How to Effectively Reduce Failure Analysis Time?

Mojdeh Golagha

Abstract Debugging is one of the most expensive and challenging phases in the software development life-cycle. One important cost factor in the debugging process is the time required to analyze failures and find underlying faults. Two types of techniques that can help developers to reduce this analysis time are Failure Clustering and Automated Fault Localization. Although there is a plethora of these techniques in the literature, there are still some gaps that prevent their operationalization in real-world contexts. Besides, the abundance of these techniques confuses the developers in selecting a suitable method for their specific domain. In order to help developers in reducing analysis time, we propose methodologies and techniques that can be used standalone or in a form of a tool-chain. Utilizing this tool-chain, developers (1) know which data they need for further analysis, (2) are able to group failures based on their root causes, and (3) are able to find more information about the root causes of each failing group. Our tool-chain was initially developed based on state-of-the-art failure diagnosis techniques. We implemented and evaluated existing techniques. We built on and improved them where the results were promising and proposed new solutions where needed. The overarching goal of this study has been the applicability of techniques in practice.

1 Introduction

We are in the era of software intensive systems. The complexity of systems is growing as they are being increasingly used in safety critical applications. These new applications have raised the need for intensive testing to assure the reliability of the systems.

M. Golagha (✉)
fortiss, München, Germany
e-mail: golagha@fortiss.org

© The Author(s) 2022

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2020*,
https://doi.org/10.1007/978-3-030-83128-8_4

These changes have consequently influenced software debugging endeavors. From the increase in the number of tests and the fast pace of delivering software have emerged the need for more automated debugging techniques. Automated diagnosis techniques can reduce the effort spent on manual debugging, which shortens the test-diagnose-repair cycle, and can therefore be expected to lead to more reliable systems, and a shorter time-to-market. Software debugging has always been recognized as a time-consuming, tiresome, and expensive task that cost billions for economies every year [35].

In our terminology, a *failure* is the deviation of actual run-time behavior from intended behavior, and a *fault* is the reason for the deviation [36]. In other words, a fault is the program element that needs to be changed in order to remove the failure. The essence of failure diagnosis is to trace back a failure to the fault or faults [11].

Developers usually get quick and yet preliminary test results from huge amounts of test runs and use this information to attack problems such as locating faults. Such a quick feedback approach may lower the development cost but puts a lot of pressure on developers. In practice, the time resource allocated for each debugging session is usually limited and predetermined. Therefore, developers need an assisting tool which increases their productivity and reduces failure analysis time. We focused on automated diagnosis techniques and tried to improve them to make them applicable in practice. We recognized two general categories of techniques for addressing reducing failure diagnosis time: *failure clustering* and *automated fault localization*.

Failure clustering methods attempt to group failing tests with respect to the faults that caused them [30]. If there are several failing test cases (TC) as the result of test execution, these failing TCs may be clustered such that tests which are in the same cluster would have failed due to the same hypothesized fault. Then, in an ideal world, testers investigate only one representative TC from each cluster to discover all the underlying faults. This process eliminates the need for analyzing each failing TC individually. Thus, there would be a significant reduction in analysis time[9].

Automated fault localization techniques aim to "identify some program event(s) or state(s) or element(s) that cause or correlate with the failure to provide the developer with a report that will aid fault repair" [21]. The debugging process is usually predicated on the developer's ability to find and repair faults. While both steps in the debugging process (fault localization and fault repair) are time-consuming in their own right, fault localization is considered more critical, as it is a prerequisite for fault repair [21]. Furthermore, Kochhar et al. have found that there is a large demand for fault localization solutions among developers [17]. Therefore, over the past ten years, a lot of research has gone into developing automated techniques for fault localization in order to help speed up the process [38].

There is a plethora of failure clustering and automated fault localization techniques in the literature [28]. Although developers find these methods worthwhile and essential [17], these techniques are not adopted in practice yet.

2 Failure Clustering

Clustering failures is effective in reducing failure analysis time [9]. Its advantages are threefold:

1. It eliminates the need to analyze each failing test individually. To achieve this goal, it is enough to select one representative for each cluster and analyze only the representatives to find all the underlying faults in case of multiple faults [9].

   In an industrial environment, usually several dozens of TCs are executed each night as regressions happen, and several hundred every weekend, which together usually lead to large numbers of failures. Developers must analyze the failing tests and find all the root causes in the short time they have before the software release. The complexity of the analysis process makes the failure diagnosis process tough and time-consuming. Since in practice, a single fault usually leads to the failure of multiple TCs, analyzing only the representative TCs helps developers to find more faults in a shorter time.
2. It provides the opportunity for debugging in parallel [13].
3. It gives an estimation of the number of faults causing the failures. Fault localization, while there are several faults in the code, is more challenging than when there is only one fault in the code. When a program fails, the number of faults is, in general, unknown, and certain faults may mask or obfuscate other faults [13].

Jones et al. [13] introduced a parallel debugging process as an alternative to sequential debugging. They suggest that in the presence of multiple faults in a program, clustering failing tests based on their underlying faults, and assigning clusters to different developers for simultaneous debugging, reduce the total debugging cost and time. They propose two clustering techniques. Using the first technique, they cluster failures based on execution profiles and fault localization results. They start the clustering process by using execution profile similarities and complete it using fault localization results. Their second technique suggests to only use the results of fault localization.

Hoegerle et al. [12] introduced another parallel debugging method which is based on integer linear programming [25]. They applied the above-mentioned second clustering technique of Jones et al. to compare it with their own debugging approach. Their results show that this clustering technique of Jones et al. is not so effective. But the first technique is effective if it is adapted to the context.

Parallel debugging reduces the analysis time. However, it does not remove the need for analyzing all the failing tests one by one. It provides segregation between faults to facilitate fault localization. But it does not provide segregation between failing tests.

Another shortcoming in this area of research is the lack of a methodology for adapting this idea to different industrial domains.

Moreover, the other similar existing approaches in the literature are either based on coverage data or use context-specific data [31]. Therefore, there is a need for

other sources of noncoverage data for the cases that the source code or execution profile is not available.

To close the above-mentioned gaps, we propose a clustering approach. In the following, first we describe our general clustering approach. Then, we explain two different techniques based on available data.

2.1 Clustering Approach

Our general approach consists of usual steps of any clustering solution: First, we run TCs and collect relevant data to use for clustering. Second, we apply hierarchical clustering [32] on data. Third, we use some metrics to choose the best number of clusters and finalize clusters. In an ideal solution, there is one cluster for each underlying fault. Therefore, the number of clusters equals the number of faults. Intuitively, we are not aware of this number beforehand. Fourth, we select one (or k) representative for each cluster to start the debugging process. To implement this clustering approach, one might need to adapt some steps based on available data. In the following, we explain two failure clustering techniques that can be utilized in different settings and at different levels of testing.

