Will This Bug-fixing Change Break Regression Testing?

Xinye Tang
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
tangxinye@nfs.iscas.ac.cn

Song Wang
Electrical and Computer Engineering
University of Waterloo, Canada
song.wang@uwaterloo.ca

Ke Mao
CREST Centre
University College London, UK
k.mao@cs.ucl.ac.uk

***Abstract*—Context:** Software source code is frequently changed for fixing revealed bugs. These bug-fixing changes might introduce unintended system behaviors, which are inconsistent with scenarios of existing regression test cases, and consequently break regression testing. For validating the quality of changes, regression testing is a required process before submitting changes during the development of software projects. Our pilot study shows that 48.7% bug-fixing changes might break regression testing at first run, which means developers have to run regression testing at least a couple of times for 48.7% changes. Such process can be tedious and time consuming. Thus, before running regression test suite, finding these changes and corresponding regression test cases could be helpful for developers to quickly fix these changes and improve the efficiency of regression testing. **Goal:** This paper proposes bug-fixing change impact prediction (*BFCP*), for predicting whether a bug-fixing change will break regression testing or not before running regression test cases, by mining software change histories. **Method:** Our approach employs the machine learning algorithms and static call graph analysis technique. Given a bug-fixing change, *BFCP* first predicts whether it will break existing regression test cases; second, if the change is predicted to break regression test cases, *BFCP* can further identify the might-be-broken test cases. **Results:** Results of experiments on 552 real bug-fixing changes from four large open source projects show that *BFCP* could achieve prediction precision up to 83.3%， recall up to 92.3%， and F-score up to 81.4%. For identifying the might-be-broken test cases, *BFCP* could achieve 100% recall.

Keywords—source code change impact analysis; regression testing; static program analysis.

I. INTRODUCTION

Software source code is frequently changed for fixing revealed bugs, which is called bug-fixing change. Previous studies [1, 5, 17, 19, 39] have shown that these bug-fixing changes might introduce unintended system behaviors, which are inconsistent with scenarios of existing regression test cases, and consequently break regression testing. However, it is difficult for developers to identify such changes manually. Especially, semantics changes [6, 19], which might introduce unintended system behaviors. To identify such changes, developers often perform regression testing, which is a key process during software development. It could help examine the quality of source code changes by checking whether they introduce unintended system behaviors which break existing regression test cases [3]. Running system regression test cases to check whether new source code changes break regression test cases is a required process before submitting source code changes during the development of many software projects. Unfortunately, for most software projects, such process can be tedious and time consuming.

First, as a software project evolves, its test suite grows in size. Given limited test resource and time, running regression test cases for all bug-fixing changes might be infeasible for many software projects, especially for large projects, e.g., Hadoop[1] has more than 1,500 regression test cases. Executing the whole test suite would cost several hours. Intuitively, a tool that can predict whether a bug-fixing change will break regression testing or not should be helpful for developers, since it provides early evaluation about the quality of the change.

Second, not all bug-fixing changes pass regression testing at first run. Our pilot study (Section II) shows that about 48.7% bug-fixing source code changes might break regression testing at the first run because of the inconsistency between changes and regression test cases, developers have to fix these changes after first run and re-run regression test suite until all regression test cases are passed. Thus, before running regression test suite finding these changes and corresponding regression test cases could be helpful for developers to quickly fixing these changes and improve the efficiency of regression testing.

Thus, a tool for evaluating bug-fixing changes with respect to whether they will break existing regression test cases would provide developers early feedback about the quality of bug-fixing changes, narrow down their code review work space [2, 19], and further improve developers' productivity.

In this paper, we propose bug-fixing change impact prediction (*BFCP*), for predicting whether a bug-fixing source code change will break regression testing before running regression test cases, by mining software source code change histories. Specifically, given a bug-fixing source code change, without executing regression test suite, *BFCP* performs the following prediction tasks:

1) *BFCP* first predicts whether this bug-fixing change will break regression testing or not;

[1] https://hadoop.apache.org

2) If the change is predicted that have a big possibility to break regression testing, *BFCP* can further identify these might-be-broken test cases.

We examine *BFCP* on 552 real bug-fixing changes from 18 release versions of four large open source projects. Results show that *BFCP* could achieve prediction precision up to 83.3% and recall up to 92.3%. For identifying the might-be-broken test cases, *BFCP* could achieve 100% recall, with an average number of identified test cases less than 20.

The main contributions of this work include:

1) We provide a list of 18 unique metrics for predicting the impact of bug-fixing changes on regression testing.

2) We propose *BFCP* to predict whether a bug-fixing change will break regression testing and identify bug-fixing changes that might break regression testing, before running regression test cases.

3) We evaluate the performance of *BFCP* on four large-scale open source projects. Results show that *BFCP* could provide efficient results.

4) To the best of our knowledge, this is the first work to predict the impact of bug-fixing source code changes on regression testing.

In the remainder of this paper, Section II presents our pilot study and motivation; Section III shows the overview of proposed approach; Section IV explains our experiment settings; Section V presents our analysis of experiments results; Section VI discusses our approach; Section VII is our related work; Section VIII presents the threats to this work; Section IX summarizes this work.

II. PILOT STUDY AND MOTIVATION

This section reports our motivation and the result of our pilot study on evaluating the impact of bug-fixing source code changes on regression testing.

Goal: In order to provide developers early information about the quality of bug-fixing changes, we try to predict whether a specific bug-fixing source code change might break regression testing or not. Intuitively, one important question against our study is: "***what's the percentage of bug-fixing source code changes that might break regression testing***?"

The answer to this question directly impacts the feasibility of our study, that is whether it worth to predict the impact of a specific bug-fixing source code change on corresponding regression test suite. If the percentage is quite significant for example, among all bug-fixing source code changes of a project, 30% changes might break regression testing, which means about 30% changes have to run regression test suite at least twice (the first run might break regression test cases and then developers might resolve this and rerun again to make sure no regression errors are triggered).

