# A Class-level Test Selection Approach Toward Full Coverage For Continuous Integration

Yingling Li*‡, Junjie Wang*‡, Qing Wang*†‡§, Jun Hu*‡
*Laboratory for Internet Software Technologies, †State Key Laboratory of Computer Science
Institute of Software,Chinese Academy of Sciences,Beijing,100089,China
‡University of Chinese Academy of Sciences,Beijing,100089,China,§Corresponding author
email:{yingling,wangjunjie,wq,hujun}@itechs.iscas.ac.cn

***Abstract*—Continuous Integration (CI) is an important practice in agile development. With the growth of integration system, running all tests to verify the quality of submitted code, is clearly uneconomical. This paper aims at selecting a proper test subset towards full coverage of all changed and affected code so as to reduce the cost of CI testing. We proposes FEST, a novel approach, which searches for the full dependencies of changed code at the class level and then selects test classes related to the changed and affected classes. We assess FEST from fault detection efficiency and cost effectiveness based on 18 open source projects with 261 continuous integration versions from Eclipse and Apache communities, and compare it with the state-of-the-art approach ClassSRTS (as baseline). Results show that FEST (1) can not only cover all faults detected by actual CI testing and baseline, but also find new faults in 25% and 18% versions respectively. (2) shows better or equal test scale benefits than actual CI testing (in 98% versions) and baseline (in 99% versions); and can compensate risk of omitting necessary tests for actual CI testing (in 62% versions) and baseline (in 73% versions).**

## I. Introduction

*Continuous Integration* (CI) is a widely-applied development practice which requires developers to integrate their code into the master codebases frequently. It can improve the productivity, facilitate fast feedback of quality problems, and help projects release more often [1], [2]. One of the best practices in CI is to make your build self-testing, called CI testing. Each programmer must do a complete build and run (and pass) all or eligible unit tests before submitting work [1], [3]. With the growth of integration system, running all tests to verify the quality of submitted code, is clearly uneconomical. The big challenge of CI testing is how to optimize the test set to reduce test size as much as possible without sacrificing quality. These are many test case selection techniques proposed to tackle this problem [2], [4]–[16].

Generally speaking, the methods of test case selection can be split into static and dynamic techniques. Dynamic selection technique collects test dependencies by dynamically running tests on the previous revision, and selects a test set that may reach the code changes [7]–[9], [15], [17]. However, dynamic test dependencies for large projects may be time-consuming to collect, and for real-time systems, dynamic selection techniques may not be applicable, because code instrumentation for obtaining dependencies may cause timeouts or interrupt normal test run [5], [15]. Besides, for programs with non-determinism (e.g., due to randomness or concurrency), dependencies collected dynamically may not cover all possible traces, leading to omission of necessary tests [5], [6]. Static selection technique does not require to execute tests; it uses static program analysis to infer the dependencies between changed code, affected code and test code [4], [5], [13], [18], [19]. It is easier to manipulate than dynamic technique thus draws more and more attention. However, the drawback of existing static approaches is that the selected test sets either cannot fully cover the necessary tests or are too redundant due to inefficient static analysis techniques [4]–[6], [13], [20].

In this paper, we propose a novel test selection approach called FEST (i.e., Full dEpendency based class-level test SelecTion) to select a proper test subset for CI testing based on class-level static dependencies. FEST first locates the changed classes of submitted commits, and generates full dependencies of source code at the class level. It then searches a complete set of affected classes and identifies affected test classes. Finally, it extends the test classes based on recently failed tests.

We experimentally evaluate FEST from fault detection efficiency and cost effectiveness on 261 CI versions of 18 projects from two large open source communities (i.e., Eclipse and Apache). For comparison, we refer the real-world practice of CI testing as the ***actual CI testing*** and apply the state-of-the-art approach ClassSRTS [5], [13] as baseline. The results show that (1) FEST can cover all faults detected by actual CI testing and baseline, and find new faults in 25% and 18% versions respectively; (2) compared with baseline, FEST shows better or equal test scale benefits in 99% versions, and higher risk compensation in 73% versions.