2.1.1 Failure Clustering with Coverage

In this approach, we use *test coverage profiles* as data for clustering. First, we run a test suite and extract an execution profile for each TC. Second, utilizing agglomerative hierarchical clustering, we build a tree of failing tests based on the similarity of execution traces. In order to cut this tree into clusters we need to know the best number of clusters. In the third step, we hence utilize fault localization techniques to decide on the best number of clusters. Then, we cut the tree into the found number of clusters. Finally, in the fourth step, we calculate the centers of the clusters and choose the failures which are closest to the centers as representative tests.

Step 1: Running Tests and Profiling Executions

The first step is to run the tests and profile executions. To profile TC executions, we instrument the code. Executing a program while instrumenting the code results in a report about which lines of code have been executed. We developed Aletheia [10] to instrument the code and prepare data for clustering.

Step 2: Generating Failure Tree

We 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. We utilize hierarchical clustering to generate a dendrogram of failing tests. We use execution profiles generated in the previous step, as our feature sets for clustering.

Step 3: Cutting the Failure Tree by Fault Localization

Like any other clustering application, the next question is regarding the best number of clusters. In our case, in an ideal solution, the number of clusters equals the number of underlying fault which we do not know a priori. Therefore, we need a strategy to predict the number of clusters. We use Spectrum-Based Fault Localization (SBFL) (see Sect. 3) to find the best number of clusters k, or the cutting point, of the dendrogram. Liu and Han [19] as well as Jones et al. [13] suggest that if the failures in two clusters identify the same entities as faulty entities, they most likely failed due to the same reason and should be merged into one cluster.

This process can be considered in a top-down manner. This technique computes the *fault localization rank* for the children of a parent to decide whether the parent is a better cluster or it should be divided into its two children. The result of fault localization is a ranked list of entities from the most to the least suspicious. To check similarity between ranked lists, we use Jaccard [24] set similarity as suggested by [13], defined on two sets A and B as follows.

$$Similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (1)$$

If the similarity of the fault localization rank of two children is smaller than a predefined threshold, they are (likely) pointing to different faults. They are dissimilar and should not be merged. Thus, the parent cluster is not a good stopping point and should be divided into its children. Otherwise, the parent is a better cluster and this is the stopping point for clustering. According to Fig. 1, the first step is to decide whether dashed line 1 is a better cutting point or dashed line 2. To answer this question, the fault localization rank at cluster c2 is compared to fault localization rank at cluster c3. If the similarity between these two sets is larger than the predefined threshold, they are similar and the parent c1 is a better clustering than dividing it into two clusters c2 and c3. As the result, line 1 is the cutting point. If line 1 is not the cutting point, the process continues to the point that no more division is needed. Because of the good results in our large-scale experiments [9], we propose 0.85 as the similarity threshold.

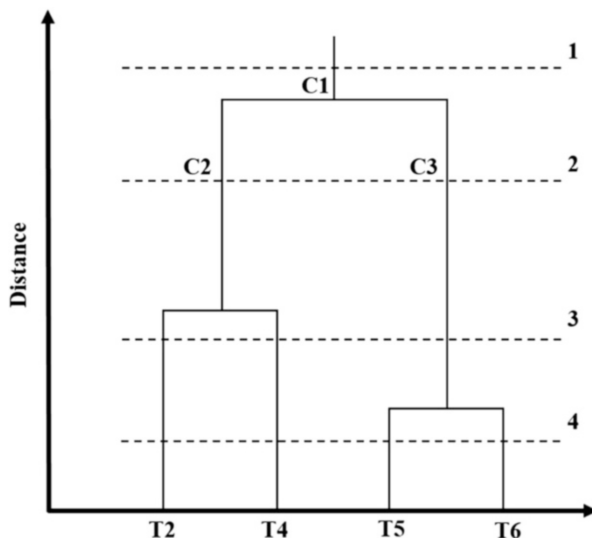

Fig. 1 Hierarchical clustering of four failing tests

Step 4: Selecting Representative TCs

We have already grouped TCs based on their hypothesized root causes by generating the failure tree and cutting it into clusters. Now, we need to suggest a representative for each cluster. Developers investigate only the representatives to find all the faults. Since it is likely that clustering is imperfect, the selection of representatives for a cluster has a great importance: We are aware that clusters are unlikely to be 100% pure [5]. We require our solution to make the representatives reliable. To avoid selecting an outlier (a failure which does not belong to the fault class that has the majority in the cluster) as a representative in clustering, we hence calculate the center of the cluster and find the k-nearest neighbors (KNN) [3] to the center. These KNNs are selected as representatives of the respective cluster. KNN search finds the nearest neighbors in a set of data for each data point. Based on discussion with test engineers, we suggest $k = 1$.

2.1.2 Failure Clustering Without Coverage

In previous section, we explained our first clustering approach that uses the coverage profile of tests as the input for clustering. However, 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 Hardware-in-Loop tests). Second, in this approach, collecting coverage information when running passing tests is also needed. Sometimes, this requirement imposes extra work on the system.

Third, instrumenting very large projects when running integration tests can be very expensive and time-consuming. In this section, we propose a clustering technique to group failing tests based on *noncoverage data*, retrieved from three different sources. These data sources make the clustering approach applicable in different stages of testing and other purposes such as test prioritization.

In this approach, first, we collect the data from different non-code-based sources, for example, Jira tickets to make a feature vector for each TC. 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 developers receive the list of representatives to investigate them. If the suggested number of clusters appears to the developers to be inaccurate, they can immediately adjust the number of clusters on the user interface and get new representatives. Steps two and four are similar to the first clustering approach. In the following, we describe all the steps in detail.

Step 1: Collecting the Input Data

Two main input data sources are: the database of test results that includes several thousand test results from the previous test runs, and the repository of the TCs, providing the source files for the tests. We can extract three sets of features (variables in a data set) using these two data sources. Since the primary objective 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 unit in an industrial setting (e.g., a single ECU in a car company). Each test run is usually called a *build*. All the feature values are extracted individually for each build. We explain each set of features in the following.

General Features The following features can be extracted from the database [18]: general information about the test; that is, its source file, component, domain, and the hardware that executes the test (if any).

If features are of categorical nature and do not follow an ordinal scale, we transform them into binary data.

Jira History We use 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 [18]. Table 1 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 2.

Table 1 Jira history [18]

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 2 Binary Jira History

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

Test Case Similarity The files used to generate TCs are usually 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 source files [18]. To facilitate the calculation of similarity between two files, we compare the high level steps taken in each test.

