Improving Spectrum-Based Fault Localization using quality assessment and optimization of a test suite

Chang Liu
Chinese Aeronautical Radio
Electronics Research Institute
Shang Hai, China

Chunyan Ma
Software College
Northwestern Polytechnical University
Xi’an Shaanxi, China
machunyan@nwpu.edu.cn

Tao Zhang
Software College
Northwestern Polytechnical University
Xi’an Shaanxi, China

Abstract—Spectral fault localization is an automatic fault-localization technique to expedite debugging, which uses risk evaluation formula to rank the risk of fault existence in each program entity after collecting testing information. To assess the potential usefulness of a test suite and improves the accuracy for spectral fault localization, methods of assessing and optimizing test suite are proposed in this paper, which. Firstly, Average Ranking Cost of the test suite quality and two kinds of constraints are defined; and test suite quality assessment method based on these definitions is given. Secondly, a new test suite optimization method based on greedy algorithm is proposed. Finally, two widely used program databases (SIR and Defects4j) and 8 SFL techniques are applied to verify the effectiveness of our method, and the fault localization cost before and after optimizing test suites of test objects are analyzed using effect size. The largest effect size reaches 0.5398 and Each SFL technology has different degrees of improvement in the rankings of faulty statements in different programs by optimizing test suite.

Keywords—Spectral fault localization, Average Ranking Cost, Test suite quality assessment

I. INTRODUCTION

The spectral fault localization (SFL), as an automatic localization technique, has been extensively studied in recent years. This approach uses risk evaluation formula to calculate suspicious factor of fault existence in each program entity after dynamically collecting testing information. The testing information includes: the execution coverage for each entity (e.g., statement, function, and basic block), and the execution result (i.e., pass or fail) for each test case. Then, all entities will be sorted in ascending order according to their suspicious factors. Debugging is then conducted on entities according to the top-to-bottom ranking list until a fault is found. For different program entities, the statement and the basic block have especially received attention. Without loss of generality, this study considers an entity as a statement. In order to make SFL techniques effective and useful, there are two things that should be considered. One is proposing new and effective SFL techniques. So far, more than 50 SFL techniques and related technologies have been proposed over the years, such as O and Op [2], DE(J) and DE(C) [1], Wong [3], Jaccard [4], Kulczynski [5], D* [6], EMF [20] and so on. The other, which [7,15-16,18-19, 28] has adopted empirical approaches according to the established experimental setups and benchmarks, such as Siemens Suite, is improving the performance of existing SFL techniques through understanding the quality of test suites.

The goal of this work is to perform such an investigation of test suite quality assessment to improve spectral fault localization technique. We propose an Average Ranking Cost with two kinds of constraints to measure test suites quality and then a test suite optimization method using greedy algorithm for SFL techniques. The major contributions of this research are summarized as follows:

- We propose an evaluation method that provides a repeatable and objective way investigating and assessing test suites through an Average Ranking Cost with two constraints.
- With the proposed evaluation method of test suites, we explore an aspect about how to improve the performance of SFL by optimizing test suite based on the greedy algorithm.
- Nine typical SFL techniques such as O, Op and two widely used program databases (SIR [7] and Defects4j [8]) are applied to verify the effectiveness of our method.

Next section provides the necessary background on SFL techniques. Section III presents the proposed quality assessment of a test suite in detail. Section IV explores test suite optimization method based on greedy algorithm. Section V provides evaluation for our method. Section VI gives the related work. Section VII draws conclusions.

II. TECHNIQUES REVISITED

For example, two SFL techniques O and O' are shown in the following equations.

\[
O(a_{t'}, a_{n}, a_{p'}, a_{p}) = \begin{cases} 
1 & (a_{t'} > 0) \\
0 & \text{otherwise} 
\end{cases} 
\]

(1)

\[
O' (a_{t'}, a_{n}, a_{p'}, a_{p}) = \frac{a_{t'}}{p-1} 
\]

(2)

