

Using Evolutionary Coupling to Establish Relevance Links Between Tests and Code Units. A case study on fault localization.

Jeongju Sohn
jeongju.sohn@uni.lu

Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
Luxembourg

Mike Papadakis
michail.papadakis@uni.lu

Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
Luxembourg

ABSTRACT

Many software engineering techniques, such as fault localization, operate based on relevance relationships between tests and code. These relationships are often inferred through the use of dynamic test execution information (test execution traces) that approximate the link between relevant code units and asserted, by the tests, program behaviour. Unfortunately, in practice dynamic information is not always available due to the overheads introduced by the instrumentation or the nature of the production environments. To deal with this issue, we propose CEMENT, a static technique that automatically infers such test and code relationships given the projects' evolution. The key idea is that developers make relevant changes on test and code units at the same period of time, i.e., co-evolution of tests and code units reflects a probable link between them. We evaluate CEMENT on 15 open source projects and show that it indeed captures relevant links. Additionally, we perform a fault localization case study where we compare CEMENT with an existing Information Retrieval-based Fault Localization (IRFL) technique and show that it achieves comparable performance. A further analysis of our results reveals a small overlap between the faults successfully localized by the two approaches suggesting complementarity. In particular, out of the 39 successfully localized faults, two are common while CEMENT and IRFL localize 16 and 21. These results demonstrate that test and code evolutionary coupling can effectively support test and debugging activities.

1 INTRODUCTION

Software Engineering textbooks note that the majority of the total effort putted in a project during its life-cycle is during its maintenance phase [28]. Consequently, many researchers are focusing on automating software maintenance related activities such as automated testing, test data generation and automated debugging [6, 7, 9, 11, 33, 35]. These studies often rely on dynamic execution information, in order to identify links between tests and code. For instance, the studies on test selection often rely on test execution traces, to select tests that are likely to exercise the recently committed changes. Fault localization studies also adopt dynamic information to identify code locations that triggered test failures by narrowing down the code affected by a test failure [33].

While precise, dynamic test execution information such test coverage is not always available, mainly due to the difficulty and the cost of data collection [1, 17, 18, 26, 37]. For instance, test coverage, one of the most frequently used dynamic code analysis information, requires instrumentation, which may or may not be possible to perform in given environments. Even when the employed environment supports test coverage collection, this functionality is

often turned-off, as it introduces significant overheads in order to log the related information [1, 18, 26]. Furthermore, when developers are under a fast-release cycle, they are unlikely to have enough time for data collection at the first place [5, 18, 37].

To overcome the absence of dynamic coverage information, researchers have proposed static approaches that aim at guessing (predicting) the relations between tests and code [2, 3, 23, 24, 31]. While encouraging, the working assumptions of these approaches are not often met. For instance, Information Retrieval based Fault Localization approaches [13, 24, 31] assume the existence of bug reports, Focal method identification techniques [8] assume particular test patterns [34] and similar naming convention that can be captured by string-matching [23, 32, 36], assumptions that often are not met.

We fill this gap by proposing CEMENT (CoEvolution between Method and Test), a static technique that automatically infers test and code relationships given the projects' historical evolution. Thus, instead of relying on dynamic or static code analysis, CEMENT it relies on how software has evolved. Precisely, CEMENT establishes links, called Evolutionary Couplings [38], between tests and code units by checking the tests and code that have co-evolved throughout the development.

The key idea is that changes, on both tests and code, made around the same time imply a probable relevance coupling between them. Consider for example bug fixing cases, these are often followed or preceded by additions/modifications of tests (to reproduce and validate the repair action). Similarly, functionality additions are frequently followed by test additions. Therefore, the co-evolution analysis of tests and code units can capture a probable relevance coupling between them.

While there are many aspects of software evolution that could potentially be exploited, i.e, commit time, developer or the context of the evolution, we stay simple by focusing only on whether tests and code have been altered in similar periods of time (e.g., within few commits to each other). CEMENT differs from existing techniques as it is independent of the dynamic execution traces and the semantics of the source code. We posit that these differences allow test and code evolutionary couplings to complement existing techniques as they are capturing largely underexplored dependencies. Furthermore, we believe that co-evolution of tests and code units (i.e., methods) reveals important and hard to capture, by other techniques, couplings.

We empirically evaluate CEMENT's ability to infer links between tests and code by investigating its ability to select relevant tests (for given code methods) and to perform fault localization, i.e., successfully localize faulty methods given failed and passing test cases. We

thus, inject some faults/mutants, in essence applying mutation testing [19], on a set of selected methods and check the ability of CEMENT to select tests that detect (kill) them. Then we design a fault localization case study, where we investigate whether faulty methods have stronger links with failing tests than the passing tests. This means that we form a novel fault localization method, on top of CEMENT, and compare its performance with that of an existing Information Retrieval-based Fault Localization technique [14] that performs particularly well on the set of the projects that we study.

Our results, conducted on the 15 open source projects of Defects4J v.2.0.0 [10], show that CEMENT can infer relevant links between tests and code methods that can support software maintenance activities (e.g., fault localization) given the past co-evolution of the software.

In summary, the technical contributions of this paper are:

- The introduction of evolutionary coupling between tests and code, a novel type of coupling established based on how tests and code have co-evolved. Additionally, since the coupling between tests and code can be captured in a static way CEMENT offers advantages when dynamic information is not available.
- Empirical evidence that CEMENT can establish evolutionary couplings between tests and code. Results from a fault localization case study show that CEMENT can be useful in fault localization.
- Empirical evidence that evolutionary coupling between tests and code improves as software becomes more mature. The comparison between the CEMENT’s fault localization results for the projects with different levels of software maturity shows that CEMENT becomes more effective when the project under inspection has actively evolved (i.e., changed).
- Empirical evidence that fault localization using CEMENT complements state-of-the-art static fault localization techniques. Our results shows that CEMENT can localize faults for which an existing fault localization technique has failed, implying the relevant links captured by CEMENT can complement this technique.

2 EVOLUTIONARY COUPLING BETWEEN TESTS AND CODE

The key idea underlying our approach is that developers make focused changes to their projects. Instead of making multiple irrelevant actions, they focus on one action at the time [9]. Even if the changes are not serving one purpose they are closely related since they are the result of the developer focus/attention. This means that the changes committed around the same period are usually relevant to each other. For instance, when developers repair faults in code, they often introduce or modify a test to evaluate the repaired part. Similarly, when they implement new functionality or update existing ones, they probably introduce or update the tests related to this functionality, at the same time or shortly after. Even when some changes are irrelevant these should be eliminated through the number of evolutions/changes as it is unlikely to have the same irrelevant changes repeatedly. Based on these, we formulate the following hypothesis, which forms the main idea of our work:

Hypothesis: Changes in code units that are followed by the changes in tests (and vice versa) imply a relevance relationship (coupling) between them. Similarly, changes in code units not followed by changes in tests imply an absence of relevance.

We use the established term evolutionary coupling [38] to name the coupling between tests and code established from the above hypothesis, i.e., guided by co-changes or, in other words, the co-evolution of tests and code. This paper aims to investigate whether this evolutionary coupling between tests and code can be useful in software testing and debugging activities, more specifically, whether they can be used to fault localization. Hence, we propose CEMENT, a static approach that automatically infers links between tests and code units relying on evolutionary coupling. We work with methods since they form discrete and localizable units, typically targeted by automated techniques such as fault localization.