Input Feature Weights We extract three set of features as input for clustering. These three sets lead to the generation of three different data sets. We measure the distance between TCs using all these three sets. Then, to have an aggregated distance value, we assign weights to each group and sum up the distance values. Based on our experience [7], test similarity, general features, and Jira history are almost equally important and therefore can be assigned similar weights, as shown in the following equation:

$$d_{aggregate}(x, y) = 0.31 * d_{general}(x, y) + 0.35 * d_{jira}(x, y) + 0.34 * d_{testFile}(x, y) \quad (2)$$

Step 2: Generating Failure Tree

Similar to the first approach, we apply hierarchical clustering on collected data.

Step 3: Cutting the Failure Tree by Fitting a Regression Model

In this approach, since we do not have coverage profile of tests, we cannot utilize FL to predict the number of clusters. Therefore, we propose using polynomial regression [16] to examine the relationship between the number of failing tests and the cutting distance on the hierarchical tree which basically shows the number of clusters. To this end, we extract the real number of faults in the previous analyzed builds from the database. Then, we calculate the cutting distances of the respective

trees. Finally, we fit the regression model to predict the cutting distance based on the number of failing tests. Figure 2 illustrates an example. However, a model should be fitted for each specific case.

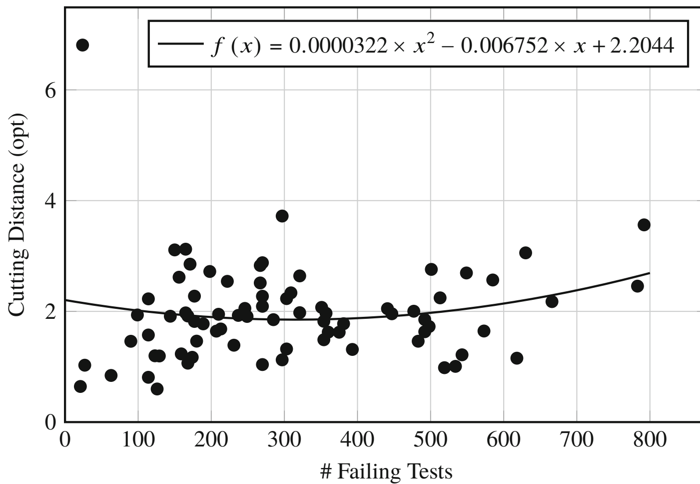

Fig. 2 Polynomial regression model for cutting distance[18]

Step 4: Selecting Representative TCs

Similar to the first approach, the nearest neighbor to the center of the cluster is considered as the representative of the cluster. Since we want the developers to be able to change the number of the cluster in real time, we pre-compute the representatives for all the clusters considering all possible cutting distances.

2.2 Industry Impact

Our large-scale evaluations in [9] and [7] show that using first and second clustering approaches, we are able to reduce more than 80% and 60% of the failure analysis time, respectively. Analyzing only the representative tests, we discovered all the underlying faults in the first approach and more than 80% of the faults using the second approach. These numbers show that our clustering approach is an effective way to reduce failure diagnosis time in an industrial setting.

The second takeaway is that if the source code is available and the highest accuracy is required, one can use approach one; if short response time is important and some level of inaccuracy is tolerable, one can use approach two.

3 Fault Localization

One of the most popular subsets of automated FL techniques is spectrum-based fault localization, known as SBFL [38]. In order to correlate program elements with failing TCs, these techniques are built upon abstractions of program execution traces, also known as program spectra, executable statement hit spectra, or code coverage [38]. These program spectra can be defined as a set of program elements covered during test execution. The initial goal of SBFL techniques is therefore to identify program elements that are highly correlated with failing tests [21]. In order to determine the correlation between program elements and TC results, SBFL techniques utilize ranking metrics to pair a suspiciousness score with each program element, indicating how likely it is to be faulty. The rationale behind these metrics is that program elements frequently executed in failing TCs are more likely to be faulty. Thus, the suspiciousness score considers the frequency at which elements are executed in passing and failing TCs. Some of the more popular ranking metrics have been specifically created for the use in FL, such as Tarantula [14] and DStar [37], whereas others have been adapted from areas such as molecular biology, which is the case for Ochiai [23]. DStar, Ochiai, and Tarantula are three of the most popular and best-performing metrics in recent studies [28].

$$DStar = \frac{(N_{CF})^{*}}{N_{UF} + N_{CS}},$$

$$Tarantula = \frac{\frac{N_{CF}}{N_f}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}},$$

$$Ochiai = \frac{N_{CF}}{\sqrt{N_F * (N_{CF} + N_{CS})}},$$

where N_F is the number of failing tests, N_S is the number of passing tests, N_{CF} is the number of failing tests that cover the element, N_{CS} is the number of passing tests that cover the element, N_{UF} is the number of failing tests that do not cover the element. DStar metric takes a parameter *. The nominator is then taken to the power of *. There is no significant difference between these metrics [28].

An example of a hit spectrum is shown in Table 3. Each • in the table means that the respective element "e" (can be at different levels of granularity, e.g., statement, method, basic block, etc.) was hit in the respective test run "t". Table 4 shows the suspiciousness scores and ranks of program elements in Table 3 using Ochiai metric. As the ranks indicate, element $e4$ is the most suspicious element.

A study on developers' expectations on automated FL [17] shows that most of the studied developers view FL process as successful only if it:

- Can localize faults in the Top-10 positions
- Is able to process programs of size 100,000 LOC
- Completes its processing in less than a minute
- Provides rationales of why program elements are marked as potentially faulty

Table 3 A hypothetical hit spectrum

Element	Test cases					
	t1	t2	t3	t4	t5	t6
e1	•	•	•	•	•	•
e2		•	•	•	•	
e3		•	•	•	•	
e4	•	•			•	•
e5		•				
Verdict	F	P	P	P	P	F

Table 4 Ochiai suspiciousness scores

Element	Suspiciousness	
	Score	Rank
e1	0.577	2
e2	0	3
e3	0	3
e4	0.707	1
e5	0	3

Considering these expectations, real-world evaluations [28] show that SBFL techniques are not yet applicable in practice. They are able to process large-size programs, but are not always able to locate the faults in top positions. This might be the consequence of considering the correlation, not causation. Although the goal of any FL technique is "to identify the code that caused the failure and not just any code that correlated with it" [21], SBFL techniques measure the correlation between program elements and test failures to compute suspiciousness scores. Thus, they do not control potential confounding bias [27]. Confounding bias is a distortion that modifies an association between an exposure (execution of a program element) and an outcome (program failure) because a factor is independently associated with the exposure and the outcome (see Sect. 3.2).

In addition, for SBFL techniques, the granularity of the program elements in the program spectra is important, not only to the effectiveness of the system but also to the preferences of developers [17]. Kochhar et al. found that among surveyed developers, method, followed by statement and basic block were the most preferred granularities. But when it comes to the effectiveness of the system, the method and statement granularities may be too coarse- or fine-grained, respectively, to properly locate the faulty program elements [21, 26]. Unfortunately, there is no golden rule to say which granularity is the best for all contexts.