The main contributions of the paper are as follows:

- We design a new class-level test selection approach (FEST) to select a proper test subset towards fully covering all changed code and affected code. It can resolve full dependency relations at the class level, which improves previous work by capturing the dependencies of hidden references; and design an algorithm to incrementally and iteratively search the affected tests.
- We conducted experiments on 261 integration versions of 18 open source projects to evaluate the fault detection efficiency and cost-effectiveness of our approach, and results are promising.

DOI reference number: 10.18293/SEKE2019-011

## II. Approach

This paper proposes an approach, called FEST (i.e., Full dEpendency based class-level test SelecTion). There are four major steps in FEST (shown in Figure 1), the following subsections will explain the details of the four steps.

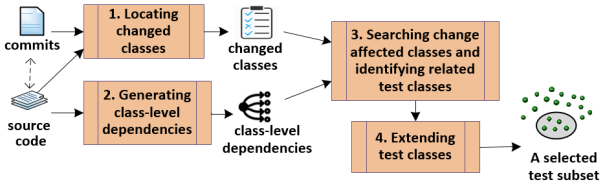


Fig. 1. The overview of FEST

### A. Locating the changed classes of submitted commits

For each CI, we have three phases to analyze and locate the changed classes. Firstly, we look for the **names of changed files** from the log message of the commits. Secondly, we look for the **changed lines of each changed file** by applying "*git diff*", and filter the blank lines and annotation lines. Thirdly, we locate the changed code on associated classes. In detail, an improved open source tool Doxygen (named as Doxygen#) is applied to analyze the structure and dependencies of source code. We further parse the *xml* files generated by Doxygen# to obtain the code structure information of each file, including class names, the start line and end line of each class, and annotation information of each method (e.g., *@Test*, *@BeforeClass*, *@Before*). Based on the code structure information, we map the changed lines to their corresponding classes, and then label them as changed classes. Note that the commits can involve both program code and test code. We also locate changed test classes for better selecting the related tests. After this step, FEST will output a set of changed classes (*C_Set*).

### B. Generating the full dependencies of code version

To ensure the full dependencies of code captured, we refer to the relations defined in *UML*. There are six relations among classes or objects defined in *UML*. Table I presents these relations and their representation in code (refer Orso's work [21]), as well as the corresponding relations in code. There are two categories of relations in code: inheritance and invocation [22]. Some of the invocation types can be directly obtained with code dependency analysis tools (e.g., Doxygen# ), and we call it as direct reference (short for *R*). Other invocation types can not be directly obtained with existing tools, and we call it hidden reference (short for *HR*). These hidden reference relations are usually ignored in the existing approaches [5], [20] or tools for code dependency analysis (e.g., Doxygen) because they are hard to resolve. We carefully consider and resolve the relation.

For inheritance and reference relations of invocation category, we directly capture them using the tags (e.g., *derivedcompoundref*, *referencedby*) in *xml* files outputted by Doxygen#. For hidden reference, we parse the *xml* files based on its representation in code (as shown in Table I) and the code structure information of class files obtained in Step 1. Take the first *dependency* relation as an example (i.e., the invocation by "*Class.forName('P.e2')*" in reflection), for each *xml* file, we check whether there is such hidden reference relation by its representation in code. If there is "*Class.forName*" in the *xml* file, we regard it as this case, and obtain the information of the referenced class (i.e., *e2* as the name of referenced class, *P* as the name of *e2*'s package), and record its position (i.e., line of code); then we map the position to corresponding class based on the code structure information, and obtain the reference class and its information (e.g., class name). Finally, the hidden reference can be built with the obtained information of reference and referenced classes.

