The Journal of Systems and Software 167 (2020) 110614

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An extensive study of class-level and method-level test case selection for continuous integration☆

Yingling Li[a,1], Junjie Wang[b,d,1], Yun Yang[e,f], Qing Wang[b,c,d,*]

[a] School of Computer Science and Technology, Southwest Minzu University, Chendu, Sichuan Province, China
[b] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, 100089, China
[c] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, 100089, China
[d] University of Chinese Academy of Sciences, Beijing, 100089, China
[e] School of Computer Science and Technology, Anhui University, Hefei, Anhui Province, China
[f] School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, Australia

ARTICLE INFO

Article history:
Received 16 September 2019
Revised 3 December 2019
Accepted 24 April 2020
Available online 28 April 2020

Keywords:
Continuous integration
Test case selection
Program dependencies
Class-level test case selection
Method-level test case selection
Dynamic execution rules

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 for continuous integration so as to reduce test cost as much as possible without sacrificing quality. We first propose a static test case selection framework *Sapient* which aims at selecting a precise test subset towards fully covering all changed code and affected code. There are four major steps: locate changes, build dependency graphs, identify related tests by searching dependency graphs, and extend tests. Based on *Sapient*, we then develop a class-level test case selection approach FEST and a method-level approach MEST. FEST captures the class-level dependencies, especially the hidden references, which facilitates the modeling of full dependency relations at the class level. MEST combines two dynamic execution rules (i.e., dynamic invocation in reflection and dynamic binding in inheritance) with static dependencies to support building the precise method-level dependencies. Evaluation is conducted on 18 open source projects with 261 continuous integration versions from Eclipse and Apache communities. Results show that both FEST and MEST can detect almost all faults of two baselines (i.e., actual CI testing and ClassSRTS), and find new faults compared with actual CI testing (in 25% and 26% versions) and ClassSRTS (in 18% and 27% versions) respectively. Furthermore, MEST outperforms FEST in reduced test size, fault detection efficiency and test cost, indicating method-level test case selection can be more effective than class-level selection. This study can further speed up the feedback and improve the fault detection efficiency of CI testing, and has good application prospects for the CI testing in large-scale and complex systems.

© 2020 Published by Elsevier Inc.

1. Introduction

Currently, many software applications are rapidly developed and deployed. Rapid iterations are used to implement new requirements while keeping high quality. *Continuous Integration* (CI) is a widely-applied development practice to make it work, which requires developers to integrate code into main codebases frequently (Hilton et al., 2016; Duvall et al., 2007; Vasilescu et al., 2015; Schermann et al., 2016). Each integration can be verified by an automated build (include compiling and integration testing), and only the code that have passed integration testing (i.e., CI testing) will be merged into the main codebases, thus it is critical to detect faults as soon as possible (Vasilescu et al., 2015; Elbaum et al., 2014). Many well-known software organizations and open source communities (e.g., Google, Microsoft, Cisco, Github, Eclipse, Apache, etc.) are campaigners of CI practice (Spieker et al., 2017; Elbaum et al., 2014; Memon et al., 2017; Hilton et al., 2016; Vasic et al., 2017; Marijan and Liaaen, 2016; Vasilescu et al., 2015; Labuschagne et al., 2017). CI tests the code automatically before it will be integrated into the main codebases to ensure its quality.

☆ This work is supported by the National Key Research and Development Program of China [grant number 2018YFB1403400], the Fundamental Research Funds for the Central Universities, Southwest Minzu University [grant number 2020NYBPY04], Natural Science Foundation of China [grant numbers 61602450, 61432001]; and is partly supported by Australian Research Council Discovery Project [grant number DP180100212] and Sichuan science and technology project [grant number 2019YJ0252].

* Corresponding author.
E-mail address: wq@itechs.iscas.ac.cn (Q. Wang).

[1] These authors contributed equally to this work.

https://doi.org/10.1016/j.jss.2020.110614
0164-1212/© 2020 Published by Elsevier Inc.

The big challenge is how to select an appropriate test case subset for continuous integration so as to reduce test size as much as possible without sacrificing quality. Running all test cases will consume a large amount of resource and get feedback in a long cycle, but selecting an improper test case subset will miss some required tests, and therefore increase quality risk (Jiang and Chan, 2016). Test case selection, which seeks to identify the test cases that are relevant to changes (Yoo and Harman, 2012), provides a perspective for tackling this challenge; thus many test case selection techniques are proposed (Parsai et al., 2014; Legunsen et al., 2016; Blondeau et al., 2016; Gligoric et al., 2015; Ekelund and Engström, 2015; Beszédes et al., 2012; Celik et al., 2017; Herzig et al., 2015; Elbaum et al., 2014; Memon et al., 2017; Spieker et al., 2017; Mondal et al., 2015; Legunsen et al., 2017; Shi et al., 2015; Ryder, 2004; Zhang, 2018; Wang et al., 2018; Nardo et al., 2015; Zhang et al., 2011; Vahabzadeh et al., 2018; Wikstrand et al., 2009).

Generally speaking, test case selection techniques can be divided into static selection and dynamic selection based on the methods of collecting dependencies. 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 (Gligoric et al., 2015; Celik et al., 2017; Herzig et al., 2015; Wikstrand et al., 2009; Zhang, 2018). However, collecting dependencies by dynamically executing tests is time-consuming, and for real-time systems, it may not even be applicable, because code instrumentation for obtaining dependencies may cause timeouts or interrupt normal test run (Legunsen et al., 2016; Zhang, 2018). 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 (Legunsen et al., 2016; Blondeau et al., 2016). 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 (Parsai et al., 2014; Ekelund and Engström, 2015; Legunsen et al., 2016; Vasic et al., 2017; Legunsen et al., 2017). It is easier to manipulate than dynamic technique thus draws more and more attention, but it might not cover all necessary tests or select some unnecessary tests (Soetens et al., 2016; Parsai et al., 2014; Blondeau et al., 2016; Legunsen et al., 2016; 2017). We further analyze existing static selection techniques, and find that 1) omission of dependencies leads to omit necessary tests; 2) imprecise dependencies result in unnecessary tests. These are due to imprecise static dependency analysis, or the fact that some dependencies are dynamically determined at runtime which could not be resolved by existing static analysis techniques (Legunsen et al., 2016; 2017; Soetens et al., 2016) and existing tools for dependency analysis (e.g., WALA[2], Doxygen[3]).

On the other hand, based on different granularities of code dependencies, test case selection techniques can also be divided into module-level selection (Vasic et al., 2017; Knauss et al., 2015), package-level selection (Ekelund and Engström, 2015), file-level selection (Gligoric et al., 2015; Celik et al., 2017; Vasic et al., 2017; Zhang, 2018), class-level selection (Legunsen et al., 2016; 2017), and method-level selection (Soetens et al., 2016; Blondeau et al., 2016; Parsai et al., 2014; Legunsen et al., 2016; Zhang, 2018). Existing studies (Vasic et al., 2017; Gligoric et al., 2015) compared the performance of selection techniques in terms of different granularities, and the results showed that coarser-grained selection (e.g., file-level) can reduce overhead, but would select more unnecessary tests; while finer-grained selection (e.g., method-level) might increase the overhead of capturing dependencies, but could collect more precise dependencies thus select a relative small test set and reduce the time of running tests. For example, prior study (Gligoric et al., 2015) showed that file-level selection may select twice as many tests as method-level selection on five Java projects from GitHub community.

In this paper, we propose a StAtic PrecIse tEst case selecTioN framework named *Sapient* which aims at selecting a precise test[4] subset towards fully covering all changed code and affected code so as to reduce test cost without sacrificing quality. There are four major steps: locate changes, build dependency graphs, identify related tests by searching dependency graphs, and extend tests considering real-world practice.

Based on *Sapient*, we then develop a class-level test case selection approach FEST and a method-level test case selection approach MEST. These two approaches adopt *Sapient* framework and mainly extend the second step, i.e., building precise dependency graph. Specifically, FEST captures the class-level hidden references to facilitate the building of the full dependency relations at the class level. MEST combines two dynamic execution rules (i.e., dynamic invocation in reflection and dynamic binding in inheritance) with static dependencies to help build the precise method-level dependencies.

Evaluation is conducted on 261 integration versions of 18 open source projects from two large open source communities (i.e., Eclipse and Apache) to evaluate FEST and MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness. For comparison, we employ two baselines, i.e., the real-world practice of CI testing and the state-of-the-art approach ClassSRTS (Legunsen et al., 2016; 2017)

We first compare each approach respectively with two baselines, then compare MEST with FEST, finally analyze why MEST performs well. Results show that FEST and MEST can detect almost all faults of two baselines, and find new faults compared with actual CI testing (in 25% and 26% versions) and ClassSRTS (in 18% and 27% versions) respectively. We then conduct comparison between FEST and MEST, and results show that (1) MEST can reduce test size by 48% compared with FEST; (2) MEST can cover all faults detected by FEST in 95% versions, and find new faults in 20% versions; (3) although MEST needs more time to capture the finer-grained dependencies, the end-to-end time of MEST is 32% of FEST. This indicates method-level test case selection approach can be more effective than class-level selection in reduced test size, fault detection efficiency, and end-to-end time.

The main contributions of the paper are as follows:

- A static test case selection framework *Sapient* which aims at selecting a precise test subset towards fully covering all changed code and affected code so as to reduce test cost without sacrificing quality.
- A class-level test case selection approach (FEST) adopting *Sapient* which can capture the class-level dependencies, especially hidden references, thus facilitate the modeling of the full dependency relations at the class level.
- A method-level test case selection approach (MEST) adopting *Sapient* which combines two dynamic execution rules with static dependencies to capture the precise method-level dependencies.
- Promising evaluation results based on 261 integration versions of 18 open source projects for the effectiveness of FEST and MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness, as well as the comparison between FEST and MEST.
- Public-accessible experimental dataset[5], which can facilitate the replication of our study and evaluation of other approaches.

[2] http://wala.sourceforge.net.
[3] http://www.doxygen.nl/.
[4] Note that, we interchangeably use *test case*, *test* and *test class* in this paper.
[5] https://github.com/edcba321/a/blob/master/CI_testing_history.rar.

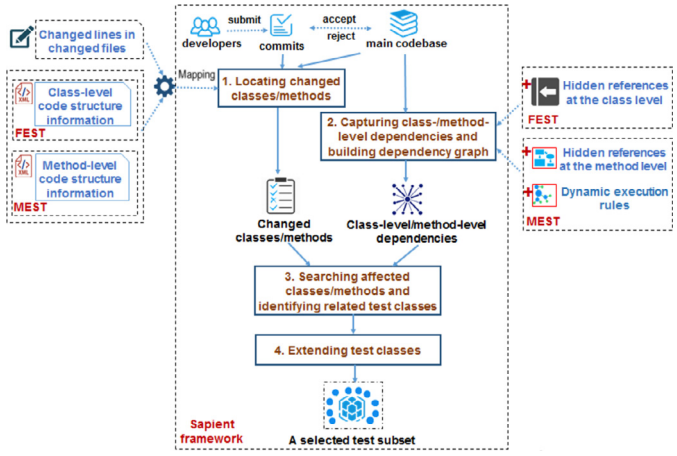

Fig. 1. The overview of *Sapient* framework.

- This paper extends a prior publication (presented at QRS'19 (Li et al., 2019b) which won the best paper award), and also adds prior work (presented at SEKE'19 (Li et al., 2019a)). The extensions of the paper are as follows: 1) new static precise test case selection framework *Sapient* (see Section 2); 2) more thorough experimental evaluation of FEST and MEST (see Section 6.1.1 and Section 6.1.2); 3) experimental comparison between FEST and MEST (see Section 6.2); 4) experimental evaluation of FEST and MEST by projects, detailed discussions of these two approaches as well as the state-of-the-art approach for test case selection (see Section 7).

The rest of this paper is organized as follows. Section 2 describes the *Sapient* framework. Following that, Sections 3 and 4 describe FEST approach and MEST approach respectively. Section 5 presents the design of experiment. Section 6 shows the results of our experiment. Section 7 discusses issues related to the proposed approaches and threats to the validity. Section 8 summarizes the related work, and Section 9 concludes the paper and points out future work.

2. *Sapient* framework

We propose *Sapient* framework which aims at selecting a precise test subset towards fully covering all changed code and affected code so as to reduce test cost without sacrificing quality. There are four major steps as shown in Fig. 1:

- Step 1: locate changed classes/methods of submitted commits;
- Step 2: capture precise class-level/method-level dependencies and build the dependency graph;
- Step 3: search for all affected classes/methods based on the dependency graph by incrementally and iteratively applying BFS, and identifying affected test classes (although our framework can analyze code dependencies at different levels, it selects test classes to aid comparison);
- Step 4: extend the test classes based on recently failed test classes considering real-world practice.

2.1. Locating changed classes/methods of submitted commits

In the first step, for each CI, we have three phases to analyze and locate the changed classes/methods. Firstly, we obtain the **names of changed files** from the log messages 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 map the changed lines to corresponding changed classes/methods. In detail, an improved open source tool Doxygen (named as Doxygen#) is applied to analyze the code structure and dependencies of source code. The improvement by our organization is to precisely identify the *override* and *overload* methods with the same name in different class files. We parse the *xml* files generated by Doxygen# to obtain class-/method-level code structure information. Based on the refined information, we map the changed lines to corresponding classes/methods, and label them as changed classes/methods.

