evaluate the test uniqueness of a covered method $m$ for test, we design a variant TF-IDF measurement. Assume that there are $N$ test cases in a project, and each test case calls a set of methods. Thus, we can construct a test-method matrix $R_{m,t}$ where each column represents a test case and each row represents a method called by at least one test case. Specifically, we calculate an IDF-like measurement for each $r(m, t)$ in $R_{m,t}$ as

$$ test\_uniq(m, t) = r(m, t) = \left\{ \begin{array}{ll} \log N \frac{N}{freq(m)} & \text{if } t \text{ covers } m \\ 0 & \text{otherwise} \end{array} \right. \quad (3) $$

In Equation 3, $N$ represents the number of test cases in the project, $freq(m)$ represents the number of test cases covering method $m$, which ranges from 1 to $N$. Thus, $\log N \frac{N}{freq(m)}$ is scaled to the range $[0, 1]$. Note that, we can pre-calculate $R_{m,t}$ in the latest revision of each project, and then use $r(m, t)$ for each bug-fixing commit efficiently.

Textual Similarity. Moreover, we further introduce a similarity function between test and $m$. Specifically, we tokenize the name of test method into a bag of words with the word “test” removed, e.g., testCalendarTimeZoneRespected is converted into a bag as $B = \{“calendar”, “time”, “zone”, “respected”\}$. Let the number of tokens in $m$ which can match any of words in $B$, as $k$, then the $sim(test, m) = \frac{k}{|B|}$, which ranges between 0 and 1.

4.1.2 Code Element Re-identification. When identifying a set of relevant methods, we need to re-identify in the past commits. Given a method $m$ in revision $r$, if we can find a method in revision $r'$ with exact the same signature with $m$ ($r' > r$), we consider $r'$ as a match of $r$ in $r'$. However, if we cannot locate such an exact match, we track their identify by defining the following similarity-based metrics:

$$ sim(m, m') = \alpha \cdot sim\_signature(m, m') + \beta \cdot sim\_body(m, m') \quad (4) $$

Algorithm 2: Test Case Migration

Input : A bug-fixing revision, $bf\_c$; a revision under investigation, $c_{invo}$; a test case, test
Output : An adapted revision $c_{invo}'$ from $c_{invo}$

1. $exec\_code = run(test, bf\_c)$
2. $\Delta code = diff\_analysis(exec\_code, c_{invo})$
3. $\Delta code' = static\_dependency\_analysis(c_{invo}, \Delta code)$
4. $c_{invo}' = version\_adaptation(\Delta code', c_{invo})$
5. return $c_{invo}'$

Intuitively, the metrics consist of the signature similarity and the code body similarity. We consider $m$ and $m'$ as a match if there similarity is above a threshold $th_m$. Limited by the space, readers may need to check our anonymous website [1] for more details.

4.1.3 Regression Potential Metric. Consequently, given a set of methods $M$, each $m \in M$ has a relevance score denoted by $m.rel$ and a historical change number $m.changes$. Thus, we quantify the final probability of a regression-fixing commit as

$$ P(rfc|bf\_c, set_f) = 1 - \prod_{m} (1 - p \times m.rel)^{m.changes} \quad (5) $$

In Equation 5, $p$ is the basic regression-inducing probability, e.g., 0.01, shared by all the method. Then, we adjust the overall probability based on the relevance score.

4.2 Test Dependency Migration

Given a test test in a bug-fixing commit, we need to migrate test along with its code dependency in a revision $c$ under investigation. Typically, we need to overcome three changes:

- **Bug-fixing change identification**: A fixing change in the bug-fixing commit cannot be migrated as the test test cannot fail in the buggy revision.
- **Dependency identification** Each time we migrate the test case in the revision $c$, we need to ensure that test can both compile and run on $c$, with its dependencies.
- **Library version reconciliation** To further avoid potential exceptions, we need to ensure that the migrated test and its dependencies can be adapted to the old libraries.

