Failure Clustering without Coverage

Mojdeh Golagha
Technical University of Munich
Munich, Germany
mojdeh.golagha@tum.de

Constantin Lehnhoff
Technical University of Munich
Munich, Germany
constantin.lehnhoff@tum.de

Alexander Pretschner
Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

Hermann Ilmberger
BMW Group
Munich, Germany
hermann.ilmberger@bmw.de

ABSTRACT

Developing and integrating software in the automotive industry is a complex task and requires extensive testing. An important cost factor in testing and debugging is the time required to analyze failing tests. In the context of regression testing, usually, large numbers of tests fail due to a few underlying faults. Clustering failing tests with respect to their underlying faults can, therefore, help in reducing the required analysis time. In this paper, we propose a clustering technique to group failing hardware-in-the-loop tests based on non-code-based features, retrieved from three different sources. To effectively reduce the analysis effort, the clustering tool selects a representative test for each cluster. Instead of analyzing all failing tests, testers only inspect the representative tests to find the underlying faults. We evaluated the effectiveness and efficiency of our solution in a major automotive company using 86 regression test runs, 8743 failing tests, and 1531 faults. The results show that utilizing our clustering tool, testers can reduce the analysis time more than 60% and find more than 80% of the faults only by inspecting the representative tests.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Failure Clustering, Debugging

ACM Reference Format:
Mojdeh Golagha, Constantin Lehnhoff, Alexander Pretschner, and Hermann Ilmberger. 2019. Failure Clustering without Coverage. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330561

1 INTRODUCTION

Recent innovations in the automotive industry have led to a steady increase in software used in cars. The software units, generated to provide new functionality, are executed within a distributed system of Electronic Control Units (ECUs). Before any software release, the new components of cars' ECUs must be tested for their correctness in multiple stages. One of these stages is the execution of regression tests in a hardware-in-the-loop (HiL) environment. In this environment, the final ECU is used as a computing unit, whereas communication with other components is simulated. In the scope of the regression tests, several hundred test cases are executed each night, and more than a thousand over every weekend, leading to hundreds of failing test cases (TC) every week. These failing tests are analyzed daily or once a week, depending on the test type, by the testers to localize and report the underlying faults (failure is the deviation of actual runtime behavior from expected behavior, and a fault is the root cause for the failure [54]. In our paper, we use "fault" and "root cause" terms interchangeably). Manual analysis of all these failing tests is costly and time-consuming.

According to our previous paper [26], one solution to the problem of excessive analysis effort is the implementation of a pre-analysis tool which clusters failing tests with respect to their root causes. After the execution of a test suite, the results of the failing tests can be passed to the clustering tool which generates and returns a set of clusters. Furthermore, a representative test case for each of the clusters can be determined. In an ideal clustering, each cluster would represent exactly one root cause, and the set of selected representative test cases would cover all the root causes. The analysis effort reduction can then be accomplished by only analyzing the representative tests instead of all failing tests.

However, in [26] we focus on clustering failing tests at the software-in-the-loop (SiL) level, where software components are tested within a simulated environment without any actual hardware. We use the coverage profile of tests as the input for their clustering tool. It is not always possible to use this kind of data in practice due to three reasons. First, the source code is not always accessible (e.g., in the case of HiL tests). Second, in this approach, collecting coverage information when running passing tests is also needed. This requirement is unnecessary and imposes extra work on the system. Third, instrumenting very large projects when running integration tests can be very expensive and time-consuming.

The other similar existing approaches in the literature (see Section 6) are either coverage based or use context-specific data. To bridge this gap, we propose a clustering tool which groups failing tests based on different *code independent* data. This tool can be used in different levels of testing, e.g., HiL, SiL, etc.

Problem. In the automotive industry, usually, several hundreds of tests are executed in a weekly or daily fashion as regressions happen. Analyzing a large number of failing tests to find and repair the root causes is time-consuming and expensive. Since a large number of failures usually happen due to a few underlying faults, is there any way to avoid analyzing all the tests that are failing because of the same reason? Is there any way to select a representative group of tests and find all the faults to reduce the effort?

Solution. As suggested by [26], we propose to cluster failing TCs with respect to their root causes but without coverage data. We select one representative for each cluster. Investigating only the representatives, lead us to find all the root causes without having to inspect all the failing tests.

Contribution. In this paper, we propose:

(1) A new set of non-code-based data to cluster failing tests. These new data make the clustering tool applicable in different levels of testing and different contexts.
(2) Our clustering tool can be used a priori for test selection and prioritization. Since for each test run, due to time and resource constraints, only a subset of tests can be executed, a clustering tool can help in grouping tests based on their similarities and introducing the cluster representatives for the next test run. This can raise the diversity in each test run.
(3) We propose a methodology for adapting the clustering idea to real contexts. We investigate the effectiveness of the adapted approach in a major automotive company with ca. 1.3 million LoC and 13000 tests.

Organization. In section II, we overview the technical background. In section III, we describe our approach. We explain the experiment setup in section IV, and the evaluation results in section V. Finally, in section VI, we review the related work and conclude the paper in section VII.

2 TECHNICAL BACKGROUND

In this section, we review the basics in automotive software test and integration.

2.1 Body Domain Controller

To meet customers' requirements concerning functionality, comfort, and safety, most of today's cars are designed as complex distributed systems, consisting of multiple ECUs. Typically each ECU is in charge of controlling a distinct subset of functions within the car. Multiple communication buses (e.g., CAN, FlexRay, Automotive Ethernet) are used to allow ECUs to exchange information about their states. The communication buses together with the ECUs form a *boardnet*.

The Body Domain Controller (BDC) is one of the central ECUs in the boardnet of every car that connects all main buses to enable boardnet-wide communication, and controls the *body* functions which describe a set of basic functions provided by every car. Each body function belongs to a *component* (e.g. air conditioning, seat functions, sunblinds) and multiple components are grouped into a *domain* (e.g. the comfort domain). Overall, the BDC is divided into 7 domains and more than 100 components.

2.2 Automated Software Integration and Analysis

To ensure the quality and safety, ECU software should be tested in several stages during its development. Following a continuous integration approach, automotive companies need to continuously select, compile, and execute a large number of test cases. Test series are either executed in a software-in-the-loop (SiL) or a hardware-in-the-loop (HiL) environment. In a HiL environment, the software is executed on the real hardware, whereas simulated hardware is used in the SiL environment. In both environments, states and responses of dependent ECUs are simulated.