Note that *the commits can involve both program code and test code, we also locate changed test classes/methods for better selecting the related tests*. After this step, we obtain the set of changed classes/methods (*C_Set*).

Definition 1: $C\_Set = \{C_1, C_2, \cdots, C_n\}$, *C_Set* is the set of changed classes/methods in the version, C_i is a changed class/method which includes at least one line of changed code in the commits.

2.2. Capturing the class-level/method-level dependencies of integrated version

The second step focuses on capturing the dependencies of integrated version and building the dependency graph. It requires the precise dependencies being modeled so as to facilitate the precise identification of affected test classes. We capture some dependencies by using tags (e.g., *derivedcompoundref, referencedby*) in *xml* files outputted by Doxygen#. In detail, for each *xml* file, we first parse it by tag *compounddef* to obtain id of the current class, and further parse by *derivedcompoundref* and *referencedby* tags to obtain all subclasses and invocation classes of the current class; finally, for each subclass or invocation class, we build the dependency between it and the current class.

There are two categories of relations in code: inheritance and invocation (Meyer, 1997). All the inheritance relations and part of the invocation types can be directly obtained with the aforementioned operation. However, some of the invocation types can not be directly obtained with existing code dependency analysis tools (we call it *hidden reference*). Besides, some dependencies are decided dynamically at runtime which can also not be directly obtained. The test case selection approaches which extend *Sapient*, such as FEST and MEST as shown below, can employ specific techniques to capture these dependencies in order to further improve the modeling of dependencies.

Definition 1 (Class-/Method-level Dependency Relation Graph - DRG)**.** An *DRG* is a 2-tuple $< N, E_v >$, where:

- N is the set of nodes representing all classes/methods in the version;
- $E_v \subseteq N \times N$ is the set of dependencies; there exists an edge $< n_1, n_2 > \in E_v$, if node n_1 invokes n_2, or node n_1 inherits n_2.

2.3. Searching affected classes/methods and identifying related test classes

Given a set of changed classes/methods (i.e., *C_Set*), and dependency relation graphs (i.e., *DRG*), we design BFS-based (i.e., Breadth-First Search) incremental and iterative search algorithm to search all affected classes/methods and identify affected test classes.

The pseudocode for searching affected classes/methods and identifying related test classes is shown in Algorithm 1.

As shown in Algorithm 1, for each changed class/method (located in step 1), firstly, we apply BFS to incrementally and iteratively search all affected classes/methods from *DRG* which can reach class/method C_i, and put them into the set of affected

Algorithm 1 Affected test classes/methods search algorithm.

Input: ~
C_Set, *DRG*
Output: ~
A set of affected test classes *T_Set*
Declare Variable:
A_Set is a set of affected classes/methods with dependency relations to changed class/method C_i;
1: **for** each class/method C_i in *C_Set* **do**
2: *A_Set*= searched affected classes/methods from *DRG* by *BFS*;
3: *A_Set* = *A_Set* + C_i;
4: **for** each class/method *a* in *A_Set* **do**
5: **if** *a* is a test class/method and the test class $\notin$ *T_Set* **then**
6: *T_Set* = *T_Set* + the class of *a*;
7: **end if**
8: **end for**
9: **end for**
10: output *T_Set*;

classes/methods *A_Set* (as shown in line 2). Note that *we also add the changed class/method C_i to A_Set, which is to ensure that the newly-added or changed tests can be selected* (as shown in line 3). Secondly, for each class/method in *A_Set*, we check whether it is a test class/method. Following existing work (Soetens et al., 2016; Blondeau et al., 2016), if a class contains at least one method (the method name starts with "test" or the method has the annotation "*@Test*", "*@BeforeClass*", "*@Before*"), we regard it as a test class; if the name of a method started with "test" or the method has the annotation "*@Test*", we regard it as a test method. If it is a test class/method and the test class is not in the set of affected tests *T_Set*, we add the test class to *T_Set* (as shown in lines 4–8). Finally, we obtain a set of affected test classes (i.e, *T_Set*).

2.4. Extending test classes

In this section, we add the recently failed test classes to redetect the deferred faults. These faults detected in the previous versions might have not been fixed and deferred to subsequent versions. Although these related tests do not have any dependencies with changed classes/methods, they can detect these deferred faults in the integrated version. This strategy has also been applied in existing studies (Yoo et al., 2011; Elbaum et al., 2014; Spieker et al., 2017). In detail, we look for the failed test classes in recent *n* versions (*n* is an input parameter), then check whether they run again. If not, we add them into *T_Set*. In this way, we obtain a final set of selected tests (*T_Set*).

3. FEST Approach

Based on the aforementioned *Sapient* framework, we develop a class-level test case selection approach FEST. It mainly extends the second step of *Sapient*, i.e., building precise dependency graph. Specifically, FEST captures the class-level dependencies of hidden references to facilitate the modeling of the full dependency relations at the class level.

Step 1: Locate changes. Based on *Sapient*, FEST additionally parses the *xml* file outputted by Doxygen# to obtain class-level structure information, including class names, the start line and the end line of each class, etc. With these information, FEST can map the changed lines to corresponding classes. In detail, for each changed line of changed file, FEST compares it with the start line and end line of each class in the changed file (i.e., a changed file may contain more than one class), if the changed line is between the start line and end line of a class, we regard the class as the corresponding changed class of the changed line.

Step 2: Build dependency graph. Based on the dependency modeling of *Sapient*, FEST additionally models the dependencies of hidden references to help build more precise dependency graph. In detail, to ensure the class-level full dependencies of code captured, FEST refers to the relations defined in *UML*. There are six relations among classes or objects defined in *UML*. Table 1 presents these relations and their representation in code (refer Orso's work (Orso et al., 2004)), as well as the corresponding relations in code. Inheritance includes *extends* and *implements* as shown in Table 1. Invocation includes class-level direct reference (short for *CDR*) and class-level hidden reference (short for *CHR*).

Both inheritance relations and *CDR* can be directly obtained by using the tags in *xml* files outputted by Doxygen# as introduced in Section 2.2. For *CHR*, we combine its representation in code and the code structure information of class files obtained in step 1 to infer it. 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, *CHR* can be built with the obtained information of reference and referenced classes.

Steps 3 and 4: Identify and extend related test classes. FEST adopts these two steps from the *Sapient* framework.

4. MEST Approach

Based on the aforementioned the *Sapient* framework, we develop a method-level test case selection approach MEST. It mainly extends the second step of *Sapient*, i.e., building precise dependency graph. Specifically, MEST combines two dynamic execution rules (i.e., dynamic invocation in reflection and dynamic binding in inheritance) with static dependencies to help build the precise method-level dependencies.

Note that, MEST considers both methods and fields when capturing the dependencies. This can ensure more precise changed information is recorded and more complete dependencies are traced. Existing work on method-level can only record the changes regarding to methods (Soetens et al., 2016; Parsai et al., 2014). In the rest of the paper, we refer to "methods" for both methods and fields for simplicity.

Step 1: Locate changes. Based on *Sapient*, MEST additionally parses the *xml* file outputted by Doxygen# to obtain method-level structure information, including method names, the start line and end line of each method, and annotation information of each method (e.g., *@Test, @BeforeClass*, *@Before*), etc. Then as FEST, for each changed line, MEST compares it with the start lines and end lines of methods, and obtains the detailed location of changes.

Step 2: Build dependency graph. Based on the dependency modeling of *Sapient*, MEST additionally utilizes dynamic execution rules to capture the dependencies at runtime so as to facilitate the modeling of precise dependencies. Besides, MEST also captures the dependencies of hidden references at method-level (different from the hidden references at class-level), which cannot be directly obtained by existing code dependency analysis tools. The following illustrates the details.

Fig. 2 presents an illustrative example to explain the method-level dependencies utilized in our study. The program classes and test classes are marked by pink and blue boxes respectively (e.g., Class *A+* is a program class, and Class *T2* is a test class), and meth-

Table 1
Mapping of class-level relations between relations in *UML* and in code.

Relationsin UML	Representationin code	Relationsin code
Generalization	A subclass *e*1 extends its superclass *e*2.	*Inheritance*
Realization	Class *e*1 implements an interface *e*2	*Inheritance*
Dependency	A method of Class *e*1 uses Class *e*2as an argument, e.g., the invocation by"*Class.forName('P.e2')*" in reflection	*Invocation(CHR)*
	A method of Class *e*1 uses Class *e*2in a cast operation.	*Invocation(CHR)*
	Class *e*1 instantiates Class *e*2 in *e*1'methods,then invokes its methods or variables.	*Invocation(CDR)*
Association	Class *e*1 uses Class *e*2as a member variable.	*Invocation(CHR)*
	A method in Class *e*1 invokes methodsor member variables of member Class *e*2.	*Invocation(CDR)*
	Class *e*1 defines a member variable, andinitiates it by invoking Class *e*2's methodsor variables, or using *e*2 in a cast operation.	*Invocation(CHR)*
Aggregation	Class *e*1 uses Class *e*2 asan argument to instance.	*Invocation(CHR)*
Composition	If Class *e*2 is a component of Class *e*1, it can only be instantiated in Class *e*1.	*Invocation(CDR)*

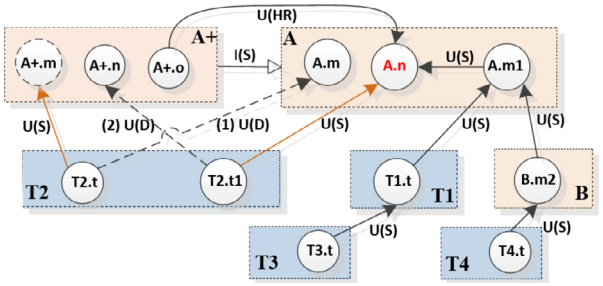

Fig. 2. An example of the method-level dependencies utilized in our study.

ods (e.g., method *A+.o*) are marked by circles within its class. The two main categories of inheritance and method-level invocation dependencies (Meyer, 1997) are marked by *I* and *U* respectively, e.g., Class *A+* is a subclass of Class *A* (marked by *I*), method *T1.t* invokes method *A.m1* (marked by *U*). The method-level hidden references and dependencies captured by dynamic execution rules are marked by *HR* and *D* respectively. In addition, the inheritance and method-level direct reference dependencies which exist hidden reference relations are marked by *S*.

Capturing method-level hidden reference dependencies: The method-level hidden reference dependencies usually occur between constructor functions, or between superclasses and subclasses. We carefully consider each situation and resolve the dependencies.

For the method-level hidden reference dependencies between *constructor functions*, we infer them by using the inheritance and direct reference dependencies obtained by Doxygen#. Based on prior's work (Meyer, 1997), for each inheritance or direct reference dependency, there is a method-level hidden reference between its constructor functions. Following this rule, when Class *A* is the superclass of Class *A+*, i.e., *inheritance dependency* (as shown in Fig. 2), it indicates that the constructor of Class *A* is implicitly invoked by the constructor of Class *A+*. We will build a dependency between constructor functions of Class *A* and Class *A+*. For this case in *direct reference dependency*, when method *A+.m* is invoked by method *T2.t* (as shown in Fig. 2), it indicates that the constructor of Class *A+* is implicitly invoked by the constructor of Class *T2*. We will build a dependency between constructor functions of Class *A+* and Class *T2*.

For the method-level hidden reference dependency that *a method of subclass invokes a method of superclass, and the subclass has another method with the same name as the involved method*, we capture it using keyword "*super*" (Booch, 1991; Meyer, 1997) and the code structure information of class files obtained in step 1. Take Fig. 2 as an example, method *A+.n* overrides method *A.n* of its superclass, then method *A+.o* invokes method *A.n* by "*super*" (e.g., "*super.n*" for *Java*, "*super::n*" for *C++*) (Booch, 1991). To obtain this hidden dependency, for each *xml* files, we firstly check whether there is such hidden reference using keyword search (e.g., "super.", "super(", "super::") in the *xml* file, and record the position (i.e., lines of code) of occurrence, and obtain the name of referenced method (i.e., *n*) and other information of method *n* (e.g., parameters and return type). Secondly, we look for the superclasses (e.g., Class *A*) of method *n*, and search the referenced method (i.e., *A.n*) from these superclasses based on the method name (i.e., *n*), parameters and return type. Meanwhile, we map the position to corresponding method based on the code structure information, and label it as the reference method (i.e., *A+.o*). Finally, the method-level hidden reference can be built with the obtained referenced and reference methods.

Capturing method-level dependencies by dynamic execution rules: As we know, some of the dependency relations are determined dynamically at the runtime of a program. Existing work (Gligoric et al., 2015; Vasic et al., 2017; Celik et al., 2017; Zhang, 2018) captured these dependency relations by dynamically executing tests, which is effort-consuming and difficult to manipulate. Nevertheless, these dependencies can be precisely inferred based on specific static rules. Two dynamic execution rules have been built to capture method-level invocation dependencies more precisely.