Figure 1. An example for SFL

Considering the example in Fig. 1, a program PG has ten statements from s1 to s10, and test suite TS has nine test cases from t1 to t9. Specifically, t1 to t6 give rise to fail runs and the remaining six test cases give rise to pass runs, as indicated in binary outcomes OC which records the testing results of TS. In OC, 1 indicates to pass and 0 indicates to fail. The elements in the ith row of matrix M represent the test coverage information of statement s1 executing t1 to t9, in which l
indicates that $s_i$ is executed by the corresponding test case, and 0 otherwise. Matrix $M_A$ is such defined that its $i$th row represents the corresponding four values of the vector $<a_{0i}, a_{1i}, a_{2i}, a_{3i}>$ for $s_i$. $SF_i$ is the suspicious factor list of statements using SFL $O$. For instance, $a_{0i}=4$ for $s_i$ means that four test cases give pass without executing $s_i$, and in $SF$, “4” represents the suspicious factor of $s_i$ which is highest in the suspicious factors of all statements and “-1” represents the suspicious factors of $s_i$ and $s_0$ which is lowest in the $SF$. $SF$ is using to rank statements from $s_j$ to $s_{10}$. The derived ranking list of statements using $O$ is $[s_1, "s_5, s_8, s_{10}", "s_4, s_6, s_7, s_9", "s_2, s_3"]$, where from left to right, the statements ranked from high to low and some statements have been marked by each “ ” have the same ranking.

For the performance measurement of SFLs, the EXAM is appropriate and majority of the SFL community use the same measurement as the EXAM or its equivalent. Therefore, we use the EXAM with tie-breaking ways as ranking cost in this paper.

$$g + e/2 \over n \tag{3}$$

- $g$ is the number of correct statements ranked strictly higher than all faults.
- $e$ is the number of all statements ranked equal to the highest ranked fault and if no correct statement ranked equal to the highest ranked fault, then $e = 0$.
- $n$ is the number of statements in the program.

For example, in Fig.1, if the $s_j$ is assumed to be faulty, then for $s_1$, $g=4$ and $e=3$, and the fault localization cost is $(4+3/2)/10 = 11/20$ using $O$. The smaller the value of fault localization cost is, the better the performance of locating a fault for the SFL technique is.

III. THE QUALITY ASSESSMENT METHOD OF A TEST SUITE

Before quantitatively test suite quality, several assumptions, which are adopted from most of the previous SFL studies [2], [19], [10], [11], [12], are first listed.

- The SFL techniques are applied to programs whose testing result of either fail or pass can be decided for any test case.
- Debuggers examine the statements one by one from the top to bottom of the ranking list returned by SFL, and once the faulty statement is examined, the fault can always be identified.
- The test suite is assumed to have 100% statement coverage and also is assumed that the test suite contains at least one passed test case and one failed test case.
- The faulty statement must be executed by all failed test cases in a given single-fault program.

For a real faulty program, it is impossible to know which test suite is better to locate the specific fault using a SFL technique, because we don’t know where the faulty statement is. The efficiency and effectiveness of a test suite cannot be evaluated when the faulty statement is unknown. In a single-fault program, one statement cannot be faulty if it is not covered by some failed execution and this view has been discussed detailly in [21]. So in order to generalize the test suite quality evaluation, we assume the single faulty statement exists in statements whose value of $a_{0i}$ is equal to 0. These statements are called possible faulty statements. Then we measure the test suite quality on Average Ranking Cost of locating all possible faulty statements. The definition of “Average Ranking Cost” of a test suite is as follows.

**Definition 1 Average Ranking Cost of a test suite**. Given a program $PG = <s_1, s_2, ..., s_n>$ with $n$ statements, a SFL and a test suite, the Average Ranking Cost is the average of the sum of performance measurement calculated in terms of equation $(3)$ for all possible faulty statements. All possible fault statements can refer to the set of statements with $a_{0j} = 0$ in the single fault scenario, or the set of statements with the top 20% of suspicious factor (exonerating about 80% of the blocks of code on average) in [22]. We take each of the possible faulty statements as a real faulty statement to calculate the ranking cost of the test suite respectively. If there are $x$ possible faulty statements, we will calculate the average ranking cost of $x$ statements to evaluate the ability of the test suite to locate faults.

For the example of fig.1, the statements $s_1, s_2, s_4, s_5, s_6, s_7, s_9$ and $s_{10}$ are the possible faulty statements because their $a_{0j}$ equal 0. The corresponding Average Ranking Cost is $(11/20+0+11/20+1/5 + 1/5 + 11/20 + 11/20 + 1/5)/8 = 9/40$ referring to equation $(3)$. The smaller the value of Average Ranking Cost of a SFL applying for a test suite, the higher the quality of this test suite.