2.3 TCF - Test Case Framework

Verifying the correctness of ECU software during its development phase can be accomplished following a black-box testing approach. For this purpose, boardnet messages are used to stimulate the ECU from the outside environment, and the ECUs' responses are compared with expected values. To facilitate the implementation of test cases for the BDC, an abstract domain specific language named TCF (Test Case Framework) is used [48]. In TCF, interaction with ECUs is encapsulated by *Actors*. As shown in Figure 1, each Actor provides a set of actions to control a specific function of the ECU, e.g., changing the volume of the audio module. In HiL testing, at compile time, these actions are translated into bus messages with the help of a *Mappings* function. Figure 2 demonstrates the a mapping example for Figure 1. A Mapping is used to map an Actor's action into an I/O signal by defining the bus type (e.g., 'LIN'), the specific bus (e.g., 'KLIN8'), the bus message (e.g., '0x2A'), as well as the signal's name.

```
Audio_Volume {
  VolumeUp(steps)
  VolumeDown(steps)
  VolumeNoDirection(steps)
  NoAction
  Invalid
}
```

Figure 1: TCF Actor

```
HiLMappings {
 Set Audio_Volume.VolumeDown(steps) {
  LIN.KLIN8.0x2A.ST_DIRRT_AUDCU_LIN = steps
 }
 Check Audio_Volume.VolumeDown(steps){
  LIN.KLIN8.0x2A.ST_DIRRT_AUDCU_LIN = steps
 }
}
```

Figure 2: TCF HiL Mapping

The other two important concepts used in the TCF language are *Codings* and *Alternatives*, as shown in Figure 3. Codings are needed to define which peripheral hardware is connected (e.g., the audio module is connected and controlled via LIN bus). *Alternatives* allow to specify different conditions for the execution of the test case (e.g., car state 'parking' vs. 'driving'). For each Alternative, a separate test is generated at compile time. For instance, the 'Audio_Example.tcf' file, illustrated in Figure 3, leads to the generation of two tests: 'Audio_Example_a1' and 'Audio_Example_a2'.

```
TestCase Audio_Example {
 HiLCoding {
  BE_AUDIO_VERB = 01 // connecting audio module
  BE_AUDIO_LIN_VAR = 01 // connecting LIN bus
 }
 Execute {
  Alternatives {
   PWF = ST_CON_VEH_PARK_IO   // parking
   PWF = ST_CON_VEH_DRIV      // driving
  }
  Audio_Volume = VolumeDown(15)
  Run($ShortMainTaskCycle)
  Audio_Volume == VolumeDown(15)
 }
}
```

Figure 3: TCF File

In this paper, we propose a clustering idea to reduce the failure analysis time in BDC HiL regression testing and debugging.

3 APPROACH

In the following, we describe our methodology. We first explain the clustering approach we proposed in [26] that we use as the base of our work. Then, we suggest the data sources that make this clustering approach applicable in different stages of testing and other purposes such as test prioritization.

First, we collect the data from different non-code-based sources, e.g., Jira tickets to make a feature vector for each test case. We binarize all of them to prepare them for hierarchical clustering. Second, utilizing agglomerative clustering, we build a tree of failing tests based on the similarity of their feature vectors. Third, using a regression model on the number of failing tests in previous test runs, we predict the number of clusters. Finally, in the fourth step, we calculate the centers of the clusters and choose the failing tests which are closest to the centers as the representative tests. The testers receive the list of representatives to investigate them. If the proposed number of clusters seems incorrect to the testers, they can adjust the number of clusters directly on the user interface and get the new representatives. Steps two and four are taken from [26]. In the following, we describe all the steps in detail.

3.1 Generating the Input Data

Two main input data sources are: the test result database, containing several million test results from the last couple of years, and the TCF test case repository, providing the source files for the executed tests. We could extract five sets of features (variables in a dataset) using these two data sources. Since the primary objective of this paper is to cluster failing tests, these data are extracted only for failing tests. In case of test selection or prioritization, they can be extracted for all tests. Typically, multiple projects (e.g., weekly, daily, nightly) are used to test a single ECU. Each test run is called a *build*. All the feature values are extracted individually for each build. We explain each set of features in the following.

3.1.1 *General Features.*

We extracted the following features from the database:

- **T_Id:** specifies the test identifier.
- **Agent:** specifies the name of hardware which is executing the tests.
- **Component:** specifies the component of the BDC the test belongs to.
- **Domain:** specifies to which domain of the BDC the test belongs to.
- **File:** specifies from which underlying TCF file the test was generated. A single TCF source file may be used to generate multiple test cases.

Table 1 shows an example of data values for these features. As all these features are of categorical nature and do not follow an ordinal scale, we transformed them into binary data.

Table 1: General Features

Test	Agent	Domain	Component	File
TC 1	Agent-1	Comfort	FBE	Audio.tcf
TC 2	Agent-2	Body	FES	FesModes.tcf
TC 3	Agent-1	Body	FES	FesModes.tcf

3.1.2 *Failed/Passed History.*

To exploit the knowledge of historical test executions, we generated Failed/Passed history matrices. The general idea of this feature set is to express the similarity between two test cases based on the times they shared the same execution result. Naturally, a test result can either have the status *Fail* (F) or *Pass* (P). However, in practice due to the resources constraints, it is not guaranteed that every test case is executed in every build. Thus, we added another value as *Not executed* (N).

Table 2: Fail/Pass History

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	P	P	F	P	F
TC 2	P	N	N	F	N
TC 3	P	N	F	P	P

Table 2 demonstrates an example. Again, we need to transform it to a binary table; thus we generate two tables, one to represent the similarity based on co-occurrences of Failed statuses and one to represent the similarity based on co-occurrences of Passed statuses. As an example, Table 2 has been transformed to the Tables 3 and 4.

The Failed/Passed distance d_{fp} between two tests x and y can therefore be defined as

$$d_{fp}(x, y) = 0.5 * d_f(x, y) + 0.5 * d_p(x, y) , \qquad (1)$$

Table 3: Binary Failed History

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	0	0	1	0	1
TC 2	0	0	0	1	0
TC 3	0	0	1	0	0

Table 4: Binary Passed History

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	1	1	0	1	0
TC 2	1	0	0	0	0
TC 3	1	0	0	1	1

where $d_f(x, y)$ denotes the distance between two tests based on failing history and $d_p(x, y)$ denotes the distance between two tests based on passing history using any specified distance metric (see Section 3.2.3).

3.1.3 Broken/Repaired History. Similar to the Failed/Passed History, the Broken/Repaired History aims at exploiting historical execution knowledge, with a small difference. In the Broken/Repaired History the focus lies on the transition of test statuses. A transition from status *Failed* to status *Passed* is defined as a *Repaired* event (R), whereas a transition from status *Passed* to status *Failed* is defined as a *Broken* event (B). Following this definition, an arbitrary Failed/Passed History can be transformed into a Broken/Repaired History. If no transition takes place, e.g., if a test case failed in two successive builds, a *No Event* (N) label is used to fill the gap. Table 5 shows an example.

Table 5: Broken/Repaired History

Test	Build_1	Build_2	Build_3	Build_4	Build_5
TC 1	N	N	B	R	B
TC 2	N	N	N	N	N
TC 3	N	N	B	R	N

Following the approach described for the Failed/Passed History, again two binary tables should be generated, one to represent the similarity based on co-occurrences of Broken events and one to represent the similarity based on co-occurrences of Repaired events. The computation of distances for two failing tests is similar to the Failed/Passed History calculations.