Algorithm 2 shows our pipeline solution to address the above challenges, which takes input as a bug-fixing revision $bf\_c$, a revision under investigation $c_{invo}$ and test case test; and generates an adapted revision $c_{invo}'$ with test and its dependencies migrated. First, we run test against $bf\_c$ to get all the covered code, i.e., $exec\_code$ (line 1). By analyzing the difference between $exec\_code$ and the revision under investigation $c_{invo}$, we locate a set of code $\Delta code$ which (1) is covered by test but (2) does not exist in $c_{invo}$ (line 2). $\Delta code$ narrows down the scope of our migrated code. Next, to minimize compilation error caused by the migration, we further expand $\Delta code$ by analyzing its static dependencies in $bf\_c$, e.g., the Java interface implemented by $\Delta code$. As a result, we have $\Delta code'$ for the completeness of migrated code. Finally, we detect the difference of library version dependencies between $bf\_c$ and $c_{invo}$. Based on the library version difference, we apply our defined heuristics on $c_{invo}$ migrated with $\Delta code'$.
Figure 2 shows simplified class dependency graphs of a regression-fixing commit bfc and a commit under investigation cinv under Common Lang project [2]. This regression lasts for 1.5 years, and undergo 493 commits between the two revisions. In Figure 2, we use container box in yellow to represent class (or inner class and interface), and container box to represent class member such as method and field. The containment represent that a class/interfaced declares a method or a field. Methods are container box whose name has ‘(’ as suffix, while fields are container boxes without such suffix. The arrows between the boxes are the call relation between methods and fields. Moreover, we represent in green the methods covered by the test case testCalendarTimeZoneRespected() , test, in the bug-fixing commit, while represent in red the methods with content changed between bfc and cinv. Last, the underlying text represents test method and the italic text represents static method or field.

**Dynamic Diff Analysis.** Our diff analysis focuses on the test-depended code elements which does not exist in the commit under investigation. In this example, RegMiner have a set of methods, i.e., appendTo(), getZoneDisplay(), format(), and getPatternForStyle(). Since the former two methods exist in cinv, only the latter two methods are identified to move along the test to form the $\Delta$code = \{test, format(), getPatternForStyle()\}.

**Static Dependency Analysis.** Observing that $\Delta$code can statically depend on unexecuted code, we further extend a closure of $\Delta$code, as $\Delta$code', by analyzing the dependencies of its elements such as declaration, call, and inheritance/implementation. With the same rationale of dynamic diff analysis, we include only the code elements which does not exist in cinv. For example, the interface DatePrinter and its override method format() will be included in the closure. In contrast, the field cDateTimeInstanceCache will not be included.

**Version Adaption.** Finally, we migrate $\Delta$code' into cinv. The migration can also results in some syntax compilation errors. If it happens, we define a set of heuristic program transformation rules based on the compilation error information (see [1] for more details). In the transformation rule, we set a list of triggers such as syntax change and library version change, each of which corresponds to one AST rewriting rule on cinv. For example, we will revise the modifier of a field f into static if introducing its dependent static method (e.g., the static getPatternForStyle() method will force RegMiner to update the modifier of cDateTimeInstanceCache into static); we will remove the "Override" annotation on format() method in FastDatePrinter when finding the supported JDK version in cinv is 1.5.

4.3 **Validation Effort Minimization**

Given a bug-fixing commit, we validate it as a regression-fixing commit by searching for a regression-inducing commit $c$ in the history where the test case fails $c$ but passes $c - 1$. A naive search algorithm can be a binary search algorithm as git bisect implementation [4]. The binary search algorithm assumes each revision provides us with a feedback (such as test case pass and failure),

\[ \text{In JDK 1.5, the “Override” annotation is not allowed to describe a method implementing the interface} \]

which guide us to either approach the revision $c$ where the $c$ can fail and $c - 1$ can pass (i.e., regression), or the initial revision where $c$ just fail (i.e., non-regression bug).

However, each revision cannot always be compilable in practice. Moreover, the test code and its dependencies migrated by RegMiner can also suffer from incompatibility, especially when some scenarios are beyond our considered program transformation rule (see Section 4.2). Different from a guiding feedback, the potential compilation errors provide less guidance. Therefore, when visiting a revision with no feedback during the search, we design an approach to either (1) search for the closest revision with guiding feedback and get rid of the "no-feedback region" in the history, or (2) quickly abandon the bug-fixing commit to proceed with the next one. Note that, there are abundant code repositories, it is affordable to abandon a few regressions for regression retrieval efficiency.

