Test Case Prioritization based on Analysis of Program Structure

Zengkai Ma
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
nicokay@sjtu.edu.cn

Jianjun Zhao
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
zhao-jj@sjtu.edu.cn

Abstract

Test case prioritization techniques have been empirically proved to be effective in improving the rate of fault detection in regression testing. However, most of previous techniques assume that all the faults have equal severity, which dose not meet the practice. In addition, because most of the existing techniques rely on the information gained from previous execution of test cases or source code changes, few of them can be directly applied to non-regression testing. In this paper, aiming to improve the rate of severe faults detection for both regression testing and non-regression testing, we propose a novel test case prioritization approach based on the analysis of program structure. The key idea of our approach is the evaluation of testing-importance for each module (e.g., method) covered by test cases. As a proof of concept, we implement Apros, a test case prioritization tool, and perform an empirical study on two real, non-trivial Java programs. The experimental result represents that our approach could be a promising solution to improve the rate of severe faults detection.

1. Introduction

Software testing plays an important role in software development, especially when the software system becomes complicated and large-scale today. In order to improve the cost-effectiveness of test activities, testing techniques such as test suite reduction [12] and test case prioritization [19] have been proposed. Test suite reduction technique is used to find the redundant test cases and remove them from test suite, so that engineer can spend less time and resource running the minimized test suite. Other than deleting needless test cases, the main purpose of test case prioritization is to rank test cases execution order so as to detect fault as early as possible. There are two benefits brought by prioritization technique. First, it provides a way to find more bugs under resource constraint condition and thus improves the reliability of the system quickly. Second, because faults are revealed earlier, engineers have more time to fix these bugs and adjust the project schedule. Here, we focus on the test case prioritization technique.

Just as several empirical studies [3,6,15,19] have shown, most of existing test case prioritization techniques are effective to improve the rate of faults detection for regression testing. However, so far the proposed prioritization techniques are mainly based on the data collected during previous testing of the software or difference between the various software versions. Therefore, they are hard to be applied to non-regression testing where neither previous execution data of test cases nor code difference information is available. Although some techniques [17, 18] attempt to combine multiple information, such as user knowledge and requirements, to prioritize test cases, they fail to consider the fault severity from the viewpoint of program structure. Therefore, the faults with serious threat to system may not be detected early.

In this paper, we propose a new prioritization index called testing-importance of module (TIM), which combines two prioritization factors: fault proneness and importance of module. TIM is a metric, which can be used to measure the severe fault proneness for module covered by test case. By computing the TIM value of covered modules, the ordering of test cases can be determined (the test case with high TIM value has high priority). The main advantages of this prioritization approach are twofold. First, the TIM value can be evaluated by analyzing program structure (e.g., call graph) alone and it also can be evaluated by incorporating program structure information and other available data (e.g., source code changes). Therefore, this approach can be applied to not only regression testing but also non-regression testing. Second, through analyzing program structure, we can build a mapping between fault severity and fault location. Those test cases covering important part of system will be assigned high priority and executed first. As a result, the severe faults are revealed earlier and the system becomes reliable at fast rate. The main contributions of this paper are:

- We propose a new approach to evaluate the testing-importance for modules in system by combining analysis of fault proneness and module importance.
- We develop a test case prioritization technique which can provide test cases priority result by handling multiple information (e.g., program structure information, source code changes) and can be applied to both new developed software testing and regression testing.
- We implement *Apros*, a tool for test case prioritization based on the proposed technique, and perform an experimental study on our approach. The result suggests that *Apros* is a promising solution to improve the rate of severe faults detection.

The rest of this paper is organized as follows. In Section 2, we briefly introduce the background knowledge of test case prioritization techniques and metrics; Section 3 gives an intuition of our approach by an example and Section 4 describes this approach in details; Section 5 presents the experimental study and discusses the results; Section 6 discusses related work and finally Section 7 gives the conclusions and future work.