TABLE I. Details of bug-fixing changes. Bug-to-changes are the valid bug-fixing source code changes. Buggy changes are the changes that can break regression testing.

Project	Version	Regression test suite size	#Fixed bugs	#Bug-to-change	#Buggy change
Ant	1.8.2	223	27	23	5
	1.8.4	230	25	21	5
Lucene	4.4.0	518	23	22	16
	4.5.0	526	16	16	14

Thus, if we can predict these bug-fixing changes before submitting the changes, by prediction models and identify these might-be-broken test cases, we would provide developers early information about the quality of bug-fixing source code changes and narrow down their code review work space, further improve their productivity. On the contrary, if only less than 1% bug fixing changes might break regression test cases, which means such change is trivial and rare, thus predicting the impact for every bug-fixing source code change is trivial.

The goal of our pilot study is to evaluate the feasibility and cost-effect of predicting whether a specific bug-fixing source code change might break regression testing or not.

Approach: We manually collect bug-fixing source code changes and corresponding regression test suite set from four randomly selected versions of open source projects: Ant and Lucene as sample data set for our pilot study. First, we collect the fixed bugs of each version from their release notes; second, we collect corresponding source code changes by analyzing patches of each reported bugs in their bug repository, both Ant and Lucene used Bugzilla[2] to track and maintain their reported bugs. Note that, not all bug-fixing source code changes are patched in corresponding bug reports maintained in Bugzilla. We use a heuristics link recovering method [21] to find links between bugs and change logs maintained in change repository, i.e., searching for specific keywords and bug IDs in change logs; then, we collect corresponding regression test suite before the submission of each bug-fixing source code change.

After collecting sample data, we manually examine the impact of each bug-fixing source code change on regression test suite by running the version of regression test suite just before the change was committed. The number of bug-to-changes is less than that of fixed bugs, because not all fixed bugs are related to source code. Changes which break regression testing are labelled as *Buggy*, changes that will not break regression testing are labelled as *Clean*. The details are shown in Table Ⅰ.

Results: As shown in Table Ⅰ, using heuristics link recovering method, we could find corresponding source code changes for over 84% bugs in each version. The average proportion of bug-fixing source code changes that might break regression testing in Ant is 22.7%, and the ratio for Lucene is 78.9%. Overall, ***the average percentage of bug-fixing source***

[2] https://www.bugzilla.org/

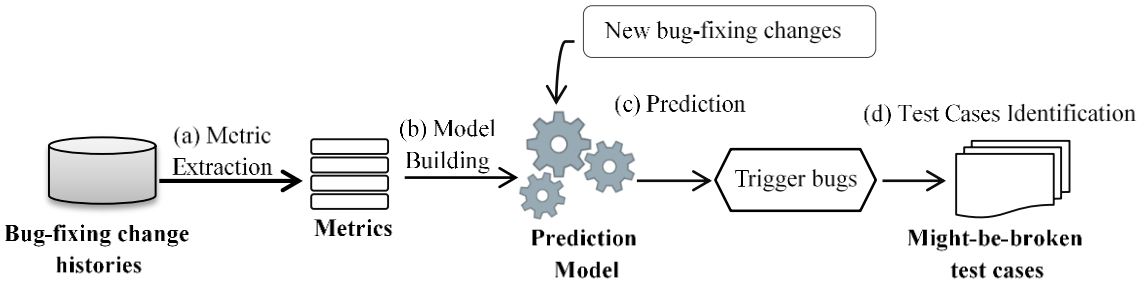


Figure. 1. Overview of *BFCP*.

code changes that might break regression testing is 48.7% on our sample data set.

The results indicate that about half of bug-fixing source code changes would break regression test suite and developers have to run regression testing at least twice, in theory a prefect prediction tool could save a lot of time by providing early and accurate information about the quality of a specific bug-fixing change.

Results of our pilot study motivate us to dig deeper in predicting the impact of bug-fixing source code changes on regression testing. In this work, we propose *BFCP* to identify bug-fixing source code changes that might break regression testing and if the change is predicted with a big possibility to break regression test cases, *BFCP* can further identify the might-be-broken test cases.

III. METHODOLOGY

Figure 1 shows that *BFCP* consists of four main processes: (1) extracting metrics from source code change histories (Section III A); (2) evaluating performance of extracted metrics (Section III B); (3) building models based on extracted metrics (Section III C); (4) using our prediction models to predict whether a bug-fixing source code change might break regression testing or not, and if a change is predicted that have a big possibility to break regression testing, our approach can further identify these might-be-broken test cases for this change (Section III D).

A. Metric Extraction

To build and train classification models for predicting the impact of bug-fixing source code changes on regression test suite of a project, this step presents how we extract metrics. First, for each change, we manually extracted 18 unique metrics from bug-fixing change histories. These metrics are widely used in change impact analysis [1, 3, 4, 30, 31, 38, 40, 41]. In order to make sure all metrics are correctly collected, we use the same manual extraction method proposed in a previous study [35]. Specifically, the first author collected all 18 metrics for each change, and then the second author re-collected metrics for each change, after that we merged the results from first and second authors, if there had conflicts, each conflict finally got resolved by a joint pair-inspection of all authors. For labelling each source code change, using the same approach in our pilot study, we examine the impact of each bug-fixing source code change on regression test suite by running the version of regression test suite just before the change wa committed. Changes which break regression testing are labelled as *Buggy*, changes that will not break regression testing are labelled as *Clean*.