TABLE I
Mapping of relations between relations in *UML* and in code

| Relations in UML | Representation in code | Relations in code |
|---|---|---|
| *Generalization* | A subclass $e1$ extends its superclass $e2$. | *Inheritance* |
| *Realization* | Class $e1$ implements an interface $e2$ | *Inheritance* |
| *Dependency* | A Method of Class $e1$ usesClass $e2$ as an argument, e.g., the invocation by "*Class.forName('P.e2')*" in reflection | *Invocation (HR)* |
| | A Method of Class $e1$ uses Class $e2$ in a cast operation. | *Invocation (HR)* |
| | Class $e1$ instantiates Class $e2$ in $e1$'methods, then invokes its methods or variables. | *Invocation (R)* |
| *Association* | Class $e1$ uses Class $e2$ as a member variable. | *Invocation (HR)* |
| | A method in Class $e1$ invokes methods or member variables of member Class $e2$. | *Invocation (R)* |
| | Class $e1$ defines a member variable, and initiates it by invoking Class $e2$'s methods or variables, or using $e2$ in a cast operation. | *Invocation (HR)* |
| *Aggregation* | Class $e1$ uses Class $e2$ as an argument to instance. | *Invocation (HR)* |
| *Composition* | If Class $e2$ is a component of Class $e1$, it can only be instantiated in Class $e1$. | *Invocation (R)* |

For convenience of step 3, FEST outputs an inheritance relation graph (*HRG*) and an invocation relation graph (*VRG*).

### C. Searching affected classes and identifying related test classes

In this section, we design a BFS-based (i.e., Breadth-First Search) search algorithm to incrementally and iteratively look for a complete set of affected classes and identify related test classes. Note that since our approach considers the class-level dependency relations, the *tests* in this paper means the test classes.

In detail, for each changed class $C_i$ in *C_Set*, firstly, we use BFS to search its all subclasses from *HRG*, and all referenced classes from *VRG* which invoke class $C_i$ or its subclasses, and add them to the set of affected classes *A_Set*. Then, for the changed class $C_i$ or each affected class, we check whether it is a test class. Note that we also check the changed class $C_i$, which ensures to select the newly-added tests or changed tests. Following existing work [6], [20], if it contains at least one method (the method name started with "test" or the method has the annotation "*@Test*", "*@BeforeClass*", "*@Before*"), we regard it as a test class. If it is a test class, we will add it to the set of related tests *T_Set*. Finally, when this loop is over, and *A_Set* is not null, for each class $a_i$ in *A_Set*, we apply BFS iteratively to search its subclasses and referenced classes until there is no new class found, and identify new test classes, add

them to *T_Set*. After the above process, we search affected classes and identify the set of related tests *T_Set*.

### D. Extending test classes

In this section, we aim to complement some tests to re-detect the deferred faults. These faults detected in the previous versions might have not been fixed and deferred to subsequent versions. This strategy has also been applied in existing studies [2], [11], [23]. In detail, we look for the failed test classes in recent *n* versions, then check whether they run again. If not, we add them into *T_Set*. We have experimented *n* from 1 to 30, and results showed that the performance is better and more stable when *n* is 10. Hence, we apply the recent 10 versions in the following experiments. Finally, we obtain a final test set (*T_Set*).

## III. Experiment Design and Evaluation Metrics

### A. Subject projects and data preparation

According to the activeness and CI testing history availability, we chose 12 Eclipse projects and 6 Apache projects to evaluate our approach presented in Section II. We collected the CI testing history from November 2017 to January 2018 for the chosen projects, which contains test classes, test methods, results, etc. From which we can obtain the failed test information for each CI version. Then we collected commits from the *Git* repository by executing "*git log*", checked out source code by executing "*git checkout*"; and built the relationship of the three datasets by the commit *id* of CI versions. Finally, we chose the versions for our experiment, in which the number of changed classes and the number of executed CI tests are greater than 0; and obtained 261 versions from the 18 projects.

We ran all experiments on a 3.40 GHz Intel Core i7-3770 machine with 8 GB of RAM, running Ubuntu Linux 14.04.3 LTS and Java 64-Bit sever version 1.8.0_73.

### B. Baseline approach

To further evaluate the performance of our approach, we take ClassSRTS [5], [13], the state-of-the-art class-level static selection technique, as baseline. Besides, we also compare FEST with the selected tests of actual CI testing, which are recorded in the repositories.