If we only use the Average Ranking Cost to evaluate the quality of a test suite, we found that sometimes Lowering the Average Ranking Cost can improve the rankings of some faulty statements, but it may also cause many statements, including faulty ones, to be ranked equal and indistinguishable. For example, in an original test suite, possible faulty statements in a program are ranked $[1, 1, 1]$, and Average Ranking Cost of these statements is 1. When a test case is removed for optimization, these statements are ranked $[1, 1, 1]$, and Average Ranking Cost is 2/3. Thus when we use a method of reducing the test case to decrease Average Ranking Cost of possible faulty statements, the suspicious factors of more statements may be equal. This situation is contrary to the SFL’s principle of making statements more distinguishable. To prevent this situation, we define some constraints for the Average Ranking Cost.

In order to give constraints for Average Ranking Cost, we first give some definitions.

**Definition 2 Partition**. The partition is a list which is formed of consecutive equal values and it at least include 2 equal values in $SF$. In Figure 1, $SF$ has three partitions which are $[3,3,3], [0,0,0,0]$ and $[-1,-1]$.

The more the number of partitions, the better the differentiation among risk rankings of statements. So Constraint 1 is proposed.

**Constraint 1 The number of the partitions cannot be reduced when the Average Ranking Cost is reduced after optimizing test cases.**

**Definition 3 Degree of a Partition**. The degree of a partition is the number of elements contained by this partition. In Figure 1, the degree of the partition $[3,3,3]$ is 3.

**Definition 4 Degree of the program**. The degree of the program is the sum of the degrees of all partitions with respect to all possible faulty statements. Taking Figure 1 as an
example, the degree of the program is 7, where $s_1$ and $s_8$, whose $\alpha_{df}$ are not 0, they do not participate in calculation.

The greater the degree of program, the worse the distinction among risk rankings of statements. So Constraint 2 is proposed.

**Constraint 2** The degree of the program cannot be increased when the Average Ranking Cost is reduced after optimizing test cases.

Combining definition 1 with constraint 1 and 2, we can calculate and compare the quality of the test suite according to the steps shown in Figure 2. For example, for any program, if we let the already available test suite be $TS'$, the test suite be $TS$ after $TS'$ is reduced; or any program corresponds to two test suites $TS$ and $TS'$. The following steps can be applied to compare $TS$ and $TS'$.

In this paper, a test suite optimization method based on greedy algorithm is proposed.

Greedy algorithm divides the problem (i.e. test suite optimization) into multiple sub-problems (i.e. Comparing the quality of test suite before and after deleting a test case). According to the chosen greedy strategy (heuristic principle), we select an optimal solution of each sub-problem, and finally constitute the optimal solution of the problem. The greedy strategy is to choose best Average Ranking Cost with two kinds of constraints in the sub-question. We build a subset of test suites by reducing test cases and then calculate the Average Ranking Cost with two kinds of constraints the test suites subset. According to the greedy strategy, a subset of test suites that reduces the Average Ranking Cost with two kinds of constraints are selected as local optimal solutions. Thus, an optimized test suite is constructed by multiple local optimal solutions within the time complexity of $O(n)$. We will introduce the process of using the greedy algorithm to solve the problem about optimization test suite from the design heuristic principle and algorithm implementation.

A. Heuristic principle

Before implementing the greedy algorithm, we need to design a heuristic principle for solving this problem, as shown below.

1) We assume a negative correlation between Average Ranking Cost with two kinds of constraints and quality of a test suite. The lower is Average Ranking Cost with two kinds of constraints, the higher is the quality of the test suite.

2) At the same time, two test cases that increase Average Ranking Cost with two kinds of constraints are removed, and the Average Ranking Cost is statistically more likely to decrease. In other words, removing test cases $t_1$ and $t_2$ separately can reduce the Average Ranking Cost with two kinds of constraints, but removing them at the same time will increase the Average Ranking Cost with two kinds of constraints. Although the above may theoretically exist, we assume that the probability of this happening is low.

B. Test suite optimization based on greedy algorithm

The main idea of the greedy algorithm in this paper is based on the heuristic principle of test suite optimization problem. A test case is removed from the original test suite to form a subset of the test cases. The optimal Average Ranking Cost with two kinds of constraints of solving the subset of the test cases is a sub-question, and the solution result of the combined sub-problem constitutes a solution to the problem.