3.1.4 Jira History. We used the Jira tickets to extract the next feature set which is based on the faults assigned to the previously analyzed failed tests. The idea is that tests which frequently shared the same cause in the past are also likely to fail due to the same cause in the future. Table 6 shows an example. Each "cause" is a Jira ticket ID that has been assigned to the failing test. One ticket may be assigned to several failing tests if the manual analysis shows that these tests are failing because of the same reason. Similar to the previous feature sets, this table should change to a binary form as shown in Table 7.

Table 6: Jira History

Test	Project	Build	Cause
TC 1	project 1	build 1	cause x
TC 2	project 1	build 1	cause x
TC 3	project 1	build 1	cause y
TC 1	project 1	build 2	cause z

Table 7: Binary Jira History

Test	Build1CauseX	Build1CauseY	Build2CauseZ
TC 1	1	0	1
TC 2	1	0	0
TC 3	0	1	0

3.1.5 TCF Test Case Similarity. TCF files used to generate test cases as described in Section 2.3 are maintained in SVN repositories. These repositories are referenced to define the source files needed to generate the desired test series. Our hypothesis is that the likelihood that two tests failed due to the same cause increases with the similarity of their underlying TCF source files. To facilitate the calculation of similarity between two TCF files, we translate TCF files into a standard machine-readable format (JSON). Figure 4 shows an example JSON output for the Audio_Example.tcf file, introduced in Section 2, Figure 3.

Using this JSON representation, the similarity between source files of any two test cases can be calculated using a combination of the following features:

- **CODING:** Each Coding step in a test is mapped to a binary feature. For each test case, it is checked whether the test case contains the Coding step or not (true = 1, false = 0). Two Coding steps are regarded as equal if they share the same name and the same value.
- **ACTOR:** Each Actor in a test is mapped to a binary feature. For each test case it is checked whether the test case uses the given actor to perform an arbitrary action in any of its test steps or not (true = 1, false = 0).
- **ACTOR_TYPE:** Each Actor/Type combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to perform an action of the given Type (e.g., set or check) in any of the test's steps (true = 1, false = 0).
- **ACTOR_VALUE:** Each Actor/Value combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to set or check the given Value in any of the test's steps (true = 1, false = 0).
- **ACTOR_TYPE_VALUE:** Each Actor/Type/Value combination in a test is mapped to a binary feature. For each test case it is checked whether the given Actor is used to perform an action of the given Type for the given Value in any of the test's steps (true = 1, false = 0).
- **BUS_TYPE:** Each Bus Type in a test is mapped to a binary feature. For each test case it is checked whether the test case

```
{
  "tcName": "Audio_Example",
  "codingSteps": [
    {"codingStepName": "BE_AUDIO_VERB",
    "codignStepValue": "01"},
    {"codingStepName": "BE_AUDIO_LIN_VAR",
    "codignStepValue": "01"}
    ],

  "executionSteps": [
    {"executionStepActor": "PWF",
      "executionStepType": "SET",
      "executionStepBusType": "LIN",
      "executionStepBus": "KLIN22",
      "executionStepBusMessage": "0xA",
      "executionStepSignal": "ST_CON_VEH_CSG_LIN",
      "executionStepValue": 2},

    {"executionStepActor": "Audio_Volume",
      "executionStepType": "SET",
      "executionStepBusType": "LIN",
      "executionStepBus": "KLIN8",
      "executionStepBusMessage": "0x2A",
      "executionStepSignal": "ST_TURN_AUDCU_LIN",
      "executionStepValue": 15}
      ]
}
```

Figure 4: JSON Representation of Audio_Example_1.tcf

performs an arbitrary action on a Bus of the given Type in any of its steps (true = 1, false = 0).

- **BUS:** Each Bus in a test is mapped to a binary feature. For each test case it is checked whether the test case performs an arbitrary action on the given Bus in any of its steps (true = 1, false = 0).
- **BUS_MESSAGE:** Each Bus Message in a test is mapped to a binary feature. For each test case it is checked whether the test case uses the given Bus Message to perform an arbitrary action in any of its steps (true = 1, false = 0).

As an example, Table 8 shows the binary values for two examples in Figures 4 (TC1) and 5 (TC2) considering only ACTOR and ACTOR_TYPE features.

Table 8: Binary TCF Similarity Features

Test	PWF	AudioVolume	PWFSet	AudioVolumeSet
TC 1	1	1	1	1
TC 2	0	1	0	1

3.2 Clustering Failing Tests

We need a clustering technique to group our failing tests with respect to their root causes. However, we are not aware of the number of clusters a priori. In fact, in an ideal solution, the number of clusters equals the number of underlying faults or root causes. Thus, we cannot use the techniques in which we must specify the number of clusters in advance. Among all the available techniques such as DBSCAN [17] or OPTICS [1] or the modified version of k-means [44], we decided to use Hierarchical clustering since it enables users to retrieve an arbitrary number of clusters without the need to re-execute the clustering algorithm. This is especially useful in practice since in a real-world scenario, it will not limit the users to a single suggested number of clusters. Users will be able to explore multiple alternatives without the need to wait for the re-execution of the clustering tool.

```
{
  "tcName": "Audio_Other_Example",
  "codingSteps": [
    {"codingStepName": "BE_AUDIO_VERBAUT",
      "codignStepValue": "01"},
    ],

  "executionSteps": [
  {"executionStepActor": "Audio_Volume",
    "executionStepType": "SET",
    "executionStepBusType": "LIN",
    "executionStepBus": "KLIN8",
    "executionStepBusMessage": "0x2A",
    "executionStepSignal": "ST_TURN_AUDCU_LIN",
    "executionStepValue": 99}
    ]
}
```

Figure 5: JSON Representation of Audio_Example_2.tcf

3.2.1 Agglomerative Clustering. Hierarchical clustering or Hierarchical Cluster Analysis (HCA) is a distance-based clustering technique in the field of unsupervised machine learning [45]. The goal of Hierarchical clustering is to compute a tree structure of data objects. The number of clusters can vary from one to the number of individual data objects (m). Agglomerative clustering is a 'bottom-up' implementation of Hierarchical clustering. Starting with m single clusters, clusters with the smallest inter-cluster distance are merged iteratively until only one cluster remains. Figure 6 shows a cluster tree for objects A, B, C, D, and E. In the first iteration, the objects B and C are merged into cluster C3, then objects D and E are merged into cluster C4. In the third iteration object A is merged into the existing cluster C3, forming the new cluster C2. Finally, clusters C2 and C4 are merged into cluster C1.

3.2.2 Clustering Methods. Clustering methods define how inter-cluster distances are calculated using a given distance metric. In our experiment, we used the following methods [45] (considering clusters B and C in Figure 6 as an example):

- **Single:** The distance between two clusters B and C is defined as the minimum distance between any two objects of B and C.
- **Complete:** The distance between B and C clusters is defined as the maximum distance between any two objects of these clusters.