**Skipping No-feedback Region.** Figure 3 shows an example where we visit a no-feedback revision when searching over the code history. Our approach estimate the no-feedback region with an exponential search algorithm as Algorithm 3. Taking a no-feedback revision $v$ and a boundary revision as input $b$ (indicating the searched boundary region cannot be beyond $(v, b)$ or $(b, v)$), Algorithm 3 aims to return a revision $b_v$ between $v$ and $b$ so that $b_v$ can either
We build RegMiner to mine Java regressions, supporting maven
repository, repo; a test case, test; boundary revision b

A boundary revision with feedback, b_r

A no-feedback revision,

Input:

We take input as a revision, a revision, a no-feedback revision, v; a code repository, repo; a test case, test; boundary revision b

Output:

A boundary revision with feedback, b_r

1 direction = v > b
2 cursor = prev = v
3 b_r = b
4 // make sure that b_future > b Past
5 b_past, b_future = set_boundary(a, b)
6 step = 1
7 while true do
8  prev = cursor
9  cursor = move(cursor, step, direction, test)
10  if cursor can feedback and prev cannot feedback then
11    b_r = update_best(b_r, cursor)
12    direction = ¬direction
13  if cursor can feedback and prev can feedback then
14    return b_r
15  if cursor is out of boundary (b_past, b_future) then
16    return b_r
17  else if cursor cannot feedback and prev cannot feedback then
18    if cursor is out of boundary (b_past, b_future) then
19      return b_r
20  else if cursor cannot feedback and prev can feedback then
21    step = 1
22    step = step * 2

(1) be a feedback revision closest to the no-feedback revision v or
(2) the boundary revision b if no such feedback revision can be found. Specifically, each time we move on the history, we double the step size to get rid of the no-feedback region sooner possible (line 21). When we reach a feedback revision from a non-feedback revision (line 8-11) or vice versa (line 19-20), we change the traverse direction and reset the step size as 1 to fine-tune the region. The boundary revision closest to the base no-feedback revision v (i.e., the input) will be preserved (line 9 and 15). Finally, the optimal b_r will be returned if the traverse on the history is out of pre-defined scope (line 13 and 17).

Overall Algorithm. Taking input as a revision head and a revision tail where tail > head, a test case test, and the code repository repo, Algorithm 4 returns a regression-inducing revision, if exists. Algorithm 4 is designed based on binary search (line 1-7, 19). However, if we visit a no-feedback revision during the binary research, we will search for the no-feedback region supported by Algorithm 3. Based on the reported region, we either (1) skip this bug-fixing commit (line 10-11, i.e., cannot find feedback revision between head and tail), or (2) reset the binary search region to continue the search.

5 EXPERIMENT

We build RegMiner with the following research questions, more details of our experiment are available at [1].

- RQ1: Whether RegMiner can predict the regression potential of bug-fixing commits well?
- RQ2: How effective and efficient RegMiner can migrate test cases and locate the regression-inducing changes?
- RQ3: Whether RegMiner can effectively and continuously harvest regressions on open source projects?

5.1 Experiment Design

5.1.1 Benchmark Construction. In order to evaluate the effectiveness and efficiency of RegMiner to locate regressions, we manually collect 18 regression-fixing commits from Github for Java projects. The scale is comparative to the number of regressions used in various regression studies (see Figure 1). We first search for closed Github issues which uniquely mention a commit as its solution. We then filter those issues based on its label and description. We prioritize the issues with label as “regression” or “bug”, or with description with keyword of “regression”. In this study, we spent one month on manually filtering out real regressions, checking evolutionary history for regression-inducing commits, and migrating test cases. As a result, we confirmed 18 regressions from 12 Java projects by this means (see [1] for more details). We apply RegMiner on the total 18 regression fixing commits to measure the completeness and record the runtime overhead. Given different machine have the runtime overhead different, we evaluate the efficiency by the number of revisions each solution needs to visit.

5.1.2 Baseline. To the best of our knowledge, RegMiner is the first technique to retrieve runnable regressions in the code repository.
In this study, we construct baselines by disabling the function of test dependency migration and validation effort minimization separately. We have RegMiner-TDM for RegMiner with test dependency migration disabled. In RegMiner-TDM, we copy the identified test case into the target revision without dependency analysis. We have RegMiner-VEMin+bsi and RegMiner-VEMin+all for RegMiner with validation effort minimization disabled. In RegMiner-VEMin+bsi, we search for regressions by using git-bisect [4] strategy. In RegMiner-VEMin+all, we search for regressions by iteratively checking each commit from the regression-fixing commit to the initial commit. It stops either when a regression-inducing commit is found or the initial commit is reached. We compare RegMiner with RegMiner-TDM, RegMiner-VEMin, and RegMiner-VEMin+all regarding their precision and recall on the benchmark.