2.1 Tests and Code Evolutionary Coupling

Evolutionary coupling between tests and code exists when tests and code have co-evolved throughout the software development: it is independent of specific changes and simply focuses only on whether the changes in tests and code occurred in similar time periods. As a result, this coupling does not require any dynamic information to be established, hence being easier to exploit in some cases. In addition, by relying on the timing when each test and method change was made, the relation captured by the evolutionary coupling is independent of any software testing and debugging tasks, but reflects important links since they reflect the developer’s intent as developers tend to do them together. Another distinct characteristic of evolutionary coupling is that it is inferred from the projects’ evolution and thus, it evolves along with the target software. The more changes performed to the software, or the more mature the software is, the better. This feature can be very helpful in development cases following the continuous integration development model, as they tend to evolve both tests and code at the same time and include finer-grained commits. Additionally, such evolutionary couplings can provide developers up-to-date guidance on which code is linked to which tests and vice versa easing comprehension.

2.2 CEMENT

For a given set of methods and tests, CEMENT identifies evolutionary couplings, between each method and test pair, by computing the average time interval between their past changes. Before going into the details of CEMENT, we detail and show the distinct nature of tests and method coupling that CEMENT aims for, which we call *the coupling asymmetry*.

2.2.1 Asymmetry in the test and method coupling. In an ideal case where each test and method assesses and implements a unique functionality, the coupling between them is symmetric: a method is associated with a test at a degree equal to $x\%$, which is also equal to the degree the test associates to the method. However, in practice, we frequently observe cases where a single test exercises, directly or indirectly, (simply relates) to multiple functionalities. We also observe methods implementing multiple functionalities. For these

cases, the coupling between methods and tests becomes asymmetric, i.e., a test may relate with a method at a different degree than the method with that test. The degree of the association reflects the strength of the relation between entire tests and code methods. For example, let us assume that method m implements a functionality f that is a part of a more extensive functionality F ; test t examines the functionality F , indirectly evaluating the functionality f .

For method m , test t is the most related one, as it evaluates its main functionality, f . However, test t has a stronger degree of coupling to another method m' that carries out the entire functionality F , making the coupling between method m and test t asymmetric. Still, test t is the one to run when we need to inspect method m . CEMENT takes into account this asymmetry and computes a coupling degree separately for a method and for a test. It then aggregates these values from both test and method sides to estimate their final coupling degree.

2.2.2 Computing the distance between the test and method. CEMENT measures the degree of evolutionary coupling between the test and method as the time interval between their past changes. For the time interval between two changes, CEMENT counts the number of commits between them. Hereafter, we will refer to this time interval as the distance. There are two additional points that we need to consider when calculating this distance. First, tests and methods are likely to be altered more than once, especially when the software under inspection has evolved actively. Secondly, the obtained distance is inherently asymmetric since it quantifies the degree of asymmetric coupling.

Algorithm 1 presents the pseudo-code of computing the distance from target t to tc . Both t and tc can be either a method or a test. We denote as t a test and tc a method. Tests and methods are likely to be modified more than once from their introduction. Therefore, we first collect a list of commits that changed test t and method tc (Line 1 and 2). Here, we are interested in inspecting whether the test and method have been changed around the same time. Thus, we consider only the distance to the *nearest* method change for each test change rather than taking all the past changes of the method into account (Line 3 to 7). If the test and method are related to each other, their changes can trigger the changes or be triggered by the changes of the other party. Thus, we use the absolute distance while we search for the nearest method change. We then aggregate these distances computed for each past test change by taking the average (Line 8). By using the average, we can reduce the risk of being affected by the outlier case where the method and test were accidentally modified at the same time period.

The distance we define is asymmetric. For example, let us assume that method M was altered at the commits c_0 and c_3 and test T at the commits c_1 , c_3 , and c_4 . For the method changes at c_0 and at c_3 , the distance to the nearest test change is 1 and 0, respectively; the final distance of method M to test T is thereby $\frac{1+0}{2} = \frac{1}{2}$. However, for test T , the distance to method M is $\frac{2}{3}$ and not $\frac{1}{2}$, as the shortest distance for individual test changes at c_1 , c_3 , and c_4 is 1 ($| - 1|$), 0, and 1, resulting in the average distance of $\frac{2}{3}$. This asymmetry of the distance has occurred for two reasons. First, we consider only the nearest change, and while doing that, we search both back and forth, allowing each change to select any change as the nearest one regardless of whether the other party

Algorithm 1: DistanceToNearest

input : a target, t , a comparison target, tc , a distance aggregation method, M_{avg}
output : the distance of a target t to the nearest changes in the comparison target tc

- 1 $Revisions_t \leftarrow \text{ChangeTarget}(t)$
- 2 $Revisions_{tc} \leftarrow \text{ChangeTarget}(tc)$
- 3 $Dists \leftarrow []$
- 4 **for** rev in $Revisions_t$ **do**
- 5 $d \leftarrow \min(\{\text{distance}(rev, rev') \mid rev' \in Revisions_{tc}\})$
- 6 add d to $Dists$
- 7 **end**
- 8 $\text{dist}_{t,tc} \leftarrow M_{avg}(Dists)$
- 9 **return** $\text{dist}_{t,tc}$

also considers it as the nearest. More importantly, the coupling between methods and tests is asymmetric. Since the distance is a way to quantify the degree of the coupling between the test and method, it naturally becomes asymmetric if the coupling is asymmetric; this is also why we did not define the distance to work in both ways. CEMENT considers this asymmetry and calculates distances from both directions, i.e., from a test to a method and the opposite, when it estimates the coupling degree, i.e., the distance, between tests and methods.

2.2.3 Establishing Links between Tests and Methods. Algorithm 2 describes how CEMENT infers relevant links between tests and methods from the distances calculated from Algorithm 1. The final output of CEMENT is a list of methods/tests (C) sorted in descending order of their degree of coupling to the test/method under inspection (T): the higher the rank of a method/test is, the stronger its link to the target test/method. For this, CEMENT first calculates the distance of target T to each candidate in the list C (Line 1). After computing the distance to the target for each candidate in C , we select the top N that are closest to T (Line 2). CEMENT then calculates the distance to T for each candidate in C_N and multiplies newly obtained distances with their matching distances from the target (Line 4 to 6). CEMENT then sorts the candidates in descending order using these updated distances; for those failed to be in the top N , we use their distance values in $D_{T \rightarrow C}$ (Line 3). This additional update for the top N is to handle the asymmetric nature of the distance. We set N to 100, which we obtained empirically. To summarize, CEMENT considers a method and a test to be relevant (likely coupled) if and only if both of them are considered to be close enough. This condition helps avoiding coincidental cases¹.

3 EXPERIMENTAL SETTINGS

3.1 Research Questions

To evaluate CEMENT we start our investigation by checking its ability to mine true links between code units and tests. Thus, we ask:

¹There is no need of having both methods and tests considering each other as the most likely-to-be relevant one. Being relatively close to each other, compared to the rest of tests/methods is sufficient to establish a link between them.

Algorithm 2: CEMENT

input : a target, T , a list of likely-relevant candidates, C , a distance aggregation method, M_{avg} , the number of top candidates, N
output : a list of candidates C sorted in descending order of the distance to the target T

- 1 $D_{T \rightarrow C} \leftarrow \text{DistanceToNearest}(T, C, M_{avg})$
- 2 $C_N \leftarrow \text{SelectTopN}(N, D_{T \rightarrow C}, C)$
- 3 $D_{T \leftrightarrow C} \leftarrow \text{Initialize}(D_{T \rightarrow C}, T, C)$
- 4 **for** c **in** C_N **do**
- 5 $d_{c \rightarrow T} \leftarrow \text{DistanceToNearest}(c, T, M_{avg})$
 $D_{T \leftrightarrow C}[c] \leftarrow d_{c \rightarrow T} \cdot D_{T \leftrightarrow C}[c]$
- 6 **end**
- 7 $C_{ranked} \leftarrow \text{Rank}(D_{T \leftrightarrow C}, C)$
- 8 **return** C_{ranked}

RQ1 Capability: can CEMENT establish static links between tests and methods based on their co-evolution?