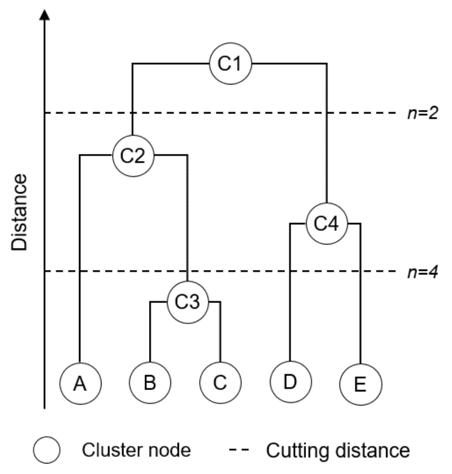

Figure 6: Hierarchical Cluster Tree

- **Average:** The distance between B and C clusters is defined as the average distance between objects of these two clusters.
- **Weighted:** The distance between two clusters A and C3 is defined as the mean of the distances between objects of B and A and between objects of C and A, since cluster C3 was formed by clusters B and C in the previous iteration.
- **Centroid:** The distance between B and C clusters is defined as the distance between the two centroids of these clusters.

3.2.3 Distance Metrics. The following distance metrics can be used to compute the distance between any two objects represented as numerical feature vectors [45]: Euclidean, Squared Euclidean, City-block, Cosine, Correlation, Hamming, Jaccard, Chebyshev, Canberra, Braycurtis, and Yule.

3.3 Predicting the Number of Clusters

The number of predicted clusters relates to the number of underlying faults, therefore an accurate prediction is important for testers. We considered two regression models, Linear regression [36] and Polynomial regression [36], to examine the relationship between the number of failing tests and the cutting distance on the Hierarchical tree. Cutting the tree at a given height will partition the tree at a selected precision. In Figure 6 example, cutting after the first row (n=2) of the tree will yield two clusters, C2 and C4, and cutting after the second row (n=4) will yield four clusters, A, C3, D, and E. The second cutting distance leads to more fine-grained clusters.

In our experiment, we extracted the real number of faults in the previous analyzed builds from the database. Then, we calculated the cutting distances of the respective trees. Finally, we fitted the regression models to predict the cutting distance based on the number of failing tests.

3.4 Selecting the Representatives

As mentioned earlier, the main goal of this paper is to reduce the analysis effort by selecting some representatives for failing tests, so that only the representatives should be analyzed instead of all the failing tests. To this end, we selected the clusters' medoids as representatives. A medoid is the object which is closest to the geometric center of the cluster [62]. Since we want the testers to be able to change the number of the cluster in real-time, we precompute the representatives for all the clusters considering all possible cutting distances.

4 EXPERIMENT

To evaluate how successful our clustering without coverage approach is in achieving its primary objective, reducing failure analysis time, we need to answer the following **research questions**:

RQ1. How effective are non-code-based data in clustering failing tests? How much reduction in analysis time is achievable? How many of the underlying faults are detectable?

RQ2. How efficient is the clustering tool?

RQ3. Which set of input features are the most important and useful in clustering failing tests?

RQ4. Which distance metrics and clustering methods are the most effective in the given context?

Since there are several parameters in our approach that we need to set before applying it, we set up a training phase that helps us select the right metrics and parameters for this context. Thus, first, in the following, we explain the available data, evaluation metrics, and the training phase. Then, we explain the evaluation results in the next section.

4.1 Industrial Data

We had access to BDC test projects. The test suite contains ca. 13000 test cases. In each test run (build) a subset of these 13000 tests is executed. We could extract the results of 86 builds that were analyzed, and the root causes were stored in the test results database. This allows us to use the testers' manual analysis reports as ground truth for assigning failures to faults.

All in all, we had access to 8743 failing tests and 1531 faults. We reserved 10% of the data for test purposes and used the remaining 90% to train the clustering algorithm. As the result, the training set consists of 77 builds, 7837 failing tests and 1360 faults [1].

4.2 Performance Metrics

To evaluate the performance of the clustering, we considered two groups of metrics, one group to measure the effectiveness from the scientific point of view and one group to measure the effectiveness from the practical point of view.

4.2.1 Practice Oriented Metrics. From the perspective of a tester, three questions should be answered:

(1) How many of the existing faults do we find analyzing only the cluster representatives?
(2) How many tests are assigned to the correct faults if we analyze the representative failures and assign the found fault to all of the cluster members?
(3) How much reduction is achievable using the clustering tool?

To answer these questions, we measure the following metrics respectively:

FoundCauses: shows the ratio of the faults found analyzing only the representatives.

[1]Details are available here: https://figshare.com/s/7e9ff8519340f0e7544e

$$FoundCauses = \frac{|C_{rep}|}{|C_{total}|}\ , \tag{2}$$

where C_{rep} is the set of unique faults found through analyzing the representative tests and C_{total} is the set of unique underlying faults (found manually and stored in the database). Score "1" means we found all the faults.

Purity: borrowed from [26], Purity shows how many failures are assigned to the correct faults if analyzing only the representative tests and assigning the same fault to all the cluster members. To compute Purity, each cluster is assigned to the class which is most frequent in the cluster. A class is a fault in our case. The accuracy of this assignment is measured by counting the number of correctly assigned objects dividing by the total number of failures.

$$Purity(\Omega, \mathbf{C}) = \frac{1}{N}\sum_{k}\max_{j}|\omega_k \cap c_j|, \tag{3}$$

where N is the total number of failures, $\Omega = \{\omega_1, \omega_2, .., \omega_k\}$ is the set of K clusters and $C = \{c_1, c_2, .., c_j\}$ is the set of J classes. Purity score "1" means all the members in the cluster failed because of the same reason and all of the them have been assigned to the same fault.

Achieved Reduction (ARed): borrowed from [26], ARed measures how much of the possible reduction we could achieve. To measure the achieved reduction, first, the absolute reduction *Red* and the ideal effort reduction *IRed* must be calculated. If we consider *the time needed for analyzing one test as one unit of analysis time*, then:

$$Red = 1 - \frac{N_p}{\#f_{tests}}\ , \tag{4}$$

$$IRed = 1 - \frac{N_t}{\#f_{tests}}\ , \tag{5}$$

where N_p is the number of predicted clusters, N_t is the number of underlying faults, and $\#f_{tests}$ is the number of failures. The achieved reduction *ARed* is then defined as the proportion of the actual reduction *Red* to the ideal reduction *IRed*:

$$ARed = \frac{Red}{IRed} \tag{6}$$

We, finally, combine these three evaluation metrics to a single metric. A single performance metric facilitates the evaluations of different parameter combinations when fitting the clustering model. Thus, we define the **Performance** metric as:

$$Performance = w_{fc} \times FoundCauses + w_p \times Purity + w_{ar} \times ARed\ , \tag{7}$$

where w_{fc}, w_{ar}, and w_p are the weights that can be defined by the testers based on their needs. After discussion with industry experts and training with different values, we found the best combination as w_{fc}=0.4, w_{ar}=0.4, and w_p=0.2. An important fact to consider when choosing the performance weights is that both the FoundCauses and the Purity metrics (unlike ARed) favor the larger number of clusters and a few members in each cluster. For instance, having each failing test in a separate cluster leads to *Purity* score "1" and subsequently FoundCauses score "1" but ARed score "0".