5.1.3 Regression Potential Evaluation and Retrieval. As for regression-fixing commit prediction, the size of ground truth regression set is too small to provide convincing results. Hence, we run RegMiner on 66 Github repositories for three weeks and collect the reported regressions for this study. Assuming the number of regression-fixing commits is $N$, we randomly select another $N$ confirmed non-regression fixing commits. Then, we let RegMiner generate regression potential metric for each fixing commit, comparing the metrics distribution of the two groups.

5.2 Results (RQ1): Regression Potential Prediction

Figure 4 shows the distribution on our estimated regression probability of two fixing-commit groups. With our mining technique, RegMiner reports 537 regressions within 3 weeks. Overall, the regression-fixing group has an average of 0.625 and median of 0.642; in contrast, the bug-fixing group has an average of 0.343 and median of 0.265. We apply unpaired two-samples Wilcoxon test on two groups and have the $p$-value smaller than $10^{-5}$. Therefore, the regression-fixing group is significantly different from the bug-fixing group. Given the regression-fixing group is a higher mean value, we conclude that our regression potential metric is effective.

Moreover, we further investigate the outlier bug-fixing commit with high regression potential. We observe that sometimes the code implementation of a feature can contribute to both regression and non-regression bugs. Once such a feature goes through many changes in the history, it makes RegMiner report its normal bug-fixing commit as a high regression potential commit. For example, in the project of FastJSON [3], we observe two test cases, testClazz and testForIssue are designed to the feature of parsing attributes of a JSON class object and reference object respectively. The former is a test created for a normal bug in commit 89f072 while the latter is that for a regression bug in commit f1ebf3. They share same code implementation of JSON parsing function, and undergo similar number of changes. In this case, RegMiner will report the bug-fixing commit as one with high regression potential. Nevertheless, from the statistical perspective, our regression potential metrics can largely filter out most bug-fixing commits and identify the most regression-fixing commits.

5.3 Results (RQ2-3): Close World Regression Retrieval

Table 1 shows that the overall regression retrieval performance. Overall, comparing to the baselines, our solutions (or technical design) proves their effectiveness. With our search condition (i.e., pass regression fixing commit, fail regression-inducing commit, and pass working commit), we verify that all the reported regressions are precise. The difference lies in the recall of each solutions. We observe that our test dependency migration technique allows us to adapt and successfully run the test on more regression-inducing commits. Comparing to bisect, our non-feedback region detection technique allows us to skip away from the non-feedback region and improve the chance to locate the regression-incurring commits. Last, iterating over all the commits can have comparative completeness, but incurs much higher overhead.

### Table 1: The overall performance of regression retrieval

| Approach     | Completeness | Overhead (#commit) |
|--------------|--------------|---------------------|
| RegMiner     | 0.61         | 25.3                |
| RegMiner-TDM | 0.27         | 21.2                |
| RegMiner-VEMin+bsi | 0.16 | 14.0              |
| RegMiner-VEMin+all | 0.61 | 79.0              |

**Why RegMiner cannot locate some regression?** Next, we investigate the scenarios where RegMiner cannot locate a regression. We observe that RegMiner has its limitation on test case adaptation, mainly suffering from project sophistication (e.g., project license requirement) and unexpected library downgrading situation. For example, in one our regression, the project sonar-cxx [5] requires that all the Java classes need to include license information in its comments. Without satisfying the requirement, its maven-license-plugin (defined in maven compilation configuration file) does not allow the project compilation. Moreover, we also observe that the library adaption sophistication between regression-fixing and regression-inducing commits are beyond our predefined AST rewriting rules. For example, the code using lambda expression in JDK1.8 cannot be adapted to a past version using JDK1.6. In this case, a code transformation technique regarding version difference
5.4 Results (RQ4): Open World Regression Retrieval

Overall, RegMiner construct a regression dataset consisting of 537 regressions over the 66 projects within 3 weeks. We recruit two students to verify that all the reported regressions are real. We construct those regressions into a replicable dataset for empirical and experimental facilities. Here, we briefly introduce its demography as follows. More detailed empirical study is reported in Section 6.

5.4.1 Life span of a regression. Figure 5 shows the details of how long a regression can last regarding the number of days and commits between regression-fixing and regression-inducing commits. Of all the 537 regressions, each regression takes on average 493.4 days (and 419.9 commits) between its introduction and fix. The longest period takes 3629 days and 3208 commits. Typically, a regression is introduced in an insidious way. Once detecting a regression, programmers may need to locate its introduction long time ago.