To answer this question, we need an oracle that decides on the link between tests and methods. This type of oracle (i.e., general associations between tests and methods), however, is hard to get and there is no guarantee that it can be accurately approximated [32]. To set such an oracle we use mutation testing [19] applied at specifically selected methods and check whether tests selected by CEMENT can indeed kill the mutants of the selected methods and contrast them with randomly selected tests. The underlying assumption here is that tests related to the methods should kill the mutants that reside on these methods, at least kill more mutants, than tests that are not related. We thus, expect the resulting test-and-killed relations between tests and methods to overlap with the links inferred by CEMENT.

Specifically, to answer this RQ, we select N tests that are the most likely to kill mutants in the methods we consider by picking the top N tests with the strongest coupling to these methods. As CEMENT establishes a relationship between a test and a method by default, we take additional steps to obtain a relationship between a test and multiple methods. To be specific, we first repeat CEMENT for each method using all tests, obtaining multiple rankings for each test; we then take either the highest (i.e., the best) or the average as the final ranking for each test. These two are notated as $\text{CEMENT}_{t \rightarrow mm}^{best}$ and $\text{CEMENT}_{t \rightarrow mm}^{avg}$; t and mm denote a test and multiple methods, respectively. We deem CEMENT capable of inferring the links between tests and methods from their co-evolution if the tests ranked within the top N by CEMENT can kill more mutants than randomly selected N tests.

After investigating the existence of links between tests and methods, we turn our attention to a more concrete task in order to investigate whether these links offer actionable information. Therefore, we ask:

RQ2 Applicability: can we use the links between tests and methods generated by CEMENT in software debugging?

To answer this question, we conduct a case study of CEMENT in fault localization. We select fault localization among different software debugging activities since we can directly relate the resulting

test and method links to fault localization by assuming the methods strongly coupled to a failing test as suspicious ones. We also compare CEMENT with an existing Information Retrieval-based Fault Localization (IRFL) technique adopted in a recent program repair technique, iFixR [14]. We choose this IRFL technique as our baseline because it combines various IR-based features of faults, summarising existing IRFL techniques.² In addition, the IRFL is a static approach, thereby allowing us to inspect further how CEMENT performs compared to existing static techniques.

After investigating the applicability of CEMENT on fault localization we check whether its performance is dependent on the project maturity. Hence, we ask:

RQ3 Impact of Software Maturity: how does the software maturity affect the effectiveness of CEMENT?

CEMENT assumes tests and methods to be relevant if they have co-evolved throughout the development. Thus, for CEMENT to be useful, the program under inspection should be mature enough to have a sufficient amount of previous changes for CEMENT to process. Hence, to answer RQ3, we divide our dataset into two levels of software maturity (i.e., Applicable and Confident) based on the number of changes made on individual tests and methods.

- Applicable: tests and methods have changed at least once
- Confident: tests and methods have changed equal to or more than the average number of times individual tests and methods have changed

We extend our fault localization case study to additionally have these two different levels of software maturity for each project and investigate how the performance of CEMENT varies depending on the software maturity. We achieve this by using these two software maturity levels as the filtering criteria: *Applicable* is to inspect the changes in CEMENT's performance after excluding the tests and methods it cannot handle inherently, and *Confident* is to simulate how CEMENT performs when the projects become more mature.

3.2 Subject

We evaluate CEMENT using Defects4J v.2.0.0 [10], a repository that contains real-world faults of 17 open source projects. We select Defects4J mainly for two reasons. First, Defects4J provides an inner command to run mutation testing on 17 projects it investigated. This inner command handles from compiling a project to running a given test suite on the injected mutants, saving us a lot of time and effort that would spend on preparing an environment for mutation testing. Second, Defects4J provides 835 real-world faults and an infrastructure to replicate them easily, making it an optimal target for our fault localization case study. Table 1 presents the overall information about the projects that we studied: out of these 17 projects, we use 14 to answer RQ1 and 15 to answer RQ2 and RQ3.

3.2.1 Subjects for Mutation Testing. Among 17 projects in Defects4J, we fail to run mutation testing on Commons Collections and Mockito using its inner mutation testing command. We simply exclude these two from our targets for mutating testing, as our goal is not about running mutation testing on every project in Defects4J. In

²iFixR performs statement-level fault localization by localizing faults at the file-level first using D&C that leverages various IR features [13]

this study, we use Git to collect the projects’ past changes, as 16 out of 17 projects in Defects4J employ Git to manage the changes between versions; JfreeChart is the only one that uses SVN instead.³ We thus, excluded JfreeChart, leaving 14 projects for our analysis.

3.2.2 Subjects for Fault Localization. For our case study on fault localization, we exclude Commons Collections and JfreeChart. We exclude JfreeChart for the same reason above and Commons Collections for a different reason. As explained in RQ2, we assume faulty methods to be more strongly related to failing tests than non-faulty ones. For CEMENT to establish these relevant links between failing tests and methods, failing tests should exist before the failure. However, for many of the faults in Defects4J, the failing tests marked by Defects4J were introduced the first time to the project when developers submitted fix patches, making them impossible for CEMENT to handle.⁴ Thus, we further filter out them, excluding Commons Collections entirely since there were no remaining faults after this exclusion. Combined, we are left with 214 faults out of 835 faults.

We use the IRFL in iFixR as the baseline for our fault localization case study. This IRFL technique also failed in JfreeChart for a similar reason to ours. Among 214 faults, the IRFL failed to localize 39 faults due to missing bug reports or bug reports related to more than one fault and 33 faults by failing to compute suspiciousness scores for faulty statements. Thus, we further exclude 72 faults and use the remaining 142 faults for the baseline comparison in RQ2. Table 1 presents the number of faults remaining after these exclusions for each project. RQ3 is about the impact of software maturity on the performance of CEMENT. We use all 214 faults to answer RQ3 since we do not need to compare CEMENT with the IRFL here.

Table 1: Test subjects. Mockito was excluded from RQ1. For RQ2 and 3, all 15 projects were used. The value in parentheses is the number of faults after removing those the IRFL cannot handle.

Project	# of faults	# of methods	# of tests	# of previous commits	
				min – max	average
Lang	23 (21)	1959.3	2201.6	3695 – 4769	4160.9
Math	32 (24)	4089.7	3288.1	3929 – 10074	7378.2
Time	3 (2)	3782.7	5112.7	8849 – 8988	8895.3
Closure	60 (27)	9857.9	8435.6	10332 – 23795	18293.5
Mockito	8 (1)	3118.4	2187.0	4296 – 5808	5305.4
Cli	12 (7)	216.9	228.2	324 – 906	445.1
Codec	6 (4)	408.0	537.7	587 – 1301	945.7
Compress (11)	12	1202.0	675.6	569 – 3077	1877.6
Csv	4 (3)	109.5	205.2	147 – 470	314.8
Gson	2 (2)	805.0	1353.5	2157 – 2160	2158.5
JacksonCore	6 (5)	1718.0	722.3	1621 – 3292	2440.3
JacksonDatabind	19 (15)	5427.3	3358.3	7187 – 10330	8786.3
JacksonXml	1 (0)	388.0	298.0	686 – 686	686.0
Jsoup	21 (18)	985.0	446.0	639 – 2134	1431.0
JXPath	5 (2)	1518.2	475.4	1947 – 2045	1993.6

³Currently, JfreeChart has successfully migrated from SVN to Git. However, Defects4J still refers to the older version of JfreeChart and generates a new git branch that contains only four commits made by Defects4J developers.

⁴Defects4J isolates each test failure and provides a separate commit that contains only a single fault and fails only with the tests related this fault. Defects4J achieves this by reversing the bug-fixes and labeling the tests related to the fixes as the failing tests.