4.2.2 *Science Oriented Metrics.*

To enhance the scientific comparability of the clustering results, we also measure the following metrics.

Precision, Recall, and F-measure: Applied in the clustering context, a true positive (TP) means assigning two failures grounded in the same fault to the same cluster; a true negative (TN) means assigning two failures grounded in different faults to different clusters; a false positive (FP) means assigning two failures grounded in different faults to the same cluster; and a false negative (FN) means assigning two failures grounded in the same fault to different clusters. Then:

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

$$F_1 = 2 \times \frac{Recall \times Precision}{Recall + Precision} \tag{10}$$

Entropy: As introduced by Manning et al. [46], Entropy measures the internal disorder of clusters. Therefore, lower values of Entropy indicate better clustering results. To calculate the Entropy for each of the K clusters:

$$E_F(f_i, C) = -\sum_{j=1}^{K}(\frac{f_{ij}}{|f_i|})log(\frac{f_{ij}}{|f_i|}), \tag{11}$$

where f_{ij} is the number of failures of type i (failing because of underlying fault i) that belongs to cluster c_j and $|f_i|$ is the total number of failures of type i. Then, the total Entropy is:

$$E_{F-TOT}(F, C) = -\sum_{i=1}^{J} E_F(f_i, C). \tag{12}$$

4.3 Parameter Setting

We developed our clustering tool as a standalone Python application using SciPy [2], NumPy [3], and Pandas [4]. Fitting our clustering model involves setting the following parameters:

(1) the optimal clustering method
(2) the optimal distance metric for each of five groups of features
(3) the optimal weights for each of five groups of features
(4) the optimal way of predicting the number of clusters

to find the best parameter for each item in the training phase, we used the *Performance* metric.

4.3.1 *Clustering Method and Distance Metric.*

The Performance values of different method and distance metric combinations do not show a significant difference. Comparing all the Performance values [5], we chose "Centroid" as the clustering Method, and "Hamming" as the distance metric for General, Failed/Passed, Broken/Repair, and Jira features sets, and "Euclidean" as the distance metric for TCF Similarity feature set. Table 9 shows the average Performance values on the training dataset.

[2]https://www.scipy.org/
[3]http://www.numpy.org/
[4]https://pandas.pydata.org/
[5]Due to space constraints, we do not include all the results. They are available here: https://figshare.com/s/cdda4b6e074f588195d0

Table 9: Performance values using Centroid method

Feature Set	Dist. Metric	Performance_Avg
General Features	Hamming	0.82
Failed/Passed History	Hamming	0.76
Broken/Repaired History	Hamming	0.75
Jira History	Hamming	0.81
TCF Similarity	Euclidean	0.83

Table 10: Best Performing Weights for Input Feature Sets

Feature Set	Symbol	Value
General Features	$w_{general}$	0.29
Failed/Passed History	w_{fp}	0.01
Broken/Repaired History	w_{br}	0.09
Jira History	w_{jira}	0.31
TCF Similarity	w_{tcf}	0.30

4.3.2 Input Feature Weights. As described in Section 3.1, we extracted five sets of features as input for our clustering. These five sets, lead to the generation of five different datasets. We use these five datasets in the following way. We measure the similarity between two tests using each of the five datasets separately. To have a final similarity value, we need to combine these five similarity values. We use the following formula to combine these values:

$$Similarity = w_{general} \times s_{general} + w_{fp} \times s_{fp} + w_{br} \times s_{br} + w_{jira} \times s_{jira} + w_{tcf} \times s_{tcf}, \tag{13}$$

where w is the weight, and s is the similarity value. Since it is not clear which sets of features are more useful in clustering to get a higher weight, we generated 1000 random weight combinations that satisfy $w_{general} + w_{fp} + w_{br} + w_{jira} + w_{tcf} = 1$ condition. To measure the similarity between two tests, first, we measure the similarity based on all the five feature sets. Then, assigning the weights to each group, we sum up the similarity values to find the combined similarity value. Table 10 shows the best combination of weights, which resulted in the highest Performance value on the training data. The results show that General Features and the Jira History, and TCF Similarity sets are the most important features, while Broken/Repaired History and Failed/Passed History are not so useful.

4.3.3 Predicting the Number of Clusters. We used Linear and Polynomial regressions to investigate the correlation between the number of failing tests and the cutting distance in the Hierarchical tree of failing tests (explained in Section 3.2.1).

The results show that applying Polynomial regression leads to a better performance than Linear regression, $P_{polynomial}$=0.80 and P_{linear}=0.76. Figure 7 depicts the fitted Polynomial regression model, where x is the number of failing tests, and the $f(x)$ shows the predicted distance to cut the tree.

5 EVALUATION

We can now answer the research questions, describe evaluation results, and explain threats to validity.

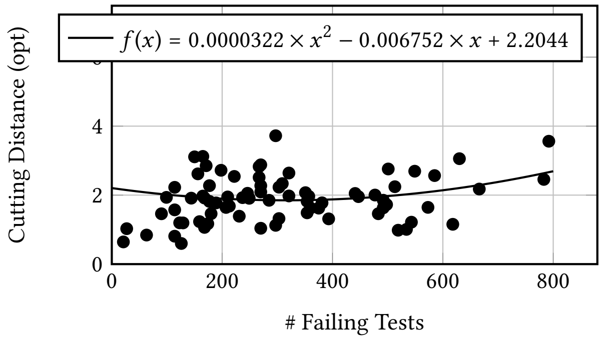

Figure 7: Polynomial Regression Model for Cutting Distance

5.1 Summary of Results

The results of the clustering evaluation are shown in Table 11. As the results show, we are able to achieve high scores of Purity, FoundCauses and also ARed. Since the number of predicted clusters is usually larger than the number of underlying faults, Recall, F-measure and Entropy metrics do not show high scores. However, we are able to achieve a huge reduction in analysis time anyway and find more than 80% of the causes by investigating only the representatives. For instance, in Build 154, the ground truth says there are 13 underlying faults. We predicted 35 clusters that are 99% pure. These 22 extra clusters (35-13) lead to poor Recall (0.13) and consequently F-measure (0.22) scores. Nevertheless, with selecting one representative for each of these 35 clusters, and analyzing only them, we reduced the analysis time 66% ((1-(35/78))/(1-(13/78))) and found all the underlying faults (FoundCauses=1.0).