2. Background

Before a software release, it should be sure that most faults of this software have been found and fixed. An appropriate test execution sequence can reveal faults early and thus leave more time for engineers to solve these problems. The general purpose of test case prioritization is to meet some objectives such as: maximizing the rate of fault detection, achieving the target coverage level as early as possible. Formal definition of this problem has been given by Rothermel et al as follows [16]:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ *such that* $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Here, PT is the set of all possible orders of T, and f is an objective function that, applied to any such order, yields an award value for that order.

The goal for most of existing test case prioritization techniques is to increase the rate of fault detection of test suites. To quantify this goal, Rothermel [15] introduced a metric APFD, which can be used to measure the weighted average of the percentage of faults detected over the life of a test suite. The APFD values for test suites can be computed according to the following Equation (1):

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \qquad (1)$$

where n is the number of test cases, m is the number of exposed faults. TF_i is the position of the first test case which reveals the fault i in the ordered test cases sequence. The prioritized test suite with fast fault detection rate has higher APFD value than those with low fault detection rate.

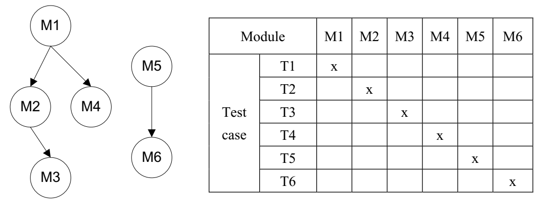
The limitation of using APFD metric is that this metric assumes that all faults have equal severity and test cases have equal costs which does not meet the practice. Therefore, an improved metric called APFDc [5] has been proposed for taking into account the fault severity and test cost. The value of APFDc can be computed according to the following Equation (2):

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \qquad (2)$$

where n is the number of test cases and t_i represents the cost of test case i, m is the number of revealed faults and f_i represents the severity of fault i. TF_i is the position of the first test case which reveals the fault i in the ordered test cases sequence. This formula can also be used to compute the APFD value if both fault severity and test case costs are identical.

3. Motivating Example

As shown in Figure 1, consider a sample system which consists of six modules: M1-M6 and there exist some call relationships between each module. A test suite includes six test cases T1-T6 which covers the M1-M6 respectively.



Module		M1	M2	M3	M4	M5	M6
Test case	T1	x					
	T2		x				
	T3			x			
	T4				x		
	T5					x	
	T6						x

Figure 1. A sample system and coverage information for test cases

In order to find the severe faults as early as possible, we should prioritize the test cases. Since here is not a regression testing phase, those prioritization techniques which rely on the information only available in such kind of testing phase can not work here. In addition, because each test case covers the same number of modules, we also can not identify the priority of test cases in terms of number of covered modules. Therefore, for improving the rate of severe fault detection for this test suite, another prioritization index should be used here.

The possibility of a test case revealing severe faults is not only related to the number of modules covered by it but also related to the character of those covered modules. Intuitively, the test cases covering high fault proneness modules are more likely to reveal faults than those covering low fault proneness modules. We can rank the test cases according to the fault proneness of modules covered by them. However, estimating the fault proneness for modules is often difficult. When such a kind of rank index is unavailable, we should find another prioritization factor. Because the faults in key modules are more severe than those in other modules, the severities of faults are various in different parts of programs. For instance, in this sample system, the module M3 provides service for module M2 and M2 provides service for M1. Therefore, the fault in M3 potentially threatens those three modules: M3, M2 and M1. Even though the information about fault proneness of modules is unavailable here, we believe that the test case T3 covering module M3 should be executed before T1 and T2. In order to combine above two prioritization factors (fault proneness, fault severity), we propose the $testing$-$importance$ metric **TIM** to measure the possibility of a module containing severe faults. And then we can prioritize the test cases in decreasing order of the summation of **TIM** value of modules covered by them. For this sample system, our approach can produce the prioritization result (T3,T6,T4,T2,T5,T1) by analyzing the structure of system.