3.3 Past Change Collection

CEMENT determines whether a given method and test are relevant using the time intervals between their past changes. Thus, to run CEMENT, we first need to find when each method and test was changed. We employ Git v.2.32.0 [4] for this because it is a widely used version control system and 15 projects we studied also utilize Git to maintain their versions. For the experiments, we use Defects4J, which works with the project’s detached HEADs it created instead of the main branch of the project. Thus, to avoid confusion, such as the mismatch between commit hashes, we gather the past changes of methods and tests from these detached HEADs. This paper aims to demonstrate that CEMENT can infer the relevant links between tests and methods based on their history of co-evolution. Hence, rather than trying to achieve the best performance by controlling the time window for past change collection, we simply collect all prior changes of methods and tests. For each past change, we record the hash of the commit that introduced the change.

3.4 Mutation Testing

As mentioned in Section 3.2, Defects4J provides an inner mutation testing command relying on Major. We thus, work with the buggy versions because it becomes easier to relate the results we obtained with mutation testing with those of the fault localization case study we perform. Moreover, working with fixed versions may lead to overestimating the performance of CEMENT, as the changes that generated the fixed versions may contain information that directly reveals the link between certain tests and methods (e.g., failing tests and faulty methods), which is rare in practice. We select the most recent version among multiple buggy versions for each project to maximize the number of past changes that CEMENT can exploit.

Mutation testing is computationally expensive [19]. Thus, we reduce the scope of the mutation testing by mutating only the top ten frequently modified classes instead of the entire code. We formulate a test set that includes 20% of the entire tests of a target project. This test set includes all the tests that are likely to execute the selected classes according to the test and code naming convention [32]. In case there is available space after the selection process, we randomly select the remaining number of tests.

3.5 Fault Localization

We conduct a case study on fault localization to evaluate the applicability of CEMENT in software testing and debugging. As for the baseline of this case study, we select an Information Retrieval-based Fault Localization (IRFL) technique adopted in a recent program repair technique called iFixR [14]; we run iFixR on 17 projects in Defects4J v.2.0.0. Since we work at the method granularity, we aggregate the statement-level fault localization results of iFixR to the method level, taking the highest statement-level suspiciousness score for each method. For CEMENT, we rank the methods in descending order of their distances to failing tests: the higher its rank is, the more suspicious the method is. Defects4J includes faults that result in multiple failing tests. In these cases, we take the highest method ranking among those computed with each failing test; similarly, for multiple faulty methods, we take the highest.

While Defects4J provides real faults, the corresponding buggy versions delivered by Defects4J have been tailored to contain only a single fault [10]. Consequently, if we run CEMENT directly on these buggy versions, we may include change information that explicitly reveals the locations of faults. To prevent this, instead of working with the buggy versions given by Defects4J, we execute CEMENT on the original buggy commits that Defects4J additionally provides. For iFixR, we follow its own configuration.

3.6 Evaluation Metrics

We evaluate the effectiveness of CEMENT in establishing the links between tests and methods from their evolutionary coupling using mutation score. Mutation Score (MS) is defined as the ratio of mutants killed by selected tests to the total number of generated mutants, as below.

$$MS(T) = \frac{\# \text{ of mutants killed by } T}{\# \text{ of total mutants}}, R_{MS}(T) = \frac{MS(T)}{MS_{max}(= MS(T_{all}))}$$

T denotes a test set that contains N_T tests. We define N_T (i.e., the number of selected tests) differently for each project, configuring this N_T to be 10% of the total number of tests in the mutation testing. We compare the MS score of test set T with the MS score of running all the tests (T_{all}). We refer to the latter one as the maximum (MS_{max}); it is the total number of killed mutants over the total number of generated mutants. Here, we want to evaluate the trade-off between the effort saved by running only N_T test and the performance degradation caused by it. Thus, we divide the MS score of test set T by the maximum MS score. We notate this ratio as R_{MS} and use it to evaluate the capability of CEMENT.

RQ2 and RQ3 are about the applicability of CEMENT in fault localization. We evaluate the fault localization performance of both CEMENT and the baseline IRFL using $acc@n$ and wef . These two metrics measure the absolute effort spent on localizing faults, following the guideline suggested by Parnin and Orso [20]. $acc@n$ counts the number of faults ranked within the top n places; for n , we use 1, 3, 5, and 10. wef or wasted effort is the number of non-faulty elements examined before inspecting the first faulty one. As wef is computed per fault, we take the average and the median.

3.7 Tie Breaking

CEMENT does not guarantee to calculate a unique distance (i.e., coupling degree) for each method and test pair; in the worst case, it may compute the same degree of coupling for all the method and test pairs. To avoid overestimating the performance, we break these potential ties between the method and test links by assigning the lowest ranking that tied methods or tests can have to all those that are tied. For a similar reasoning, we assign the lowest possible ranking to the methods tied by having the same suspiciousness score while evaluating the fault localization performance.

3.8 Implementation & Environment

CEMENT is implemented in Python version 3.9.9. All the experiments were run on the machine equipped with Intel Core i7 CPU and 32GB RAM. The replication package and all the results are publicly available from <https://doi.org/10.5281/zenodo.6366615>.

4 RESULTS

4.1 RQ1. Capability

Table 2 records the changes in mutation score when executing subsets of the accompanied test suites, composed of 10% of the total number of tests, using R_{MS} score. Overall, we obtained higher R_{MS} scores in CEMENT (i.e., CEMENT^{best_{t→mm}} and CEMENT^{avg_{t→mm}}) than in the random test selection. For example, for JacksonCore, R_{MS} improves from 0.19 to 0.60, increasing the number of killed mutants more than three times. Among 14 projects we investigated, CEMENT successfully outperforms the random test selection baseline in 11 projects by taking the highest ranking for each test among those it obtained with the considered methods (CEMENT^{best_{t→mm}}). When using the average instead of the highest (CEMENT^{avg_{t→mm}}), the number of projects where CEMENT is superior than the random selection decreases by two. However, CEMENT^{avg_{t→mm}} still performs better than the random in nine out of 14 projects. In fact, CEMENT consistently outperforms the random test selection either by CEMENT^{best_{t→mm}} or CEMENT^{avg_{t→mm}}.

The maximum Mutation Score (MS) assumes running all tests participated in the mutation testing. Since we selected 10% of these tests, all the R_{MS} scores in Table 2 being greater than 0.1, even in the random, suggests that there exist large overlaps between the tests in terms of the mutants they killed. Nonetheless, CEMENT were able to select 10% of the tests that killed much more mutants than the same number of randomly selected tests. For instance, in JacksonXml, we can kill 74% of the total killed mutants by running tests selected by CEMENT, whereas with the randomly selected tests, we can kill only 28%. Based on these results of CEMENT consistently outperforming the random, we posit that CEMENT can establish relevant links between tests and methods given the project’s evolution.

Answer to RQ1: Tests selected by CEMENT consistently kill more mutants than those killed by randomly selected tests. These results suggest that CEMENT successfully captures probable links between tests and methods that can identify tests related to mutated methods.

4.2 RQ2. Applicability

Table 3 presents the results of our case study of CEMENT on fault localization. Compared to the Information Retrieval-based Fault Localization (IRFL) technique used in a recent program repair method, iFixR, CEMENT acquires comparable performance in terms of $acc@1$: CEMENT places 18 faults at the top of the rankings, whereas the IRFL places 23 faults. For the sake of simplicity, we will call the IRFL technique used in iFixR *the IRFL*, hereafter. While the IRFL surpasses CEMENT consistently in localizing faults, it also requires more effort to build the FL model and collect data for training and evaluation: the IRFL collects 17 features (seven from bug reports and ten from source code files) for fault localization. Moreover, the IRFL assumes the existence of bug reports that in many cases is not available and often requires an additional data preprocessing step, which can be costly [14]. In contrast, CEMENT needs only the hashes of commits that changed the methods and tests. Hence,

Table 2: The changes in mutation scores when running only 10% of the entire tests. Each cell contains R_{MS} , the ratio of the mutation score obtained by the approach (row) to the maximum mutation score. When CEMENT (i.e., $\text{CEMENT}_{t \rightarrow mm}^{best}$ and $\text{CEMENT}_{t \rightarrow mm}^{avg}$) outperforms the random test selection (Random), the corresponding ratio (R_{MS}) is highlighted in bold. For Random, its R_{MS} score is highlighted in bold if and only if it outperforms both $\text{CEMENT}_{t \rightarrow mm}^{best}$ and $\text{CEMENT}_{t \rightarrow mm}^{avg}$.