We did not select any dynamic approach as baseline because in the most recent work [5], ClassSRTS is compared to the state-of-the-art dynamic approach, and results show that ClassSRTS is comparable with the dynamic approach. The reason we did not compare with dynamic selection is because it needs to run tests to obtain the dependency information, which is costly and limits its realization in our study.

### C. Evaluation metrics

Following existing work, faults denote failed test classes where a test class is regarded as fail if at least one of the test cases in the test class fails, otherwise, it is regarded as pass. For each CI version, we compare the selected tests and the failed tests in the CI testing history, and count the failed test classes contained in the set of selected tests to get the number of detected faults. Regarding a new test class which is not included in the current set of actual CI testing, it will be labeled as a detected fault if its first run in the subsequent versions fails by following previous work [24]. It is because it should be detected in the previous version, but it was omitted by actual CI testing. **To facilitate understanding, we present basic metrics used in this paper in Table II**.

TABLE II
Basic metrics used in the study

| Metrics | Description |
|---|---|
| $Ct$ | A set of actual CI tests (i.e., actual tests ran during CI testing). |
| $St$ | A set of selected tests by FEST or ClassSRTS (marked by *St@F* and *St@C* respectively). |
| $Cft$ | A set of faults detected by actual CI testing. |
| $Tft$ | A set of total faults detected by FEST or ClassSRTS (marked by *Tft@F* and *Tft@C* respectively). |

*1) Fault detection efficiency:* In this dimension, we define two metrics to evaluate fault detection efficiency.

*(a)* ***Fault coverage*** *(Fcovg)* means the percentage of the faults found by test selection approach (i.e., FEST or ClassSRTS) compared with the faults detected in actual CI testing. Following existing work [4], [12], [20], [25], it is defined as Equation 1.

$$Fcovg = \frac{|Tft \cap Cft|}{|Cft|} \times 100\% \quad (1)$$

In it, *Tft* is the faults detected by the measured approach, it can be *Tft@F* (for FEST) or *Tft@C* (for ClassSRTS). This metric is also used to evaluate the coverage of the faults detected by FEST compared with the faults detected by ClassSRTS, where *Cft* will be replaced by *Tft@C*. When $|Cft|$ is 0, we treat *Fcovg* as 100%.

*(b)* ***Fault detection enhancement*** *(Fenhm)* means the new faults which are detected by the measured approach compared with actual CI testing or ClassSRTS. It is defined as *NewFt*.

*2) Cost effectiveness:* In this dimension, we evaluate the cost effectiveness from test scale benefits and risk compensation ability.

*(a)* ***Test scale benefits*** *(Testscal)* evaluates the return on investment (ROI), i.e., the number of detected faults divided by the number of running tests, as the existing work [9], [26], [27]. As we could not obtain the exact set of all faults, we use the union set of the faults detected by all investigated approaches and actual CI testing, as previous work [12], [20]. Therefore, we apply a penalty coefficient to adjust ROI, which is the proportion of the number of faults detected by current approach to the number of all faults. Hence, *Testscal* is defined as Equation 2. The higher value of *Testscal* the better.

$$Testscal = \frac{|Tft|}{|St|} \times \frac{|Tft|}{|Cft \cup Tft@F \cup Tft@C|} \quad (2)$$

Here, *Tft* is the fault set of the measured approach, i.e., it can be *Cft* (for actual CI testing), *Tft@F* (for FEST) or *Tft@C* (for ClassSRTS). When $|St|$ or $|Cft \cup Tft@F \cup Tft@C|$ is 0, we simply treat *Testscal* as 0.

*(b)* ***Risk compensation ability*** *(Riskcomp):* Running insufficient tests for CI could lead to omit some potential faults, while running necessary and sufficient tests will decrease the

quality risk even if sometimes they did not find more faults. We define risk compensation ability to evaluate the ability of compensating the omitted necessary tests by actual CI testing. It is defined as Equation 3.

$$Riskcomp = \frac{|St| - |(St@F \cup St@C) \cap Ct|}{|St@F \cup St@C|} \quad (3)$$

Here, *St* is the test set of the measured approach, it can be *St@F* or *St@C* (for *Riskcomp* of FEST or ClassSRTS). $St@F \cup St@C$ means the union set of tests selected by FEST and ClassSRTS respectively. $(St@F \cup St@C) \cap Ct$ is the test set running in actual CI testing and also selected by FEST or baseline. The positive *Riskcomp* means that the measured approach can compensate risk for actual CI testing, the higher value of *Riskcomp* the better. In contrast, the negative *Riskcomp* means it can not compensate risk. When $|St@F \cup St@C|$ is 0, we simply treat *Riskcomp* as 0.