Despite ongoing research and improvements, the real-world evaluations show that FL techniques are not always effective. Considering above-mentioned gaps, we suggest the following improvements on SBFL.

3.1 Syntactic Block Granularity

As mentioned previously, the two main program spectra granularities used by practitioners are statement and method [17]. Unfortunately, neither of these options are perfect, as they both have their limitations.

Due to its fine-grained nature, the statement granularity has a number of drawbacks. First, simple profile elements like statements cannot properly characterize and reveal nontrivial faults. Statements might be too simple to describe some complex faults, such as those that are induced by a particular sequence of statements [21]. In the Defects4J data set [15], which is the largest available database of Java faults, also the most used one for FL studies, the median size of a fault is four lines, with 244 bugs having faults spread across multiple locations in the program [33]. Furthermore, due to the nature of the statement granularity, it is incapable of locating bugs due to missing code, known as a fault of omission [38]. For example, of the 395 bugs in the Defects4J data set, only 228 can be localized by the granularity of statements.

Unfortunately, it is unclear whether developers can actually determine the faulty nature of a statement by simply looking at it, without any additional information [26, 38]. As a possible solution to the drawbacks of the statement granularity, Masri suggests the usage of more-complex profiling types with higher granularity [21]. Previous empirical studies have shown that the effectiveness of SBFL techniques improves when the granularity of the program elements is increased [21]. For that reason, among others, many practitioners prefer to use the method granularity.

However, due to its coarse-grained nature, the method granularity has a handful of drawbacks when used for calculating SBFL scores. Sohn and Yoo suggest two drawbacks to the method granularity due to the nature of methods themselves [34]. First, methods on a single call chain can share the same spectrum values, resulting in tied SBFL scores. Second, if there are TCs that only execute non-faulty parts of a faulty method, they will decrease the overall suspiciousness score of the given method. Furthermore, when given a list of methods ranked by their suspiciousness, a programmer would still have to walk through all the statements in each method while looking for the bug, which can result in a lot of wasted effort, especially if the methods are large. Finally, the method granularity also lacks any sort of context and may not provide any further information to the developer. For instance, if there are failing TCs that focus on testing one specific method, such as a unit test, the developer will already know that the fault is contained within the failing method, so the method granularity results are of no additional help.

As both the statement and method granularities exhibit drawbacks, there is a clear need for a new granularity level that has a higher granularity than statements, without the added wasted effort and lack of context of methods. As a possible solution, we propose the usage of the *syntactic block granularity*. Based on different syntactic components found in the program's source code (see Table 8 for syntactic blocks in Java), it considers a wide range of program elements in an effort to provide more context to the developer with minimal added cost.

Fig. 3 Faulty *mid*() method. Each syntactic block is enclosed by a box

```
 1: public int mid(int x, int y, int z) {
 2:     int mid = z;
 3:     if (y < z) {
 4:         if (x < y) {
 5:             mid = y;
 6:         }
 7:         /* FAULT: missing the following
 8:         else if (x < z) {
 9:             mid = x;
10:         }
11:         */
12:     }
13:     else {
14:         if (x > y) {
15:             mid = y;
16:         }
17:         else if (x > z) {
18:             mid = x;
19:         }
20:     }
21:     return mid;
22: }
```

To illustrate the drawbacks of the statement and method granularities, as well as highlight the benefits of the syntactic block granularity, consider the sample program in Fig. 3. The method *mid*() takes as input three integers and outputs the median value. The method contains a fault of omission, where the proper implementation should include the else-if block from lines 8–10. Tables 5, 6, and 7 contain the coverage information from a test suite containing six different TCs for each of the three different granularities. In each table, each TC corresponds to a column, with the top of the column corresponding to the inputs, the black dots corresponding to coverage, and the status of the test at the bottom of the column. To the right of the test case columns are the Ochiai score [1] and the corresponding rank for each element.

As mentioned previously, it is not possible to localize a fault of omission using the statement granularity. Therefore, no SBFL technique using the statement granularity will be able to localize the fault in the *mid*() method.

To localize the fault using the method granularity, the only information provided to the developer is that the fault is contained in the *mid*() method (see Table 6). However, due to the unit test nature of the TCs, this fact is obvious. A developer would still have to go through all the statements in the method to find the fault.

Table 5 *mid*() Statement granularity Ochiai calculations

Line #	Test cases						Ochiai	Rank
	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1	•	•	•	•	•	•	0.577	2
2	•	•	•	•	•	•	0.577	2
3	•	•	•	•	•	•	0.577	2
4	•	•			•	•	0.707	1
5		•					0	6
13			•	•			0	6
14			•	•			0	6
15			•				0	6
17				•			0	6
18							0	6
21	•	•	•	•	•	•	0.577	2
Verdict	F	P	P	P	P	F		

Table 6 *mid*() Method granularity Ochiai calculations

Line #'s	Test cases						Ochiai	Rank
	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1–22	•	•	•	•	•	•	0.577	1
Status	F	P	P	P	P	F		

Table 7 *mid*() Syntactic block granularity Ochiai calculations

Block type	Line #'s	Test cases						Ochiai	Rank
		3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
MD	1–22	•	•	•	•	•	•	0.577	3
ICS	3	•	•	•	•	•	•	0.577	3
ITB	3–12	•	•			•	•	0.707	1
ICS	4	•	•			•	•	0.707	1
ITB	4–6		•					0	5
IEB	13–20			•	•			0	5
ICS	14				•	•		0	5
ITB	14–16			•				0	5
ICS	17					•		0	5
ITB	17–19							0	5
Verdict		F	P	P	P	P	F		

As seen in Table 7, when using the syntactic block granularity, the fault is localized to the then block of the if statement (*ITB* block) from lines 3–12 with a rank of 1. As an improvement over the method granularity result, the developer would only have to look through one portion of the method to find the fault. Furthermore, the developer would have a further intuition as to the location/type of fault that exists. Due to the average depth of faults within the program elements

of the syntactic block granularity, the developer would expect the fault to likely be an issue with the direct children elements of the block, or the block itself. In this case, the faulty missing element would in fact be a direct child of the top ranked *ITB* block from lines 3–12. Furthermore, due to the 0 suspiciousness score of the *ITB* block from lines 4–6, the developer can infer that the fault is either with the if statement conditional at line 4 or is a fault of omission.

This added information provided by the syntactic block granularity, as well as the reduced number of statements required to search through to localize the fault, helps illustrate the benefits of the syntactic block granularity over the other two granularities.