Based on the industry partner's needs and experts' advice, we assigned higher weights to the combination of FoundCauses and Purity (0.6) rather than the ARed (0.4). Due to this reason, the results tend to a larger number of clusters which are almost pure. Nevertheless, we have always achieved more than 60% reduction in the analysis time. Assigning higher weights to ARed in the training phase leads to different parameter setting and consequently different results. To compare the results considering different weights, we considered two alternatives: 1. (w_{fc}=0.6, w_{ar}=0.4, and w_p=0.0) and 2. (w_{fc}=0.4, w_{ar}=0.6, and w_p=0.0). Table 12 shows the average scores of performance metrics.

Based on the above results, we answer the research questions as follows:

RQ1. Considering the Performance metric which is a combination of FoundCauses, Purity, and ARed metrics, our idea in using non-code-based data for clustering failing test is quite effective. In the evaluation, on average 90.4% of the faults are found when only analyzing the representative tests. Furthermore, an average Purity of 96.7% is achieved, which means that only 3.3% of the failing tests would be assigned to a wrong fault when assigning the root cause of the cluster's representative to all members of the cluster. The results show an average of 83.1% reduction in analysis time.

RQ2. Clustering based on non-code-based data is more efficient than clustering based on code-based execution profile (if the source code is available). In our approach, we need to compute the data for similarity comparisons one time a priori; then we can use them several times. Code-based techniques such as [26] need to instrument

Table 11: Evaluation Results - Performance = 0.4 * FoundCauses + 0.2 * Purity + 0.4 * ARed

Build	# Failures	# Faults	# Clusters	Precision	Recall	F-measure	Entropy	FoundCauses	Purity	ARed	Performance
31	142	9	44	0.99	0.14	0.25	0.04	0.89	0.98	0.74	0.85
54	77	26	29	0.87	0.68	0.76	0.15	0.88	0.95	0.94	0.92
75	86	12	13	1.00	0.79	0.88	0.05	0.92	0.94	0.99	0.95
102	96	25	35	0.96	0.63	0.76	0.05	0.96	0.99	0.86	0.93
114	37	20	20	0.94	0.71	0.81	0.05	0.95	0.98	1.00	0.98
124	185	10	65	0.97	0.03	0.07	0.03	0.90	1.00	0.69	0.83
134	84	23	26	0.89	0.84	0.86	0.14	0.83	0.95	0.95	0.90
144	121	33	63	0.39	0.20	0.26	0.40	0.82	0.94	0.66	0.78
154	78	13	35	0.93	0.12	0.21	0.05	1.00	0.99	0.66	0.86

Table 12: Average Scores of Performance Metrics Using Different Weights

Metric	Initial Setup	Alternative 1	Alternative 2
Purity	0.9674	0.9740	0.8989
ARed	0.8312	0.7990	0.9520
FoundCauses	0.9049	0.9151	0.6566
Performance	0.8879	0.8686	0.8338

the code which can be expensive in case of integration testing. Our data collection and pre-processing took 5 hours while running the clustering itself took 3 minutes in the largest case. Collected data can be updated at longer intervals such as every two months.

RQ3. We set up a training phase to answer this question. We randomly generated 1000 weight coefficients for our five groups of feature sets. As shown in Table 10, General Features, Jira History, and TCF Similarity are the most important groups. There is not any significant difference between these groups.

RQ4. Like RQ3, we found the answer to this question in the training phase. Our experiment confirms our previous finding in [26] that the impact of the clustering method and distance metric on the clustering performance is negligible. However, choosing a suitable strategy to determine the number of clusters is highly important.

Cross-validation. To test our model's ability to predict new data and to give an insight on how the model will generalize, in addition to the previous setting, we did 10-fold cross-validation. In our cross-validation setting, we used the training part to select the best weights for feature combination and to fit the Polynomial regression model. However, we preserved the distance metric and the clustering method unchanged since they do not have a significant impact on the results. The average scores are:

Foundcauses=0.8813, **Purity=0.9700**, and **ARed=0.8813**.

Considering w_{fc}=0.4, w_{ar}=0.4, and w_p=0.2, these scores yield **Performance=0.8813**. For obvious reasons, we could not just report the cross-validation results. Since we could not report the average scores of metrics such as "# of Faults", "# of Failures", and "# of Clusters", we selected a snap of data as the test data set to be able to show and discuss the results in Table 11.

Comparing with [26]. Although we cannot compare the results directly since the experiments have been done in different contexts, we include a summary of the results. In [26], we evaluated our technique using 11 SiL builds, 499 failing tests, and 36 faults. On average, we achieved Purity=0.96, FoundCauses=1, and ARed= 0.9177. Using source code as the data for clustering inevitably leads to more accurate clusters meaning higher scores of Purity and FoundCauses and lower scores of ARed. On a larger experiment with 86 HiL builds, 8743 failing tests, and 1531 faults, we achieved the same Purity score and a higher ARed score while losing 10% of the root causes. However, the advantage of our new technique is its applicability in all stages of testing, even in the cases that source code is not available. In addition, collecting and re-using data for several experiments is making it more efficient. Finally, our new technique can also be used a priori in combination with coverage data for test selection and prioritization.

Generalizability. We believe that our proposed approach is applicable in other industrial domains because we have chosen domain independent features. In the case of TCF test similarity features, the idea is to extract high-level steps taken in the tests and compare them. In the case of general features, the idea is to extract the components or sub-systems or domains' names which the tests belong to and compare them. In the case of other features, the idea is to extract the history of test executions and analysis. All these features are usually accessible in all companies or organizations.

A priori test selection and prioritization. To use our clustering tool as a test selection tool, one can feed the tool all the tests before running them. The tool finds the clusters and returns a representative test for each cluster. The representative tests are the subset of tests that should be executed in a limited time. However, weights assigned to the five sets of features should be optimized for this goal.

5.2 Threats to Validity

We evaluated our approach in a large scale industrial case. For both training and evaluation, we used real-world systems, failures, and faults. Nevertheless, we do not claim that our experimental setup and results will be valid for all other contexts.

Proper data recording has a great impact on our clustering approach. We offered five feature types to measure the semantic similarity between test cases. However, if failure analysis results, root causes information, patches and changes are not stored frequently in database, SVN, and Jira systems, the clustering results will not be accurate.

To find the best solution for our case, we set up a training phase. To set the parameters and assign the weights, we considered industry needs and experts' advice. Changing the parameters based on different needs could affect the final results. Nevertheless, we tried to propose a methodology for applying this idea in different environments.

6 RELATED WORK

We developed our work based on the clustering idea we proposed previously in [26]. We investigated the impact of clustering failing tests based on coverage profiles on reducing analysis time. The reduction in our earlier work is achieved by selecting and analyzing cluster representatives. Having access to the source files, we instrument the code and generate function-level coverage which later is used to calculate distances between the failing tests. By applying Agglomerative clustering, we obtain a cluster tree and use fault localization techniques to predict the number of clusters. The selection of cluster representatives is then achieved by finding the nearest neighbor to the clusters' centers. The main difference between our new work and the previous work is in the data used for clustering. In this paper, we suggest some non-code-based data from different resources rather than the source code. Our new solution makes the clustering more efficient and applicable in different stages of testing.