## IV. Experiment Results and Analysis

Based on the number of faults detected by actual CI testing and FEST, we divide the experiment project versions (261) into four categories. The following subsections will present their results and analysis respectively.

- Category *Dual_F* (9%:23): both actual CI testing and FEST detected faults.
- Category *FEST_F* (20%:51): actual CI testing did NOT detect faults, while FEST detected faults.
- Category *Dual_NF* (70%:184): both actual CI testing and FEST did NOT detect faults.
- Category *CI_F* (1%:3): actual CI testing detected faults, while FEST did NOT detect faults.

### A. Category Dual_F

In Figure 2, the size of bubbles refers to the number of tests. We order versions by reduced test size between FEST and actual CI testing, then assign the ID sequentially.

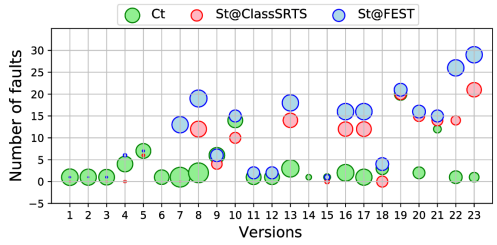


Fig. 2. Number of selected tests and number of detected faults for Dual_F

We can easily see that blue bubbles are either higher or inside other bubbles. It means that FEST can find more or equal number of faults than ClassSRTS or actual CI testing. Particularly, in some versions, the difference is quite large.

*1) Fault detection efficiency:* In Figure 3, FEST can not only cover all faults detected by actual CI testing and ClassSRTS, but also find new faults in 65% (15/23) and 83% (19/23) versions respectively. While ClassSRTS can cover the faults detected by actual CI testing only in 43% (10/23) versions, and find new faults in 48% (11/23) versions.

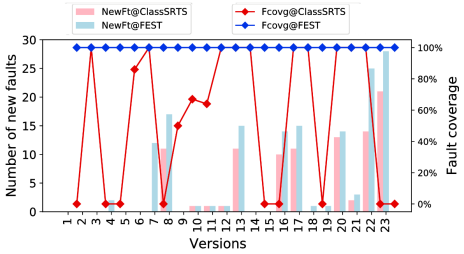


Fig. 3. Fault coverage and detection enhancement compared with actual CI testing for Dual_F

*2) Cost effectiveness:* In Figure 2, we observe that FEST selects slightly more tests than ClassSRTS in some versions. We further analyze the cost effectiveness of FEST.

***Test scale benefits:*** From Figure 4, we can see that FEST shows better *Testscal* than actual CI testing in 96% (22/23) versions. Only in one version (v21), both FEST and ClassSRTS show slightly lower *Testscal* than actual CI testing. Compared with ClassSRTS, FEST shows better or equal *Testscal* in 87% (20/23) versions; in other three versions (v2, v19, v20), FEST shows lower *Testcal*. For these 4 versions where FEST shows lower *Testcal*, it has higher *Riskcomp* (see the next paragraph), which indicates our proposed approach can mitigate the risk of omitting necessary tests.

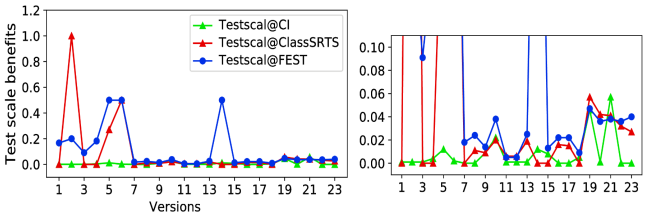


Fig. 4. Test scale benefits for Dual_F (the enlarged figure on the right shows the details)

***Risk compensation ability:*** In Figure 5, compared with actual CI testing, FEST has positive *Riskcomp* in all versions, while ClassSRTS shows negative *Riskcomp* in 35% (8/23) versions. It means that in these 35% versions, ClassSRTS omits some necessary tests, which might lead to omit faults.