4. Our Approach

4.1. General Process

Our approach generally consists of three processes, as shown in Figure 2.

- **Evaluating TIM for Modules**: in this process, $Apros$ evaluates the fault proneness (**FP**) and importance (**IM**) for each module by incorporating analysis of program structure and other available information (e.g., user requirements, source code changes, previous execution data of test cases). Then the **FP** and **IM** are combined to get **TIM**.
- **Analyzing Test Case Coverage**: in this process, $Apros$ analyzes the program and test suite to obtain the coverage information for each test case.
- **Identifying Test Case priority**: at last, through computing the **TP** value (summation of **TIM** value of covered modules) for each test case, the sequence of prioritized test cases can be created.

4.2. Evaluating TIM for Module

In this paper, we use **TIM** to measure the possibility of containing severe faults for module. Its value can be computed by the following Equation (3):

$$TIM = FP \times W_{fp} + IM \times W_{im} \tag{3}$$

where the FP represents the fault proneness of this module and the W_{fp} is the weight for the FP. The IM indicates

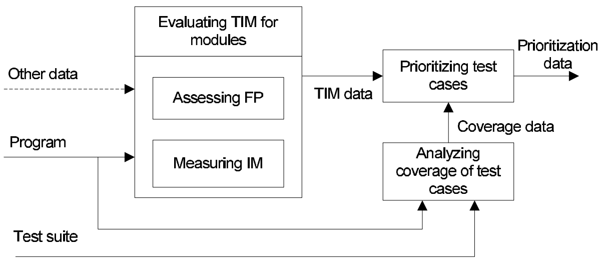


Figure 2. General processes of our approach

the importance of this module and the W_{im} is the weight for the IM. The weights W_{fp} and W_{im} are determined according to the context of testing. For instance, when the fault proneness is hard to estimate, the W_{fp} can be set to a smaller value. In contrast, if the fault proneness is proved to be effective such as the source code changes in the context of regression testing, then the W_{fp} can be set to a bigger value. A default value can be assigned, which gives each factor equal weight.

4.2.1 Fault Proneness of Module

As a factor of **TIM**, **FP** represents the possibility of a module containing faults. Some kinds of information are often used to estimate this property in previous research such as: complexity of requirement implementation [17] and source code modifications. Let there are n metrics used to calculate the value of **FP** for module, and the value of metric i is expressed by M_i and the weight for this metric is expressed by W_i. **FP** can be computed according to Equation (4). A default value can be assigned in the case of giving each factor equal weight.

$$FP = \sum_{i=1}^{n} W_i \times M_i \tag{4}$$

4.2.2 Importance of Module

The importance of a module should be considered from two viewpoints: the importance for user (**IU**) and the importance for system (**IS**). The fault which is related to the important module for user is more severe than the fault related to the other modules. And also the fault with serious threat to system is more severe than the fault with low impact on system. The value of **IM** of module can be calculated by Equation (5).

$$IM = IU \times W_{iu} + IS \times W_{is} \tag{5}$$

where the W_{iu} is the weight for the IU, and the W_{is} is weight for the IS. Similar to Equation (3), the weight can be adjusted according to the context of testing and also can

```
EIS Algorithm
Input: MCG(M, E)
Output: R: set of modules with updated $IS$ value
Declare: NM: set of modules which have no incoming edge
Begin EIS
1.  select the $NM$ from $M$
2.  while $NM$ is not empty do
3.    for each $m \in NM$
4.      let $isv$ be the $IS$ value of $m$
5.      let $od$ be the $out\text{-}degree$ of $m$
6.      $isvplus \leftarrow isv/(od+1)$
7.      $isv \leftarrow isvplus$
8.      move $m$ from $M$ to $R$
9.      for each $e \in outgoing\ edges\ of\ m$
10.        let $isv$ be $IS$ value of the destination module of $e$
11.        $isv \leftarrow isv + isvplus$
12.        remove $e$ from $E$
13.      end for
14.    end for
15.    select the $NM$ from $M$
16.  end while
17.  if $M$ is not empty then
18.    move $\forall m \in M$ to $R$
19.  end if
20.  return $R$
End EIS
```

Figure 3. EIS Algorithm for computing IS value for modules.

be assigned a default equal value. The **IU** can be determined by user according to the requirements but **IS** should be evaluated through analyzing the program structure. We proposed the EIS algorithm to compute **IS** value for modules in system.