5.4.2 Changes made in regression-fixing and regression-inducing commits. Table 2 shows the landscape of changes in regression-fixing and regression-inducing commits. We show the change types (in terms of hunk, line, and file) and their quantities respectively. Overall, both regression-fixing and regression-fixing commits undergo radical changes. Nevertheless, comparing to regression-fixing commits, regression-inducing commits incur twice more changes (10.9 vs 37.1 regarding hunk number). We observe the minimum hunk number in regression-inducing commit is 0 (see last column in Table 2), our investigation shows that the commit only adds a binary file, which is not counted in Table 2.

Overall, RegMiner achieves the largest regression dataset within the shortest period. Moreover, the search facility shows larger potential to harvest (1) more regressions by parsing continuously growing Github repositories and (2) more diversified (or more interested) regressions by setting up different regression requirement (e.g., relevant to concurrency and network protocol, etc.).

Table 2: Types and quantities of change in regression-fixing and regression-inducing Commits, regarding hunk, line, and source files

| Change Unit & Type | Regression-Fixing | Regression-Inducing |
|--------------------|-------------------|---------------------|
|                    |        avg | min | max |         avg | min | max | min |
| hunk               | 10.9    | 6   | 2   | 288   | 37.1 | 13  | 882 | 0   |
| line deletion      | 108.4   | 42  | 2   | 3820  | 354.4| 103 | 12227| 0   |
| line addition      | 49.4    | 5   | 0   | 10350 | 141.3| 15  | 6378| 0   |
| file deletion      | 0.1     | 0   | 0   | 59    | 0.3  | 0   | 24  | 0   |
| file addition      | 1.0     | 1   | 0   | 17    | 1.8  | 0   | 111 | 0   |
| file modification  | 3.3     | 2   | 1   | 62    | 7.7  | 3   | 289 | 0   |

5.5 Threats to Validity

This experiment has two external threats. The first threat lies in that our experiment design only focus on the regression-fixing commits with tests modified, some regressions can be fixed without equiping a test case. Nevertheless, our open-world study shows that those regressions with test case addition/modification has already provided us with the largest regression dataset in the community. In the future, we will investigate test generation techniques such as EvoSuite [19] and Randoop [48] to synthesize potential regression test to further improve the regression retrieval capability. The second threat lies in that our implementation is based on Java projects, further studies on other popular programming language such as Python and C++ are still needed to generalize our findings.

6 EMPIRICAL STUDY

Based on the regression dataset we construct in Section 5, we investigate the following empirical research questions:

- **ERQ1.** What do the regression-inducing commits do?
- **ERQ2.** How likely a regression can be fixed by reverting the regression-inducing changes in practice?
- **ERQ3.** What are the follow-up potential research opportunities?

6.1 Study Design

6.1.1 Semantic Analysis. In this study, we investigate the general semantics of changes made in regression-inducing commits. Specifically, we generate rough semantic labels for each commit in three types of labels, i.e., refactoring, feature enhancement, and bug-fixing for a commit. We let a commit have multiple semantic labels.

Given a commit $c$ consisting of a set of hunks $H$. A hunk $h \in H$ is a sequence of added or deleted lines, denoted as $h = (Add, Del)$ where $Add$ represents the set of added lines and $Del$ represents the set of deleted lines. Moreover, we use the function $\text{code}(.)$ to have the code body of either part of a hunk. For example, $\text{code}(h.Add)$ represents the code body of the added code in hunk $h$. Also, we use $\text{len}(.)$ to represent the length of the code body. We attach the semantic label with the following rules:

- **Refactoring:** We regard a commit to have a refactoring label if $\exists h_i \in H, h_j \in H (h_i \neq h_j)$, so that
  (1) Either $\text{code}(h_j.Add)$ is similar to $\text{code}(h_j.Del)$ or vice versa, and
  (2) $|\text{code}(h_i.Add)|$ (or $|\text{code}(h_i.Del)|$) should be larger than a threshold $th_{len}$. 

