Application of Software Testability Measurement Model SPM to Software Testing

XU Ping, WANG Yang-ling, SHEN Zhong-nan
Dept. of System Engineering of Engineering Technology
Bei Hang University
Beijing, China

Abstract—Towards the contradiction in software testing between requirements of high efficiency and limited resources, research on application of software testability measurement model SPM to software testing is conducted. Through analyzing execution probability (E), infection probability (I), propagation probability (P) and testability value (T) in determining the quantity of software test case, software testing intensity, regression testing intensity, optimizing software testing sequence and analyzing performance bottleneck, an application approach of SPM model to software testing is proposed. By validation, it is proved that it is effective and can improve testing efficiency.

Keywords—software testability; measurement; software testing; SPM; PIE

I. INTRODUCTION

Nowadays because of being restricted by progress, budget, manpower, resources and so on, software testing occupies a large portion in software life cycle. In view of decreasing testing resources and improving testing efficiency, there is a wide concern on software testability. Especially software testability design and software testability measurement are researched at home and abroad, this paper focuses on the latter, software testability measurement. In this area, it is currently researched mainly that improvement of software testability measurement approach. Large amounts of remarkable achievements have been achieved, such as object-oriented measurement and process-oriented measurement. For the former, VC measurement is a representation of the object-oriented measurement. For the latter, the static analysis and the dynamic analysis are included. Verification and measurement of software component testability Ming-Chih Shih proposing is a representation of the dynamic analysis approach, as in [1]. In the static analysis have there existed SPM model, as in [2-3], and have there existed Software Testability Measurements Derived from Data Flow Analysis proposed by Pu-Lin Yeh, as in [4]. However, it is rare that research how to apply testability measurement to software testing, judge whether software testing is difficult and provide the probability of exposing the faults in the tested procedure. This undoubtedly goes against the application of testability measurement to the practice. Taking into account SPM model is currently more comprehensive approach, combing with the practice, this paper will carry out the research on applying the SPM model to software testing.

II. AN APPROACH OF APPLYING SPM MODEL TO SOFTWARE TESTING

A. SPM model

SPM model is on the assumption of the fault/failure, based on the static PIE analysis approach, as in [5-8]. Towards the shortage of the PIE, SPM model introduces the “relay” from the RELAY model for error detection, as in [9]. That is to say, (a) a potential failure is originated when a fault is introduced into the procedure; (b) the potential failure will be transferred as a state failure when it is computed; (c) the state failure will be transferred as another state failure by computation transfer and information flow transfer, finally be transferred as the procedure output, and an external error is revealed. According to this, SPM model make the refinement of the calculation process of infection probability and propagation probability for the sub-process of infection, namely, a fault will originate a potential failure and the sub-process of propagation, the potential failure will originate the next potential failure by computation transfer and information flow transfer, and finally it is transferred as the output of the procedure. SPM model is based on the static PIE analysis and make the computation of the two key metrics more perfect which are infection probability and propagation probability. The measurement of SPM model is more credible than that of the static PIE analysis.

1) Estimate for execution probability (E)

For any location , there are paths from the program beginning to -- with branches on every path. The execution estimate of would be the summation of execution rate along i-th path. Thus,

\[ E_i = \sum_{i=1}^{n} \prod_{j=0}^{k} e_j \]  

(1)

is the domain/range ratio, (1-domain/range) ratio, or branch weight at the j-th branch on i-th path before .

2) Estimate for infection probability (I)

The estimate for “I” is much related with the number of tokens--operands and operators--in a location. The following is the calculation of “I” for location :
\[ I_l = \sum_{i=1}^{n} (I_{op} \times I_{op-op(func)}) \bigg/ (opr + opd) \quad (2) \]

opr: the number of the operators or built-in functions on the right-side of the assignment, boolean predicate, or in the <condition> part of if or while statements; opd: the number of the operands on the right-side of the assignment, boolean predicate, or in the <condition> part of if or while statements; I(op): the potential failure probability of the operators and operands; \( I_{op-op(func)} \): the transfer infection probability of operators and operands through the transfer of the mathematic expressions or <control> statement.

3) Estimate for propagation probability(P)

In view of the complexity of program, before calculating “P”, the program is divided into five kinds of typical program structures, which are P1, P2, P3, P4 and P5. The following is the calculation of “P” for location l:

a) For a specified location l, find the variable v, which is defined.

b) From location l, find location l’, where v is used after it is executed at location l. And l’ is the new specified location.

c) Get I_l of location.

d) Repeat step a)-c) until we reach the output statement.

\[ P_l = \prod I_l \quad (3) \]

The structure of location l’ is M. If M is P1, calculate I_l; if M is P2, calculate I_l = \( \max(l_{l'}) \); if M is P3, P4 or P5, calculate I_l = \( \sum_{i=1}^{m} (E_l \times I_{l'}) + \sum_{i=1}^{m} (E_l \times I_{l'}) \).