4.2.3 EIS algorithm

We use EIS algorithm given in Figure 3 to evaluate **IS** for each module. The EIS algorithm takes as input a module call graph (MCG) which consists of a set of modules (M) and a set of call edges (E). At the beginning, the **IS** value of each modules in M has been initialized to the equal value 1. At the end, the algorithm outputs a set of modules with updated **IS** value.

Since the call relationships between modules exist, the impact of a fault might not be confined to the internal area of a module. The threat of a fault in module will back propagate along the call edge. The faults in callees potentially impact on callers but the faults in callers cannot affect callees. Therefore, the callees should be tested earlier than callers. By processing of the EIS algorithm, the modules asking for other modules' service should share its **IS** value with those service providers and any modules providing service can get the plus **IS** value from the service recipients. As a result, the callers generally have smaller **IS** value than those callees. However, we do not evaluate the **IS** for a module by simply using the number of its ancestors (NAM) in MCG, but through EIS algorithm. The Table 1 describes the **IS** value, the ancestors and the NAM for each module in the sample system showed by Figure 1.

Following analysis explains the reason of using EIS algorithm instead of using NAM.

Module	IS	NAM	Ancestors
M1	0.33	0	
M2	0.66	1	M1
M3	1.66	2	M1,M2
M4	1.33	1	M1
M5	0.5	0	
M6	1.5	1	M5

Table 1. The IS, NAM and Ancestors.

We assume that a module is cleared only if the test cases, which cover this module, have been executed and the revealed faults have been fixed, and also all the modules called by it are cleared. Suppose that M1 and M5 have been tested and the revealed faults have been fixed. Considering M1 and M5 call the untested modules, the set of cleared modules in this system is empty yet. However, if we then choose one test case from the untested module set (T2, T4 and T6) to execute, the set of cleared modules would depend on the chosen test case as following described.

- **Choosing to run T2.** Because M2 needs service provided by untested module M3, fixing the faults in M2 cannot guarantee that M2 is cleared. Therefore, the set of cleared modules is still empty even if the T2 has been executed and the revealed faults have been fixed.
- **Choosing to run T4.** M4 does not call any other modules, and thus when the faults in M4 are fixed, M4 is cleared.
- **Choosing to run T6.** Similar to M4, M6 would not be affected by the faults in other modules. In addition, because M6 is the only service provider for M5, if the faults in M6 are fixed, the M5 can be considered cleared. Therefore, M5 and M6 are the cleared modules.

From above analysis and Table 1, we can find that the **IS** value indicates the testing priority for modules, even if their NAM values are the same. Considering the major process of EIS algorithm is similar to topological sorting, the time complexity of EIS can be $O(m{+}e)$.

4.3. Evaluating Test Case Priority

We evaluate the test case priority according to its **TP** value. After obtaining the modules coverage information, we use Equation (6) to compute the **TP** value for each test case.

$$TP_j = \sum_{i=1}^{m} TIM_i \tag{6}$$

where the TP_j represents the TP value of test case j, the TIM_i represents the TIM value of module i, and m is the number of the modules covered by test case j. Then, the test cases are prioritized by decreasing the value of **TP** for each test case.

5. Experiment and Result Analysis

To investigate the effectiveness of our approach, we have implemented $Apros$, a test case prioritization tool. In our implementation, we analyze the call graph of system and test suite to compute the **TIM** value for methods and identify the methods coverage for test cases, and then produce the prioritization results.