Extracting Syntactic Blocks

Each syntactic block consists of a set of statements that syntactically belong together to form a program element. For instance, every statement in a method declaration belongs together, as whenever the method is called, the contained statements may be run. Furthermore, each element may further be broken into multiple sub-elements. For example, an if statement has three components: the *condition statement*, the *then block*, and the *else block*. Each of these sub-elements may be run separately from each other. For instance, the *condition statement* will always be executed, but depending on the Boolean value of the conditional, either the *then block* or the *else block* will be executed. As a result, for Java programs, the syntactic block granularity consists of the 18 different types of program elements found in Table 8. For ease of use, each syntactic block type has an ID associated with it. An example of each type of syntactic block can be found in bold in the last column in Table 8.

Like the method granularity, for each syntactic block, if any of the contained statements are executed, the syntactic block is also marked as executed. Originally, *Class*, *Interface*, and *Enum* declarations were also considered as types of syntactic blocks. However, due to their average size compared to all other types of blocks, the added benefit of encompassing class level faults (such as missing method declarations or incorrect class variables) was outweighed by the overall added wasted effort associated with inspecting whole classes for a fault.

Due to the hierarchical nature of syntactic blocks, it is possible for one block to completely encapsulate another. For example, in Fig. 3 the *ITB* block from lines 3–12 encapsulates the *ICS* block at line 4 and the *ITB* block from lines 4–6. Because of this, sections of code will appear multiple times as a programmer walks through the ranked list of elements. In order to prevent unnecessary work, any block completely encapsulated by another block with a higher suspiciousness score can be ignored and removed from the final ranking.

While our work focused on faults in Java programs, the concept would be similar for other programming languages with similar syntax, for example, C, C++, C#, Go, PHP, and Swift.

Table 8 Java syntactic block types

Base program element	Syntactic block type	Type ID	Example
–	Constructor declaration	CND	`class X { X() { ... } }`
–	Method declaration	MD	`public int foo() { ... }`
–	Initializer declaration	IND	`static { a = 3; }`
–	Do statement	DS	`do { ... } while (a == 0);`
Foreach statement	Condition statement	FECS	`for (Object o : objects) { foo(); }`
	Body block	FEBB	`for (Object o : objects) { foo(); }`
For statement	Condition statement	FCS	`for (int a = 3; a <10; a++) { foo(); }`
	Body block	FBB	`for (int a = 3; a <10; a++) { foo(); }`
If statement	Condition statement	ICS	`if (a == 5) foo() else bar();`
	Then block	ITB	`if (a == 5) foo() else bar();`
	Else block	IEB	`if (a == 5) foo() else bar();`
Switch statement	Statement	SS	`switch(a) { ... }`
	Entry statement	SES	`case 1: foo(); break;`
Try-catch statement	Try block	TCTB	`try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }`
	Catch block	TCCB	`try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }`
	Finally block	TCFB	`try { foo(); } catch (Exception e) { bar(); } finally { x = 0; }`
While statement	Condition statement	WCS	`while (a > 0) { bar(); }`
	Body block	WBB	`while (a > 0) { bar(); }`

3.2 Re-ranking Program Elements

Using Fig. 4, we explain an example of confounding bias in SBFL results. The code snippet indicates a hypothetical faulty program. Assume that a fault in method F1 propagates only through the left branch where method F2 is triggered, while the right branch, where method F3 is called, executes correctly. Put differently, although F1 contains a fault, only those tests taking the left branch are failing. In this case,

an SBFL technique gives the highest suspiciousness score to the method F2, since it is executed more frequently in failing executions and less frequently in passing executions (F1: 1 failing, 1 passing, F2: 1 failing, and F3: 1 passing). However, method F1 is the faulty element.

```
public void F1(int i) {
  if (i < 0) {
    ... // Faulty
    F2(x);
  } else {
    ...
    F3(x);
  }
}
```

Fig. 4 A hypothetical faulty method with two branches

"Confounding bias happens when a seeming causal impact of an event on a failure may be in fact due to another unknown confounding variable, which causes both the event and the failure" [21]. Given a program and a test suite, assume that all failing TCs induce dependence chain (chain of executed program elements in our case) $e_1 \rightarrow e_2 \rightarrow e_{bug} \rightarrow e_3 \rightarrow e_4 \rightarrow e_{fail}$ and all passing TCs induce $e_1 \rightarrow e_2$ only, where e_{bug} indicates the execution of faulty element and e_{fail} indicates a failure. A correlation-based approach such as SBFL would assign the same suspiciousness score to any of e_{bug}, e_3, or e_4, thus resulting in two false positives, whereas a causation-based approach that considers dependencies to have causal effect would assign e_4 the lowest suspiciousness score and e_{bug} the highest suspiciousness score. This means, when computing the suspiciousness scores, the confounding bias to consider for e_4 would involve e_3 and e_{bug}, for e_3 it would involve e_{bug}, and no confounding is involved when computing the suspiciousness score of e_{bug} [21].

In our analysis on SBFL results, we observed that often if the most suspicious element s* does not contain a fault, one of its parents or grandparents contains a fault, as shown in Fig. 4. Method F2 is the most suspicious element, while its parent F1 contains a fault. To improve SBFL effectiveness, we propose a re-ranking strategy based on this observation.

In this approach, we augment SBFL with a combined dynamic call and data-dependency graphs of failing tests. First, using any similarity metric such as DStar, we find the most suspicious method s* of the program. It has rank 1 on the suspiciousness ranking list. We locate it on the combined dynamic call graph of the failing tests and list all of its parents and grandparents. Then, we inject this list between rank 1 and 2 of the ranking list and re-rank all the elements accordingly. The re-ranking approach gives the second rank to the parents (if exists) of s* and the third to its grandparents. We start inspecting the suspicious elements based on the newly ranked list. Visual representation of call graph while highlighting the most

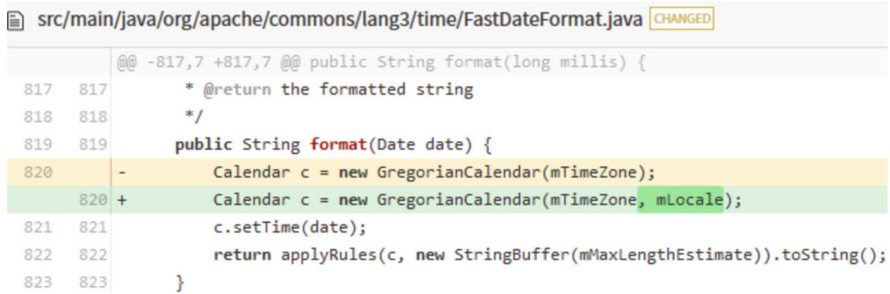

Fig. 5 Human patch to fix Lang-26 [4]

suspicious elements on it aids users in better understanding the problem. In the following, we use a real fault, Lang-26, from Defects4J database as our motivating example to explain each step. This bug is a wrong method reference that causes one test to fail [33] (Fig. 5).