Rule 1: **Dynamic invocation in reflection**. A method is dynamically invoked through a string constant or string variable (e.g., "*Method m = getMethod ('m2', int.class, String.class);*"). In order to capture this dependency, for each *xml* file, we firstly check whether there is such hidden reference using keyword search (e.g., "*getMethod*", "*getDeclaredMethod*") in the *xml* file. We then record the position (i.e., lines of code) of occurrence, and obtain the information of referenced method (e.g., *m2* as method name, *"int.class, String.class"* as the list of parameter types, the class name of *m2*). When *m2* is a string constant or a local variable, we can directly obtain it via parsing the code within current method; if it is a string variable being a parameter of current method, we look for the method that invokes current method to obtain the value of this string variable. Then, based on the code structure information of the class file obtained in step 1, we map this position of occurrence to corresponding reference method and its information. Finally, this dynamic dependency can be built with the obtained referenced and reference methods.

Rule 2: **Dynamic binding in inheritance**. This rule can be separated into two sub-scenarios, and we illustrate them below. The first scenario is that the subclass does not override the method of superclass. As shown in Fig. 2, method *T2.t* invokes method *A+.m* (marked by dotted line), however, method *m* is not directly in Class *A+* but in its superclass *A*. In this case, the dependency captured by static analysis is the invocation between method *T2.t* and method *A+.m* (marked by solid orange line), which is imprecise. Our MEST adds the invocation dependency that method *T2.t* invokes method *A.m* (marked by dotted black line), and removes the invocation between method *T2.t* and method *A+.m* to avoid selecting unnecessary tests.

The second scenario is that the subclass overrides the non-static method of superclass. As shown in Fig. 2, the object of superclass *A* is instantiated by subclass *A+* (e.g., *"A instance_a = new A+();"*), method *T2.t1* invokes non-static method *A.n*; and method *A.n* is overridden by method *A+.n* with the two methods having the same method name, parameters and return type. In this case, the program will dynamically execute method *A+.n* rather than method *A.n*. Similar to the first scenario, the dependency captured by static analysis is the invocation between *T2.t1* and method *A.n* (marked by solid orange line) which is imprecise. Our MEST adds the invocation dependency that method *T2.t1* invokes method *A+.n* (marked by dotted black line) and removes the invocation between method *T2.t1* and method *A.n*.

The process of capturing the dependencies with the dynamic Rule 2 is shown in Algorithm 2. In detail, for each invocation dependency in static dependencies *Es_Set*, we first check that the referenced method *A+.m* (take Fig. 2 as an example) is defined in its class *A+*, if not, we iteratively search the superclasses of class *A* by using BFS until the definition of method *m* is searched, then update the referenced method of invocation dependency based on the first scenario of Rule 2 (as shown in lines 2–10). Besides, for these invocation dependencies in terms of the second scenario of Rule 2, if method *A.n* in Fig. 2 is a non-static referenced method of the dependency, we search its overriding method (i.e., *A+.n*) from its subclasses based on its method name, parameters and return type. If it exists and the object of superclass *A* is instantiated by subclass *A+*, we update the referenced method by *A+.n* (as shown in lines 11–14).

Algorithm 2 The dependency analysis algorithm based on Rule 2.

Input: ~
A set of static dependencies *Es_Set*
Output: ~
A set of updated dependencies *Es_Set* based on Rule 2
Declare Variable:
Sc_Set is a set of superclasses of method $A+.m$;
$A+.n$ is a method, which overrides method *A.n* of its superclass *A*;

```
1:  for each invocation dependency i in Es_Set do
2:    if referenced method A+.m of i not in Class A+ then
3:      Sc_Set = searched the superclasses of Class A+ from Es_Set
        by BFS;
4:      for each class Sc in Sc_Set do
5:        if method m in class Sc then
6:          update A+.m of i by method Sc.m;
7:          break;
8:        end if
9:      end for
10:   end if
11:   A+.n = searched the overriding method of non-static referenced method A.n in its subclasses from Es_Set;
12:   if A+.n is not null and the object of A is instantiated by A+ then
13:     update A.n of i by A+.n;
14:   end if
15: end for
16: output updated Es_Set;
```

Steps 3 and 4: Identify and extend related test classes. MEST adopts these two steps from the *Sapient* framework.

Take Fig. 2 as an example, if method *A.n* is a changed method, its affected test set *A_Set* is {*A.n, A+.o, A.m1, T1.t, B.m2, T3.t, T4.t*}. Then we check whether each test method in *A_Set* is a test class, and obtain a set of affected test classes (e.g., *T_Set* of *A.n* is {*T1, T3, T4*}).

5. Experiment design and evaluation metrics

In this section, we introduce the design of experiment and the selected baseline approaches.

5.1. Subject projects and data preparation

According to the activeness and CI testing history availability, we chose 12 projects from Eclipse community and 6 projects from Apache community to evaluate our approaches presented in Section 2. We collected the CI testing history from November 2017 to January 2018 for the chosen projects, which contains test classes, test methods, durations, results, status, etc. Based on these information, we can obtain the failed test information for each CI version. Then we collected commits from the *Git* repository by executing "*git log*", and checked out source code by executing "*git checkout*". Next, we built the relationship of the three datasets by the commit *id* of CI versions.

We chose the versions for our experiment by following the criteria: (1) the number of changed classes was greater than 0; (2) the number of the executed CI tests was greater than 0. Finally, we obtained 261 versions from the 18 projects. Table 2 shows basic statistics for the chosen projects. For one project, *kLoc* is the number of thousands of lines of source code; *Size* refers to the size of projects based on lines of code determined by the criteria in prior work (Engström et al., 2010); *L* means the large size; *M* means the medium size; *Revs* is the number of versions; *Language* refers to the programming languages used in the project, and *T[m]* is duration in minutes to run actual CI tests. The last row shows the average across all projects.

Regarding "extending test classes" in step 4 of FEST and MEST, we conducted the experiments with the parameter n from 1 to 30. The results indicate that 10 is the sweet point to get the better and more stable performance. Hence, we adopt 10 as the number of versions to backward inspection in the following experiments. Hence we apply the recent 10 versions in the following experiments. 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 server version 1.8.0_73.

5.2. Baseline approaches

To further evaluate the performance of our approaches, we compare them with the following two baselines.

Actual CI testing. This is the real-world practice of CI testing recorded in the project repositories. In detail, from the collected CI testing history, we obtain the test classes executed in each CI testing, and treat them as the selected tests. Note that, not all stored test classes are executed, and developers usually employ experience-based or heuristic-based guidelines for the test selection.

ClassSRTS. This is the state-of-the-art class-level static selection technique (Legunsen et al., 2016; 2017). It uses the class firewall (Kung et al., 1995) technique to find class-level dependencies by reasoning inheritance and reference relationships based on a class dependency graph. ClassSRTS selects a test subset that transitively depends on any changed classes in the dependency graph. Besides, ClassSRTS outperformed the state-of-the-art method-level approach (Legunsen et al., 2016). Hence for comparison with our method-level test case selection approach MEST, we also utilize ClassSRTS as baseline. We did not select any dynamic approach as baseline because in the most recent work (Legunsen et al., 2016), ClassSRTS is compared to the state-of-the-art dynamic approach (Gligoric et al., 2015), and results show that ClassSRTS is comparable to the dynamic approach. The reason we did not compare with dynamic selection is because it needs to run tests to obtain

Table 2
Projects used in the study.

ID	Project Name	Language	kLoc	Size	Revs	T[m]
Ep1	eclipse.jdt.core	Java	1478	L	6	110
Ep2	eclipse.jdt.debug	Java	227	L	5	23
Ep3	eclipse.jdt.ui	Java	697	L	9	63
Ep4	eclipse.pde.ui	Java	378	L	4	37
Ep5	eclipse.platform.debug	Java	142	L	21	15
Ep6	eclipse.platform.resources	Java	99	M	8	40
Ep7	eclipse.platform.runtime	Java	58	M	10	15
Ep9	eclipse.platform.team	Java	295	L	9	26
Ep10	eclipse.platform.text	Java	144	L	31	12
Ep11	eclipse.platform.ua	Java	171	L	8	12
Ep12	eclipse.platform.ui	Java	639	L	72	48
Ep13	rt.equinox.framework	Java	132	L	3	21
Ap3	Apache OOZIE	Java	198	L	5	93
Ap5	Apache ratis	Java	32	M	4	15
Ap6	Apache Streams	Java	61	M	3	40
Ap8	Beam	Java,Python	336	L	18	109
Ap10	Hadoop	Java	2697	L	7	75
Ap11	Hadoop HBASE	Java,Python, C+,PhP	773	L	38	150
Avg			475		15	50

Table 3
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 selection approaches (e.g., *St@F* for FEST, *St@M* for MEST, *St@C* for ClassSRTS).
Cft	A set of faults detected by actual CI testing.
Tft	A set of total faults detected by selection approaches (e.g., *Tft@F* for FEST, *Tft@M* for MEST, *Tft@C* for ClassSRTS).

the dependency information, which limits its application in our experiment.

5.3. Evaluation metrics

Following existing work (Marijan et al., 2013; Soetens et al., 2016), when a test class is observed failed, we treat it as a fault; and only when at least one test case in the test class fails, we treat the test class as failed. 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 (Beszédes et al., 2012). It is because it should be detected in the previous version, but it was omitted by actual CI testing.

We evaluated our approaches from four dimensions: reduced test size, fault detection efficiency, test cost and cost effectiveness. **To facilitate understanding, we present basic metrics used in our paper in** Table 3.

5.3.1. Reduced test size (rts)

Reducing test size is an effective way to speed up the feedback of CI testing. In this dimension, we use reduced test size to evaluate our proposed approaches. *Rts* means the percentage of the number of reduced tests by the measured approach *A* (e.g., FEST) compared with the number of tests of approach *B* (e.g., actual CI testing). Following the existing work (Soetens et al., 2016; Gligoric et al., 2015; Parsai et al., 2014; Shi et al., 2017; Marijan et al., 2017), it is defined as Eq. 1.

$$Rts = \frac{|St_B| - |St_A|}{|St_B|} \times 100\% \quad (1)$$

Here, St_B is the test set of approach *B*, e.g., it will be *Ct* when comparing with actual CI testing. St_A is the test set of the measured approach *A*, e.g., it can be *St@F* (when measuring *Rts* of FEST). The positive *Rts* means that the measured approach can reduce test size compared with approach *B*, the higher the better. In contrast, the negative *Rts* indicates that the measured approach selects more tests than approach *B*.

5.3.2. Fault detection efficiency

The set of reduced tests should meet the fault detection efficiency. In this dimension, we evaluate fault detection efficiency from fault coverage and new faults.

(a) Fault coverage (Fcovg) means the percentage of the faults found by the measured approach *A* (e.g., FEST) compared with the faults found by approach *B* (e.g., actual CI testing). Following existing work (Soetens et al., 2016; 2013; Parsai et al., 2014; Mondal et al., 2015), it is defined as Eq. 2.

$$Fcovg = \frac{|Tft_A \cap Tft_B|}{|Tft_B|} \times 100\% \quad (2)$$

In it, $Tft_A \cap Tft_B$ denotes the intersection set of two fault sets detected by the measured approach *A* and approach *B* respectively. For example, when *Fcovg* is used to evaluate the fault coverage of FEST compared with actual CI testing, Tft_A and Tft_B will be replaced by *Tft@F* and *Cft* (for actual CI testing) respectively. When $|Tft_B|$ is 0, we treat *Fcovg* as 100%.

(b) New faults (NewFt) means the number of new faults detected by the measured approach *A* (e.g., FEST) compared with approach *B* (e.g., actual CI testing). It is used to measure the fault detection enhancement of approach *A*.

5.3.3. Test cost

In this dimension, following previous work (Gligoric et al., 2015; Legunsen et al., 2016; 2017), we evaluate test cost by end-to-end time that includes the time to select tests (i.e., analyze changed

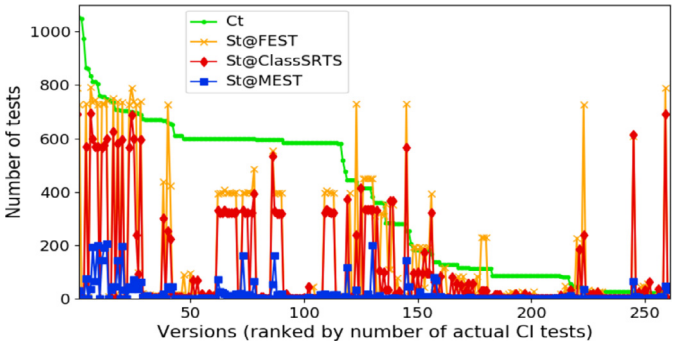

Fig. 3. Number of selected tests in FEST, MEST, ClassSRTS or actual CI testing.