Figure 3 gives the Test suite optimization algorithm based on greedy algorithm. In line 1 to line 4 of this algorithm, it retains all test cases and calculates their Average Ranking Cost, the number of partitions (constraint 1) and the degree of the program(constraint 2). Next, for each test case, we remove a test case which execute the program and pass it on line 7 and then calculate Average Ranking Cost, the number of partitions (constraint 1) and the degree of the program(constraint 2) on line 11. If Average Ranking Cost is lower or unchanged, degree of the program is higher or unchanged and the number of the partitions for statements is more or unchanged, we should keep removing the test case and go to the next test case.
Otherwise, re-add this removed test case to the test case set on line 16.

**Average Ranking Cost** is the average of Exam for program statements (excluding statements with \( a_i \neq 0 \)), so first we need to calculate the Exam cost for each possible faulty statements. We calculate the suspicious factor matrix \( SF \) of the program through the \( MA \) matrix. For each statements, we count the number of statements whose ranking is higher than the ranking and the number of statements in the program, and then the Exam value is calculated by the formula (2).

\[
\text{Exam cost for each possible faulty statements} = \frac{n}{n+1}\text{ Ranking cost after optimization} - \frac{n}{n+1}\text{ Ranking cost before optimization}
\]

Figure 3. Test suite optimization based on greedy algorithm

V. EVALUATION

A. Experimental objects

This experiment uses seven C language programs and 3 language Java programs. This C language experimental objects consists of five programs in the Software-artifact Infrastructure Repository [7]. TABLE 1 lists all information of programs. The TABLE 2 describes Java experimental objects from Defects4j through the number of versions and the maximum and minimum values of the number of test cases in these programs.

**Table 1: Experimental objects using C language**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Number of versions</th>
<th>Number of test cases</th>
<th>Number of lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>Air collision avoidance system</td>
<td>41</td>
<td>1500</td>
<td>174</td>
</tr>
<tr>
<td>print</td>
<td>Lexical analyzer</td>
<td>7</td>
<td>4130</td>
<td>726</td>
</tr>
<tr>
<td>tokens</td>
<td>Priority scheduler</td>
<td>10</td>
<td>2710</td>
<td>374</td>
</tr>
<tr>
<td>schedule2</td>
<td>Mode replacement</td>
<td>32</td>
<td>5542</td>
<td>564</td>
</tr>
<tr>
<td>replace</td>
<td>Information statistics</td>
<td>23</td>
<td>1052</td>
<td>565</td>
</tr>
<tr>
<td>tot_info</td>
<td>Expression parsing and arithmetic</td>
<td>13</td>
<td>1361</td>
<td>1039</td>
</tr>
<tr>
<td>expression</td>
<td>Comparison of various sorting</td>
<td>10</td>
<td>1500</td>
<td>2512</td>
</tr>
</tbody>
</table>

**Table 2: Experimental objects using Java language**

<table>
<thead>
<tr>
<th>Program name</th>
<th>Project name</th>
<th>Number of test cases</th>
<th>Number of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>JFreeChart</td>
<td>26</td>
<td>1591–2193</td>
</tr>
<tr>
<td>Math</td>
<td>Apache commonsmath</td>
<td>106</td>
<td>817–4378</td>
</tr>
<tr>
<td>Time</td>
<td>Joda-Timet</td>
<td>27</td>
<td>3749–4041</td>
</tr>
</tbody>
</table>

B. Evaluation method

In order to compare the quality of the test suite before and after optimization, we use the effect size to evaluate the comparison results. The effect size is a value used to express the degree of correlation between two sets. Currently, there are two commonly used effect size measures:

- Pearson correlation coefficient. It is mainly used to calculate the degree of correlation between two sets.
- Cohen’s \( d \) metric. It is mainly used to calculate the difference between two sets.