Step 1: Finding the Most Suspicious Method(s) Using SBFL
As mentioned earlier, we arbitrarily use DStar (*=4) to find the most suspicious method in the first step. DStar calculations on Lang-26 spectrum places "lang3.time.FastDateFormat-TextField-1171" and "lang3.time.FastDateFormat-StringLiteral-1130" methods at rank **1**, as the most suspicious methods. Method "lang3.time.FastDateFormat-820" which is the faulty method gets rank **17**.

Step 2: Locating the Most Suspicious Method(s) on the Dynamic Call and/or Data-Dependency Graph
In the second step, we generate a graph which includes dynamic method calls and/or explicit data-dependencies of all failing tests. A dynamic call graph is generated at runtime by monitoring the program execution. The graph contains nodes that indicate the executed methods and edges between methods that represent method calls. We consider dynamic call graph to inspect real, not potential (as is the case in static call graphs), dependencies. An explicit data-dependency graph indicates dependencies between program elements introduced by a common variable used in multiple program elements. A data-dependency exists when two program elements exchange data using a variable. This happens when one program element writes to a field, and another element reads that field later. The result is a data-dependency between the first and second elements.

Considering Fig. 6, a call graph contains only the solid lines. A data-dependency graph contains only the dashed line. A combined graph can be helpful in SBFL. If method *compute* is faulty, method *getResult* will also return a wrong result. Thus, it will be labeled as a suspect. Looking into the combined graph, one can improve the labeling by adjusting for the confounder.

Figure 7 shows the call graph of Lang-26 failing test. Due to space constraints, it is only depicting the left branch. Thick red boundaries highlight the most suspicious methods. All nodes are annotated with their ranks.

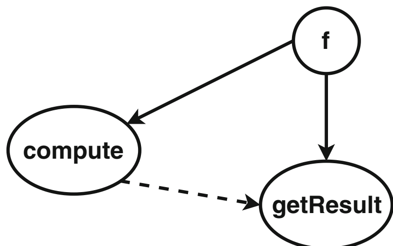

Fig. 6 Combined call/data-dependency graph

Step 3: Re-ranking Program Methods

In step 3, we find the parents and grandparents of the DStar's most suspicious methods. Then, we inject them between rank 1 and rank 2 and change all the ranks accordingly. Parents get rank 2 and grandparents get rank 3. In our example, "lang3.time.FastDateFormat-888" is the parent and "lang3.time.FastDateFormat-820" is the grandparent. Thus, in our new list, we place them right after ranked 1 methods and change their ranks to 2 and 3, respectively. Users get the new ranking list. As annotations on Fig. 7 show, the rank of faulty method "lang3.time.FastDateFormat-820" changes from **17** to **3** which is a considerable improvement.

3.3 Evaluation

Our evaluation results in [6] show that using our proposed re-ranking techniques, we can improve the average effectiveness of SBFL to 73.5% when it comes to locating faulty elements in Top-10 ranks. This means, in 75.3% of the cases, SBFL with re-ranking listed the faulty element in top ten ranks.

In addition, using syntactic blocks, we add context to SBFL results, provide additional insight into the possible location of the fault, and cover more types of faults than both popular statement and method granularities. Syntactic block granularity exhibits ranking behavior similar to the method granularity, while having a wasted effort (# lines that have to be inspected before finding the fault) equivalent to, if not better than, the statement granularity. Finally, when compared to the method granularity, it exhibits up to a 92.48% improvement when it comes to the locality of the program elements to the fault, a characteristic that provides the user with a better insight into the possible location of the faults. When inspecting program elements containing multiple statements, it is important to be able to have some insight as to which statements are more suspicious than others. For example, when inspecting a method for a fault, it would be helpful for the user to know where to start looking for the fault, instead of having to walk through each statement one-by-one. By knowing certain characteristics of the program elements in each granularity, a user may be able to localize the fault easier. One possible characteristic to consider would be the proximity of the most suspicious faulty program element

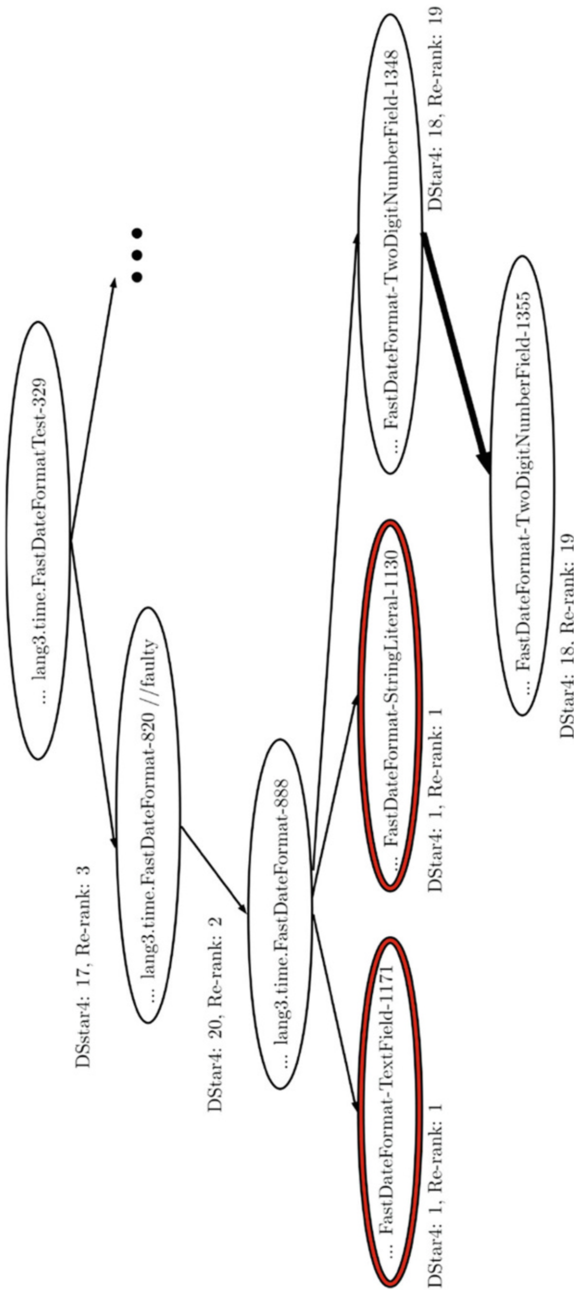

Fig. 7 Left branch of Lang-26 call graph. Numbers in the nodes indicate the line numbers of methods in the source code

e* to the fault itself. If the average depth of the fault in the abstract syntax tree (AST) of the program element e* is low, it will be easier to find the fault, as the user could work their way down the AST, giving higher priority to the shallower elements.