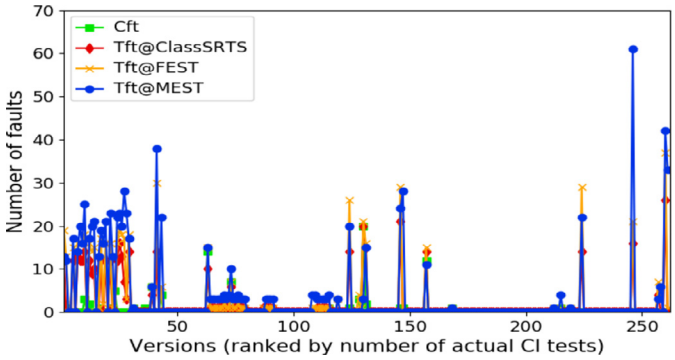

Fig. 4. Number of faults detected by FEST, MEST, ClassSRTS or actual CI testing.

classes/methods, capture dependencies and search tests) and time to run the selected tests. For actual CI testing, the end-to-end time refers to the duration of CI testing.

5.3.4. Cost effectiveness

In this dimension, we evaluate the cost effectiveness from test scale benefits.

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 (Mirarab et al., 2012; EngstrÖm et al., 2011; Herzig et al., 2015). 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 (Soetens et al., 2016; Mondal et al., 2015). 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 Eq. 3. The higher value of *Testscal* the better.

$$Testscal = \frac{|Tft|}{|St|} \times \frac{|Tft|}{|Cft \cup Tft@X \cup Tft@C|} \tag{3}$$

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

6. Experiment results and analysis

In this section, we first present the effectiveness of FEST and MEST compared with two baselines respectively, then compare MEST with FEST, and finally analyze why MEST performs well.

6.1. Effectiveness of FEST and MEST

We present the effectiveness of FEST and MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness compared with actual CI testing and ClassSRTS respectively. Fig. 3 and Fig. 4 present an overview of the number of selected tests and detected faults by FEST, MEST, ClassSRTS or actual CI testing. We mixed the versions in different projects together, sorted them by number of actual CI tests and then assigned the ID sequentially.

From Fig. 3, we can easily see that in most cases, MEST obviously selects less tests than actual CI testing, ClassSRTS and FEST. In detail, compared with actual CI testing, it can reduce test size in 99% (258/261) versions; especially, in 44% (116/261) versions, it can reduce more than 500 tests, and in 69% (181/261) versions it can reduce more than 100 tests. More than that, MEST detects more faults than actual CI testing (in 67 versions) and ClassSRTS (in 68 versions) as shown in Fig. 4.

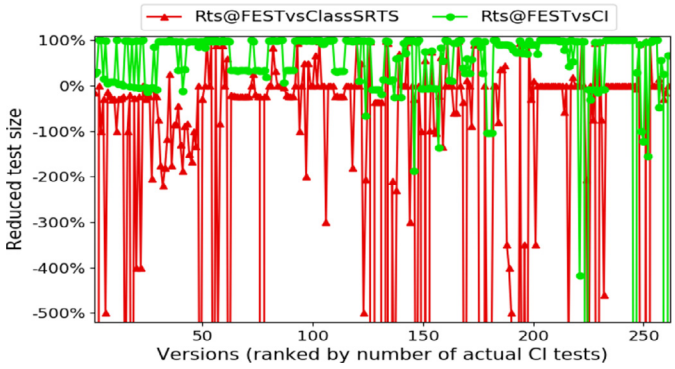

Fig. 5. Reduced test size of FEST compared with actual CI testing or ClassSRTS.

Meanwhile, FEST can detect more faults than actual CI testing (in 66 versions) and ClassSRTS (in 48 versions) as shown in Fig. 4. Besides, FEST can reduce the test size in 84% versions; on average, it reduces the test size by 60% in all versions, as shown in Fig. 3.

The following subsections present more details of FEST and MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness compared with actual CI testing and ClassSRTS respectively.

6.1.1. Effectiveness of FEST

We present effectiveness of FEST from reduced test size, fault detection efficiency, test cost and cost effectiveness in the following subsections.

Reduced test size: Fig. 5 shows the reduced test size (*Rts*) of FEST compared with actual CI testing or ClassSRTS. Compared with actual CI testing, FEST can reduce test size by an average of 60%. In detail, FEST shows positive *Rts* in 84% (219/261) versions; in other 16% (42/261) versions, FEST shows negative *Rts*, but it can find much more faults in 20 out of those 42 versions, in the resting 22 versions, they detect the same number of faults (as shown in Fig. 4).

Compared with ClassSRTS, FEST shows positive *Rts* in 18% (46/261) versions, selects equal number of tests as ClassSRTS in 50% (131/262) versions. In other 32% (84/261) versions, FEST shows negative *Rts*, but it can detect more faults in 10 out of 84 versions, and equal number of faults in the resting 74 versions. It indicates that ClassSRTS omits some necessary tests which results in a high risk of omitting faults, while FEST resolves full dependency relations at the class level to select tests, which can mitigate the risk of omitting necessary tests although it selects more tests in some versions. On average, FEST can reduce test size by -46% compared with ClassSRTS.

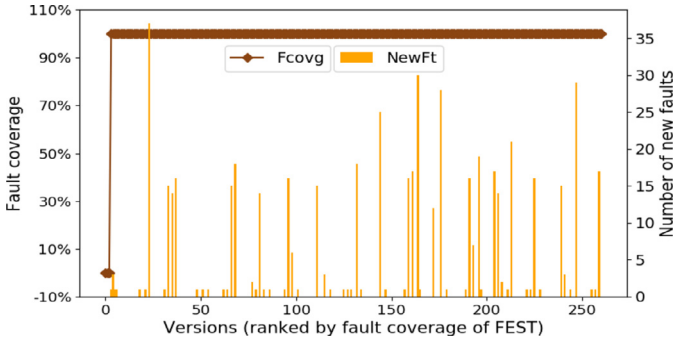

Fig. 6. FEST's fault coverage and the number of new faults compared with actual CI testing.

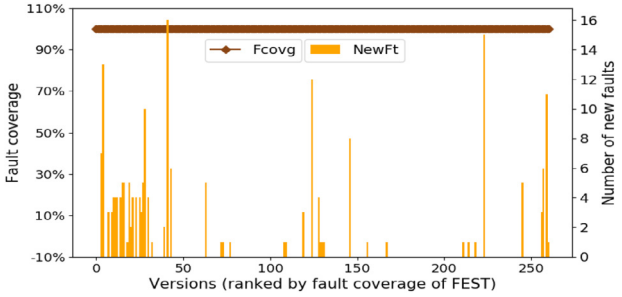

Fig. 7. FEST's fault coverage and the number of new faults compared with ClassSRTS.

For the versions whose *Rts* is less than -500% compared with actual CI testing (3 versions) and ClassSRTS (32 versions), we simply set them as -500% in Fig. 5 to improve the readability.

Summary 1.1.1: On average, FEST can reduce test size by 60% and -46% compared with actual CI testing and ClassSRTS respectively.

Fault detection efficiency: We evaluate fault detection efficiency of FEST compared with actual CI testing and ClassSRTS respectively.

(a) Comparison with actual CI testing: Fig. 6 shows the fault coverage and the number of new faults of FEST compared with actual CI testing. We can see that FEST can cover all faults detected by actual CI testing in 99% (258/261) versions, and find new faults in 25% (66/261) versions. However, in the other 3 (1%) 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 it is in fact not due to the drawback of FEST.

(b) Comparison with ClassSRTS: Fig. 7 shows the fault coverage and the number of new faults of FEST compared with ClassSRTS. We can see that FEST can cover all faults detected by ClassSRTS in all versions, and can find new faults in 18% (48/261) versions. Especially, in 6 versions, FEST can find more than 10 new faults.

Summary 1.1.2: FEST can cover all faults detected by actual CI testing (in 99% versions) and ClassSRTS (in 100% versions), and find new faults in 25% and 18% versions respectively.

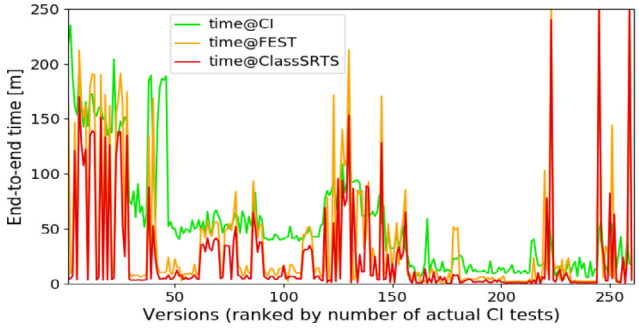

Fig. 8. End-to-end time of actual CI testing, ClassSRTS or FEST.

Test cost: We first compared the time for selecting tests. Regarding FEST, the average time of locating changed methods and capturing the dependencies of source code is 5.9 minutes, the average time of running search algorithm is 2.48 minutes. Hence, the time of test case selection for FEST is about 8.3 minutes while the time of test case selection for ClassSRTS is 3.6 minutes on average. The reason is that FEST needs more time to capture the full dependencies at the class level.

We then compared the end-to-end time of actual CI testing, FEST and ClassSRTS in Fig. 8. We can see that in 77% (202/261) versions, the end-to-end time of FEST is much lower than actual CI testing, especially in 154 out of 202 versions, FEST can reduce more than 50% end-to-end time than actual CI testing. Only in other 23% (59/261) versions, FEST shows higher end-to-end time, but it can find new faults in 35 of these versions. Specifically, in 21 out of the 35 versions, FEST can detect more than 10 new faults than actual CI testing. This implies that actual CI testing ran insufficient tests and omitted some necessary tests, which leads to omit some faults.

Compared with ClassSRTS, we can see that FEST shows less end-to-end time in 10% (27/261) versions. In other 90% (234/261) versions, FEST shows higher end-to-end time than ClassSRTS, but in 47 out of these versions, it can detect more faults. It implies that ClassSRTS might omit necessary tests which leads to a high risk of omitting faults, while our FEST captures full dependencies at the class level to select all affected tests, which can mitigate the risk of omitting necessary tests.

Note that in 2 or 3 versions, the end-to-end time of ClassSRTS and FEST are all greater than 250, we simply set them as 250 in Fig. 8 to improve the readability. On average, the end-to-end times of our FEST and ClassSRTS are about 37 and 25 minutes respectively. In our study, 78% (14/18) experimental projects belong to large-sized projects and the average duration of actual CI testing is 50 minutes, as shown in Table 2. Hence, the end-to-end time of our FEST is 74% (37/50) and 148% (37/25) of actual CI testing and ClassSRTS respectively.

Summary 1.1.3: On average, the end-to-end time of FEST is 74% of actual CI testing; compared with ClassSRTS, the end-to-end time of FEST is 48% higher, but FEST can detect more faults.

Cost effectiveness: Fig. 9 shows the test scale benefits (*Testscal*) of actual CI testing, ClassSRTS or FEST. We can see that FEST has better or equal *Testscal* in 98% and 99% versions than actual CI testing and ClassSRTS respectively. For other versions where FEST shows lower *Testscal* than actual CI testing or ClassSRTS, FEST can detect more faults. It indicates that FEST can mitigate the risk of omitting necessary tests, although it selects more tests in these versions.

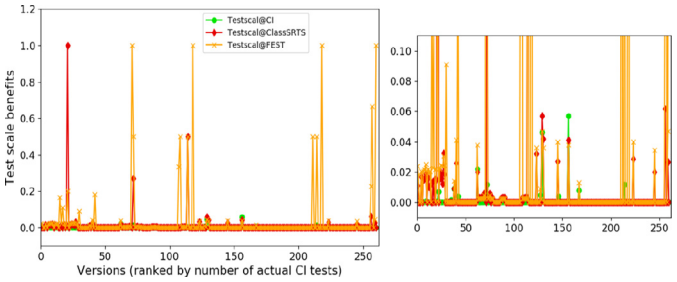

Fig. 9. Test scale benefits of actual CI testing, ClassSRTS or FEST (the enlarged figure on the right shows the details).

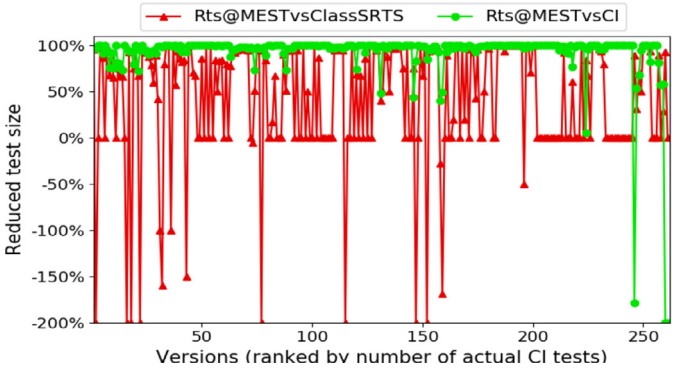

Fig. 10. Reduced test size of MEST compared with actual CI testing or ClassSRTS.

Summary 1.1.4: FEST shows better or equal *Testscal* than actual CI testing (in 98% versions) and ClassSRTS (in 99% versions).

6.1.2. Effectiveness of MEST

In this section, we present the effectiveness of MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness compared with actual CI testing and ClassSRTS respectively.

Reduced test size: Fig. 10 shows the reduced test size (*Rts*) of MEST compared with actual CI testing or ClassSRTS. MEST can reduce test size by an average of 92% compared with actual CI testing. Specifically, in 99% versions, MEST shows positive *Rts*; only in three versions (v246, v260, v261), MEST shows negative *Rts*, but it can find more faults as shown in Fig. 4.