Prior to [26]'s clustering experiment, [3, 11, 34, 41, 53] suggested using coverage profiles, Markov model and fault localization for segregating between failing and passing executions and also segregating between failing executions with different root causes. Later, [13, 31, 52, 55] proposed similar solutions in different contexts such as database applications, with different feature types extracted from their own specific contexts. To the best of our knowledge, we are the first to use our proposed five set of features to cluster failures specifically at the HiL level of testing.

Recently, Pham et al. [51] grouped failing tests based on the similarity between symbolic execution paths, generated by the symbolic execution engine KLEE [5]. They argue that their symbolic analysis approach generates more fine-grained clusters that segregate different faults even if the call stack of failures are the same. However, symbolic execution has scalability issues. For large industrial cases such as automotive companies, utilizing symbolic execution would be expensive.

Another related line of research involves clustering crash or bug reports [9, 10, 23, 37, 42, 49, 58]. Our focus is clustering failing tests rather than crash reports, bug reports or failing traces. These techniques explore run-time information collected in the field where the software systems are deployed.

Our second goal, using our clustering tool a priori to select tests for the next regression test run, is somewhat related to test selection or prioritization area of research.

The aim of test prioritization [7, 12, 14, 19–21, 33, 38, 59, 66, 68, 69] is to minimizing test time and maximizing efficiency with running fault revealing tests first. Thus, these techniques rely on data such as statement coverage for fault detection and hope that satisfying these coverage goals will lead to an increasing fault detection rate. Most of these test case prioritization techniques are coverage based.

Regression test selection is a well-studied research topic [4, 15, 16, 18, 24, 25, 28, 30, 39, 47, 50, 61, 65, 70, 72]. These techniques compute test dependencies statically or dynamically to find the tests that are affected by the code changes. Rerunning only the affected tests reduces regression testing costs. Similarly, test-suite reduction helps in speeding up the regression test by eliminating the redundant tests [2, 6, 8, 22, 27, 29, 32, 35, 40, 43, 56, 57, 60, 63, 64, 67, 68, 71, 73]. All these techniques are based on the source code. However, we believe that utilizing our clustering technique, we will be able to add semantic similarity analysis to these approaches to improve their effectiveness. In their real world experiment report, Shi et al. [60] conclude that researchers should develop novel test suite reduction techniques that either miss fewer failed builds or at least provide more predictable fault detection loss.

7 CONCLUSION AND FUTURE WORK

Analyzing failing tests in the scope of automotive hardware-in-the-loop regression testing is a time-consuming task and requires a significant amount of manual work. To reduce the excessive analysis effort, we developed a clustering tool that groups failing tests with respect to their underlying faults and does not use coverage information. The analysis time is then reduced by proposing a single representative test for each cluster so that testers only have to analyze the representative tests instead of all failing tests. We suggested five different non-code-based data source to measure the similarity between failing tests. We evaluated the effectiveness and efficiency of our proposed idea in an automotive context with 86 regression test runs containing 8743 failing tests and 1531 faults.

The results show that out of the five suggested data feature sets, Jira History, General Features, and TCF Similarity are the most useful features in this context. While clustering method and distance metric do not have a significant impact on the results, the technique used for predicting the number of clusters has high importance. We fitted a Polynomial regression model based on the number of failing tests to predict the number of clusters. With this approach, we are able to achieve an average of 83.1% reduction in analysis time.

In the future, we will consider two more data sources, bus traces and log messages. Also, we plan to replace the current random weight optimization routine for feature sets with a genetic algorithm. The genetic algorithm can find the most promising combination of weights.

In addition, we will evaluate the effectiveness of our tool in scheduling and prioritizing test cases. Since tests which are in the same cluster tend to fail due to the same reason, selecting a subset of tests extracted from different clusters may help in finding the same amount of faults using less testing resources.

In our experiment, we only used non-coverage data. However, we believe that with combining them with coverage data, we can improve the fault detection rate of the reduced test suites. Thus, we plan to evaluate our approach considering a combination of non-code-based and coverage data for both failure clustering and test selection/prioritization.

REFERENCES

[1] M Ankerst, MM Breunig, HP Kriegel, and J Sander. 1999. OPTICS: Ordering points to identify the clustering structure. *SIGMOD RECORD, VOL 28, NO 2 - JUNE 1999: SIGMOD99: PROCEEDINGS OF THE 1999 ACM SIGMOD - INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA* (1999).

[2] J. Black, E. Melachrinoudis, and D. Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. In *Proceedings. 26th International Conference on Software*

Engineering. 106–115.

[3] James F. Bowring, James M. Rehg, and Mary Jean Harrold. 2004. Active learning for automatic classification of software behavior. *ACM SIGSOFT Software Engineering Notes* (2004).

[4] L.C. Briand, Y. Labiche, and S. He. 2009. Automating regression test selection based on UML designs. *Information and Software Technology* 51, 1 (2009), 16 – 30. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[5] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *USENIX* (2008).

[6] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, and B. Xie. 2017. How Do Assertions Impact Coverage-Based Test-Suite Reduction?. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 418–423.

[7] Jinfu Chen, Lili Zhu, Tsong Yueh Chen, Dave Towey, Fei-Ching Kuo, Rubing Huang, and Yuchi Guo. 2018. Test Case Prioritization for Object-oriented Software. *J. Syst. Softw.* 135, C (Jan. 2018), 107–125.

[8] T.Y. Chen and M.F. Lau. 1998. A new heuristic for test suite reduction. *Information and Software Technology* 40, 5 (1998), 347 – 354.

[9] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. 2012. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *2012 34th International Conference on Software Engineering (ICSE)*. 1084–1093.

[10] T. Dhaliwal, F. Khomh, and Y. Zou. 2011. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 333–342.

[11] William Dickinson, David Leon, and Andy Podgurski. 2001. Finding failures by cluster analysis of execution profiles. *Proceedings - International Conference on Software Engineering* (2001).

[12] William Dickinson, David Leon, and Andy Podgurski. 2001. Pursuing Failure: The Distribution of Program Failures in a Profile Space. *SIGSOFT Softw. Eng. Notes* 26, 5 (Sept. 2001), 246–255.

[13] Nicholas DiGiuseppe and James A. Jones. 2012. Concept-Based Failure Clustering. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*.

[14] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing Test Cases for Regression Testing. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*. ACM, New York, NY, USA, 102–112.

[15] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. 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 (ESEM '08)*. ACM, New York, NY, USA, 22–31.

[16] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 11–20.

[17] X Ester, M., Kriegel, H. P., Sander, J., & Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Kdd* (1996).

[18] Facebook. [n.d.]. Buck. https://buckbuild.com/. Accessed: 2019-01-28.