In this work, these 18 metrics are grouped into three categories derived from the bug-fixing source code change histories, which are: Size metrics, Atomic Change metrics, and Semantic Change metrics. We describe each of these three categories of metrics as follows:

Size Metrics: This type of metrics are used for represent statistical source code changes, and have been widely used in software defect prediction [30, 31, 38], and source code change impact analysis [1, 3, 4]. Intuitively, a large change has a higher chance of introducing new bugs or changing existing semantics, thus results in regression test cases failed. In this study, we also use size metrics to measure the impact of a bug-fixing change on regression test suite. Overall, five unique factors cover the statistical information of involved classes, methods of bug-fixing source code changes are considered, the detail of each type of size metric is shown in Table Ⅱ.

Atomic Change Metrics: This type of metrics were first introduced by Ryder et al. [39] to analyze the impact of source code changes and predict whether software source code changes would introduce bugs to programs. Along this line, many source code change impact analysis related studies [3, 4, 40, 41] have employed these metrics to predict the impact of source code changes on software quality. In our study, although our goal is not to predict whether a source code change might introduce latent bugs or not, we still believe these metrics would be helpful to measure the impact of a bug-fixing source code changes on regression test suite, e.g., given a bug-fixing source code change, if the value of metric **DM** (whether this bug-fixing source code change has deleted method) is true, which means at least one method is deleted from source code in this change, so if this method has corresponding test cases, when running regression test suite these test cases would be failed without any modification. The details of these metrics are presented in Table Ⅱ.

Semantic Change Metrics: To capture the impact of bug-fixing source code changes on regression test suite, we have further explored metrics to represent semantic changes. Existing work [6, 19] shown that semantic changes are easily introduced to program and difficult to manually find for developers. However, regression test cases usually are designed for particular logic or semantic of corresponding source code.

TABLE II. Details of metrics used in *BFCP*.

Metric category	Metric name	Type	Definition
Size	**LA**	Numeric	lines of code added
	LD	Numeric	lines of code deleted
	LF	Numeric	number of files changed
	NC	Numeric	number of changed class
	NM	Numeric	number of changed method
Atomic Change	**AC**	Boolean	has added classes
	DC	Boolean	has deleted classes
	AM	Boolean	has added methods
	DM	Boolean	has deleted methods
	CM	Boolean	has changed methods body
	MR	Boolean	has renamed methods
	PC	Boolean	has changed parameters of methods
Semantic Change	**CC**	Numeric	number of changed dependencies
	DD	Numeric	number of deleted dependencies
	AD	Numeric	number of added dependencies
	A/RF	Boolean	has added/removed for blocks
	A/RW	Boolean	has added/removed while blocks
	A/RI	Boolean	has added/removed if blocks

If semantic information of source code is changed, these test cases have a big possibility to be broken, when running regression test suite.

For obtaining representative semantic change metrics, we randomly selected 88 bug-fixing source code changes from our pilot study, and manually examined these changes to explore the common semantic changes that might break regression testing. In total, we found that 40 of the 88 bug-fixing source code changes would break regression test cases. Specifically, among the 40 changes, we found that 15 changes were involved with control flow, i.e., for block, while block, and if block, so **A/RF**, **A/RW**, and **A/RI** metrics which measure the changes of control flow were added to our metrics list. Additionally, we found that eight changes involved dependency changes, so **CC**, **DD**, and **AD** were added to our metrics list. In Table Ⅱ we presented the details of Semantic Change metrics.

B. Metric Evaluation

Since for most of these metrics are not designed for predicting the impact of bug-fixing changes on software regression testing. After we collected all 18 metrics, we evaluated their performance in the two ways: first, we evaluated whether all metrics are relevant for our prediction task; second, we further examined the performance of proposed three different categories of metrics. The details of our metrics evaluation are presented in Section V **RQ1** and **RQ2**.

C. Model Building

We build our prediction model based on the 552 bug-fixing source code changes. In this work, predicting the impact of bug-fixing source code changes on regression testing is modeled as a binary classification problem. To find a better model, we try different machine learning algorithms: Support Vector Machine (SVM), Naive Bayes (NB), Alternating Decision Tree (ADTree), Logistic Regression (Logistic). These algorithms are widely used to solve classification and prediction problems in software engineering [30, 31, 36]. We use Weka [37] to re-implement all these algorithms.

D. Prediction and Test Case Identification

We try to leverage built models to predict the impact of bug-fixing source code changes on regression test suite. Specifically, we sort bug-fixing changes in chronological order in that the earlier changes have an impact on the following ones. We divide them into 4 folds, each fold has the same number of *Buggy* changes, then we use data from the first three folds to train our models, and test the models on the data from the last fold.

The last step of *BFCP* is to identify the might-be-broken test cases, when a bug-fixing source code change is predicted that might break regression testing. For finding these might-be-broken test cases we employ static call graph analysis technique, which is widely used in existing program analysis work [12, 13, 14, 15, 16]. In this work, when a bug-fixing source code change is predicted to break regression testing, we then identify these might-be-broken test cases. Specifically, two kinds of test cases will be recommended to developers: the first type is test cases that directly test classes, methods involved in the bug-fixing source code change; the second type is test cases that test other classes, methods that call the classes, methods involved in the bug-fixing source code change. We use the example program in Figure 2 to illustrate how *BFCP* identifies might-be-broken test cases.

Figure 2 (a) shows two versions of a small example project, which contains three classes: `A`, `B`, and `C`. Class B is extended from class `A`, and class `C` calls function `f1()` of class A. Here, the original version of the program consists of all program statements except the `if` block shown in dash boxes. The edited (for fixing bugs) version is generated by adding all the boxed code statements. As shown in Figure 2 (b), associated with the source code of example program is are three Junit regression test classes: `TestA`, `TestB`, and `TestC`. In class `TestA`, test case `test1()` is designed to test a regression error in function `f1()` of class `A`, and `test2()` in calss `TestB`, `test3()` in class `TestC` are designed to test regression bugs in `f2()` of class `B` and `f3()` of class `C`, respectively. In this

example program, these regression test cases will be used with both the original and edited versions of the program.