Since the main purpose of this experiment is to explore the difference between ranking costs of real faulty statements before and after the test suite optimization, Cohen’s \( d \) metric is used as a formula for calculating the effect size. Cohen’s \( d \) is formula (4).

\[
d = \frac{u_1 - u_2}{s}
\]

In formula 5, \( u_1 \) represents the average of the first sample (that is ranking cost of the faulty statement for each faulty version after test suite optimization) and \( u_2 \) represents the average of the second sample (that is ranking cost of the faulty statement for each faulty version after test suite optimization). \( s \) represents the combined standard deviation of two samples, the formula of which is given by (5).

\[
s = \sqrt{\frac{(n_1 - 1) \times s_1^2 + (n_2 - 1) \times s_2^2}{n_1 + n_2 - 2}}
\]

In the above formula, \( n_1 \) and \( n_2 \) are the sizes of the two samples, respectively, and \( s_1^2 \) and \( s_2^2 \) represent the variance of the two samples.

In this paper, ranking costs of real faulty statements before optimization is used as the data with subscript 1 in the above formula, and ranking costs of real faulty statements after optimization is used as the data with subscript 2. A positive number indicates that the Average Ranking Costs of faulty statements on the optimized test suite is lower than before optimization, and vice versa. The larger the absolute value is, the greater the difference between the two sets of data (before and after optimization). In short, the larger the effect size is, the better the quality of test suite after optimization will be.

C. Experimental method

We perform experiments on the above eleven programs, compare the ranking of faulty statements calculated on the optimized test suite with the ranking of faulty statements before optimization, and use the effect size to quantify the experimental results.
The overall experimental process is shown in Figure 4. First we run the original test suite in the test program. The matrix $M$ and $OC$ are constructed based on the collected operational information, and then the vector $<a_{ef}, a_{op}, a_{of}, a_{op}>$ of each statement which is used to construct the $MA$ matrix is statistically calculated. We use nine suspicious factor calculation formulas to calculate the suspect factor for each statements to get the matrix $SF$. We can rank each statements of the program based on the value in $SF$ and calculate the Average Ranking Cost of the original test suite. Based on the heuristic principle, we use greedy algorithm to get the sub-test suite whose Average Ranking Cost is lower than the original test suite and which makes less degrees of program and more the number of equivalence class for statements. To prove that the optimized test suite is higher quality than the original test suite, we use the optimized test suite to test program and use SFL technology to get a suspicious factor ranking sequence for each statements. We check the ranking and cost change of the corresponding faulty statements according to the line number of the specific faulty statement in the experiment object. That is based on the line number of the faulty statement and the $SF$ matrix, a ranking sequence of faulty statements before and after optimization is constructed. Finally, we use effect size to quantify the contrast between the two.

D. Analysis of experimental results

Figure 5 shows the effect size of faulty statements before and after the test suite optimization for each programs under each formula. As shown in Figure 5, after using the greedy algorithm to optimize the test suites, in most cases the rankings of faulty statements have increased Effect Sizes. The largest effect size reaches 0.5398. For each SFL excluding Jaccard, Ochiai and SBI, the ranking costs with two kinds of constraints decrease after optimizing test cases. So the rankings of faulty statements in different programs has different degrees of improvement by optimizing test suite.

For Jaccard, Ochiai and SBI, the ranking cost of some faulty versions increased slightly after optimizing test cases. Due to the randomness of a real fault of programs, when test suite quality is assessed, all possible fault statements are involved in performance calculations in this paper. Our method takes the Average Ranking Cost as a measurement, the high-quality test suite could improve the average localization performance of faulty programs when we do not know where the real fault is. However, this does not show that it must improve the localization performance of a real fault in a given program. In the case of guaranteeing average performance, two kinds of constraints can further reduce the cost of finding faulty statements in practical applications.

At the same time, it is shown in the Figure 5 that the effect amount on SIR[12] is significantly higher than that on Defects4j [13]. Because there are a large number of repetitive (redundant) test cases in the SIR test suite, it is better to optimize the test suite by reducing the number of test cases. There are no redundant test cases in Defects4j, so the results are not obvious. From the results, there are still some negative effects in the partial program under some formulas.
improve fault-localization effectiveness. The rankings of some statements may be more distinguished than before via removing the redundant test cases, but this situation is not necessarily occur. These methods were unable to reveal the underlying rationale for all of their observations, and did not consider how and why removing test cases from the test suite can improve the fault localization.