Approach	Lang	Math	Time	Closure	Cli	Codec	Compress	Csv	Gson	Jackson Core	Jackson Databind	Jackson Xml	Jsoup	JXPath
Random	0.22	0.31	0.47	0.43	0.28	0.36	0.30	0.49	0.20	0.19	0.15	0.28	0.49	0.29
$\text{CEMENT}_{t \rightarrow mm}^{best}$	0.22	0.40	0.58	0.49	0.33	0.72	0.18	0.67	0.38	0.60	0.16	0.74	0.52	0.45
$\text{CEMENT}_{t \rightarrow mm}^{avg}$	0.54	0.38	0.36	0.40	0.52	0.47	0.49	0.65	0.14	0.13	0.33	0.64	0.48	0.49

concerning the cost spent up to the localization,⁵ we posit that CEMENT shows comparable performance to the IRFL.

An important discrepancy in the evaluation of IRFL happens when bug reports explicitly specify the code elements that have the reported bugs. In these cases, we do not need to localize faults in the first place, as they have been already identified by the person reporting them. The IRFL localizes faults based on the similarity between bug reports and source code. Therefore, if a bug report already contains the identifier of a fault, we might overestimate the performance of the IRFL, especially when it directly exploits the identifiers of code elements in a bug report. Hence, we divide faults into two groups based on whether their identifiers are already in bug reports and examine whether the localization performance differs between these two groups. We treat a bug report to contain the identifier of a fault if it has both the class and the method name of the fault.

The leftmost column of Table 3 presents the localization results of the faults whose identifiers are in bug reports; out of 142 faults we examined, 54 already have their identifiers in bug reports. The middle column shows the localization results of the faults without their identifiers in bug reports; the rightmost column describes the combined results. Overall, the IRFL performs better when bug reports contain fault identifiers: the IRFL places 15 out of 54 faults at the top ($acc@1$) for the group of faults with their identifiers in bug reports, whereas it places only eight at the top for the other group without the identifiers, even though more faults belong to the latter group. We observe similar trends in $acc@3$, 5, 10.

Compared to the IRFL, having fault identifiers in bug reports does not have the same effect on CEMENT. While CEMENT localizes more faults near the top for the group where bug reports have fault identifiers, the difference is smaller; for example, for the group without fault identifiers, the $acc@1$ decreases almost by half in the IRFL, whereas, in CEMENT, it decreases only by two, from 10 to 8. Even this small decrease is from elsewhere, as CEMENT does not leverage both source code and bug reports in the first place; we suspect that the observed decreases are coincidental and are attributed to the characteristics of faults in each group.⁶ Furthermore, CEMENT becomes more comparable to the IRFL for the

faults without their identifiers in the bug reports: compared to the IRFL, CEMENT ranks the same number of faults at the top and within the top three (i.e., $acc@1$ and $acc@3$) and locates only one less fault within the top five (i.e., $acc@5$). CEMENT fails to compete with the IRFL in *wef*, although the difference between CEMENT and the IRFL becomes smaller in the median compared to the average. Regarding the previous observation in $acc@n$, this result is likely from CEMENT completely failing on some faults, assigning the lowest rank to them. Nevertheless, CEMENT still achieves comparable performance in terms of localizing faults near the top, suggesting that CEMENT can be useful in fault localization.

Answer to RQ2: CEMENT achieves comparable performance to the recent Information Retrieval-based Fault Localization (IRFL) technique, especially for the cases where this IRFL technique performs less effective.

4.3 RQ3. The Impact of Software Maturity

Table 4 presents the fault localization results of CEMENT for the complete set of 214 faults with two variations on software maturity: *Applicable* and *Confident*. Methods or tests being "Applicable" means they have changed at least once, and being "Confident" implies that they have been altered more frequently than the average. We apply these two maturity criteria to each buggy version of target projects, filtering out the methods and tests that failed to meet them. We did not differentiate faulty methods and failing tests from other methods and tests while applying these criteria. As a result, 14 and 129 faults out of 214 faults were excluded by *Applicable* and *Confident* criteria, respectively.

The results in Table 4 show that the performance of CEMENT can be improved as the software becomes more mature. For instance, when we apply *Applicable* criterion (*Applicable**), we observe small improvements in the percentage of faults localized near the top, that are around 1% to 3%. The improvement is more evident in *wef* where the average decreases by half and the median by around 15. When we further filter out immature methods and tests using *Confident* criterion (*Confident**), this improvement becomes more prominent: the percentage of localized faults further increases by 6% at the top and by around 6 to 8% within the top three, five and ten compared to the results of applying *Applicable*

⁵With CEMENT, each fault localization task itself was done within seconds, and the past change collection, which covers over thousands of commits here, took on average within 5 minutes, without any optimization. This cost can be further reduced in practice, as we do not have to process all prior commits every time; the changes can be collected incrementally.

⁶If a method evolves actively, this method is more likely to have failed in the past than those rarely changed. Subsequently, if the method frequently fails, a reporter

may already know that this method is the trigger when it causes a failure, and thereby, includes its identifier in the bug report.

Table 3: Comparison between the fault localization by CEMENT and the IRFL in iFixR. CEMENT becomes more comparable to the IRFL when focusing only on the faults whose identifiers are not already in bug reports (Without Faulty Methods).