4) Estimate for testability value(T)

For the program statement: \( I_l = E_l \times I_l \times P_l \); for the program block: \( T_{block} = \sum_{l=1}^{n} I_l \bigg/ n \).

See references for detailed calculation.

B. Application Approach

SPM model, based on static PIE approach, gives four measurement value, including execution probability, infection probability, propagation probability and testability value. Their calculation process is similar to process of ascertaining fault location in hardware. However, SPM model provides the probability of fault location instead of ascertaining specific fault location. By applying SPM model to practice, we can sum up that SPM plays the role of guiding on software testing, as is shown in the following TABLE I.

<table>
<thead>
<tr>
<th>SPM Analysis</th>
<th>The Number of Test Case</th>
<th>Testing Intensity</th>
<th>Regression Testing Intensity</th>
<th>Optimizing Testing Sequence</th>
<th>Performance Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1) the minimum number of test case(N)

The minimum number of test case required to test conditions and judgment paths can be determined. As is well known, the smaller execution probability of statement is, the more difficult it is to cover all paths. Thus, more test cases will be required. We can conclude that the number of test case is to some extent determined by execution probability. We can get the number of test case as follows: at first, execution probability is divided into fraction and the numerator is changed into 1, as in

\[ E_l = 1/m_l \quad (4) \]

Then the denominator \( m_l \) in the last formula is rounded up and we can get the number of test case required to cover the statement l, as in

\[ m'_l = \lceil m_l \rceil + 1 \quad (5) \]

Similarly, it is available that it requires the number of test case to cover every statement in the program, of which we can choose the maximum value as the minimum number of test case required to cover all conditions and judgment paths of the program, as in

\[ N = \max \{ m'_1, m'_2, ..., m'_n \} \quad (6) \]

2) improving efficiency of code reviews

According to the formula of infection probability, we can conclude that infection probability lies on the amount of operator and operand. The more complex calculation is, the smaller infection probability is. Thus, according to infection probability, it is very cushy for programmers and code reviews staff to judge the bug probability of logics, operations and mathematical functions in mathematical expressions. This can not even be realized by the code rule checking and most of the automated testing tools. Simultaneously, testability value can reflect whether code testing is easy. While some codes have a very small testability value, potential failure in these codes will not easily be exposed. Aiming at this, testers should allocate more test resources to strengthen code testing. Now that there may exist some sort of positive connection among infection probability, testability value and code testing, we can draw the conclusions that importance of code reviews (ICR) is a binary group ling on infection probability (I) and testability value (T), as in

\[ ICR = \{ I, T \} \quad (7) \]

Among the importance of code reviews (ICR) and infection probability (I), testability value (T) is there inverse proportion connection. The smaller both of statement infection probability (I) and testability value (T) are, the more possible it is for the statement to contain the potential failure. Since that, it is
inevitable for the statement to have the bigger importance of code reviews, namely, we have to spend more test resources on it. Then, testers can judge which statements should be keystone of testing, optimize testing sequence and improve efficiency of code reviews. Particularly, this method is more suitable for the software containing a mass of calculation.

3) performance bottleneck
The performance, velocity and efficiency, may depend on whether it is easy to calculate program variables. And difficulty degree of calculation of program variables will finally determine whether it is easy to export the results of program variables. In the calculation of propagation probability, the use of program variables in subsequent statements is also taken into consideration. Persist until program exit like this. Propagation probability can reflect difficulty degree of variables output and make an effect on the program performance. The smaller propagation probability of statement is, the more possible it is for the statement to affect the program performance. By comparing propagation probability, it is undoubtedly feasible for code reviews staff to find out performance bottleneck of the program.

4) regression testing intensity (RTI)
When the statement is changed, the defect may be introduced into the program. Passing through the propagation of program, whether the defect can be found at program exit depends on propagation probability of program. It means propagation probability can reflect the probability of the defect introduced by changing the statement. Thereby, it can provide testers with the certainty of regression testing. The smaller propagation probability is, the bigger regression testing intensity caused by changing the statements is.

\[ RTI = \frac{F}{P} \] (8)

F, constant, lies on the program structure. If some statement has a very big propagation probability, after changing the program, its regression testing intensity will be very small. Testers can spend less test resource on it and improve testing efficiency.

5) test termination and code modifies
Towards the embedded software containing massive calculation, depending on SPM model, we can find out the key program block needing emphatic test, judge whether test should be terminated and make clear which parts of the program should be changed and how to be changed.