Figure 5: Time span for a regression to be fixed.

is needed. While the complication goes beyond the focus of this work. We will pursue this direction in the future.
Intuitively, it means that a piece of code is moved to another location in the project. Without losing generality, we use \( \text{code}(h_1, \text{Add}) \) and \( \text{code}(h_j, \text{Del}) \) to illustrate similarity calculation. Specifically, the similarity is calculated as:

\[
\frac{2 \times \text{LCS}(\text{code}(h_1, \text{Add}), \text{code}(h_j, \text{Del}))}{\min(\text{code}(h_1, \text{Add}), \text{code}(h_j, \text{Del}))}
\]

where \( \text{LCS}(\_, \_) \) is the longest common subsequence between two pieces of code. If the similarity is above a threshold \( t_{\text{sim}} \), we consider a refactoring happens.

- **Feature Enhancement:** We regard a commit to have a feature enhancement label if
  (1) \( \exists h_i \in H \), so that
    (a) Either \( |\text{code}(h_i, \text{Add})| = 0 \) or \( |\text{code}(h_i, \text{Add})| > th_r \). Here \( th_r \)
        is a user-defined threshold to measure the size difference
        between added and deleted code, and
    (b) \( \text{code}(h_i, \text{Add}) \) is larger than the threshold \( th_{l_{\text{len}}} \), and
    (c) \( \forall h_j \in H \) so that \( \text{code}(h_i, \text{Add}) \) is similar to \( \text{code}(h_j, \text{Del}) \).

or,

\[
\sum_i |\text{code}(h_i, \text{Add})| \geq th_{ar} \quad \text{and} \quad \sum_j |\text{code}(h_j, \text{Add})| \text{ should be}
\]

larger than the threshold \( th_{l_{\text{len}}} \). Here, \( h_i \) and \( h_j \) should not
be refactoring hunks. Overall, the ratio of the length of accumulated addition over that of accumulated deletion should be larger than a threshold \( th_{ar} \).

Intuitively, we require that we have a hunk which includes a large number of non-refactoring additions.

- **Bug Fixing:** We regard a commit to have a bug fixing label if the message of \( e \) include keywords as “fix” and “bug”. Typically, we cannot infer the bug fixing semantics without any specifications. With limited information, we use commit message as the only reference to infer the bug fixing label.

6.1.2 Revert Analysis. Reverting the change back to the past, is the fundamental rationale of the most influential regression debugging technique – delta debugging [11, 33, 46, 58, 64]. In this study, we investigate how far we can infer a regression fix, assuming that delta-debugging can successfully recommend a failure-inducing change. Given a regression-fixing commit has change set \( H_F \) and its corresponding regression-inducing commit has change set \( H_I \), we first define the notation \( \text{same}\_\text{location}(h_i, h_j) \) (where \( h_i \in H_F \) and \( h_j \in H_I \)). Specifically, \( \text{same}\_\text{location}(h_i, h_j) \) return true if (1) the method \( m_i \) where \( h_i \) happens in the regression-fixing revision can be re-identified to the method \( m_j \) where \( h_j \) happens in the regression-inducing revision, and (2) the relative location of \( h_i \) in \( m_i \) can overlap with that of \( h_j \) in \( m_j \). Readers can refer to Section 4.1.2 for how we re-identify code elements across the history. Then, we conduct the following analysis for our empirical questions:

- **Revert Analysis:** We consider the fix of a regression adopt change revert if \( \exists h_i \in H_F, h_j \in H_I \) so that
  (1) either \( \text{code}(h_i, \text{Add}) \) is similar to \( \text{code}(h_j, \text{Del}) \) or \( \text{code}(h_i, \text{Del}) \) is similar to \( \text{code}(h_j, \text{Add}) \), and
  (2) \( \text{same}\_\text{location}(h_i, h_j) \) is true.

- **Estimated Distance to Fix:** Given \( h_i \in H_F, h_j \in H_I \), we categorize the distance of regression fix change from the regression-inducing change \( cat_d(h_i, h_j) \) as:
  (1) **Level 0**: the method where \( h_i \) happens can be re-identified to the method where \( h_j \) happens, \( \text{code}(h_i) \) relatively overlaps with \( \text{code}(h_j) \).

Figure 6: Semantic labels of regression-inducing commits

(2) **Level 1**: the method where \( h_i \) happens can be re-identified to the method where \( h_j \) happens, but \( \text{code}(h_i) \) does not relatively overlap with \( \text{code}(h_j) \).

(3) **Level 2**: the file where \( h_i \) happens can be re-identified to the file where \( h_j \) happens.

(4) **Level 3**: the file where \( h_i \) happens cannot be re-identified to the file where \( h_j \) happens.