Proj. (w/wo/all)	With Faulty Methods (CEMENT/ iFixR)						Without Faulty Methods (CEMENT/ iFixR)						All (CEMENT/ iFixR)					
	acc			wef			acc			wef			acc			wef		
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Lang (16/5/21)	3/7	8/9	9/11	10/12	60.9/4.2	2.0/1	0/0	2/1	3/2	3/3	441.8/11.8	3.0/6	3/7	10/10	12/13	13/15	151.6/6.0	3.0/3
Math (12/12/24)	1/2	1/6	2/7	3/11	562.0/4.0	52.0/2	0/3	0/6	0/8	0/11	787.8/11.6	86.0/2	1/5	1/12	2/15	3/22	674.9/7.8	83.0/2
Time (1/1/2)	1/0	1/1	1/1	1/1	0.0/1.0	0.0/1	0/0	0/0	1/0	1/0	4.0/60.0	4.0/60	1/0	1/1	2/1	2/1	2.0/30.5	2.0/30
Closure (1/26/27)	0/0	0/0	0/0	0/0	4315.0/20.0	4315.0/20	2/1	3/2	4/2	6/4	2225.6/60.3	242.0/22	2/1	3/2	4/2	6/4	2303.0/58.8	301.0/22
Mockito (0/1/1)	-/-	-/-	-/-	-/-	-/-	-/-	0/0	1/0	1/0	1/0	1.0/10.0	1.0/10	0/0	1/0	1/0	1/0	1.0/10.0	1.0/10
Cli (3/4/7)	2/1	2/3	2/3	2/3	18.0/1.0	0.0/1	2/0	3/1	3/1	3/1	4.2/13.8	0.0/17	4/1	5/4	5/4	5/4	10.1/8.3	0.0/2
Codec (3/1/4)	0/2	0/3	0/3	1/3	98.7/0.7	96.0/0	0/0	0/0	0/0	0/0	212.0/16.0	212.0/16	0/2	0/3	0/3	1/3	127.0/4.5	146.0/1
Compress (6/5/11)	0/1	2/1	3/2	4/2	152.2/19.7	5.0/14	0/0	0/0	0/0	0/2	55.6/16.6	50.0/19	0/1	2/1	3/2	4/4	108.3/18.3	21.0/18
Csv (0/3/3)	-/-	-/-	-/-	-/-	-/-	-/-	0/2	1/3	1/3	1/3	22.3/0.3	18.0/0	0/2	1/3	1/3	1/3	22.3/0.3	18.0/0
Gson (2/0/2)	1/0	1/0	1/0	1/0	15.0/52.5	15.0/52	-/-	-/-	-/-	-/-	-/-	-/-	1/0	1/0	1/0	1/0	15.0/52.5	15.0/52
JacksonCore (1/4/5)	0/0	0/0	0/0	0/1	1022.0/9.0	1022.0/9	2/0	3/0	3/0	3/0	77.2/130.0	0.0/127	2/0	3/0	3/0	3/1	266.2/105.8	1.0/16
JacksonDatabind (5/10/15)	0/0	0/2	0/3	0/3	620.2/20.0	398.0/4	0/1	0/2	0/2	0/4	1819.5/38.4	1044.0/12	0/1	0/4	0/5	0/7	1419.7/32.3	540.0/11
Jsoup (4/14/18)	2/2	2/3	2/3	3/3	170.5/8.5	4.0/1	2/1	3/1	4/3	5/7	215.1/35.9	81.0/10	4/3	5/4	6/6	8/10	205.2/29.8	39.0/7
JXPath (0/2/2)	-/-	-/-	-/-	-/-	-/-	-/-	0/0	0/0	0/0	0/1	279.5/120.0	280.0/120	0/0	0/0	0/0	0/1	279.5/120.0	280.0/120
Total (54/88/142)	10/15	17/28	20/33	25/39	335.8/9.4	14.0/2	8/8	16/16	20/21	23/36	1047.5/41.3	80.0/12	18/23	33/44	40/54	48/75	776.8/29.2	63.0/8

criterion. In *wef*, the average and the median decrease by around one fourth when compared to those of the *Applicable*.

We exclude the methods and tests that failed to meet our maturity criteria to allow CEMENT running on software systems with different maturity levels. Since we use absolute metrics (i.e., *acc@n* and *wef*) in the evaluation, we consider the possibility that the performance improvement that we observed may come from the decrease in the total number of methods and tests after the filtering. To verify that the improvement comes from the project maturity, we apply the maturity criteria *only* on faulty methods and failing tests, excluding faults that failed to meet these criteria. We then rerun CEMENT for the remaining faults *without* any maturity filtering. This way, we focus on faults that appeared at the mature part of the code without risking to overestimate. As shown in Table 4, our results are almost the same as before, localizing one more or fewer faults near the top, confirming that the improvement we witnessed indeed relates to the software maturity.

Answer to RQ3: The performance of CEMENT improves when we focus only on the mature part of the code (i.e., have more than the average number of past changes), implying CEMENT can enhance along with software maturity.

5 DISCUSSION

5.1 The Impact of Evolutionary Couplings in Software Debugging

CEMENT establishes relevance links between tests and code only when they have co-evolved. Thus, CEMENT may complement existing software debugging and testing techniques that rely on dynamic or static code analysis. To check for this potential complementarity, we revisit the results of RQ2, but at this time, inspect which faults are localized by CEMENT and by the baseline IRFL technique.

Table 5 reports the number of faults localized by CCCT, the IRFL, and both. Overall, CEMENT and the IRFL localize different faults near the top. For the groups of faults whose identifiers are already in bug reports, the faults localized at the top by both CEMENT

Table 4: Complete fault localization results of CEMENT. *N* is the total number of faults in each project. "*" means that *Applicable/Confident* filtering criterion was applied to the projects before extracting the evolutionary couplings, and without "*" indicates that the filtering criterion was used only to exclude faults.

Project	N	acc@n				wef	
		1	3	5	10	mean	median
Lang	23	3 (0.13)	10 (0.43)	12 (0.52)	13 (0.57)	145.3	4.0
Math	32	2 (0.06)	3 (0.09)	4 (0.12)	7 (0.22)	692.0	79.5
Time	3	1 (0.33)	1 (0.33)	2 (0.67)	2 (0.67)	867.3	4.0
Closure	60	7 (0.12)	11 (0.18)	12 (0.20)	16 (0.27)	2130.1	228.5
Mockito	8	2 (0.25)	3 (0.38)	3 (0.38)	5 (0.62)	58.1	7.0
Cli	12	5 (0.42)	7 (0.58)	8 (0.67)	8 (0.67)	10.4	1.0
Codec	6	0 (0.00)	0 (0.00)	1 (0.17)	2 (0.33)	97.7	85.0
Compress	12	0 (0.00)	2 (0.17)	3 (0.25)	4 (0.33)	101.3	23.0
Csv	4	0 (0.00)	1 (0.25)	1 (0.25)	1 (0.25)	33.2	32.5
Gson	2	1 (0.50)	1 (0.50)	1 (0.50)	1 (0.50)	15.0	15.0
JacksonCore	6	2 (0.33)	3 (0.50)	3 (0.50)	3 (0.50)	431.5	154.5
JacksonDatabind	19	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	1369.1	540.0
JacksonXml	1	0 (0.00)	0 (0.00)	1 (1.00)	1 (1.00)	3.0	3.0
Jsoup	21	4 (0.19)	5 (0.24)	6 (0.29)	8 (0.38)	221.6	64.0
JXPath	5	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	444.4	400.0
Total	214	27 (0.13)	47 (0.22)	57 (0.27)	71 (0.33)	906.2	64.5
Total (Applicable)	198	27 (0.14)	47 (0.24)	57 (0.29)	71 (0.36)	432.2	50.5
Total (Applicable*)	198	27 (0.14)	47 (0.24)	56 (0.28)	71 (0.36)	432.2	50.5
Total (Confident)	85	16 (0.19)	27 (0.32)	32 (0.38)	38 (0.45)	180.5	16.0
Total (Confident*)	85	17 (0.20)	27 (0.32)	31 (0.36)	37 (0.44)	115.3	14.0

and the IRFL is only two; CEMENT and the IRFL locate respectively eight and 13 different faults at the top. While there are more in common between the faults localized by CEMENT and the IRFL within the top three, five and ten, many faults are still localized by only one of them. In cases where bug reports do not contain fault identifiers, fewer faults are localized by both techniques. At the same time, the number of faults localized only by CEMENT now becomes similar to that of the IRFL; for example, both CEMENT and the IRFL localize eight different faults at the top (*acc@1*). In total, CEMENT localizes 16 faults for which IRFL has failed at the top; this is around 41% of total faults ranked at the top by either approach. Within the top ten, 22 out of 97 faults are localized exclusively by CEMENT. These results indicate that the relationships

between tests and code captured by CEMENT through the software co-evolution are different from those exploited in the current IR-based fault localization techniques. Thus, we posit that CEMENT has the potential to complement current software debugging and testing activities by establishing the relationships that have remained underexplored.

5.2 Evolutionary Couplings and Traceability Links between Tests and Code

There are some studies specialized in modelling test-and-tested relationships among various relationships that tests and code can have [21, 23, 32]. These relationships are called test-to-code traceability links and aim at capturing the intent of tests, i.e., identify the key functionality that is tested/asserted by a test, leaving out indirectly tested functionality. This means that the traceability links are abstract and not exact. Compared to test-to-code traceability links, the evolutionary couplings established by CEMENT reflect a more relaxed relation. For example, let us suppose the given test and method belong to the same component and thereby have frequently changed around the same time by developers. In this case, even if the test does not directly test the method, CEMENT will likely regard it as relevant to the method since they have co-evolved. To further examine how CEMENT differs from or relates to existing studies of test-to-code traceability links, we employ TCTRACER, a state-of-the-art approach that automatically establishes test-to-code traceability links [32]. We replicate the method-level traceability links prediction study in the TCTRACER paper and investigate whether CEMENT can predict these links.