In this paper, section IV and section V could guide us to assess test suite quality and find an optimal subset of a test suite which leads to high Average Ranking Cost with two kinds of constraints. These works above are specific cases of our approach could explain their experimental results; and they evaluated the effects of test suite reduction using some programs, and thus, they are unable to definitively state that their findings will hold for programming in general. For example, for the reference [13], after so-called redundant test cases are removed, the effectiveness cannot be improved even lower with respect to the existing test suites and the Op (or O); for the example in Figure 1 which has the same meaning with the Figure 1, where s10 is faulty, if the first column test case and the seventh test case which are redundant test cases are removed, the rankings of s7 to s10 are 9/7, 2/7, 11/7, 9/7, 11/7, 11/7, 9/7, 2/7, 9/7 and 11/7, thus, the ranking of s10 are reduced from being higher than s5 and s6 to equaling s5 and s6, and the performance of Op are lower than before removing redundant test cases.

Lei, Y. et al. [19] have also identified “in a test suite, the passing test cases that do not execute the faulty statements and the failing test cases have a positive impact on the fault localization effectiveness, whereas the passing test cases that exercise the faulty statements have a negative effect on localization performance”. Their result is drawn from a large-scale empirical analysis on the localization effectiveness with respect to randomly sampled test suites and improve fault localization performance by removing those passing tests. Our quality assessment method for a test suite can shows “the failing test cases may have a positive impact on the fault localization effectiveness or no impact, and the passing test cases may have a positive impact or negative”. For example, if the faulty statement is one of s3, s4, s6 or s10 and a new failed test case \( f_{10} = [1,0,1,1,1,1,1,1,1,1] \) is added, then the suspiciousness has not changed in Figure 1, so the localization performance has not changed. We argue against that “the failing test cases must have a positive impact on the fault localization effectiveness”. Our work does not distinguish the impact of the passing test cases, and Lei, Y. et al. break the passing test cases down into two cases. Lei, Y. et al. also proposed a method for improving fault localization performance by removing those passing tests. Through our example analysis, this method is not a stable method to improve performance and the result is random. For example, in Figure 1, according to the author’s method, we remove \( t_1 \) and \( t_6 \) by that the feasible percentage value for PTD-TO is 90%.

- The derived ranking list of statements using \( O \) is \(["s_5, s_6, s_7", "s_1, s_4, s_7"]. If \( s_1 \) is faulty, its is changed from the previous ranking \( l([s_5, s_6, s_7, s_4, s_8]) \) to the current ranking \( 3 \); and if \( s_5 \) or \( s_6 \) is faulty, their rankings have been improved.

- The derived ranking list of statements using \( GP01 \) is \(["s_5, s_6, s_7", "s_1, s_4, s_7", "s_2, s_5", "s_3, s_8"] \) to the current ranking \( 4 \); and if \( s_2 \) or \( s_5 \) is faulty, their rankings have been improved.

- The derived ranking list of statements using \( GP02 \) is \(["s_5, s_6, s_7", "s_1, s_4, s_7", "s_2, s_5", "s_3, s_8", "s_4, s_7, s_9", s_2, s_5]) \) to the current ranking \( 3 \); and if \( s_2 \) or \( s_9 \) is faulty, their rankings have been improved.

Our method takes the Average Ranking Cost with two kinds of constraints as a measurement, the high-quality test suite could improve the average localization performance of faulty programs when we do not know where the real fault is. Although this does not show that it must improve the localization performance of a real fault in a given program, it is a big probability event for all possible faulty statements. In the case of guaranteeing Average Ranking Cost, two kinds of constraints can further reduce the cost of finding faulty statements in practical applications.

**VII. CONCLUSIONS**

The quality evaluation of test suite can promote the activity of improving the quality of test suite for the better localization performance of SPL techniques. This could be of significant practical benefit for larger programs. This study has a proposed a method of test suite quality assessment through Average Ranking Cost and two kinds of constraints. Based on test suite quality assessment method, the greedy algorithm is used to optimize the test suite, which improves the accuracy of spectral fault localization. And so if an available test suite exists, the greedy algorithm can guide us to reduce test cases from an available test suite to improve the fault localization performance.

**ACKNOWLEDGMENTS**

This work was supported by Key laboratory project of China aviation science foundation (20175553028 and 20185853038) and the China Aerospace funded project entitled “Fault prediction and location technology of aerospace embedded software based on variables and values”. We also thank all the anonymous reviewers for their constructive comments.

**REFERENCES**