5.1. Experiment Setting

We apply our approach to two Java programs with JUnit test cases: $xml\text{-}security$ and $jtopas$. The two programs with seeded faults are obtained from the Subject Infrastructure Repository (SIR) [1] which provides the Java and C programs for experimentation of source code analysis and testing techniques.

Object	Classes	Test Classes	Test Methods	Faults
jtopas-v1	19	10	126	9
jtopas-v2	21	11	128	9
jtopas-v3	50	18	209	17
xml-sec-v1	177	18	91	8
xml-sec-v2	192	18	94	14
xml-sec-v3	143	14	84	18

Table 2. Subject programs.

$xml\text{-}security$, including 16.8 KLOCs (the total number of lines of code, excluding comments), implements security standards for XML. $jtopas$, including 5.4 KLOCs, is a Java library used for parsing text data . Both programs have been used as benchmarks by several test case prioritization empirical studies [2, 3]. We select three sequential versions of the two programs described as Table 2 to simulate the newly developed software testing (NDS testing) and the regression testing. In this experiment, we regard a method as a module and a fault caused many methods fail to work as a severe fault. Therefore, we use the ratio of fault-affected methods to total methods of program to evaluate the severity of fault as described in Table 3. And the Table 4 shows the distribution of severe faults in our subject programs.

The information used in test cases prioritization can be various in different testing phases, for instance, in NDS testing, the information about source code changes is unavailable but such information is very useful for ranking test cases in regression testing. Therefore, in order to judge the effectiveness of proposed approach, we apply it to two kinds of test phases: NDS testing and regression testing. In NDS testing, we prioritize test cases based on the analysis of call graph of subject programs. The results have been compared

Fault Severity	Ratio of affected methods
1	$0 - 1\%$
2	$1\% - 2\%$
3	$2\% - 4\%$
4	$4\% - 8\%$
5	$8\% - 16\%$
6	$16\% - 32\%$
7	$32\% - 64\%$
8	$64\% - 100\%$

Table 3. Fault severity.

Object	Fault Severity							
	1	2	3	4	5	6	7	8
jtopas-v1	2	3	0	2	0	2	0	0
jtopas-v2	3	4	0	0	0	2	0	0
jtopas-v3	12	1	2	2	0	0	0	0
xml-sec-v1	0	0	7	1	0	0	0	0
xml-sec-v2	0	1	6	6	1	0	0	0
xml-sec-v3	4	1	5	8	0	0	0	0

Table 4. Distribution of severe faults.

against untreated, random, optimal prioritization techniques and also two widely used techniques (methods coverage and additional methods coverage) to verify whether $Apros$ performs better than other techniques and gets closed enough to optimal technique. In regression testing, we prioritize test cases based on incorporating the analysis of call graph of subject programs and the source code changes provided by Celadon [20]. The results also have been compared against untreated, random, optimal prioritization techniques and two widely used techniques in regression testing (changed methods coverage and additional changed methods coverage). There is no prior knowledge about the subject programs to help us determine the weights in Equations (1), (2) and (3). Therefore, we use default weights (equal value) in our experiment. Below is a brief description of the prioritization techniques compared with $Apros$.

- **Untreated (Unt):** The original ordering of test cases provided by the objects.
- **Random (Ran):** The test cases are ordered randomly.
- **Optimal (Opt):** The test cases are ordered to maximize the rate of severe fault detection. This kind of ordering is obtained by greedily selecting a next test case with the highest severity of faults revealed by it.
- **Total methods coverage (MT):** The test cases are ordered in terms of the total number of the methods covered by them. The test case with higher number of covered methods has higher priority.
- **Additional methods coverage (MA):** The test cases are prioritized by the number of covered methods which are not yet covered by those previous executed test cases. The test case with higher number of covered additional methods has higher priority.
- **Total different methods coverage (DMT):** The test cases are prioritized based on the number of covered

methods which differ from those methods in the preceding version (e.g., modified, added, deleted methods).