Table 6 compares the performance of CEMENT and TCTRACER in predicting test-to-code traceability links at the method level.⁷ Here, we assume that CEMENT generates a stronger link for a method to the test that evaluates its functionality than to those that do not. Thus, for each test, we rank methods in descending order of their strength of the link to it. We then simply regard the top five methods as having traceability links with the test. The performance of CEMENT varies depending on the software maturity. Thus, we extend this study by applying the two software maturity criteria (i.e., *Applicable* and *Confident*) in order to focus on the traceability links coming from the more mature part of the code. We achieve this by excluding the traceability links that failed to meet the maturity criteria from the evaluation. Table 6 reports the number of test-to-code traceability links we initially have and the number of links after the filtering.⁸

TCTRACER generally outperforms CEMENT in predicting test-to-code traceability links. We believe that this is due to the following three reasons. First, we simply take the top five methods for the prediction. Thus, even if CEMENT ranks a related method at the top, we end up with four false positives, explaining the low performance, especially in precision. Secondly, because the oracle of test-to-code traceability links was formulated manually, it might be biased toward the methods and tests with similar names. This

may give some advantages to TCTRACER, for it leverages the textual similarity between the names. Finally and most importantly, a test and a method can have an evolutionary coupling between them without being considered in the test-and-tested relationship as it could be an indirect link. Despite these, CEMENT excels TCTRACER in Chart when inspecting only the links that met the *Confident* criterion: out of the remaining four test-to-code traceability links, two of them are correctly predicted only by CEMENT.

When we inspected these cases, we found that these two are when there are no common terms between test and method names and when a test calls multiple methods, especially after calling the related method. Since five out of eight techniques that TCTRACER combines compare test and method names in order to predict the traceability links, TCTRACER could be less effective if the test and method have entirely different names. TCTRACER employs techniques that exploit dynamic execution traces, such as Last Call Before Assertion, to complement this weakness. However, like the second case that we observed, if there are many methods between the test and the ground-truth method in an execution trace, TCTRACER becomes less successful in the prediction. CEMENT leverages neither the dynamic execution traces nor the source code. As a result, it is inherently free from all the issues that may arise from using them; this allows CEMENT to handle the cases for which TCTRACER has failed. Hence, we argue that CEMENT can complement existing techniques of predicting test-to-code traceability links, especially when working with mature software systems.

6 THREATS TO VALIDITY

A primary threat to validity of our work is the absence of the oracle for the evolutionary couplings between tests and methods. To mitigate this threat, we used mutation testing as a substitute of this oracle. Since mutation testing has been often employed as a test oracle [19], we posit that it can also be useful for our case. In addition to the capability of CEMENT in establishing the evolutionary couplings, this study also investigates the usefulness of the resulting couplings in software maintenance activities. For this, we select fault localization as our target for the case study, as it is one of the most actively studied areas in software maintenance [33]. We select a recent Information Retrieval (IR) based fault localization technique as the baseline because it combines multiple existing IR-based techniques and, thereby, can summarize the current trend in fault localization to some extent. [13, 14].

The threats to external validity relate to whether our findings on the effectiveness of the evolutionary coupling can be generalized to other projects. We use 15 open source projects in Defects4J, a benchmark of real-world faults, as our targets for evaluation. Still, additional studies on industrial projects may be needed to fully verify our hypothesis.

Threats to construct validity relate to the evaluation metrics we use. To assess the capability of CEMENT to select likely-related tests to given methods, we employ mutation score, a widely adopted metric in mutation testing [19]. For the case study, we select three absolute evaluation metrics that have been frequently employed in fault localization [15, 20, 27, 30].

⁷The results of TCTRACER provided by the authors contain only method and test pairs predicted to have a traceability link rather than the complete prediction results. Hence, we only investigate precision, recall, and F1 score and exclude AUC and MAP for this replication study.

⁸Because the current CEMENT implementation handles only methods and not constructors for Java, we exclude two additional traceability links for each project

Table 5: Comparison between faults localized by CEMENT and the IRFL in iFixR. The value on the right (*either*) is the total number of faults localized by either CEMENT or the IRFL (i.e., union), whereas the values on the left denote the number of faults localized only by CEMENT, only by the IRFL, and by both (i.e., intersection), respectively. When CEMENT/the IRFL localizes faults on which the other failed, the corresponding cell is highlighted in bold text.

	N	acc@1		acc@3		acc@5		acc@10	
		CEMENT/ iFixR / both	either	CEMENT/ iFixR / both	either	CEMENT/ iFixR / both	either	CEMENT/ iFixR / both	either
With Faulty Methods	54	8/13/2	23	6/17/11	34	8/21/12	41	8/22/17	47
Without Faulty Methods	88	8/8/0	16	14/14/2	30	15/16/5	36	14/27/9	50
All	142	16/21/2	39	20/31/13	64	23/37/17	77	22/49/26	97

Table 6: Comparison between CEMENT and TCTRACER. The three values next to the project are the number of traceability links without any filtering, with *Applicable* and *Confident* filtering. The left and right values are the results of CEMENT and TCTRACER, respectively, and they are all in percentage.

Conf	Lang (74/44/7)			IO (40/40/14)			Chart (35/25/4)		
	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
All	10 / 86	16 / 78	12 / 82	9 / 67	22 / 82	13 / 74	5 / 23	9 / 74	6 / 35
Applicable	12 / 59	26 / 89	17 / 71	9 / 67	22 / 82	13 / 74	5 / 17	12 / 76	7 / 28
Confident	8 / 12	22 / 89	12 / 21	6 / 29	14 / 100	9 / 45	21 / 2	75 / 50	33 / 4

7 RELATED WORK

Our definition of co-evolution depends on the past changes in tests and methods. Several studies have leveraged past changes in tests and methods. Defect prediction aims at predicting faults before executing them using code quality metrics that include past changes in the software [12, 16, 22]. However, these usages of past changes are often limited to enrich the description of individual code elements concerning a specific problem: e.g., methods that have changed frequently are more likely to contain faults than those that have not [22]. In other words, they use the past changes to describe the characteristics of faults.

Sohn and Yoo [25, 27] considered the past changes of methods to improve fault localization performance; similarly to defect prediction, they used the past changes as another feature to describe faults. In automated program repair, Saha et al. used past software changes to further locate the code that is likely to undergo similar repairs [25]. Although the ways they used the past changes varies, all these studies utilize past software changes as additional data to enhance their approaches; they can validate their main idea without using the past changes. Furthermore, they inspect past changes per code element rather than investigating them together to grasp the overall picture of how software has changed. In contrast to the existing work, CEMENT establishes couplings between tests and methods, *directly* from how they have changed throughout the development.

Association between tests and methods can be useful in various software maintenance activities, as it can give developers hints on how their changes affect or will affect others. Consequently, there have been many studies on automatically mining this information, either dynamically or statically. Studies on test-to-code traceability links aim at setting explicit links between tests and code [21, 23, 32]. Rompaey and Demeyer investigated diverse sources of information, ranging from naming convention to static and dynamic code

analysis, to automatically generate traceability links between tests and code that can pinpoint which tests examine which part of code [23]. Qusef et al. proposed SCOTCH that identifies these traceability links using dynamic slicing; SCOTCH uses dynamic slicing to locate the area affected by the last assertion in each unit test case [21]. Mohammad et al. tried to improve the quality of the traceability links by further findings the method that implements the core functionality under testing based on the changes in the object states [8]. Recently, White et al. combined multiple techniques that automatically mine test-to-code traceability links, allowing these techniques to complement each other [32]. Unlike all these approaches that somehow employ dynamic program analysis to generate these links between tests and code, CEMENT is purely static. Perhaps more importantly, the traceability links focus on test intents resulting in abstract relationship between test-and-code, whereas the evolutionary couplings we use aim at capturing important dependencies.