However, we observed an issue when it comes to its application in industrial domains: the effectiveness of SBFL varies from case to case. It is not clear for a user whether using SBFL would help in reducing analysis time or waste time. Therefore, to continue our endeavor in developing practical solutions, first we need to understand what is the reason behind this observation. Why is the effectiveness of these techniques so unpredictable? What are the factors that influence the effectiveness of fault localization? Can we accurately predict fault localization effectiveness? Answering these questions can help in two ways: (1) We would know how to improve the code to facilitate SBFL. (2) We would be able to predict the effectiveness of SBFL. This helps users in deciding where and when to use it to avoid wasting time and money.

3.4 Predicting the Quality of SBFL

We tried to shed light on the observations above and understand what factors explain such variations. Knowing influencing factors and being able to predict the effectiveness of SBFL can improve the user's trust. If SBFL is expected to be bad, we don't use it—and the other way around. Doing this will not always help programmers, but it doesn't frustrate them either with bad predictions of the fault location.

To learn the aspects of the projects with the strongest influence on fault localization effectiveness, we investigated a large number of potential factors affecting SBFL effectiveness and built models, using standard machine learning techniques and a set of carefully selected metrics, to predict effectiveness. We wanted to determine whether we can build a model, using collected metrics as features, that is accurate enough to be used for SBFL effectiveness predictions in practice.

We grouped metrics into three groups according to the source of information they are based on. Metrics were considered potentially relevant when we had a hypothesis about why they could influence SBFL effectiveness. Considering code metrics, a metric had to be actionable as well, meaning that a developer could, using an appropriate programming style, improve its value. Static metrics measure the static aspects of the whole source code. Dynamic metrics measure the dynamic aspects considering the test runs. Test suite metrics are correlated to the test suite. In total, we collected 46 metrics and computed them for the buggy versions of five Defects4J projects. Tables 9, 10, and 11 illustrate the collected metrics.

Based on the metrics defined above, we need to predict if SBFL is effective and define class labels to predict. To define effectiveness, we relied on method-level Ochiai results. If the rank of the faulty method was between 1 and 10, we labeled SBFL as "effective" or "1", otherwise "ineffective" or "0". Using collected metrics as variables or features and SBFL effectiveness as labels, we formed a data set. We

Table 9 Static metrics

ID	Metric	Definition
S1	PML	% of methods with LoC>30
S2	PHFS	% of files with LoC>750
S3	PMFS	% of files with 300<LoC<750
S4	PHND	% of methods with nesting depth>5
S5	PMND	% of methods with 3<=nesting depth<=5
S6	CC	Mean cyclomatic complexity [22]
S7	MCC	# of methods with 10<CC<20
S8	AFFC	Mean afferent coupling [2]
S9	EFFC	Mean efferent coupling [2]
S10	ABKD	Mean block depth
S11	ADIH	Mean depth of inheritance hierarchy
S12	ALOCPM	Mean # of LoC per method
S13	ANOCPT	Mean # of constructors per type
S14	ANOFPT	Mean # of fields per type
S15	ANOMPT	Mean # of methods per type

applied regression analysis and classification algorithms on the data set with the goal to find the most relevant and influential metrics. A model generated using influential metrics helps decide whether or not to proceed with fault localization, thus avoiding misleading recommendations and waste of time.

Our evaluation results in [8] show that the best model is based on random forest; combines 15 static code, dynamic execution, and test suite metrics; and shows an excellent discrimination power (AUC = 0.88). The most influential metrics are: four static metrics (% Methods with LoC>30, % Methods with Nesting Depth>5, % Methods with 3<=Nesting Depth<=5, Mean # of Fields per Type), four dynamic metrics (Mean Node Degree, Max. Node Out-Degree, Graph Diameter, Response for Class), and two test metrics (% Method Coverage, % Methods Covered in Failing Tests). A slightly less accurate model (AUC = 0.87), based on only 10 metrics, relies on logistic regression. There is a considerable overlap of eight metrics which yield an AUC of 0.86 when using logistic regression and 0.87 when using random forest. These prediction models can be used in two ways: (1) to decide whether or not to proceed with fault localization and (2) to guide improvements in code quality and test suites.

4 Contribution and Limitation

We make the following contributions:

- **A failure clustering methodology**. We propose a failure clustering technique and a methodology for adapting the idea of debugging in parallel to a real context.

Table 10 Dynamic metrics

ID	Metric	Definition[a]
D1	VC	# of Nodes in call graph
D2	EC	# of Edges in call graph
D3	MAXVD	Max. node degree
D4	MVD	Mean node degree
D5	MAXVI	Max. node in-degree
D6	MVI	Mean node in-degree
D7	MAXVO	Max node out-degree
D8	MVO	Mean node out-degree
D9	MSND	Avg. start node degree
D10	GD	Graph diameter
D11	GR	Graph radius
D12	MGD	Mean geodesic distance
D13	VCON	Node connectivity
D14	ECON	Edge connectivity
D15	ClCo	Mean clustering coefficient [20]
D16	SCOV	% of statement coverage
D17	CBO	Coupling between objects [2]
D18	RFC	Response for class [2]

[a]D1: # methods. D2: # method calls. D3: max. # edges connected to a node. D5: max. # edges entering a node. D7: max. # edges leaving a node. D10: greatest distance between any two nodes. D11: min. distance between any two nodes. D12: # edges in a shortest path between any two nodes. D13: min. # nodes that must be removed to break all paths between two nodes. D14: min. # edges that must be removed to break all paths between two nodes. D15: measures degree to which nodes in a graph tend to cluster

Table 11 Test suite metrics

ID	Metric	Definition
T1	T	# of tests
T2	M	# of methods
T3	PPT	% of passing tests
T4	PFT	% of failing tests
T5	D	Density [29]
T6	G	Diversity [29]
T7	U	Uniqueness [29]
T8	DDU	Density × diversity × uniqueness [29]
T9	MatSpar	Matrix sparsity
T10	MetCov	% of method coverage
T11	COVPT	% of methods covered in passing tests
T12	COVFT	% of methods covered in failing tests
T13	AVGMV	Mean covered methods per test

- **A collection of data sources for failure clustering**. We introduce a list of coverage and noncoverage data that are useful in the clustering of failing tests.
- **A new data granularity for failure diagnosis**. We propose a new granularity for program spectra called the syntactic block granularity which considers 18 different types of program elements.
- **A new ranking strategy**. We propose a ranking approach for SBFL techniques which leverages dynamic call and data-dependency graphs of failing executions.
- **A model to predict the effectiveness of SBFL**. We introduce a set of metrics which influence the effectiveness of SBFL. Using these metrics, we build a model to predict the effectiveness of SBFL. This model can be helpful in facilitating fault localization as well.
- **A tool-chain for failure diagnosis**. We introduce a tool-chain for failure diagnosis which puts failure clustering and fault localization in a pipeline.

However, we are aware that our work has its limitations. The difficulty of gathering data for evaluation prevented us from reporting on several case studies. We evaluated our solution ideas on either C++ programs or Java programs. However, a comprehensive evaluation should contain benchmarks from both languages. Comparing the results, finding similarities, and understanding dissimilarities can help us better understand the important factors in general and in each category.