[19] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. 2014. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal* 22, 2 (01 Jun 2014), 335–361.

[20] ChunRong Fang, ZhenYu Chen, and BaoWen Xu. 2012. Comparing logic coverage criteria on test case prioritization. *Science China Information Sciences* 55, 12 (01 Dec 2012), 2826–2840.

[21] Yang Feng, Zhenyu Chen, James A. Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*.

[22] Jingyao Geng, Zheng Li, Ruilian Zhao, and Junxia Guo. 2016. Search Based Test Suite Minimization for Fault Detection and Localization: A Co-driven Method. In *Search Based Software Engineering*, Federica Sarro and Kalyanmoy Deb (Eds.). Springer International Publishing, Cham, 34–48.

[23] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 103–116.

[24] M. Gligoric, L. Eloussi, and D. Marinov. 2015. Ekstazi: Lightweight Test Selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 713–716.

[25] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 211–222.

[26] Mojdeh Golagha, Alexander Pretschner, Dominik Fisch, and Roman Nagy. 2017. Reducing failure analysis time: An industrial evaluation. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*.

[27] Arnaud Gotlieb and Dusica Marijan. 2014. FLOWER: Optimal Test Suite Reduction As a Network Maximum Flow. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 171–180.

[28] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, Memphis, TN, USA, October 15-18, 2018*. 112–122.

[29] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. 1993. A Methodology for Controlling the Size of a Test Suite. *ACM Trans. Softw. Eng. Methodol.* 2, 3 (July 1993), 270–285.

[30] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. ACM, New York, NY, USA, 312–326.

[31] Chien Hsin Hsueh, Yung Pin Cheng, and Wei Cheng Pan. 2011. Intrusive test automation with failed test case clustering. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*.

[32] D. Jeffrey and N. Gupta. 2007. Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. *IEEE Transactions on Software Engineering* 33, 2 (Feb 2007), 108–123.

[33] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. 2009. Adaptive Random Test Case Prioritization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 233–244.

[34] James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in Parallel. In *ISSTA*.

[35] J. A. Jones and M. J. Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering* 29, 3 (March 2003), 195–209.

[36] André I. Khuri. 2013. Introduction to Linear Regression Analysis, Fifth Edition by Douglas C. Montgomery, Elizabeth A. Peck, G. Geoffrey Vining. *International Statistical Review* (2013).

[37] S. Kim, T. Zimmermann, and N. Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 486–493.

[38] Y. Ledru, A. Petrenko, and S. Boroday. 2009. Using String Distances for Test Case Prioritisation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 510–514.

[39] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 583–594.

[40] Jun-Wei Lin and Chin-Yu Huang. 2009. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology* 51, 4 (2009), 679 – 690.

[41] Chao Liu and Jiawei Han. 2006. Failure proximity: A fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

[42] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09)*. ACM, New York, NY, USA, 557–566.

[43] Xue-ying Ma, Bin-kui Sheng, and Cheng-qing Ye. 2005. Test-Suite Reduction Using Genetic Algorithm. In *Advanced Parallel Processing Technologies*, Jiannong Cao, Wolfgang Nejdl, and Ming Xu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 253–262.

[44] James MacQueen. 1967. Some Methods for classification and Analysis of Multivariate Observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability 1967*.

[45] Oded Maimon and Lior Rokach. 2010. *Data Mining and Knowledge Discovery Handbook 2ed.*

[46] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA.

[47] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale Continuous Testing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 233–242.

[48] Sroka Michal, Nagy Roman, and Fisch Dominik. 2014. Specification-Based Testing Via Domain Specific Language. *Research Papers Faculty of Materials Science and Technology Slovak University of Technology* 22, 341 (December 2014), 1–6.

[49] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet. 2007. Automatically Identifying Known Software Problems. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*. 433–441.

[50] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, New York, NY, USA, 241–251.

[51] Van Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. 2017. Bucketing failing tests via symbolic analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

[52] Andreas Podelski, Martin Schäf, and Thomas Wies. 2016. Classifying Bugs with Interpolants. In *Tests and Proofs*, Bernhard K. Aichernig and Carlo A. Furia (Eds.). Springer International Publishing, Cham, 151–168.

[53] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.*

[54] Alexander Pretschner. 2015. Defect-Based Testing. In *Dependable Software Systems Engineering*.

[55] Erik Rogstad, Lionel Briand, and Richard Torkar. 2013. Test case selection for black-box regression testing of database applications. *Information and Software Technology* (2013).

[56] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. 2002. 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.* 130–140.

[57] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* 34–43.

[58] P. Runeson, M. Alexandersson, and O. Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering (ICSE'07).* 499–510.

[59] Pavi Saraswat, Abhishek Singhal, and Abhay Bansal. 2019. A Review of Test Case Prioritization and Optimization Techniques. In *Software Engineering*, M. N. Hoda, Naresh Chauhan, S. M. K. Quadri, and Praveen Ranjan Srivastava (Eds.). Springer Singapore, Singapore, 507–516.

[60] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-suite Reduction in Real Software Evolution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018).* ACM, New York, NY, USA, 84–94.

[61] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-suite Reduction and Regression Test Selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015).* ACM, New York, NY, USA, 237–247.

[62] Anja Struyf, Mia Hubert, and Peter Rousseeuw. 1997. Clustering in an Object-Oriented Environment. *Journal of Statistical Software, Articles* 1, 4 (1997), 1–30.

[63] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. 1995. Effect of Test Set Minimization on Fault Detection Effectiveness. In *1995 17th International Conference on Software Engineering.* 41–41.

[64] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. 1997. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97).* 522–528.

[65] G. Xu and A. Rountev. 2007. Regression Test Selection for AspectJ Software. In *29th International Conference on Software Engineering (ICSE'07).* 65–74.

[66] S. Yan, Z. Chen, Z. Zhao, C. Zhang, and Y. Zhou. 2010. A Dynamic Test Cluster Sampling Strategy by Leveraging Execution Spectra Information. In *2010 Third International Conference on Software Testing, Verification and Validation.* 147–154.

[67] Shin Yoo and Mark Harman. 2007. Pareto Efficient Multi-objective Test Case Selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07).* ACM, New York, NY, USA, 140–150.

[68] S. Yoo and M. Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* 22, 2 (March 2012), 67–120.

[69] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. 2009. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09).* ACM, New York, NY, USA, 201–212.

[70] L. Zhang. 2018. Hybrid Regression Test Selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* 199–209.

[71] L. Zhang, M. Kim, and S. Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* 23–32.

[72] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2011. An Empirical Study of JUnit Test-Suite Reduction. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering (ISSRE '11).* IEEE Computer Society, Washington, DC, USA, 170–179.

[73] Hao Zhong, Lu Zhang, and Hong Mei. 2008. An Experimental Study of Four Typical Test Suite Reduction Techniques. *Inf. Softw. Technol.* 50, 6 (May 2008), 534–546.