8 CONCLUSION

We propose CEMENT, a static approach that mines relevant links between tests and code units without any dynamic or static code analysis but through their history/evolution. The key idea of CEMENT is that tests and code that are relevant to each other are likely to be changed, multiple times, around the same time by developers. Thus, CEMENT infers such relevance relationships using the past co-evolution of the software under analysis. We empirically evaluate the capability of CEMENT in capturing such relevance relationships using 15 open-source projects. We further conducted a fault localization study to investigate the applicability and actionability of these relationships in software debugging. The results show that CEMENT can establish and use such relationships and that it is capable of achieving comparable performance to an existing Information Retrieval-based fault localization technique. Further analysis reveals that CEMENT can capture the relationships between tests and code that are different from those captured by dynamic and static code analysis and thus, evidencing that CEMENT can complement the current approaches.

Our work forms the first attempt to establish evolutionary couplings between tests and code and thus, it opens a number of interesting research directions. In particular, the exploration of additional aspects of software evolution such as actual commit time, developers and the context of the evolution could strengthen our relationships. Additionally, the formulation of hybrid techniques combining evolutionary coupling with traceability linking techniques [21, 23, 32] and historical evolution analysis techniques such as

refactoring miner [29] could lead to much stronger results. We hope to explore these directions in the near future.

REFERENCES

- [1] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3377811.3380369>
- [2] Sam Blackshear, Nikos Goriogiannis, Peter W. O’Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOP-SLA, Article 144 (oct 2018), 28 pages. <https://doi.org/10.1145/3276514>
- [3] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (feb 2013), 82–90. <https://doi.org/10.1145/2408776.2408795>
- [4] Scott Chacon and Ben Straub. 2014. *Pro git*. Apress.
- [5] Sebastian Elbaum, Gregg Rothmel, and John Penix. 2014. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (Hong Kong, China) (FSE 2014)*. ACM, New York, NY, USA, 235–245. <https://doi.org/10.1145/2635868.2635910>
- [6] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite: On the Challenges of Test Case Generation in the Real World. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13)*. IEEE Computer Society, USA, 362–369. <https://doi.org/10.1109/ICST.2013.51>
- [7] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. <https://doi.org/10.1109/TSE.2017.2755013>
- [8] Mohammad Ghafari, Carlo Ghezzi, and Konstantin Rubinov. 2015. Automatically identifying focal methods under test in unit test cases. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 61–70. <https://doi.org/10.1109/SCAM.2015.7335402>
- [9] Mark Harman and Peter O’Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23. <https://doi.org/10.1109/SCAM.2018.00009>
- [10] René Just, Dariouh Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (San Jose, CA, USA) (ISSTA 2014)*. ACM, New York, NY, USA, 437–440. <https://doi.org/10.1145/2610384.2628055>
- [11] Yasutaka Kamei and Emad Shihab. 2016. Defect Prediction: Accomplishments and Future Challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 33–45. <https://doi.org/10.1109/SANER.2016.56>
- [12] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. <https://doi.org/10.1109/TSE.2012.70>
- [13] Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. *CoRR abs/1902.02703* (2019). [arXiv:1902.02703](http://arxiv.org/abs/1902.02703) <http://arxiv.org/abs/1902.02703>
- [14] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report Driven Program Repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Tallinn, Estonia) (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 314–325. <https://doi.org/10.1145/3338906.3338935>
- [15] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Athens, Greece) (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. <https://doi.org/10.1145/3468264.3468580>
- [16] Shane McIntosh and Yasutaka Kamei. 2018. Are Fix-Inducing Changes a Moving Target? A Longitudinal Case Study of Just-In-Time Defect Prediction. *IEEE Transactions on Software Engineering* 44, 5 (2018), 412–428. <https://doi.org/10.1109/TSE.2017.2693980>
- [17] Sonu Mehta, Farima Farmahinifarahani, Ranjita Bhagwan, Suraj Guptha, Sina Jafari, Rahul Kumar, Vaibhav Saini, and Anirudh Santhiar. 2021. *Data-Driven Test Selection at Scale*. Association for Computing Machinery, New York, NY, USA, 1225–1235. <https://doi.org/10.1145/3468264.3473916>
- [18] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242. <https://doi.org/10.1109/ICSE-SEIP.2017.16>
- [19] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Advances in Computers*, Vol. 112. Elsevier, 275–378. <https://doi.org/10.1016/bs.adcom.2018.03.015>
- [20] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (Toronto, Ontario, Canada) (ISSTA 2011)*. ACM, New York, NY, USA, 199–209.
- [21] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2011. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 63–72. <https://doi.org/10.1109/ICSM.2011.6080773>
- [22] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. 432–441. <https://doi.org/10.1109/ICSE.2013.6606589>
- [23] Bart Van Rompaey and Serge Demeyer. 2009. Establishing Traceability Links between Unit Test Cases and Units under Test. In *2009 13th European Conference on Software Maintenance and Reengineering*. 209–218. <https://doi.org/10.1109/CSMR.2009.39>
- [24] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 345–355.
- [25] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering (Montreal, Quebec, Canada) (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 13–24.
- [26] Jeongju Sohn, Gabin An, Jingun Hong, Dongwon Hwang, and Shin Yoo. 2021. Assisting Bug Report Assignment Using Automated Fault Localisation: An Industrial Case Study. In *Proceedings of the 14th IEEE International Conference on Software Testing, Verification and Validation*.
- [27] J. Sohn and S. Yoo. 2019. Empirical Evaluation of Fault Localisation Using Code and Change Metrics. *IEEE Transactions on Software Engineering* (2019), 1–1. <https://doi.org/10.1109/TSE.2019.2930977>
- [28] Gregory Tassej. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Technical Report. National Institute of Standards and Technology.
- [29] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. Refactoring-Miner 2.0. *IEEE Transactions on Software Engineering* (2020), 21 pages. <https://doi.org/10.1109/TSE.2020.3007722>
- [30] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 1–11.
- [31] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating Bugs from Software Changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (Singapore, Singapore) (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 262–273. <https://doi.org/10.1145/2970276.2970359>
- [32] Robert White, Jens Krinke, and Raymond Tan. 2020. Establishing Multi-level Test-to-Code Traceability Links. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (Seoul, South Korea) (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 861–872. <https://doi.org/10.1145/3377811.3380921>
- [33] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (August 2016), 707.
- [34] Jianwei Wu and James Clause. 2020. A pattern-based approach to detect and improve non-descriptive test names. *Journal of Systems and Software* 168 (2020), 110639. <https://doi.org/10.1016/j.jss.2020.110639>
- [35] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated Test Input Generation for Android: Are We Really There yet in an Industrial Case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 987–992. <https://doi.org/10.1145/2950290.2983958>
- [36] Benwen Zhang, Emily Hill, and James Clause. 2015. Automatically Generating Test Templates from Test Names (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 506–511. <https://doi.org/10.1109/ASE.2015.68>
- [37] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 199–209. <https://doi.org/10.1145/3180155.3180198>

[38] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. 2004. Mining Version Histories to Guide Software Changes. In *26th International Conference on Software Engineering (ICSE 2004)*, 23-28

May 2004, Edinburgh, United Kingdom. IEEE Computer Society, 563-572.
<https://doi.org/10.1109/ICSE.2004.1317478>