We assume that, the edited version fixes a revealed bug by adding an `if` block. Using our prediction model, if these changes in edited version are predicted as *Buggy*, in other words, these changes will break regression testing.

To identifying these might-be-broken test cases, two kinds of static call graphs are generated from the original version of example program. As shown in Figure 2 (c), we first generate static call graphs based on the relation between test code and source code to find test cases that directly test the changed program statements. In this example, by analyzing this static call graph, we could identify that test cases `test1()` in `TestA` is directly test function `f1()` in class `A`.

The other kind of static call graph we generated is between source code and source code. As shown in the left dash box in Figure 2 (c), function `f1()` of class `A` is called by function `f2()` in class `B` and function `f3()` in class `C`. The reason why we consider this kind of call graph is that, when a change is made in class `A`, since class `B` and class `C` also call class `A`, if the semantic of class `A` is changed, the semantic of class `B` and `C` can also be changed ,so test cases that test functions in class `B` and `C` could also be broken when running regression test cases.

With the two kinds of static call graphs, we can leverage Algorithm 1: *TestCaseIdentification* to identify these might-be broken test cases. In the example program, for function `f1()`in edited version of class `A`, the might-be-broken test cases are `test1()` in class `TestA`, `test2()` in class `TestB`, and `test3()` in class `TestC`.

IV. Experiment Setup

A. Data Collection

In order to evaluate our approach, we collected data from four open source projects, i.e., Ant, Log4j, Lucene, and Hadoop. We randomly selected five versions data for Ant and Log4j, and six versions data for Lucene. For Hadoop, we selected two versions data. In total, in our experiment we examined 552 bugs, among which we can find 448 valide bug-fixing source code changes. Also, for each change, we collected the regression test suite just before the change was submitted. Note that, in a bug-fixing source code change, some resource or configuration files can also be changed, in this work, we did not explore the impact of these changes on regression test suite. We only focus on source code changes. This is why the number of bug-to-changes is less than that of fixed bugs. Details of our dataset are shown in Table III.

Algorithm 1 *TestCaseIdentification*(*C*, *CG*, *R*, *RCG*)
Input: changed class *C* and its corresponding call graph *CG*, regression test suite *R*, call graph set of regression test cases *RCG*.
Output: A set of might-be-broken test cases *T*.

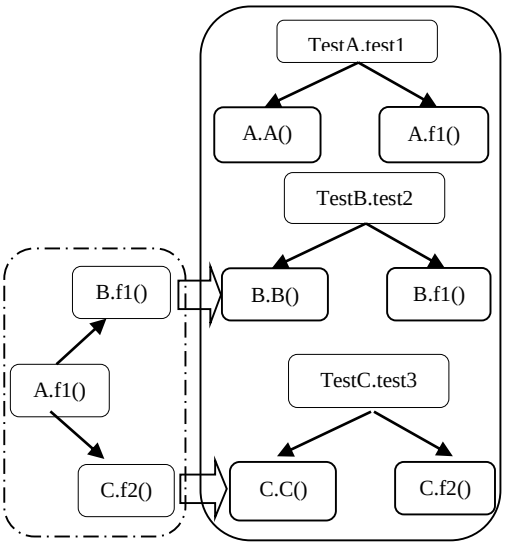
```
1  T = NULL;
2  for test t in R do
3     if C in RCG.getCalledClass(t) then
4        T.add(t);
5     endif
6     for class c in CG.getCalledClass(C) do
7        CG= generateCG(c); // generate call graph for c
8        TestCaseIdentification (c, CG, R, RCG);
9     end for;
10 end for;
11 return T;
```

```
Class A{
public f1(){
+if(){
+ ...
+ }
 }
}

Class B extends A{
pubic f1(){}
}

Class C{
public f2(){
A a = new A();
a.f1();
  }
}
```

```
Class TestA{
public test1(){
A a = new A();
a.f1();
Assert.assertTure(expression);
  }
}
Class TestB{
public test2(){
B b = new B();
b.f1();
Assert.assertTure(expression);
  }
}
Class TestC{
public test3(){
C c = new C();
c.f2();
Assert.assertTure(expression);
  }
}
```



(a) (b) (c)

Figure. 2. (a) Original and edited versions of example project. The edited version is generated by adding all boxed code statements. (b) Test cases associated with the example project. (c) Static call graphs based on the relation between source code and source code and the relation between test code and source code.

TABLE III. Details of our data used in our work. LOC is line of code. Bug-to-changes are the valid bug-fixing source code changes. Buggy changes are Buggy changes are the changes which can break regression testing.

Project	Version	Avg. LOC	#fixed bugs	#bug-to-changes	#clean changes	#buggy changes	Avg. regression test suite size
Ant	1.8.0RC1, 1.8.2, 1.8.4, 1.9.0, 1.9.2	230K	68	59	43	16	225
Log4j	1.2.13, 1.2.14, 1.2.15, 1.2.16, 1.2.17	110K	173	145	91	52	390
Lucene	4.4.0, 4.5.0, 4.6.1, 4.7.0, 4.7.1, 4.8.0	293K	89	88	16	72	524
Hadoop	2.1.1, 2.4.1	596K	222	156	48	108	1,518

B. Experiment Context

We ran all the experiments on a PC with a 2.8GHz CPU and an 8GB RAM. As described in Section III D, in our experiemnt, we sort bug-fixing changes in chronological order and divide them into four folds, each fold has the same number of Buggy changes, then we use data from the first three folds to train our models, and test the models on the data from the last fold. For obtaining better prediction performace, we employ Support Vector Machine (SVM), Naive Bayes (NB), Alternating Decision Tree (ADTree), and Logistic Regression (Logistic).