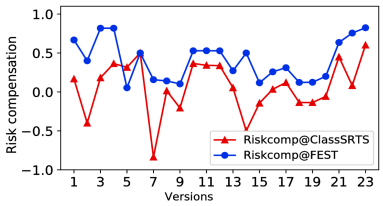


Fig. 5. Risk compensation ability for Dual_F

Compared with ClassSRTS, FEST shows higher *Riskcomp* in 96% (22/23) versions. It indicates that FEST can compensate more necessary tests than ClassSRTS, even though FEST selects more test in some cases. Only in 1 versions (v5), FEST shows lower *Riskcomp* than ClassSRTS. This is because when computing *Riskcomp*, we treat the union set of tests selected by FEST and ClassSRTS as ground truth of necessary tests. However, ClassSRTS has selected some unnecessary tests. In detail, it treats the selected tests as the difference between the set of all tests and the set of non-affected tests (tests not affected by changed code). However, the set of all tests could include some unnecessary classes, e.g., the basic classes which

do not have test cases. Hence, the lower *Riskcomp@FEST* does not indicate the low effectiveness of our approach.

### B. Category FEST_F

Similar to category Dual_F, in Figure 6, the size of bubbles refers to the number of tests.

*1) Fault detection efficiency:* In Figure 6, all green bubbles lie on X-axis due to $|Cft| = 0$. It indicates that compared with actual CI testing, the fault coverage of both FEST and ClassSRTS are all 100%. More than that, FEST can cover all faults detected by ClassSRTS and detect more new faults in 57% versions.

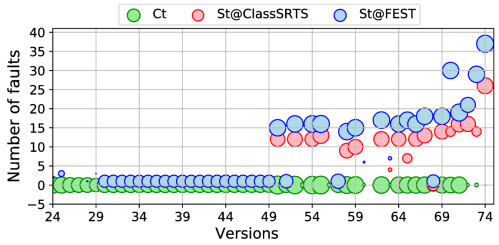


Fig. 6. Number of selected tests and number of detected faults for FEST_F

*2) Cost effectiveness:* In Figure 6, test size of FEST is smaller than actual CI testing, and similar or slightly larger than ClassSRTS.

***Test scale benefits:*** In this category, *Testscal@CI* = 0 in all versions. From Figure 7, we can observe that FEST shows better *Testscal* than actual CI testing, and better or equal *Testscal* than ClassSRTS in all versions. Especially, in 8 versions, *Testscal* of FEST is more than three times as large as that of ClassSRTS.

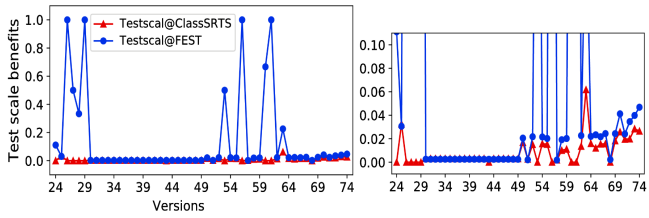


Fig. 7. Test scale benefits for FEST_F (the enlarged figure on the right shows the details)

***Risk compensation ability:*** In Figure 8, FEST has positive *Riskcomp* in 90% (46/51) versions and none negative *Riskcomp*, while ClassSRTS shows negative *Riskcomp* in 24% versions. Meanwhile, compared with ClassSRTS, FEST has better or equal *Riskcomp* in 96% (49/51) and 2% (1/51) versions respectively. The reason of lower *Riskcomp* in other 1 version (v63) is similar with the discussion in Section IV-A2.