- **Additional different methods coverage (DMA):** This technique is exactly the same as the **MA** just described, except the covered methods are those differ from preceding versions.

We use the metric APFDc [5] to measure our technique and other proposed techniques in terms of the rate of severe faults detection. The higher APFDc value implies faster detection of severe faults. In addition, based on the assumption that a detected fault would be fixed, we introduce a metric called $average\ of\ the\ percentage\ of\ fault\text{-}affected\ modules\ cleared\ per\ test\ case$ (APMC). We use this metric to measure the effect of various prioritization techniques in terms of the rate of fault-affected methods clearance and the APMC value can be computed by Equation (7).

$$APMC = \frac{\sum_{i=1}^{n} CM_i}{nm} \tag{7}$$

where n is the number of test cases and m is the number of fault-affected modules. CM_i is the number of cleared modules after the test case i executed (assume that the faults detected by test case i will be fixed). APMC values range from 0 to 100% and the prioritization technique with higher APMC value indicates the ability of faster eliminating the impact of faults.

5.2. Result Analysis

Figure 4 shows the comparison of APFDc values obtained by various prioritization techniques on six versions of two Java programs. Compared with **Ran** and **Unt**, our approach significantly improved the rate of severe faults detection in both NDS testing and Regression testing. For example, we have achieved an average improvement of 45.9% over **Ran**, and 19.7% over **Unt**. *Apros* also performs better than two widely used techniques, 8.5% better than **MT**, 3.4% than **MA** on average in NDS testing and 15% better than **DMT**, 10% than **DMA** on average in regression testing.

Although among several prioritization techniques, our approach is always close to optimal prioritization, we observed that there is no large improvement in APFDc values when using our approach in NDS testing. We believe that the reason for this lies in the coarse granularity of JUnit test cases. In JUnit test framework, a test case class is often regarded as a test case and a class consists of several test methods which cover some code area to be tested. In *Apros*, the **TP** value of a test class is the accumulation of the **TP** value of test methods, which belong to this test class. Therefore, although some test methods have been assigned a high **TP** value to indicate the bigger likelihood to

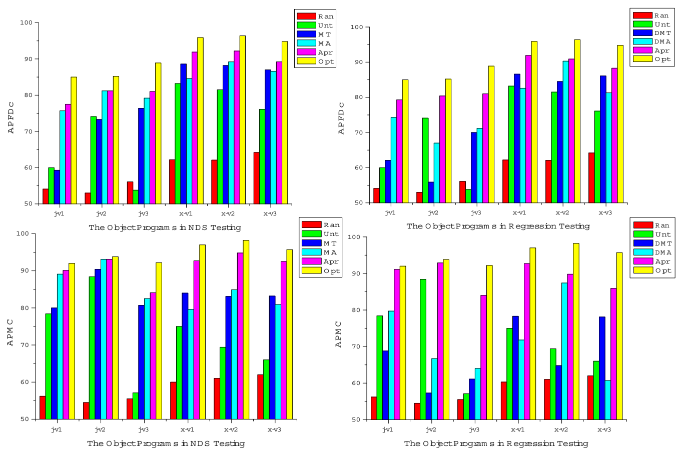


Figure 4. APFDc values and APMC values for *Apros*, compared with untreated, random, optimal values, MT, MA, DMT, DMA values in simulated NDS testing and regression testing.

reveal severe faults, in the same test class, the other test methods with lower **TP** value counteract this identity. As a result, compared with test method level, the indication ability of **TP** value is weakened to some extent in class level. As shown in Table 2, the *xml-security* programs have more test classes and less test methods than *jtopas* programs, in the other word, the granularity of test suites in *xml-security* is smaller than the granularity of test cases in *jtopas*. And as shown in Figure 4, the performance of *Apros* on *xml-security* programs is proved to be better than that on *jtopas*. The result supports our conjecture.