C. Evaluation Measures

In this work, the performance of predicting the impact of changes on regression test suite is measured in terms of Precision, Recall and F-measure, which are widely used to evaluate the performance of information retrieval approaches [37]. Our experiments can lead to four kinds of results: a change that we identified is true breaking regression testing (***TP***), a change we identified is not true breaking regression testing (***FP***), a change we predicted will not break regression testing is actual a ture change that can break regression testing (***FN***), and a change we predicted will break regression testing is actual a change that can not break regression testing (***TN***). The definitions of these metrics are as follows:

Precision: $\frac{TP}{TP+FP}$

Recall: $\frac{TP}{TP+FN}$

F-Measure: The harmonic mean of precision and recall, i.e., $2 \times \text{Precision} \times \text{Recall}/(\text{Precision} + \text{Recall})$.

To measure the performance of our proposed test cases identification approach, we only focus on **Recall**, because the purpose of identifying might-be-broken test cases in our approach is to help developers to find out all might-be-broken test cases and reduce the number of test cases might be further examined when a bug-fixing source code change break regression testing. Thus, a high prediction recall is crucial for the adoption of *BFCP* in practice. The recall for measure our might-be-broken test cases identification approach is calculated as follows:

Recall: $\frac{\#\text{test cases we identify is true that will be broken}}{\#\text{ground truth test cases that will be broken}}$

V. Results and Analysis

This section presents the experimental results. We focus on discussing the performance of our proposed approach and answering the following research questions (RQ):

A. RQ1: Is there any irrelevant metric among the 18 metrics we proposed?

To build an efficient prediction model, we collected all 18 metrics, however most of these metrics are not designed for predicting the impact of bug-fixing changes on software regression testing.

For answering RQ2, we employed *BestFirst* feature selection algorithm which built in Weka [37] and also we conducted the following experiments. As we presented in Section III D, for each of the four larger projects, we divided data into four folds and each fold has the same number of *Buggy* changes. We try to build prediction model using data of one fold and test it on the next fold of data. So for each project, we conducted three prediction experiments, that is: trained on Fold1 and tested on Fold2, trained on Fold2 and tested on Fold3, trained on Fold3 and tested on Fold4. In total, we have 12 different experiments.

For each of the prediction experiments, we conducted *BestFirst* feature selection algorithm to find the set of metrics that achieve the best performance. If there has any metrics that is selected zero time, then we can say that this metric is irrelevant of our models. The results are shown in Figure 3.

From Figure 3, we see that all 18 metrics are selected at least once in all projects. A metric that is not selected for one fold may be selected for another fold. Furthermore, a metric such as metric f3 (LF) that is less important in one project, such as Ant, may be more important in another project, such as Lucene. Therefore, the answer to RQ1 is all metrics appear to be relevant for the prediction model.

B. RQ2: Among the three categories of metrics, which category performs best?

As we presented in RQ1, all proposed 18 metrics are relevant for the prediction model. However, a further fine-grained question about the performace of our proposed metrics is: among the three categories of metrics, which performs best?

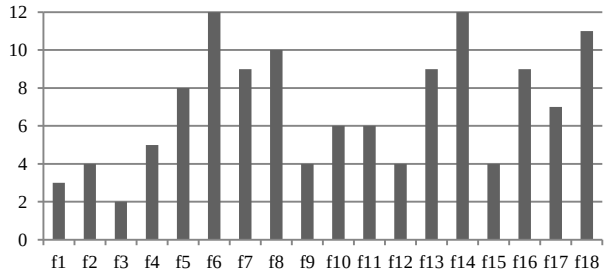
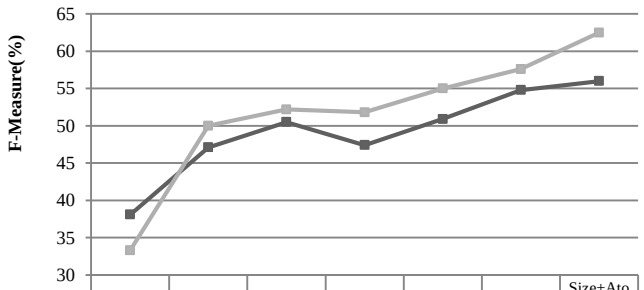
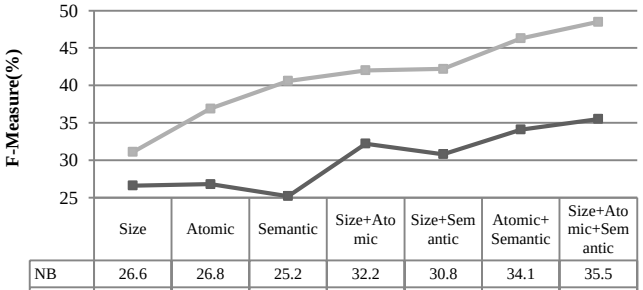


Figure. 3. Feature selection histogram.



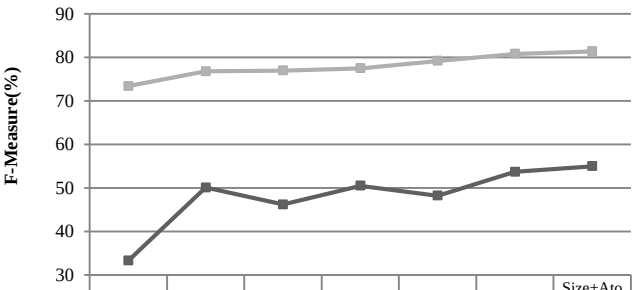
	Size	Atomic	Semantic	Size+Atomic	Size+Semantic	Atomic+Semantic	Size+Atomic+Semantic
NB	38.1	47.1	50.5	47.4	50.9	54.8	56
Logistic	33.3	50	52.2	51.8	55	57.6	62.5

Figure. 4. Comparison of different metrics for Ant.