Here, we measure the minimum \( cat_d(h_i, h_j) \) for each regression, indicating the most optimistic estimation on how close a fix is to the regression-inducing change.

Note that, we do not run a specific delta-debugging algorithm. Instead, we assume the theoretically best performance of delta-debugging to return the failure-inducing change, and observe how it can leads to a final fix solution.

6.1.3 Runtime Configuration. In this study, we choose the threshold of changed length for refactoring \( th_{l_{\text{len}}} = 5 \), the threshold of code similarity \( th_{\text{sim}} = 0.8 \), the threshold of the accumulated length for feature enhancement \( th_{ar} = 15 \), and the threshold of changed length for feature enhancement \( th_{l_{\text{len}}} = 10 \).

6.2 Study Results

6.2.1 Results (ERQ1): Figure 6 shows the distribution of regression-inducing commits on different semantic labels. Overall, the regression-fixing commits are almost evenly distributed in the three semantic labels (i.e., 29.1% for bug fixing, 30.9% for refactoring, and 19.7% for feature enhancement). Moreover, we cannot attach any labels on 231 regression-fixing commits. The major reason is that regression-fixing commits can involve huge number of deletions, which can hardly fit in the three categories. Thus, we regard the semantic of those commits as unknown. Moreover, an individual regression-inducing commit can serve very diverse purposes. We observe that 106 commits share at least two labels, and 9 commits share all three labels.

Not like normal bugs where a specification can be described by only a limited number of test cases, regression bugs have a working revision serving as the specification which allows us to derive abundant test cases to guard the program behaviors. Therefore, we deem that an automatic solution to decide when regression test generation is required can be practical remedy to prevent regression bugs incurred by refactoring, bug fixing, and feature enhancement.
we have rough estimation of semantic labels, which may not be 100% accurate in practice. For example, the semantic label of feature enhancement are estimated as large code additions. Some bug-fixing may also require to add much code. Second, when investigating the distance of regression fix to the regression introduction, we apply optimistic estimation. A more precise measurement is required to further the future investigation. The external threat lies in that our empirical results depend on a set of user-defined thresholds (see Section 6.1.3). We select those threshold based on our expertise and experience in academic and industry. In the future work, we will adjust more sets of thresholds and adapt more accurate solutions to generalize our conclusion.

7 RELATED WORK

7.1 Bug Dataset Construction

Bug datasets are fundamental infrastructure which lays empirical and experimental foundation for various SE/PL tasks such as software testing, debugging, fault localization, and repair. Researchers have proposed many bug datasets in the community. Do et al. [17] pioneered bug dataset construction work and contribute SIR dataset. Following their work, various datasets are constructed from programming assignments and competitions (e.g., Marmoset [54] QuixBugs [38], IntroClass [37], CodeFlaws [56], etc.), open source projects (e.g., DbgBench [9], Defects4j [31], BugsJS [23], Bugs.jar [50], etc.), and runtime continuous integration scenarios (e.g., BEARS [43] and BugSwarm [57]). The most relevant dataset is proposed by Marcel et al. [8], which construct Corebench as the largest regression dataset consisting of 70 C/C++ regressions. Those bug datasets are constructed manually, which largely impairs the scalability and representativeness of the bug dataset. Dallmeier et al. [16] make the first attempt to construct bug dataset in a semi-automatic way, via analyzing bug issues and their relevant commits. Zhao et al. [67] further propose to replicate bugs based on Android bug reports. Recently, Jiang et al. [30] propose BugBuilder to construct dataset by isolating the bug-relevant changes. RegMiner is different from these approaches in that we target for regression bug dataset, and address new technical challenges such as regression-fixing commit prediction, test dependency migration, and validation effort minimization. Within the shortest period, we construct the largest Java regression dataset.

7.2 Regression Research

Regression research includes regression fault localization [11, 26, 27, 33, 46, 58–60, 64, 65], regression testing [7, 20, 35, 45], and regression explanation [59, 60]. Zeller et al. [64] pioneers the delta debugging algorithm, which is followed by a number of variants in specific scenarios [26, 27, 33, 46]. The most recent delta-debugging variant is proposed by Wang et al. [58], as probabilistic delta debugging, to further lower the algorithm complexity while improve the accuracy. Tan et al. [55] propose a regression bug repair technique by searching over the regression revision through predefined code transformation rules. As for regression explanation, Wang et al. [59] proposes a state-of-the-art regression explanation technique by proposing a novel alignment slicing algorithm on the execution traces of the regression version and past working version. However, researchers usually either prepare their own dataset with limited size or inject mutated regressions which are less representative for the real-world regressions.