We observed that a prioritization technique with high APFDc value also have high APMC value and a technique with low APFDc value also have low APMC value. The reason which accounts for this same trend is obvious. If an ordering of test cases reveals most severe faults early, the APFDc value of this ordering tends to be high. Due to the most severe faults have been exposed and fixed, the most fault-affected modules in program become cleared and APMC value of this ordering also tends to be high. Therefore, in this experiment, both APFDc and APMC can be used to measure the rate of severe fault detection. However, compared with the APFDc metric, our approach illustrates more benefits than other techniques in terms of APMC metric as shown in Figure 4. For instance: our approach perform 62.2% better than **Ran**, 30.5% than **Unt**, 13% than **MT**, 11.2% than **MA**, 31.3% than **DMT** and 24.7% than **DMA** on average. The explanation is that, several faults have been assigned the same severity according to the number of methods affected by them but regardless of whether the affected method is multiple-affected (the method affected by several faults) or solely affected (the method affected by just one fault). In many cases, the effect of eliminating the impact of faults is different even if those fixed faults have equal severities just as described in Section 4.2.3. Therefore, compared with APFDc metric, the APMC metric is more accurate to evaluate the rate of fault-affected methods clearance. The APMC value obtained by *Apros* illustrates that our approach effectively improved the rate of the fault-affected methods clearance, which is in line with our expectation.

Compared with in NDS testing, the benefits gained by using *Apros* seem to be more significant in regression testing. The reason is that both the **DMT** and **DMA** techniques prioritize test cases completely rely on the coverage of modified methods. However, in these two programs, not all the test cases covering the modified methods will reveal faults and not all the faults' impact have been confined to modified area. Therefore, just as shown by experimental result, *Apros* which prioritizes test cases by handling multiple information (e.g., call graph, source code changes) tends to be more adaptive than traditional prioritization techniques.

5.3. Threats to Validity

There are several potential threat to the validity of our studies. Although the subject programs we studied are real, non-trivial systems with original test suites created by developers, a main external threats is that the subject programs we studied are relatively small (5 KLOCs -16 KLOCs). And another external threat is that, the faults previously seeded in programs are not enough widely distributed. Therefore, the objects we used may not be representative of border complex industrial programs and faults severities seen in practice. On the other hand, the main internal threats to validity are the potential errors in our tool implementation and measure of experimental result. In order to reduce this kind of threats, we have applied our tool to some small subject programs created by us and have manually validated the produced results.

6. Related Work

To improve the software testing activity, researchers have proposed many techniques for test case prioritization in recent years. Wong et al propose a way to prioritize test cases according to the criterion of "increasing cost per additional coverage" [19]. The prioritized test cases are selected from the test suites by a modification-based regression test selection technique and the authors suggest that this kind of prioritization approach could be especially useful in the situation of limited resources. Rothermel et al present a formal definition of test case prioritization problem and provide metrics for measuring the rate of fault detection of test suites [4, 5, 15, 16]. Based on investigating several prioritizing techniques, they suggest that expensive techniques might not be as cost-effective as other less expensive techniques [15] and version-specific prioritizing technique can significantly improve the rate of fault detection of test suites [6].

Most of existing researches on test case prioritization focuse on improving the rate of fault detection in regression testing. Jones et al present a way to incorporate modified condition/decision coverage information into test cases prioritization [8]. In [10, 11], Korel et al propose a model-based test prioritization approach, which uses the different information about the system model and its behavior to prioritize the test suite for system retesting. Jeffrey and Gupta prioritize test cases using relevant slices [7]. Besides based on total statement coverage, this approach takes into account the number of statements executed that influence or potentially influence the output produced by the test case. In [21], Zhang et al incorporate varying testing requirement priorities and test case costs into test case prioritization. Mirarab and Tahvildari present a prioritization approach which utilizes Bayesian Network to integrate various sources of information to prioritize test cases for regression testing [13]. In [14] Qu et al propose a approach to prioritize test cases in black box environment. Considering the historical execution data of test cases, Kim and Porter propose a prioritization approach using history information such as: how many faults the test has revealed recently and how often it has been executed lately [9]. Although the proposed techniques are empirically proved to effectively improve the regression testing activity, they may not be directly applied to non-regression testing where the code changes and the previous execution data of test cases is unavailable.