Compared with ClassSRTS, MEST can reduce test size by an average of 43%. In detail, in 60% (157/261) versions MEST shows positive *Rts*. Specially, *Rts* of MEST is greater than 50% in 143 of 157 versions. In 34% (89/261) verions, MEST and ClassSRTS select equal number of tests, but MEST can detect more faults in 10 of 89 versions. In other 6% (15/261) versions, MEST shows negative *Rts*, but it can find more or equal number of faults in these versions.

Summary 1.2.1: On average, MEST can reduce test size by 92% and 43% compared with actual CI testing and ClassSRTS respectively.

Fault detection efficiency: We evaluate fault detection efficiency of MEST compared with actual CI testing and ClassSRTS respectively.

(a) Comparison with actual CI testing: Fig. 11 shows the fault coverage and the number of new faults of MEST compared with actual CI testing. We can see that MEST can cover all faults detected by actual CI testing in 97% (254/261) versions, and find new faults in 26% (67/261) versions. For the other 7 (3%) versions, we further examine the details of them (i.e., v1, v2, v3, v4, v5, v6, v7 as shown in Fig. 11), and summarize three reasons:

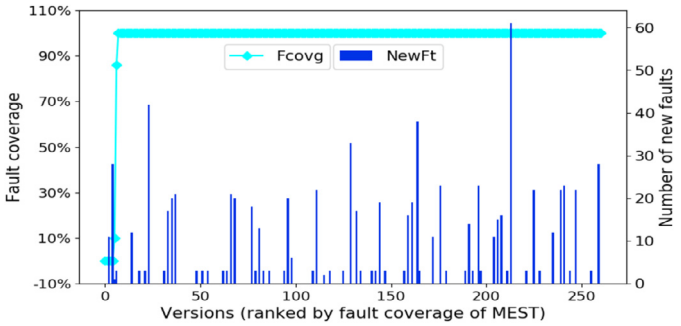

Fig. 11. MEST's fault coverage and the number of new faults compared with actual CI testing.

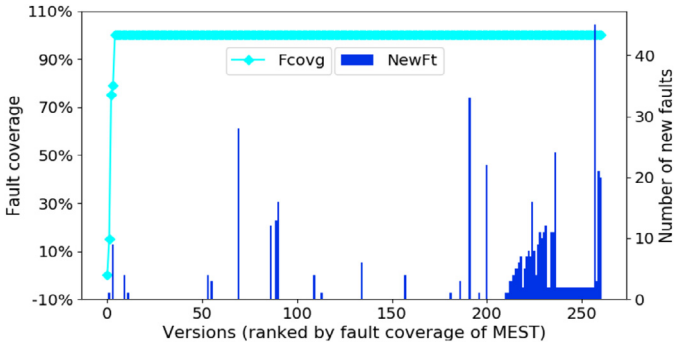

Fig. 12. MEST's fault coverage and the number of new faults compared with ClassSRTS.

- *Deferred faults*: in v1, v2, v4, we carefully analyzed the code and found that the faults which are not covered by MEST did not have any dependencies with the changed and affected code. We also looked for the test history of the faults, and did not find any related records in the recent 10 versions. This implies that these faults were committed in earlier versions, and should be detected earlier, but first ran in current integrated version. This indicates that this is not due to the power of MEST.
- *Unresolved dependency*: in v3 and v7, a particular kind of invocation dependency (i.e., a method invokes another method of current class in "*if*" judgment condition) could not be resolved by Doxygen#.
- *Third party libraries*: in v5 and v6, the faults which are not covered by MEST were caused by the third party libraries. In detail, the application uses external libraries or frameworks for which the source code is not available. In this case, the static analysis of the code can not trace dependencies through the third-party code execution. Existing selection approaches based on method-level static dependencies of source code also can not resolve the dependency (Blondeau et al., 2016; Soetens et al., 2016).

(b) Comparison with ClassSRTS: Fig. 12 shows the fault coverage and the number of new faults of MEST compared with ClassSRTS. We can see that MEST can cover all faults detected by ClassSRTS in 98% (257/261) versions, and find new faults in 27% (70/261) versions. Specifically, MEST can find more than 10 new faults in 17 versions. In other 4 versions (v1, v2, v3 and v4), it is due to the third-party libraries (discussed in the previous paragraph). The reason why ClassSRTS can select these tests is

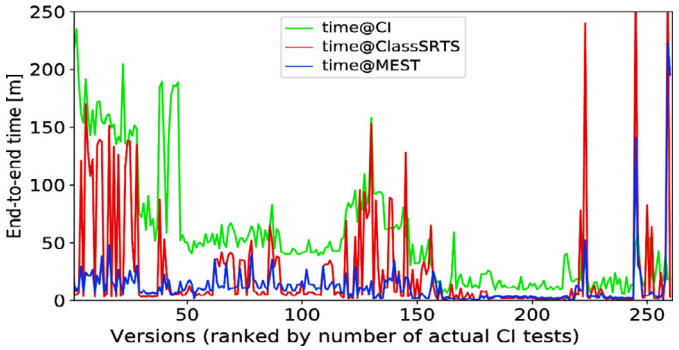

Fig. 13. End-to-end time of actual CI testing, ClassSRTS or MEST.

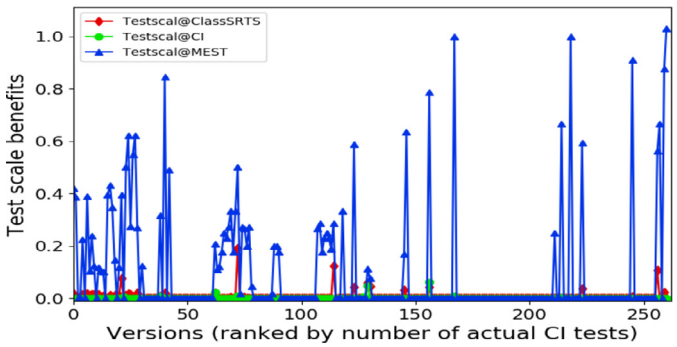

Fig. 14. Test scale benefits of actual CI testing, ClassSRTS or MEST.

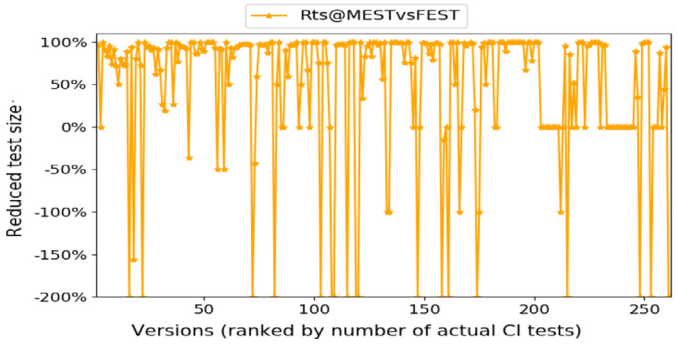

Fig. 15. Reduced test size of MEST compared with FEST.

that it selects tests based on coarse dependencies (i.e., class-level) which is not affected by the invocation of the third party libraries (Legunsen et al., 2016).

> **Summary 1.2.2**: MEST can cover all faults detected by actual CI testing (in 97% versions) and ClassSRTS (in 98% versions), and find new faults in 26% and 27% versions respectively.

Test cost: We first compare the time for selecting tests. For MEST, an average of 6.96 minutes are needed for locating changed methods and capturing the dependencies of source code, while an average of 2.11 minutes are needed for running search algorithm. Hence the end-to-end time for test case selection of MEST is about 9 minutes, while for ClassSRTS this time is 3.6 minutes. This is because MEST needs more time to capture the precise finer-grained dependencies. Despite of that, MEST selects quite a smaller number of tests which leads to the less end-to-end time, as shown below.

Fig. 13 shows the end-to-end times of actual CI testing, MEST or ClassSRTS. We can see that in 99% (258/261) versions, the end-to-end time of MEST is much less than actual CI testing. In other 3 versions, MEST can find much more faults by running more test classes with more time. It also implies that actual CI testing ran insufficient tests, which leads to omit some faults.

Compared with ClassSRTS, we can see that MEST shows less end-to-end time in most versions. The end-to-end times of our approach MEST and ClassSRTS are 12 and 25 minutes on average respectively. In our study, 78% (14/18) experimental projects belong to large-sized projects and the average duration of actual CI testing is 50 minutes, as shown in Table 2. Hence, the end-to-end time of our MEST is 24% (12/50) and 48% (12/25) of actual CI testing and ClassSRTS respectively.

> **Summary 1.2.3**: On average, the end-to-end times of MEST are 24% and 48% of actual CI testing and ClassSRTS respectively.

Cost effectiveness: Fig. 14 shows the test scale benefits (*Testscal*) of actual CI testing, ClassSRTS or MEST. We can see that MEST has better or equal *Testscal* in 98% (256/261) and 100% versions than actual CI testing and ClassSRTS respectively. Only in four versions, MEST shows lower *Testcal* than actual CI testing. The detailed reasons are due to deferred faults, unresolved dependency or third party libraries, which were discussed in "*Comparison with actual CI testing*" of Section 6.1.2.

> **Summary 1.2.4**: MEST shows better or equal *Testscal* than actual CI testing (in 98% versions) and ClassSRTS (in 100% versions).

6.2. The comparison of MEST and FEST

In the previous section, we present the effectiveness of FEST and MEST compared with two baselines; here we further compare MEST with FEST from reduced test size, fault detect efficiency, test cost and cost effectiveness, to demonstrate the advantages of method-level test selection.

Reduced test size: Fig. 15 shows the reduced test size (*Rts*) of MEST compared with FEST. On average, MEST can reduce test size by 48% compared with FEST. In detail, we can see that MEST shows positive *Rts* in 72% (189/261) versions; in 17% (43/261) versions, MEST and FEST select equal number of tests (i.e., $Rts@MESTvsFEST = 0$). In other 11% (29/261) versions, MEST shows negative *Rts*. However, in 12 of 29 versions, MEST can detect more faults than FEST; in the resting 17 versions, they detect equal number of faults.

Note that for the versions whose *Rts* is less than -200% (18 versions), we simply set them as -200% in Fig. 15 to improve readability.

> **Summary 2.1**: On average, MEST can reduce test size by 48% compared with FEST.

Fault detection efficiency: Fig. 16 shows the fault coverage and the number of new faults of MEST compared with FEST. We can see that MEST can cover all faults detected by FEST in 95% (247/261) versions, and find new faults in 20% (52/261) versions. In

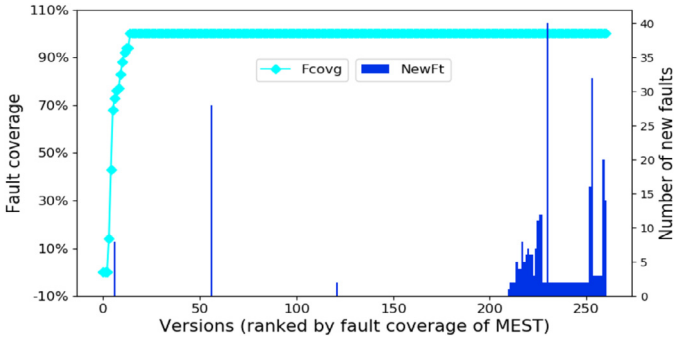

Fig. 16. MEST's fault coverage and the number of new faults compared with FEST.

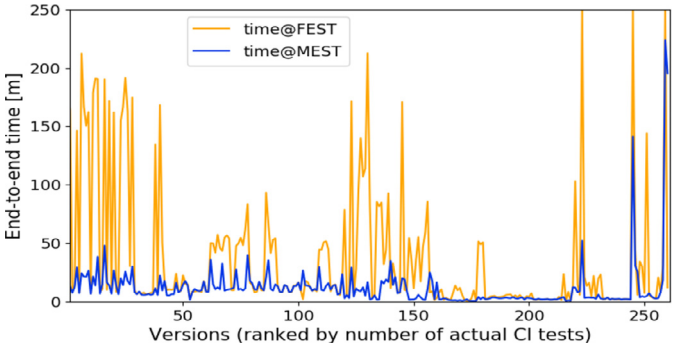

Fig. 17. End-to-end time of FEST or MEST.

other 5% (14/261) versions, MEST does not cover all faults detected by FEST. We further analyzed these versions where MEST seems not cover all faults detected by FEST, and found that the reasons are due to deferred faults, unresolved dependency and third party libraries, which were discussed in "*Comparison with actual CI testing*" of Section 6.1.2.

Summary 2.2: MEST can cover all faults detected by FEST in 95% versions, and can find new faults in 20% versions.

Test cost: We first compare the time for selecting tests. Regarding FEST, the time of test case selection is about 8.3 minutes while the time of test case selection for MEST is 9 minutes on average. The reason is that MEST needs more time to capture the more precise finer-grained dependencies including static dependencies and the dependencies occurring at runtime. Although MEST shows more time for test case selection than FEST, it selects a quite smaller number of tests which leads to less end-to-end time (see the next paragraph).

We then compare the end-to-end time of FEST and MEST in Fig. 17. We can see that MEST shows less or equal end-to-end time in 68% (177/261) and 3% (8/261) versions. In other 29% (76/261) versions, MEST shows higher end-to-end time than FEST, but it can find more new faults in 12 of 76 versions. It implies that in most cases, MEST shows less overhead, yet higher fault detection efficiency.