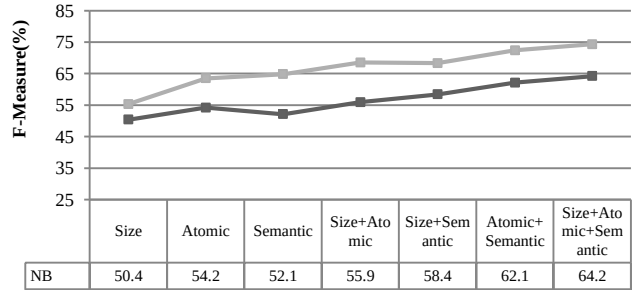
	Size	Atomic	Semantic	Size+Atomic	Size+Semantic	Atomic+Semantic	Size+Atomic+Semantic
NB	26.6	26.8	25.2	32.2	30.8	34.1	35.5
Logistic	31.1	36.9	40.6	42	42.2	46.3	48.5

Figure. 5. Comparison of different metrics for Log4j.



	Size	Atomic	Semantic	Size+Atomic	Size+Semantic	Atomic+Semantic	Size+Atomic+Semantic
NB	33.3	50.1	46.2	50.5	48.2	53.7	55
Logistic	73.4	76.8	77	77.5	79.2	80.8	81.4

Figure. 6. Comparison of different metrics for Lucene.



	Size	Atomic	Semantic	Size+Atomic	Size+Semantic	Atomic+Semantic	Size+Atomic+Semantic
NB	50.4	54.2	52.1	55.9	58.4	62.1	64.2
Logistic	55.3	63.5	64.8	68.5	68.3	72.4	74.3

Figure. 7. Comparison of different metrics for Hadoop.

For predicting the impact of bug-fixing source code changes on regression test suite, 18 unique metrics from three types are used. In this RQ, we try to further explore the performances different combinations among these three types of metrics. Specifically, we conducted seven groups of experiments with different metrics: **Size**, **Atomic**, **Semantic**, and **Size** + **Atomic**, **Size** + **Semantic**, **Atomic** + **Semantic**, and **Size** + **Atomic** + **Semantic**. We used data from the four open source projects shown in Table Ⅲ. Since we are not to find the best machine learning classifier in this RQ, we only use Naïve Bayes and Logistic to build prediction models, which are reported had good performance in change impact analysis studies [3]. For comparing the performance, we use F-Measure.

Figures 4, 5, 6, and 7 present the comparison results. To answer RQ2, in all of the four projects, **Atomic** and **Semantic** metrics outperform metrics of **Size**. This is reasonable since metrics of **Size** can only capture statistical information of changes, and metrics of **Atomic** and **Semantic** can obtain semantic information, which is more likely to reveal the root causes that why the changes break regression test cases. Further, from Figures 4 to 7, we can also find that our prediction models will achieve their best performance, when all three categories of metrics are used.

C. RQ3: What prediction performance can BFCP achieve?

In our RQ1 and RQ2, we shown that all proposed 18 metrics are relevant with our proposed prediction models, and the combination of all these 18 metrics might achieve the best performance. In this RQ, we further examined the classification performance of our proposed approach.

We examined the four types of machine learning classifiers described in Section IV B. Results are shown in Table Ⅳ.

As we can see from Table Ⅳ, among the four classifiers, Logistic regression outperforms the other three on at least one measure in Precision, Recall, F-Measure. *BFCP* could achieve prediction precision up to 83.3% and recall up to 92.3%.

D. RQ4:What test case identification performance can BFCP achieve?

We further examined the performance of our test cases identification approach, note that, as presented in Section IV C, we only use recall to measure our test cases identification approach to help developers to find might-be-broken test cases.

For test cases identification, *BFCP* can achieve 100% recall, which means we could find all might-be-broken test cases. Further, in Table Ⅳ, we presented the average number of identified might-be-broken test cases, for all the four projects, the average number of identified test cases are less than 20, regard to the the size of corresponding test cases, *BFCP* could reduce the total nubmer of needed reviewed test cases from hundreds to less than 20, which are needed to be further reviewed.

TABLE IV. Prediction results. P represents for Precision, R represent for Recall, and F1 represents for F1-measure. TCI is represent for test cases Identification.

Project	Algorithm	Changes prediction (%)			R of TCI (%)	#Avg. TCI
		P	*R*	*F1*		
Ant	SVM	75.0	12.5	21.4	100	10.3
	Naive Bayes	46.7	70.0	56.0		
	ADTree	55.6	20.8	30.3		
	Logistic	**83.3**	50.0	**62.5**		
Log4j	SVM	60.0	22.1	32.1	100	15.5
	Naive Bayes	54.5	28.6	37.5		
	ADTree	41.2	33.3	36.8		
	Logistic	**66.7**	**38.1**	**48.5**		
Lucene	SVM	73.3	84.6	78.6	100	18.2
	Naive Bayes	78.6	42.3	55.0		
	ADTree	71.4	75.0	73.2		
	Logistic	72.7	**92.3**	**81.4**		
Hadoop	SVM	69.2	85.9	71.9	100	19.5
	Naive Bayes	94.4	48.6	64.2		
	ADTree	91.3	60.0	72.4		
	Logistic	**74.3**	74.3	**74.3**		

TABLE V. Time consuming of *BFCP* (s: second).

Project	Model building and Prediction time (avg)	Test case identification time (avg)
Ant	< 0.06s	15.8s
Log4j	< 0.05s	28.0s
Lucene	< 0.1s	40.2s
Hadoop	< 0.2s	98.6s

VI. DISCUSSION

As presented in our work, given a new source code change, *BFCP* will perform two tasks to provide early information before they actually run the time-consuming regression testing. However, the time cost of *BFCP* will be a concern for applying it into practice. In order to evaluate the time cost of *BFCP*, we collected the time cost for building prediction model and identifying might-be-broken test cases for the four projects, and the average time is shown in Table Ⅴ.