In our fault localization evaluations, we assumed that each buggy version contains only one bug. However, this is not the case in practice. Although we suggest that before any fault localization attempt, one should do clustering on failing tests to segregate between faults, a comprehensive evaluation should also consider cases with multiple bugs.

5 Summary and Outlook

We proposed techniques and enhancements to facilitate failure diagnosis experience for developers. The proposed techniques can be used stand-alone or form a tool-chain; applying SBFL on fault-focused clusters yields more accurate results. The following is how developers and testers can benefit from using our proposed approaches.

Tester X is responsible for the quality of software development and deployment. She is involved in performing automated and manual tests to ensure the software created by developers is fit for purpose. Every week, she runs 1000 scheduled tests. Test runs take about 2 days. While running tests she collects coverage information. She has also provided a database of relevant data to measure the similarity between TCs.

After each test run, developers have two days to analyze failures, find bugs, repair them, and mark the analyzed failed tests as ready for the next test run. Thus, tester X collects failing tests and clusters them. The clustering results include a prediction of the numbers of bugs, groups of failing tests that are failing because of

the same reason, and one representative test for each cluster of failing tests. Tester X, then, assigns each group of failures to one developer and asks them to analyze the representative failing tests to find the reasons behind failures as soon as possible.

Developers Y and Z use SBFL prediction model. Developer Y receives promising results. She continues using SBFL techniques to get some insight regarding the fault in her code. Developer Z, on the other hand, does not receive promising results. She continues the debugging process without using SBFL. However, later she uses the model to improve the quality of her code to facilitate fault localization.

Note Automated failure diagnosis techniques can be very helpful in reducing failure analysis time regardless of the programming language. However, a minimum code and test quality is needed. If software is of poor quality, not only our tool-chain but any other tool or technique cannot be highly effective on it. Thus, there is a prerequisite for using such tools and techniques.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proceedings—Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007 (2007)
2. Bundschuh Manfred, D.C.: Object-Oriented Metrics, pp. 241–255. Springer, Berlin (2008)
3. Cover, T., Hart, P.: Nearest neighbor pattern classification. IEEE Trans. Inform. Theory **13**(1), 21–27 (1967)
4. Defects4J Dissection. http://program-repair.org/defects4j-dissection/#!/bug/Lang/26. Accessed: 2021-10-05
5. DiGiuseppe, N., Jones, J.A.: Fault density, fault types, and spectra-based fault localization. Empir. Softw. Eng. **20**, 928–967 (2015)
6. Golagha, M.: A framework for failure diagnosis. Ph.D. Thesis, Technical University of Munich, Germany (2020)
7. Golagha, M., Lehnhoff, C., Pretschner, A., Ilmberger, H.: Failure clustering without coverage. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019 (2019)
8. Golagha, M., Pretschner, A., Briand, L.: Can we predict the quality of spectrum-based fault localization? pp. 4–15 (2020)
9. Golagha, M., Pretschner, A., Fisch, D., Nagy, R.: Reducing failure analysis time: an industrial evaluation. In: Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP ’17, pp. 293–302. IEEE Press, Piscataway (2017)
10. Golagha, M., Raisuddin, A.M., Mittag, L., Hellhake, D., Pretschner, A.: Aletheia: a failure diagnosis toolchain. In: 2018 IEEE/ACM 40th International Conference on Software Engineering Companion (2018)
11. Hatton, L.: Characterising the diagnosis of software failure. IEEE Software—SOFTWARE (2008)
12. Hogerle, W., Steimann, F., Frenkel, M.: More debugging in parallel. In: Proceedings—International Symposium on Software Reliability Engineering, ISSRE, pp. 133–143 (2014)
13. Jones, J.A., Bowring, J.F., Harrold, M.J.: Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis—ISSTA ’07, p. 16 (2007)

14. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, pp. 273–282. ACM, New York (2005)
15. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering—FSE 2014, pp. 654–665 (2014)
16. Khuri, A.I.: Introduction to Linear Regression Analysis, 5th edn. In: Montgomery, D.C., Peck, E.A., Vining, G.G. (eds.) International Statistical Review (2013)
17. Kochhar, P.S., Xia, X., Lo, D., Li, S.: Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, pp. 165–176. ACM, New York (2016)
18. Lehnhoff, C.: Reducing failure analysis time: a data-driven approach. Master's Thesis, Technical University of Munich (2019)
19. Liu, C., Han, J.: Failure proximity: a fault localization-based approach. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pp. 46–56. ACM, New York (2006)
20. Luce, R.D., Perry, A.D.: A method of matrix analysis of group structure. Psychometrika **14**(2), 95–116 (1949)
21. Masri, W.: Chapter three—automated fault localization: advances and challenges, pp. 103–156. Elsevier, Amsterdam (2015)
22. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **SE-2**(4), 308–320 (1976)
23. Ochiai, A.: Zoogeographical studies on the soleoid fishes found in Japan and its neighbouring regions-II. Nippon Suisan Gakkaishi **22**, 526–530 (1957)
24. Pang-Ning, T., Steinbach, M., Kumar, V.: Introduction to Data Mining (2006)
25. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Upper Saddle River (1982)
26. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA '11, pp. 199–209. ACM, New York (2011)
27. Pearl, J.: Causality: Models, Reasoning, and Inference. Cambridge University Press, Cambridge (2000)
28. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering (2017)
29. Perez, A., Abreu, R., van Deursen, A.: A test-suite diagnosability metric for spectrum-based fault localization approaches. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 654–664 (2017)
30. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J.S.J., Wang, B.W.B.: Automated support for classifying software failure reports. In: 25th International Conference on Software Engineering, 2003. Proceedings, vol. 6, pp. 465–475 (2003)
31. Rogstad, E., Briand, L.C.: Clustering deviations for black box regression testing of database applications. IEEE Trans. Reliab. **65**(1), 4–18 (2016)
32. Rokach, L., Maimon, O.: Chapter 15—Clustering Methods. The Data Mining and Knowledge Discovery Handbook, p. 32 (2010)
33. Sobreira, V., Durieux, T., Delfim, F.M., Monperrus, M., de Almeida Maia, M.: Dissection of a bug dataset: anatomy of 395 patches from defects4j. CoRR abs/1801.06393 (2018)
34. Sohn, J., Yoo, S.: Fluccs: using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2017, pp. 273–283. ACM, New York (2017)
35. Trembly, A.C.: Software bugs cost billions annually. Nat. Underwriter/Life Health Financ. Serv. **106**(31), 43 (2002)
36. Utting, M., Legeard, B., Pretschner, A.: A taxonomy of model-based testing. Softw. Testing Verif. Reliab. **22**(April) (2006)

37. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2014)
38. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Trans. Softw. Eng. **PP**(99) (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.