On average, the end-to-end times of our FEST and MEST are about 37 and 12 minutes respectively. Hence, the end-to-end time of our MEST is 32% (12/37) of FEST. Note that the 3 versions where the end-to-end time of FEST is greater than 250, we simply set them as 250 in Fig. 17 to improve the readability.

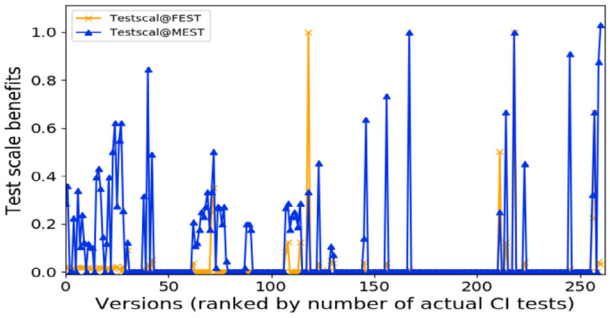

Fig. 18. Test scale benefits of FEST or MEST.

Summary 2.3: On average, the end-to-end time of MEST is 32% of FEST.

Cost effectiveness: Fig. 18 shows the test scale benefits (*Testscal*) of FEST or MEST. We can see that MEST has better or equal *Testscal* than FEST in 26% (68/261) and 72% (188/261) versions respectively. In other 5 (2%) versions, MEST shows lower *Testcal* than FEST, the reasons are due to deferred faults, unresolved dependency or third party libraries, as discussed in "*Comparison with actual CI testing*" of Section 6.1.2.

Summary 2.4: MEST shows better or equal *Testscal* than FEST in 98% versions.

To summarize, the above results of comparing FEST and MEST show that MEST outperforms FEST in reduced test size, fault detection efficiency, test cost and cost effectiveness in most cases, indicating method-level test case selection can be more effective than class-level selection.

6.3. Why MEST performs well?

As aforementioned, MEST shows better performance than baselines and FEST. We think it is due to the advantage of dynamic execution rules which have been adopted in MEST. The dynamic rules only can be addressed at the method level when the finer-grained relationship were analyzed. In this section, we ran 3 experiments (i.e., MEST.s, MEST.s-r1 and MEST.s-r2) to explore the contribution of dynamic execution rules.

- MEST.s: selecting tests based on the dependencies only with static analysis;
- MEST.s-r1: selecting tests based on the dependencies with static analysis and dynamic execution Rule 1;
- MEST.s-r2: selecting tests based on the dependencies with static analysis and dynamic execution Rule 2.

In order to compare with MEST, the three variants of MEST all extend the selected test classes based on recently failed test classes.

6.3.1. Selected test size

Fig. 19 presents an overview of the number of selected tests by MEST, MEST.s-r2, MEST.s-r1 or MEST.s. We sorted all versions by number of actual CI tests and then assigned the ID sequentially.

From Fig. 19, we can easily see that in all versions, the test set selected by MEST is the largest, the test set selected by MEST.s is

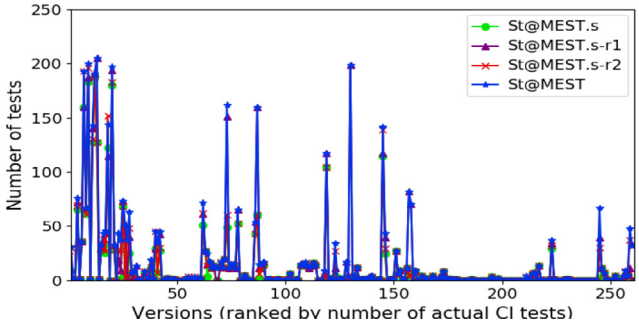

Fig. 19. Number of tests selected by MEST.s, MEST.s-r1, MEST.s-r2 or MEST.

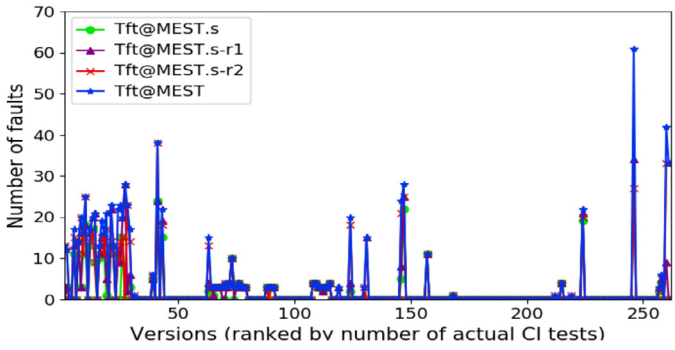

Fig. 20. Number of faults detected by MEST.s, MEST.s-r1, MEST.s-r2 or MEST.

the smallest, while the test sets selected by MEST.s-r1 and MEST.s-r2 are in medium size. This is because MEST considers the maximum number of dependencies, while MEST.s considers the minimum number of dependencies (i.e., only static dependencies). We further compared the test sets selected by the four approaches, and results show that MEST can cover all tests selected by MEST.s-r2, MEST.s-r1 and MEST.s respectively, which is as expected.

In detail, MEST, MEST.s-r1 and MEST.s-r2 select more tests than MEST.s in 28% (73/261), 22% (57/261) and 18% (46/261) versions respectively; especially, in 40, 20 and 23 versions respectively, they select more than 10 tests than MEST.s. It implies that the two dynamic execution rules contribute to collect more precise dependencies, and can select some necessary tests which are omitted by test case selection based on static dependencies.

> **Summary 3.1**: There are respectively 28%, 22% and 18% versions where MEST, MEST.s-r1 and MEST.s-r2 selects more tests than MEST.s. This implies that the two dynamic execution rules all contribute to test case selection through collecting dependencies from different points of view.

6.3.2. Fault detection efficiency

Fig. 20 presents an overview of the number of faults detected by MEST, MEST.s-r2, MEST.s-r1 or MEST.s. We can easily see that MEST detects the most faults, MEST.s detects the least faults. In detail, MEST detects more faults than MEST.s (in 46 versions), than MEST.s-r1 (in 35 versions), and than MEST.s-r2 (in 29 versions). Meanwhile, we compared the fault sets detected by MEST, MEST.s-r2, MEST.s-r1 and MEST.s respectively, and results show MEST can cover all faults detected by MEST.s-r2, MEST.s-r1 and MEST.s, which is as expected. The reason is that MEST considers all dependencies used in MEST.s-r2, MEST.s-r1 and MEST.s. The following presents more details and further analysis of the results.

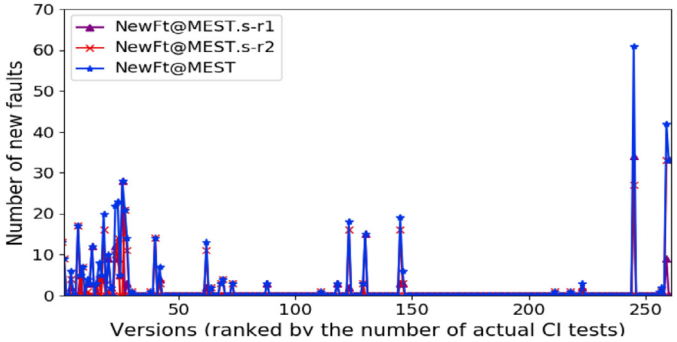

Fig. 21. Number of new faults found by MEST, MEST.s-r2 or MEST.s-r1 compared with MEST.s.

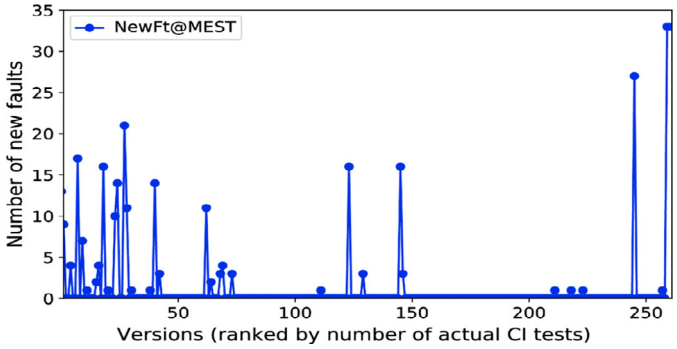

Fig. 22. Number of new faults found by MEST compared with MEST.s-r1.

(a) Comparison with MEST.s: We first compared the fault sets detected by the four approaches respectively, and results show that MEST, MEST.s-r2 and MEST.s-r1 can cover all faults detected by MEST.s because the three approaches all consider static dependencies when selecting tests.

Fig. 21 shows the number of new faults found by MEST, MEST.s-r2 or MEST.s-r1 compared with MEST.s. We observe that MEST, MEST.s-r1 and MEST.s-r2 can find new faults in 18% (46/261), 11% (29/261) and 13% (35/261) versions respectively. Especially, MEST, MEST.s-r1 and MEST.s-r2 can find more than 10 new faults than MEST.s in 18, 5 and 14 versions respectively.

The above results show that MEST, MEST.s-r1 and MEST.s-r2 all have better fault detection efficiency than MEST.s. This indicates that dynamic execution rules can bring great benefits to fault detection efficiency, and only considering static dependencies (i.e., MEST.s) can lead to a risk of omitting faults.

(b) Comparison with MEST.s-r1: We compared the fault sets detected by MEST and MEST.s-r1 respectively, and results show that MEST can cover all faults detected by MEST.s-r1.

Fig. 22 shows the number of new faults found by MEST compared with MEST.s-r1. We observe that MEST can find new faults in 13% (35/261) versions. Especially, MEST can find more than 10 new faults than MEST.s-r1 in 14 versions.

The above results indicate that MEST has better fault detection efficiency than MEST.s-r1. This further indicates dynamic execution Rule 2 can additionally contribute to fault detection.

(c) Comparison with MEST.s-r2: We compared the fault sets detected by MEST and MEST.s-r2 respectively, and results show that MEST can cover all faults detected by MEST.s-r2.

Fig. 23 shows the number of new faults found by MEST compared with MEST.s-r2. We observe that MEST can find new faults in 11% (29/261) versions. Especially, in 5 versions, MEST can find more than 10 new faults than MEST.s-r2.

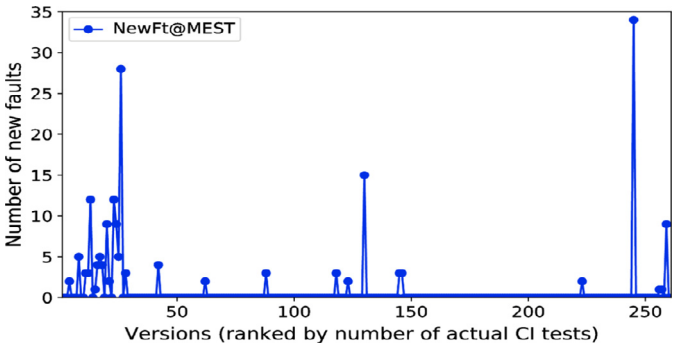

Fig. 23. Number of new faults found by MEST compared with MEST.s-r2.

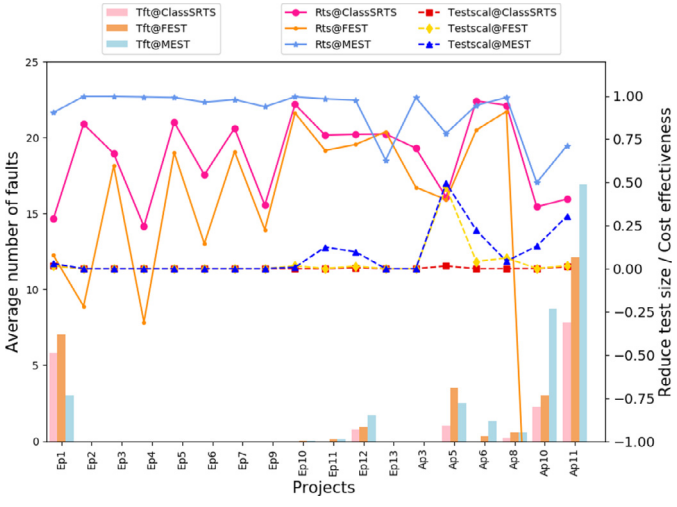

Fig. 24. Evaluation by projects in terms of reduced test size, fault detction effciency and cost effectiveness.

The above results indicate that MEST has better fault detection efficiency than MEST.s-r2. This further indicates dynamic execution Rule 1 can additionally contribute to fault detection.

> **Summary 3.2**: MEST can cover all faults detected by MEST.s, MEST.s-r1 and MEST.s-r2; and find new faults in 18%, 13%, 11% versions respectively. It indicates that the two dynamic execution rules all contribute to fault detection efficiency.