Only a few proposed techniques [17, 18] aim to improve the faults detection rate for not only the regression testing but also non-regression testing. In [18] Tonella et al treat the test case prioritization problem as a machine learning problem. The authors propose a prioritization technique based on machine learning algorithm which combines multiple prioritization indexes and utilizes information extracted from user knowledge to prioritize test cases. By combining the information elicited from the user, this technique represents an improvement over existing methods in terms of APFD metric. However, this work dose not allow for varying fault severities. In [17] Srikanth et al present a system-level test case prioritization approach based on fixed weights specified by user. The technique identifies the important requirement which would increase customer-perceived software quality and make the test cases related to that requirement run earlier. However, our approach evaluates fault severity not only from users' viewpoint, but also from the viewpoint of program structure. We identify the key part of system and make the faults located in that part revealed earlier.

7. Conclusion and Future Work

This paper described a novel test case prioritization approach which can be used to improve the rate of severe fault detection for both regression testing and non-regression testing. For each module to be tested, our approach focuses on evaluating its fault proneness and the potential impact of faults by analyzing the program structure. In addition, our approach is also open to combine a variety of available information (e.g., user requirements, previous execution profiles of test cases) into prioritization. Therefore, compared with previous techniques, our approach has a wider scope of application. In the experimental study, we compared the effectiveness of our approach with traditional prioritization techniques which are based on methods coverage or modified methods coverage. The experimental result suggests that our approach can improve the rate of severe faults detection of test suites and can effectively eliminate the impact of faults at a fast rate.

As our future work, we intend to investigate the effect of adjusting weights of factors in Equations (3), (4) and (5) in different test phases/contexts. In addition, we plan to conduct an experimental study on more complex industrial programs from different domains to obtain results that are more representative.

Acknowledgements

This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Sai Zhang, Si Huang and Cheng Zhang for their discussion on this work.

References

[1] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[2] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, 32:733–752, 2006.

[3] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *International Symposium on Software Reliability Engineering*, pages 113–124, 2004.

[4] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112, 2000.

[5] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *International Conference on Software Engineering*, pages 329–338, 2001.

[6] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering*, 28(2):159–182, 2002.

[7] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Computer Software and Applications Conference*, pages 411–420, 2006.

[8] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, pages 195–209, 2001.

[9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *International Conference on Software Engineering*, pages 119–129, 2002.

[10] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *International Conference on Software Maintenance*, pages 34–43, 2007.

[11] B. Korel, L. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *International Conference on Software Maintenance*, pages 214–223, 2002.

[12] H. K. N. Leung and L. White. Insights into regression testing. In *International Conference on Software Maintenance*, pages 60–69, 1989.

[13] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases based on bayesian networks. *Lecture Notes in Computer Science*, 4422:276–290, 2007.

[14] B. Qu, C. Nie, B. Xu, and X. Zhang. Test case prioritization for black box testing. In *Computer Software and Applications Conference*, pages 465–474, July 2007.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–188, 1999.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *Software Engineering*, 27(10):929–948, 2001.

[17] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *International Symposium on Empirical Software Engineering*, Nov 2005.

[18] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *International Conference on Software Maintenance*, pages 123–133, 2006.

[19] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *8th IEEE International Symposium on Software Reliability Engineering*, pages 264–274. Nov 1997.

[20] S. Zhang, Z. GU, Y. Lin, and J. Zhao. Change impact analysis for aspectj programs. In *International Conference on Software Maintenance (to appear)*, Sep 2008.

[21] X. Zhang, C. Nie, B. Xu, and B. Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *International Conference on Quality Software*, pages 15–24, Oct 2007.