From Table Ⅴ, we can see that the model building and predicting time varies from less than 0.06 seconds to about 0.2 seconds. The results suggest that the tool is efficient enough to be used in software practice. Moreover, most of the time is consumed on identifying might-be-broken test cases. As the project size increases, this time increases approximately linearly, thus *BFCP* should scale to large projects.

VII. RELATED WORK

A. Change impact analysis

Change impact analysis which aims to predict the ripple effects and prevent side effects of a source code change has been widely studied. Kim et al. [1] proposed a technique for predicting latent bugs in software source code changes using machine learning classifiers based on features from software project change history. Wloka et al. [3] presented an analysis-based technique for determining changes that can be committed without compromising the integrity of the repository. Stoerzer et al. [4] explored how change classification can focus programmer attention on failure-inducing changes by automatically labeling changes Red, Yellow, or Green, indicating the likelihood that they have contributed to a test failure. Sukkerd et al. [8] conducted empirical studies to understand regression failures by exploring test-passing and test-failing code changes. Torchiano et al. [10] used the information available in software repositories, in particular code comments and version control logs to study impact analysis. Ahsan et al. [11] used resolved software change requests to predict the files that have to be changed. Jiang et al. [31] and Tan et al. [30] leveraged machine learning approaches to predict whether a change is buggy at the time of the commit. Orso et al. [34] presented an empirical comparison of dynamic impact analysis techniques.

B. Software code co-change

Zaidman et al. [33] investigated whether software source code and the accompanying tests co-evolved by mining projects' version system. Their results show that co-evolution between software source code and test code is sometimes not optimal for many projects. Zimmermann et al. [26] build an annotation graph based upon the identification of lines across several versions of a file to identify co-changing lines. Wu et al. [27] visualized the co-evolution of software source code using spectrographs. Fluri et al. [28] examined whether source code and associated comments are changed together alongside the evolutionary history of a software system. Godfrey et al. [29] used source code metrics to characterize the evolution of software projects to investigate whether open source software and commercial software have different change patterns. Shane et al. [32] mined the source and test code changes that required accompanying build changes to better understand the co-change relationship. Tao et al. [19] conducted an empirical study to explore how developers understand code changes. Pinto et al. [7] presented an extensive empirical study of how test suite evolve. Tian et al. [9] proposed an automated approach to infer commits that represent bug fixing patches.

Wloka et al. [18], and Hurdugaci et al. [20] developed tools that help developers to identify the unit tests that need to be altered and executed after a source code change using test cases coverage information. In this work, when a bug-fixing source code change is predicted that might break regression testing, we also further identify these might-be-broken test cases. Different with their work, we use static call graph analysis technique. The key advantage of our approach is that our approach finds not only test cases that directly test the changes, but also test cases that test other functions, classes that call the functions, and the classes in the changes.

C. Regression test case prioritization

Regression test case prioritization techniques aim to rearrange the execution order of test cases to maximize specific objectives. Many test case prioritization algorithms have been proposed and well-studied, such as coverage based algorithms [22, 24], which assign higher priorities to a test case that executes more statements in a program; fault-exposing

potential based test case prioritization algorithms [23, 25], which aim to sort test cases so that the rate of failure detection of the prioritized test cases can be maximized. Yoo et al. [42] surveyed regression testing minimization, selection and prioritization.

In this work, different from test case prioritization, we do not intend to sort all regression test cases, when a bug-fixing source code change is predicted that might break regression testing, we further identify these might-be-broken test cases.

VIII. Threats to Validity

A. External Validity

In this work we investigate the performance of proposed *BFCP* on bug-fixing source code changes of 4 large scale open source projects. However, it is possible that our approach may not work well on some closed-source software (e.g., commercial software) or small scale open source software projects. Whether our proposed approach is feasible for these software projects should be further investigated.

The purpose of this work is to study the impact of bug-fixing source code changes on regression test suite, however, not all projects maintain valid regression test suite. Our approach is not suitable for these projects without regression test cases.

B. Internal Validity

In this paper, we study the impact of bug-fixing source code changes on corresponding regression test suite. A list of 18 metrics that cover a wide range of change characteristics that would induce impact of bug-fixing source code changes are used. However, other metrics that we have overlooked may also improve the performance of our classifiers.

Our proposed approach could predict whether a bug-fixing source code change will break corresponding regression test cases. However, we cannot explain questions like: what causes the regression test cases broken? How to fix the broken test cases? Further studies should be done for answering these questions.

IX. Conclusion and Future work

This paper proposed *BFCP* to predict the impact of a bug-fixing source code change on corresponding regression testing before running regression test cases, by mining software source code change histories. Specifically, given a bug-fixing change, without executing regression test suite, our approach first predicts whether this bug-fixing change will break regression testing or not. Second, if the change is predicted not regression error free, *BFCP* can further identify the might-be-broken test cases.

We evaluate *BFCP* on 552 real bug-fixing source code changes from 18 release versions of 4 large open source projects. Results of experiments show that *BFCP* could achieve prediction precision up to 83.3% and recall up to 92.3%; for test case identification, *BFCP* could achieve 100% recall, and the average number of identified test cases are less than 20.

We plan to extend our work from the following two aspects: first we plan to generate explainable prediction results, i.e., automatically finding root reason that why a bug-fixing source code change will break regression testing. Second, after identifying might-be-broken test cases, we plan to automatically generate fix patchs for fixing buggy changes.

Acknowledgment

This research was supported in part by National Natural Science Foundation of China under Grant Nos. 91218302, 91318301, 71101138, and 61303163.

References

[1] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering.* 34, 2, 181-196.

[2] B. Amiangshu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In Empirical Software Engineering and Measurement (ESEM ‘13), 2013 *ACM/IEEE International Symposium on*, pp. 133-142. IEEE, 2013.

[3] J. Wloka, B. Ryder, F. Tip, and X. Ren. 2009. Safe-commit analysis to facilitate team software development. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE’ 09), USA, 507-517.