7. Discussion

7.1. Evaluation on projects

In Section 6, we present the detailed results and analysis in terms of 261 *versions*. This section discusses the performance of our proposed FEST and MEST organized in terms of 18 *projects* to provide further demonstration of their effectiveness. Fig. 24 shows the results of FEST, MEST or ClassSRTS presented by projects in terms of reduced test size, fault detection efficiency and cost effectiveness. Y-axis on the left side refers to the average number of faults across all projects, Y-axis on the right side refers to reduced test size (*Rts*) or test scale benefits (*Testscal*). Note that the 2 projects (i.e., Ap10, Ap11) where the *Rts@F* (for FEST) is lower than -1, we simply set them as -1 in Fig. 24 to improve its readability.

From Fig. 24, we observe that FEST can reduce test size in 77.8% projects; and shows higher *Rts* (in 1 project) and lower *Rts* (in 17 projects) than ClassSRTS, but it can detect more faults in 9 out of

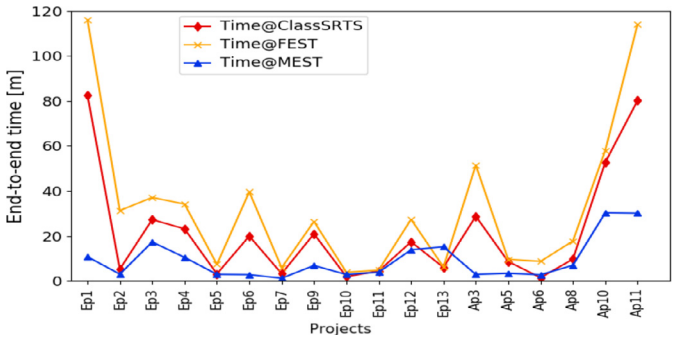

Fig. 25. Evaluation by projects in terms of test cost.

these 17 projects. On the other hand, MEST shows higher *Rts* than ClassSRTS in 89% projects; only in Ap6 and Ep13, it has lower *Rts*, but MEST can detect more faults in Ap6 and equal *Rts* in Ep13. Compared with FEST, MEST shows higher *Rts* than FEST in 94% (17/18) projects; only in Ap6, it has less *Rts*, but MEST can detect more faults than FEST.

For fault detection efficiency, there is no fault in 50% projects (e.g., Ep2, Ep3, Ep4). That is because actual CI testing passed in the versions of these projects, and no faults were detected; FEST and MEST also did not detect any faults in these projects. In other 9 projects having faults, FEST detects more faults than ClassSRTS in 100% (9/9) projects. On the other hand, MEST detects more faults than ClassSRTS in 89% (8/9) projects, especially in Ap10, MEST detects 3 times as many faults as ClassSRTS. Only in Ep1, ClassSRTS detects slightly more faults than MEST. The detailed reasons for worse performance of MEST in these projects are due to deferred faults, unresolved dependency or third party libraries, which were discussed in "*Comparison with actual CI testing*" of Section 6.1.2. Compared with FEST, MEST has better or equal *Tft* (in 78% projects), and fewer *Tft* (in 22% projects).

For cost-effectiveness, in the 9 projects having faults, FEST and MEST shows better *testscal* than ClassSRTS in 89% (8/9) and 100% (9/9) of projects respectively. Compared with FEST, MEST has higher *testscal* (in 78% projects) and lower *testscal* (in other 22% projects).

For test cost, Fig. 25 shows test cost of FEST, MEST or ClassSRTS. We can see that MEST obviously shows less end-to-end times than ClassSRTS (in 83% projects) and FEST (in 94% projects) respectively. In other 3 projects, MEST shows higher time cost than ClassSRTS or FEST, but it can detect more faults in 1 project and equal number of faults in other 2 projects. On the other hand, FEST shows more end-to-end time than the baseline ClassSRTS in most cases. That is because FEST captures full dependencies at the class level to select more tests in some cases, it can mitigate the risk of omitting necessary tests.

Hence, in terms of projects, MEST shows better reduced test size (in 89% projects) and test cost (in 83% projects), better or equal fault detection efficiency (in 94% projects) and cost effectiveness (in 100% projects) compared with ClassSRTS; for FEST, it shows better or equal performance than ClassSRTS in fault detection efficiency (in 100% projects) and cost effectiveness (in 94% projects), while it underperforms ClassSRTS in terms of test cost and reduced test size.

7.2. Comparison of FEST and MEST

Most previous studies have revealed that finer-grained test case selection (e.g., method-level) can usually collect more precise dependencies than coarser-grained test case selection (e.g., class-level), thus would select a relatively accurate test set. However, recent research (Legunsen et al., 2016) pointed out that method-

level approach is much less effective than class-level approach ClassSRTS. It implies that researchers should carefully model the dependencies on different granularities. The experimental evaluations in this paper further support this common statement. In detail, by modeling precise method-level dependencies, MEST can reduce test size by 48% compared with FEST. Furthermore, because some dependencies are determined dynamically at runtime, the dependencies which are modeled solely based on static analysis are not accurate. Our proposed MEST models dependencies with static analysis and dynamic execution rules, and generates a more precise dependency graph, so that MEST can not only cover all faults detected by FEST in 95% versions, but also find new faults in 20% versions.

From another point of view, previous studies also revealed that finer-grained test case selection (e.g., method-level) usually has more overhead than coarser-grained test case selection (e.g., class-level). The experimental evaluations presented in this study provide new insight about the time cost of test case selection approaches in different granularities. Specifically, it is true that MEST needs slightly more time for test case selection, i.e., 9 minutes vs. 8.3 minutes of FEST, because of the modeling of precise finer-grained dependencies. But end-to-end time (i.e., time to select tests and time to run the selected tests) of MEST is only 32% of FEST because MEST selects quite a smaller test subset guided by the precise dependencies. This indicates that as long as the method-level dependencies can be precisely modeled as our proposed MEST, method-level test case selection can be more effective not only for fault detection efficiency but also for the time cost. In this way, this study provides new guidelines for choosing method-level or class-level test case selection approach.

However, for the ultra large systems, the cost for resolving the method-level dependencies will be much higher. Furthermore, if the integration version only undergoes a minor change, the cost for running tests would be ignorable. In this case, FEST will be more promising. In addition, we found that there is no test cases for some classes in our experimental projects, even if MEST and FEST can search affected tests. Under this situation, FEST would be easier to map the changed codes to affected tests, because the coarser-level trace links usually could be easier to be established.

7.3. Innovation

This paper proposes a class-level test case selection approach (FEST) and a method-level test case selection approach (MEST), which aim at selecting a precise test subset towards fully covering all changed code and affected code so as to reduce test cost without sacrificing quality. We pursue this goal by three ways:

Firstly, the two proposed approaches can capture the full and precise dependencies on class-level or method-level. In detail, FEST can capture the class-level dependencies (i.e., hidden references) to facilitate the modeling of the full dependency relations at the class level. On the other hand, MEST combines dynamic execution rules with static dependencies (e.g., method-level hidden references) to capture the method-level dependencies more precisely. In comparison, existing static analysis approaches could not capture all static dependencies and the dependencies dynamically determined at runtime, e.g., ClassSRTS only considers static dependencies and can not deal with some special static dependencies (e.g., the dependency of reflections).

Secondly, FEST and MEST search all affected classes/methods and find related test classes incrementally and iteratively by applying BFS on the captured dependencies. Coppin proved the completeness of BFS in his work (Coppin, 2004). On the other hand, ClassSRTS 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). They claimed that this setup is to ensure that the newly-added tests (our approaches can also select these tests) can be included. However, the set of all tests can include some unnecessary classes, e.g., the basic classes which do not have test cases, which results in an inaccurate selected test set.

Thirdly, considering real-world practice, we add the recently failed test classes to re-detect the deferred faults to further improve its effectiveness.

Facilitated by these three aspects, our proposed approaches can select a proper test subset for CI testing which can reduce test cost as much as possible without sacrificing quality.

7.4. Threats to validity

The **internal** validity of our study arises from the implementation of ClassSRTS. It is implemented by strictly following the procedures described in the original paper (Legunsen et al., 2017). For both ClassSRTS and two of our proposed approaches, we have employed 187 test cases to verify their functionality. For the suspicious results, we have manually checked the source code, and it turned out that they are not due to the defect in code. In addition, following previous approaches, we treat the failed test cases in the first run of enhanced test of the subsequent versions as a fault. But we did not consider the cases that the fault is caused by new changes. Further exploration is needed. Besides, the dynamic binding of our approach might miss some dependencies in very rare cases. For example, when the instance of a subclass is assigned to the object of its superclass as a parameter, our approach would miss this dependency. However, compared with existing algorithms for dynamic binding (e.g., CHA, RTA, CFA), our approach can build more precise dependencies thus select a more proper tests set.

The **external** validity concerns about the experimental dataset. We can not guarantee that the presented results can be fully generalized to other projects. However, the size of the experimental dataset (18 projects with 261 versions) and the diversity of domains relatively reduce this risk. Besides, one may question about the choice of the baselines. One baseline is the actual CI testing which is the most commonly-used one; and the other is ClassSRTS (Legunsen et al., 2016), which is the state-of-the-art approach for static test case selection of class-level. Besides, ClassSRTS has better performance than the state-of-the-art method-level approach (Gligoric et al., 2015). Nevertheless, more evaluation is needed to compare our approach with other test case selection techniques (e.g., dynamical test case selection) to further prove the effectiveness of our proposed approaches.

8. Related work

CI test case selection in terms of different granularities of code dependencies: Many test case selection techniques have been proposed in terms of different granularities of code dependencies. Some selection techniques are based on coarser-grained dependencies, e.g., module-level (Knauss et al., 2015; Vasic et al., 2017), package-level (Ekelund and Engström, 2015), file-level (Gligoric et al., 2015; Vasic et al., 2017; Celik et al., 2017), to select tests. However, these techniques show low precision because they tend to select unnecessary tests which lead to relative large size of selected test set (Blondeau et al., 2016; Zhang, 2018). Therefore, some researchers proposed selection techniques based on finer-grained dependencies (e.g., method-level). These finer-grained dependencies are more precise thus can help select more precise tests and obtain a relatively smaller size of selected test set. However collecting them is more difficult than collecting the coarser-grained dependencies and may introduce substantial overhead (Gligoric et al., 2015; Zhang, 2018). For example, Legunsen et al. (Legunsen et al., 2016; 2017) implemented class-level and method-level test

case selection techniques based on static dependencies. Results showed that the class-level approach would still omit some tests because it can not obtain all dependencies (e.g., the dependencies in reflection); while the method-level approach shows worse performance in fault detection and time cost. Other selection approaches (Soetens et al., 2016; Blondeau et al., 2016; Parsai et al., 2014) based on method-level dependency can select a small size of tests, but there are some limitations, e.g., they only dealt with changes in methods (not including fields), and can not capture the invocation between constructor functions and the inheritance dependencies of test code. The main challenge of these aforementioned studies based on finer-grained dependencies is the inadequate and imprecise modeled dependencies.

CI test case selection in terms of different analysis techniques: Test case selection techniques can also be divided into dynamic selection (Gligoric et al., 2015; Vasic et al., 2017; Celik et al., 2017; Zhang, 2018) and static selection (Parsai et al., 2014; Soetens et al., 2016; Legunsen et al., 2016; 2017) in terms of different methods of collecting dependencies. Dynamic selection collects dependencies by executing tests on the previous revision, and static selection uses static program analysis to infer dependencies. In terms of dynamic selection techniques, Gligoric et al. (Gligoric et al., 2015) proposed a test selection approach based on dynamic dependencies of source code for *Java* projects. Then Vasic et al. (Vasic et al., 2017) implemented the approach for.NET language. To further improve precision of dynamic selection, Zhang (Zhang, 2018) and Celik et al. (Celik et al., 2017) proposed hybrid test selection technique and designed dynamic test selection across *JVM* boundaries respectively. When it comes to static selection techniques, Soetens et al. (Soetens et al., 2013) proposed an approach to select tests on method-level. It did not deal with polymorphism invocation, which could omit some necessary tests. To address this, Parsai et al. (Parsai et al., 2014) improved this approach by incorporating polymorphism relation. Legunsen et al. (Legunsen et al., 2016; 2017) implemented class-level and method-level test selection techniques based on static dependency. But the approach would still omit some tests because it can not obtain all dependencies (i.e., the dependency of reflections). Next, Shi et al. (Shi et al., 2019) investigated three static techniques and two hybrid techniques for reflection detection, the two approaches showing better performance than others, are safer (i.e., without losing dependencies) than our approach in capturing reflections, but they would include some unnecessary dependencies. In comparison, our aim is to build precise dependencies and select a proper set of tests. Existing studies (Soetens et al., 2016; Blondeau et al., 2016) compared the dynamic and static techniques, and results showed that the shortcomings with existing dynamic selection technique is long running tests, non-determinism, and real-time constraints. Static selection technique is easier to manipulate than dynamic selection technique, but its shortcomings lie in omitting some tests or selecting some unnecessary tests. The main reason is that some dependencies are dynamically determined at runtime and can not resolved by static analysis.

Considering the advantages and challenges of static test case selection, our proposed approaches aim at selecting a precise test subset towards fully covering all changed code and affected code.