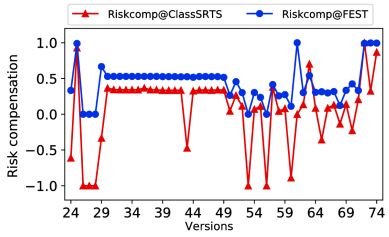


Fig. 8. Risk compensation ability for FEST_F

### C. Category Dual_NF

In all versions of this category, neither actual CI testing nor FEST or ClassSRTS detected faults (i.e., *Fcovg*, *Fenhm* and *Testscal* are 0). Therefore, we only discuss risk compensation ability for this category.

***Risk compensation ability:*** In Figure 9, FEST has positive *Riskcomp* in 49% versions and none negative *Riskcomp*, while ClassSRTS shows positive and negative *Riskcomp* in 28% and 47% versions respectively. Compared with ClassSRTS, FEST presents higher or equal *Riskcomp* than ClassSRTS in 91% (118/261) versions, among which *Riskcomp* of FEST is twice as large as that of ClassSRTS in 109 versions. The reason of lower *Riskcomp* in 9% versions is similar with the discussion in Section IV-A2.

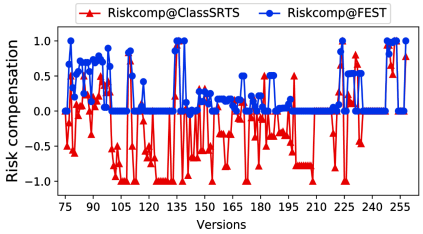


Fig. 9. Risk compensation ability for Dual_NF

### D. Category CI_F

*1) **fault detection efficiency**:* In all three versions (i.e., v259, v260 and v261) of this category, actual CI testing detected a few faults while ran a large size of tests. Both FEST and ClassSRTS selected very small test sets while detected no faults. We further examine the detail of these 3 versions.

In the 3 versions, the faults which are not covered by FEST do not have any dependencies with the changed and affected code, and have no failed test history in recent 10 versions. In other words, these faults were committed in earlier versions, and should be detected earlier. Hence, detecting these faults is not the responsibility of current CI testing. This indicates that the bad performance does not due to the drawback of FEST.

*2) Cost-effectiveness:* In this category, both FEST and ClassSRTS did not detect any faults, *Testscal* of both FEST and ClassSRTS are zero, lower than actual CI testing.

***Risk compensation ability:*** Regarding risk compensation ability, both FEST and ClassSRTS positively compensate risk in all versions, but FEST can get much higher *Riskcomp* for all of them even though it did not find any faults. It is because actual CI testing ran some redundant tests while omitting necessary tests which results in a high risk of omitting faults. Meanwhile, FEST presents better or equal *Riskcomp* than ClassSRTS in all versions.

**Summary**: Based on the detailed results and analysis of the 4 categories, we obtain the summary of FEST across all categories: (1) FEST can cover all faults detected by actual CI testing and baseline, and find new faults in 25% and 18% versions respectively; (2) FEST shows better or equal *Testscal* than actual CI testing (in 98% versions) and ClassSRTS (in 99% versions); can compensate risk of omitting necessary tests for actual CI testing (in 62% versions) and baseline (in 73% versions).

## V. Validity

The **internal** validity of our study arises from the implementation of baseline and FEST. We implemented baseline by strictly following the steps described in the original paper [13]. For both baseline and FEST approaches, we have employed 213 test cases to test their functionality. For the suspicious results, we have manually checked the code, and found they do not due to the defect in code.

The **external** validity concerns about our experimental dataset. We can not guarantee that our results can be fully generalize to other projects. However, the size of the dataset (18 projects with 261 versions) and the diversity of domains relatively reduce this risk.

## VI. Related Work

**CI test selection based on dynamic dependency:** Gligoric et al. [7] and Vasic et al. [19] implemented a test selection approach based on dynamic dependencies at the file level for $Java$ and .NET respectively, and analyzed the strengths and drawbacks of file-level and module-level test selections. Because of that, Zhang [15] proposed a hybrid test selection technique that combined file-level and method-level analysis. Furthermore, Celik et al. [8] designed a file-level test selection across *JVM* boundaries, which can find more dependencies and improve precision of selection.