Our RegMiner solution can largely mitigate the challenges, laying foundation for the future regression analysis. Moreover, our empirical study shows that there is still a large gap between reporting failure-inducing change and regression repair solution, motivating both empirical and experimental facilities for the follow-up regression research.
8 CONCLUSION AND FUTURE WORK

In this work, we propose RegMiner which can automatically construct regression dataset. We address the challenges of predicting regression-fixing commits, migrating test and its dependencies, and minimizing the regression validation effort. Our close-world experiment shows that RegMiner achieves acceptable recall. Our open-world experiment shows that RegMiner has constructed the largest regression dataset within the shortest period.

In the future, we will further improve the regression retrieval efficiency such as synthesizing regression tests for enlarging the pool of regression-fixing commits, and designing more sophisticated migration techniques. Moreover, we will continuously harvest more regressions into our dataset, facilitating new data-driven research in the software engineering community.

REFERENCES

[1] [n. d.]. Anonymous Website. https://sites.google.com/view/regminer/home. Accessed: 2021-09-03.
[2] [n. d.]. Apache Common Lang. https://github.com/apache/commons-lang. Accessed: 2021-09-03.
[3] [n. d.]. FastJson Website. https://github.com/alibaba/fastjson. Accessed: 2021-09-03.
[4] [n. d.]. Git-bisect Website. https://git-scm.com/docs/git-bisect. Accessed: 2021-09-03.
[5] [n. d.]. SonarQube C++ Community plugin. https://github.com/SonarOpenCommunity/sonar-cxx. Accessed: 2021-09-03.
[6] Cyrille Artho. 2011. Iterative delta debugging. International Journal on Software Tools for Technology Transfer 13, 3 (2011), 223–246.
[7] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 1–12.
[8] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In Proceedings of the 2014 international symposium on software testing and analysis. 105–115.
[9] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In Proceedings of the 2017 11th joint meeting on foundations of software engineering. 117–128.
[10] Chandrasekhar Boyapati, Sarfaraz Khursid, and Darko Marinov. 2002. Korat: Automated testing based on Java predicates. ACM SIGSOFT Software Engineering Notes 27, 4 (2002), 123–133.
[11] Robert Brummayer and Armin Biere. 2009. Fuzzing and delta-debugging SMT solvers. In Proceedings of the 7th International Workshop on Satisfiability Modular Theories. 1–5.
[12] Ben Buhse, Thomas Wei, Zhiqiang Zang, Aleksandar Milicevic, and Milos Gligoric. 2019. VeDebug: regression debugging tool for Java. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 15–18.
[13] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. 2018. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 184–191.
[14] Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In Proceedings, 27th International Conference on Software Engineering, 2005. ICSE 2005. IEEE, 342–351.
[15] Dekel Cohen and Amiram Yehudai. 2015. Localization of real world regression bugs using single execution. arXiv preprint arXiv:1505.01286 (2015).
[16] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. 433–436.
[17] Hyeonsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Software Engineering 10, 4 (2005), 405–435.
[18] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2005. A genetic programming approach to automated software repair. In Proceedings of the 11th International Conference on Genetic and evolutionary computation. 947–954.
[19] Gordon Fraser and Andrea Arcuri. 2011. Evostitute: automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 416–419.
[20] Patrice Godefroid, Daniel Lehmann, and Marina Polikachuk. 2020. Differential regression testing for REST APIs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 312–323.
[21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08386 (2020).
[22] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. 2005. Locating faulty code using failure-inducing chops. In Proceedings of the 26th IEEE/ACM international Conference on Automated software engineering. 263–272.
[23] Péro Gysinets, Béla Vancsics, Andrea Stocco, Davood Mazmazian, Arpad Beszédes, Rudolf Ferenc, and Ali Meshah. 2019. BugFix: a benchmark of JavaScript bugs. In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). IEEE, 90–101.
[24] Mark Harman and Phil McMinn. 2009. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Transactions on Software Engineering 36, 2 (2009), 226–247.
[25] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. 2010. Search based software engineering: Techniques, taxonomy, tutorial. In Empirical software engineering and verification. Springer, 1–59.
[26] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In Proceedings of the 7th International Workshop on Automating Test Case Design,