CI test case selection based on other information: There are some studies based on the correlations between historical test case failures and source code changes (Ekelund and Engström, 2015; Knauss et al., 2015) or testing history for test case selection (Yoo et al., 2011; Elbaum et al., 2014; Memon et al., 2017; Spieker et al., 2017; Kwon and Ko, 2017). Other studies used test case diversity and code coverage to select tests (Beszédes et al., 2012; Mondal et al., 2015). Besides, cost-based selection technique (Herzig et al., 2015) and refactoring-aware selection technique (Wang et al., 2018) were also proposed to further improve the cost effectiveness of tests. These non-code information is also helpful for test case selection, we plan to combine code dependencies and non-code information to improve the performance of test case selection in future.

9. Conclusion and future work

This paper proposes a StAtic PrecIse tEst case selecTioN framework named *Sapient* which aims at selecting a precise test subset towards fully covering all changed code and affected code so as to reduce test cost without sacrificing quality. Based on *Sapient*, we then develop a class-level test case selection approach FEST and a method-level test case selection approach MEST. Evaluation is conducted on 261 integration versions of 18 open source projects from two large open source communities (i.e., Eclipse and Apache) to evaluate FEST and MEST from reduced test size, fault detection efficiency, test cost and cost effectiveness. Results show that both FEST and MEST can detect almost all faults of two baselines, and find new faults compared with actual CI testing (in 25% and 26% versions) and ClassSRTS (in 18% and 27% versions) respectively. Furthermore, MEST performs FEST in reduced test size, fault detection efficiency and test cost, indicating method-level test case selection can be more effective than class-level selection. Our study can help practitioners manage CI testing, and improve the fault detection efficiency of current practices of CI testing.

In future work, we plan to deal with the unresolved dependencies to further improve the safety of our approaches. Meanwhile, we will try to build the tool on the project bytecode in the future version, so as to save more analysis time and further improve the selection performance. Besides, we also try to explore other features (e.g., test history information, non-code changes) and adopt machine learning approach to improve the performance of test case selection.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Beszédes, A., Gergely, T., Schrettner, L., Jász, J., Langó, L., Gyimóthy, T., 2012. Code coverage-based regression test selection and prioritization in webkit. In: IEEE International Conference on Software Maintenance, pp. 46–55. https://doi.org/10.1109/icsm.2012.6405252

Blondeau, V., Etien, A., Anquetil, N., Cresson, S., Croisy, P., Ducasse, S., 2016. Test case selection in industry: An analysis of issues related to static approaches. Software Quality Journal 1–35. https://doi.org/10.1007/s11219-016-9328-4

Booch, G., 1991. Object oriented design: With applications. Benjamin-Cummings Publishing Company, Subs of Addison Wesley Longman, Inc.

Celik, A., Vasic, M., Milicevic, A., Gligoric, M., 2017. Regression test selection across JVM boundaries. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 809–820. https://doi.org/10.1145/3106237.3106297

Coppin, B., 2004. Artificial intelligence illuminated. Jones and Bartlett Publishers, Inc..

Duvall, P.M., Matyas, S., Glover, A., 2007. Continuous integration. In: Addison-Wesley Professional, pp. 25–45.

Ekelund, E.D., Engström, E., 2015. Efficient regression testing based on test history: An industrial evaluation. In: IEEE International Conference on Software Maintenance and Evolution, pp. 449–457. https://doi.org/10.1109/icsm.2015.7332496

Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 235–245. https://doi.org/10.1145/2635868.2635910

EngstrÖm, E., Runeson, P., Ljung, A., 2011. Improving regression testing transparency and efficiency with history-based prioritization - an industrial case study. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, pp. 367–376. https://doi.org/10.1109/icst.2011.27

Engström, E., Runeson, P., Skoglund, M., 2010. A systematic review on regression test selection techniques. Information and Software Technology 52 (1), 14–30. https://doi.org/10.1016/j.infsof.2009.07.001

Gligoric, M., Eloussi, L., Marinov, D., 2015. Practical regression test selection with dynamic file dependencies. In: International Symposium on Software Testing and Analysis, pp. 211–222. https://doi.org/10.1145/2771783.2771784
Herzig, K., Greiler, M., Czerwonka, J., Murphy, B., 2015. The art of testing less without sacrificing quality. In: International Conference on Software Engineering, pp. 483–493. https://doi.org/10.1109/icse.2015.66
Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D., 2016. Usage, costs, and benefits of continuous integration in open-source projects. In: Automated Software Engineering Conference, pp. 426–437. https://doi.org/10.1145/2970276.2970358
Jiang, B., Chan, W.K., 2016. Testing and debugging in continuous integration with budget quotas on test executions. In: IEEE International Conference on Software Quality. https://doi.org/10.1109/qrs.2016.66
Knauss, E., Staron, M., Meding, W., Soder, O., Nilsson, A., Castell, M., 2015. Supporting continuous integration by code-churn based test selection. In: International Workshop on Rapid Continuous Software Engineering, pp. 19–25. https://doi.org/10.1109/rcose.2015.11
Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y., 1995. Class firewall, test order, and regression testing of object-oriented programs. Journal of Object-Oriented Programming 8 (2), 51–65. https://dblp.org/rec/bib/journals/joop/KungGHLT95
Kwon, J.H., Ko, I.Y., 2017. Cost-effective regression testing using bloom filters in continuous integration development environments. In: Asia-Pacific Software Engineering Conference, pp. 160–168. https://doi.org/10.1109/apsec.2017.22
Labuschagne, A., Inozemtseva, L., Holmes, R., 2017. Measuring the cost of regression testing in practice: A study of java projects using continuous integration. In: Joint Meeting on Foundations of Software Engineering, pp. 821–830. https://doi.org/10.1145/3106237.3106288
Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., Marinov, D., 2016. An extensive study of static regression test selection in modern software evolution. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 583–594. https://doi.org/10.1145/2950290.2950361
Legunsen, O., Shi, A., Marinov, D., 2017. Starts: Static regression test selection. In: Automated Software Engineering Conference, pp. 949–954. https://doi.org/10.1109/ase.2017.8115710
Li, Y.L., Wang, J., Wang, Q., J., H., 2019. A class-level test selection approach toward full coverage for continuous integration. In: Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering, pp. 49–55. https://doi.org/10.18293/seke2019-011
Li, Y.L., Wang, J., Yang, Y., Wang, Q., 2019. Method-level test selection for continuous integration with static dependencies and dynamic execution rules. In: IEEE International Conference on Software Quality, Reliability, and Security, pp. 350–361. https://doi.org/10.1109/qrs.2019.00052
Marijan, D., Gotlieb, A., Sen, S., 2013. Test case prioritization for continuous regression testing: An industrial case study. In: Proceedings of 29th IEEE International Conference on Software Maintenance, pp. 540–543. https://doi.org/10.1109/icsm.2013.91
Marijan, D., Liaaen, M., 2016. Effect of time window on the performance of continuous regression testing. In: IEEE International Conference on Software Maintenance and Evolution, pp. 568–571. https://doi.org/10.1109/icsme.2016.77
Marijan, D., Liaaen, M., Gotlieb, A., Sen, S., Ieva, C., 2017. Titan: Test suite optimization for highly configurable software. In: International Conference on Software Testing, Verification and Validation, pp. 524–531. https://doi.org/10.1109/icst.2017.60
Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J., 2017. Taming google-scale continuous testing. In: International Conference on Software Engineering, pp. 233–242. https://doi.org/10.1109/icse-seip.2017.16
Meyer, B., 1997. Object-Oriented software construction. Prentice Hall, New York, N.Y., second edition.
Mirarab, S., Akhlaghi, S., Tahvildari, L., 2012. Size-constrained regression test case selection using multicriteria optimization. IEEE Transactions on Software Engineering 936–956. https://doi.org/10.1109/tse.2011.56
Mondal, D., Hemmati, H., Durocher, S., 2015. Exploring test suite diversification and code coverage in multi-objective test case selection. In: International Conference on Software Testing, Verification and Validation, pp. 1–10. https://doi.org/10.1109/icst.2015.7102588
Nardo, D.D., Alshahwan, N., Briand, L., Labiche, Y., 2015. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. Journal of Software Testing, Verification and Reliability 25 (4), 371–396. https://doi.org/10.1002/stvr.1572
Orso, A., Shi, N., Harrold, M.J., 2004. Scaling regression testing to large software systems. ACM SIGSOFT Software Engineering Notes 29 (6), 241–251. https://doi.org/10.1145/1029894.1029928
Parsai, A., Soetens, Q.D., Murgia, A., Demeyer, S., 2014. Considering polymorphism in change-based test suite reduction. In: International Conference on Agile Software Development, pp. 166–181. https://doi.org/10.1007/978-3-319-14358-3_14
Ryder, B., 2004. Chianti: A tool for change impact analysis of java programs. ACM Sigplan Notices 39 (10), 432–448. https://doi.org/10.1109/icse.2005.1553643
Schermann, G., Cito, J., Leitner, P., Gall, H.C., 2016. Towards quality gates in continuous delivery and deployment. In: IEEE International Conference on Program Comprehension. https://doi.org/10.1109/icpc.2016.7503737
Shi, A., Hadzi-Tanovic, M., Zhang, L. M., Marinov, D., Legunsen, O., 2019. Reflection-aware static regression test selection. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1–29 doi:10.1145/3360613.
Shi, A., Thummalapenta, S., Lahiri, S.k., Bjorner, N., Czerwonka, J., 2017. Optimizing test placement for module-level regression testing. International Conference on Software Engineering, pp. 689–699. https://doi.org/10.1109/icse.2017.69
Shi, A., Yung, T., Gyori, A., Marinov, D., 2015. Comparing and combining test-suite reduction and regression test selection. In: Joint Meeting on Foundations of Software Engineering, pp. 237–247. https://doi.org/10.1145/2786805.2786878
Soetens, Q.D., Demeyer, S., Zaidman, A., 2013. Change-based test selection in the presence of developer tests. In: European Conference on Software Maintenance and Reengineering, pp. 101–110. https://doi.org/10.1109/csmr.2013.20
Soetens, Q.D., Demeyer, S., Zaidman, A., Pérez, J., 2016. Change-based test selection: An empirical evaluation. The Journal of Empirical Software Engineering 21 (5), 1990–2032. https://doi.org/10.1007/s10664-015-9405-5
Spieker, H., Gotlieb, A., Marijan, D., Mossige, M., 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: International Symposium on Software Testing and Analysis, pp. 12–22. https://doi.org/10.1145/3092703.3092709
Vahabzadeh, A., Stocco, A., Mesbah, A., 2018. Fine-grained test minimization. In: International Conference on Software Engineering, pp. 210–221. https://doi.org/10.29007/qd4q
Vasic, M., Parvez, Z., Milicevic, A., Gligoric, M., 2017. File-level vs. module-level regression test selection for.Net. In: ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 848–853. https://doi.org/10.1145/3106237.3117763
Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V., 2015. Quality and productivity outcomes relating to continuous integration in github. In: Joint Meeting on Foundations of Software Engineering, pp. 805–816. https://doi.org/10.1145/2786805.2786850
Wang, K., Zhu, C.G., Celik, A., Kim, J., Batory, D., Gligoric, M., 2018. Towards refactoring-aware regression test selection. In: International Conference on Software Engineering, pp. 233–244. https://doi.org/10.1145/3180155.3180254
Wikstrand, G., Feldt, R., Gorantla, J.K., Zhe, W., White, C., 2009. Dynamic regression test selection based on a file cache: An industrial evaluation. In: International Conference on Software Testing, Verification and Validation, pp. 299–302. https://doi.org/10.1109/icst.2009.42
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: A survey. Journal of Software Testing, Verification and Reliability 22 (2), 67–120. https://doi.org/10.1002/stv.430
Yoo, S., Nilsson, R., Harman, M., 2011. Faster fault finding at google using multi objective regression test optimization. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1–4.
Zhang, L., 2018. Hybrid regression test selection. In: International Conference on Software Engineering, pp. 199–209. https://doi.org/10.1145/3180155.3180198
Zhang, L.M., Kim, M., Khurshid, S., 2011. Localizing failure-inducing program edits based on spectrum information. In: International Conference on Software Engineering, pp. 23–32. https://doi.org/10.1109/icsm.2011.6080769

Yingling Li is a lecturer at Southwest Minzu University, and received her phD degree from Institute of Software, Chinese Academy of Sciences (ISCAS) in 2019. Her research focuses on intelligent software process and continuous integration testing; she received the IEEE Best Paper Award at QRS in 2019.

Junjie Wang is an associate researcher at ISCAS. She received the PhD degree from ISCAS in 2015. Her research interests include mining software repositories, intelligent software engineering and crowdtesting. She has more than 20 high-quality publications, and received the ACM SIGSOFT Distinguished Paper Award at ICSE in 2019, IEEE Best Paper Award at QRS in 2019.

Yun Yang received his PhD degree from the University of Queensland, Australia, in 1992, in computer science. He is currently a full professor in the School of Software and Electrical Engineering at Swinburne University of Technology, Melbourne, Australia. His research interests include software technologies, cloud and edge computing, workflow systems, and service computing.

Qing Wang is a researcher at ISCAS. She is also the deputy chief engineer of ISCAS, and director of Laboratory for Internet Software Technologies of ISCAS. Her research lies in the area of software process, software quality assurance, requirement engineering, knowledge engineering, big data and artificial intelligence for software engineering. She has edited/co-edited 5 books, and published more than 100 papers in international high-level conferences and journals.