[4] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. 2006. Finding failure-inducing changes in java programs using change classification. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (SIGSOFT '06/FSE-14). ACM, NY, USA, 57-68.

[5] D. Yuan, Y. Luo, X. Zhuang, G. Renna R., X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. 2014. Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (OSDI'14). USA, 249-265.

[6] W. R. Harris, G. Jin, S. Lu, and S. Jha. 2013. Validating library usage interactively. In *Proceedings of the 25th international conference on Computer Aided Verification* (CAV'13), Natasha Sharygina and Helmut Veith (Eds.). Springer-Verlag, Berlin, Heidelberg, 796-812.

[7] L. S. Pinto, S. Sinha, and A. Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proc. of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (FSE '12). ACM, Article 33, 11 pages.

[8] R. Sukkerd, I. Beschastnikh, J. Wuttke, S. Zhang, and Y. Brun. 2013. Understanding regression failures through test-passing and test-failing code changes. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1177-1180.

[9] Y. Tian, J. Lawall, and D. Lo. 2012. Identifying Linux bug fixing patches. In*Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 386-396.

[10] M. Torchiano and F. Ricca. 2010. Impact analysis by means of unstructured knowledge in the context of bug repositories. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM '10). ACM, New York, USA, Article 47, 4 pages.

[11] S. N. Ahsan and F. Wotawa. 2010. Impact analysis of SCRs using single and multi-label machine learning classification. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM '10). ACM, New York, NY, USA, Article 51, 4 pages.

[12] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. 2004. Chianti: a tool for change impact analysis of java programs. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (OOPSLA '04). ACM, New York, USA, 432-448.

[13] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. on Programming Language and Systems*, 23(6):685–746, 2001.

[14] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated inclusion constraints. *In Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems*, pages 43–55, October 2001.

[15] G. Shu, B. Sun, T. A.D. Henderson, A. Podgurski. JavaPDG: A New Platform for Program Dependence Analysis. *In Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation, Testing Tools Track*, Luxembourg, March18-22, 2013.

[16] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. 2010. Matching dependence-related queries in the system dependence graph. *In Proceedings of the IEEE/ACM international conference on Automated Software engineering* (ASE '10). ACM, New York, NY, USA, 457-466.

[17] S. Lee; S. Kang; S. Kim; Staats, M. The Impact of View Histories on Edit Recommendations. *Software Engineering, IEEE Transactions on*, vol.41, no.3, pp.314,330, March 1 2015.

[18] J. Wloka, B. G. Ryder, and F. Tip. 2009. JUnitMX - A change-aware unit testing tool. In *Proceedings of the 31st International Conference on Software Engineering* (ICSE '09). IEEE Computer Society, Washington, DC, USA, 567-570.

[19] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (FSE '12). ACM, NY, USA, Article 51, 11 pages.

[20] V. Hurdugaci and A. Zaidman. "Aiding software developers to maintain developer tests." *In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 11-20. IEEE, 2012.

[21] R. Wu, H. Zhang, S. Kim, and S. Cheung. 2011. ReLink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (ESEC/FSE '11). ACM, New York, NY, USA, 15-25.

[22] R. Gregg and M. J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on* 22, no. 8 (1996): 529-551.

[23] G. Rothermel, R.H. Untch, and M. J. Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Software Engineering*, Oct 2001, v27, n10, pp.929-948.

[24] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. 2009. Adaptive Random Test Case Prioritization. *In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (ASE '09). Washington, DC, USA, 233-244.

[25] S. Yoo and M. Harman, "*Regression testing minimization, selection and prioritization: a survey*," Software Testing, Verification & Reliability 2012, 22(2), 67-120.

[26] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr.. 2006. Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories* (MSR '06). ACM, USA, 72-75.

[27] J. Wu, H. R.C., H. A.E., Exploring software evolution using spectrographs. Reverse Engineering, 2004. *Proceedings. 11th Working Conference on*, vol., no., pp.80, 89, 8-12.

[28] B. Fluri, M. Wursch, H.C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp.70-79, 28-31.

[29] M. Godfrey and Q. Tu. Evolution in open source software: A case study. *In Proc. of the Int'l Conf. on Software Maintenance* (ICSM '00), pages 131–142. IEEE, 2000.

[30] M. Tan, L. Tan, S. Dara, C. Mayeux. Online Defect Prediction for Imbalanced Data *In Proceedings of the 2015 International Conference on Software Engineering* (ICSE '15).

[31] T. Jiang, L. Tan, S. Kim. Personalized defect prediction. *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp.279-289, Nov. 2013.

[32] M. Shane, B. Adams, M. Nagappan, and A. E. Hassan. Mining Co-Change Information to Understand when Build Changes are Necessary. *In Software Maintenance and Evolution (ICSME), IEEE International Conference on*, pp. 241-250. IEEE, 2014.

[33] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. V. Deursen Mining software repositories to study co-evolution of production & test code. *In Software Testing, Verification, and Validation (ICST), on*, pp. 220-229. IEEE, 2008.

[34] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. *In Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, pp. 491–500.

[35] K. Herzig, S. Just, and A. Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, USA, 392-401.

[36] S. Wang, W. Zhang, Y. Yang, and Q. Wang. DevNet: Exploring Developer Collaboration in Heterogeneous Networks of Bug Repositories. Empirical Software Engineering and Measurement. *ACM / IEEE International Symposium on*, Oct. 2013.

[37] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. SIGKDD'09, vol. 11, no. 1, pp. 10–18.

[38] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.* 17, 4 (December 2010), 375-407.

[39] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (PASTE '01). ACM, USA, 46-53.

[40] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering* (ICSE '11). ACM, NY, USA, 746-755.

[41] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, NY, USA, 1047-1058.

[42] Yoo, Shin and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." Software Testing, Verification and Reliability 22.2 (2012): 67-120.