**CI test selection based on static dependency:** Soetens et al. [25] and Parsai et al. [4] proposed an approach to select tests on method-level static dependencies and improved it to deal with invocation in polymorphism. Legunsen et al. [5], [13] implemented class-level and method-level test selection techniques based on static dependencies. But the class-level approach would still omit some tests because it can not obtain all dependencies (e.g., the dependencies in reflection and cast operation), while the method-level approach shows worse performance in fault detection and time cost.

Generally speaking, the drawbacks with existing dynamic selection approaches are long running tests, non-determinism, and real-time constraints; while the drawbacks of existing static selection approaches lie in omitting some tests or selecting some unnecessary tests [4]–[6], [13], [20]. Our approach belongs to static test selection which improves existing approaches by resolving full dependency relations and selecting a test set towards fully covering all changed and affected code.

## VII. Conclusion

In this paper, we propose FEST to select a proper test subset towards full coverage of all changed and affected code so as to reduce the cost of CI testing. Evaluations are conducted on 18 projects with 261 CI versions from Eclipse and Apache communities. Results show that FEST can outperform actual CI testing and the state-of-the-art ClassSRTS in terms of fault detection efficiency and cost-effectiveness in most cases.

## VIII. Acknowledgments

We sincerely thank Yun Yang for his contribution to the paper. This work is supported by National Natural Science Foundation of China under Grant No.61602450, No.61432001.

## References

[1] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in FSE'15, pp. 805–816.
[2] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in FSE'14, pp. 235–245.
[3] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in ASE'16, pp. 426–437.
[4] A. Parsai, Q. D. Soetens, A. Murgia, and S. Demeyer, "Considering polymorphism in change-based test suite reduction," in Agile'14, pp. 166–181.
[5] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in FSE'16, pp. 583–594.
[6] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse, "Test case selection in industry: An analysis of issues related to static approaches," SQJ'16, pp. 1–35.
[7] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in ISSTA'15, pp. 211–222.
[8] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across JVM boundaries," in FSE'17, pp. 809–820.
[9] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in ICSE'15, pp. 483–493.
[10] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in ICSE'17, pp. 233–242.
[11] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in ISSTA'17, pp. 12–22.
[12] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in ICST'15, pp. 1–10.
[13] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in ASE'17, pp. 949–954.
[14] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in FSE'15, pp. 237–247.
[15] L. Zhang, "Hybrid regression test selection," in ICSE'18, pp. 199–209.
[16] K. Wang, C. G. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection," in ICSE'18, pp. 233–244.
[17] G. Wikstrand, R. Feldt, J. K. Gorantla, and C. Zhe, W.and White, "Dynamic regression test selection based on a file cache: An industrial evaluation," in ICST'09, pp. 299–302.
[18] E. D. Ekelund and E. Engström, "Efficient regression testing based on test history: An industrial evaluation," in ICSME'15, pp. 449–457.
[19] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .Net," in FSE'17, pp. 848–853.
[20] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: An empirical evaluation," ESE'16, vol. 21, no. 5, pp. 1990–2032.
[21] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," Acm Sigsoft Software Engineering Notes, vol. 29, no. 6, pp. 241–251, 2004.
[22] B. Meyer, Object-Oriented Software Construction. Prentice Hall, New York, N.Y., second edition, 1997.
[23] S. Yoo, R. Nilsson, and M. Harman, "Faster fault finding at google using multi objective regression test optimization," in ESEC/FSE'11, pp. 1–4.
[24] A. Beszédes, T. Gergely, L. Schrettner, J. Jász, L. Langó, and T. Gyimóthy, "Code coverage-based regression test selection and prioritization in webkit," in ICSM'12, pp. 46–55.
[25] Q. D. Soetens, S. Demeyer, and A. Zaidman, "Change-based test selection in the presence of developer tests," in CSMR'13, pp. 101–110.
[26] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-constrained regression test case selection using multicriteria optimization," TSE'12, pp. 936–956.
[27] E. EngstrÖm, P. Runeson, and A. Ljung, "Improving regression testing transparency and efficiency with history-based prioritization - an industrial case study," in ICST'11, pp. 367–376.