Passing through a large number of practical applications provide the sufficient data, we can choose a T value as threshold (M) which can help us judge whether the testability of program meets the requirements. Thus the threshold can be the basis for test termination.

Once the following conditions are reached the test can be terminated.

The testability value of program reaches the prescriptive threshold and no faults are found out throughout the prescriptive amount of test case.

\[ T \geq M \] (9)

If the testability value can not meet the requirements,

\[ T < M \] (10)

We should analyze execution probability, infection probability and propagation probability of every statement to determine by which the low testability of program is caused. Once this measurement is found out, we can find application approach of this measurement in the software testing according to the above 1)--4). Then, the shortage of the program can be found and the corresponding suggestions to change the program will be provided. Eventually, we can improve the test efficiency designedly.

III. VERIFICATION
We will combine with some procedure, calculate and analyze the measurement results, and judge whether the application approach of SPM model in the software testing is effective.

A. Case Example
The following procedure which is from some real embedded software is a time scheduler of some real-time system. The paragraph of code realizes the clock synchronization among the nodes. Every a period of time (re-synchronization interval: 100 second), it will conduct the clock synchronization operation. The clock of every node gives its own clock reading, which can be achieved by all other nodes throughout the real-time network. After getting the average value of clock readings of all nodes, the clock synchronization operation is conducted.

```c
#include <stdio.h>
#include <math.h>
#define kNodeNo 4
void int main()
{
    unsigned int  Time[kNodeNo];
    int      DataTime[kNodeNo];
    int    AdjustTime[kNodeNo];
    double     StdTime = 0;

    for (i = 0; i < kNodeNo ; i++)
    {
        Time[i] = 0;
        DataTime[i] = 0;
        AdjustTime[i] = 0;
    }
    Read (Time[]);
    for(i = 0; i < kNodeNo ; i++)
    {
        int i;
        Time[i] = 0;
        DataTime[i] = 0;
        AdjustTime[i] = 0;
    }
    Read (Time[]);
    for(i = 0; i < kNodeNo ; i++)
    {
```
if (Time[i] == 0)
{
    break;
}
else
{
    [A]
    StdTime = StdTime + Time[i];
}

B. Results Analysis

In accordance with the formulas in A of II, we can obtain the calculation results in TABLE III of execution probability, infection probability, propagation probability and testability value of A, B, C and D statements.

1) the number of test case

For the A statement, \( E_A = 0.225 \approx 1/4.4 \), round the denominator up. \( N \) equals with 5. According to the application approach, we need at least 5 test cases to cover the A statement. The B statement which is located in the trunk procedure can be covered by arbitrary test case. By the same way, 4 test cases are needed to cover the C statement. The result of the D statement is the same as that of C. The maximum number of all the numbers is 5. According to the application approach, we should choose 5 as the minimum number of test case required by the program.

2) testing intensity and optimizing testing sequence

In accordance with the application approach this paper putting forward, the importance of code reviews (ICR) lies on infection probability (I) and testability value (T). Infection probability plays a leading role, and testability value plays a subsidiary role. When all the statements have the same infection probability, analysis of testing intensity is on the basis of testability value (T). Then the importance of code reviews (ICR) of the program sorts from big to small: ICR_D > ICR_C > ICR_B > ICR_A.

<table>
<thead>
<tr>
<th>Location</th>
<th>E</th>
<th>I</th>
<th>P</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.225</td>
<td>0.9667</td>
<td>0.1563</td>
<td>0.0340</td>
</tr>
<tr>
<td>B</td>
<td>1.0000</td>
<td>0.8743</td>
<td>0.1617</td>
<td>0.0414</td>
</tr>
<tr>
<td>C</td>
<td>0.2500</td>
<td>0.9667</td>
<td>0.1849</td>
<td>0.0447</td>
</tr>
<tr>
<td>D</td>
<td>0.2500</td>
<td>0.7650</td>
<td>0.1913</td>
<td>0.0366</td>
</tr>
</tbody>
</table>

3) performance bottleneck

According to Table II, the A statement of all the statements has the smallest propagation probability. It is the most possible for the A statement to be performance bottleneck of the program. However, because the program is of small scale and the calculation is very simple, the loop body where the A statement exists does not make a severe impact on propagation probability. Otherwise the program is very complex, we will have to modify the algorithm of the loop body to improve propagation probability.

4) regression testing intensity

According to the application approach, the bigger propagation probability is, the smaller regression testing intensity is. Regression testing intensity of this program which is caused by changing the procedure sorts form big to small: RTI_A > RTI_B, RTI_C > RTI_D.

5) test termination and code modifies

Up to now, application approach of SPM model to software testing has not been widely put into practice. Since the sufficient experiment data can not be achieved, it may be impossible to choose a T value as testability threshold (M). It will be difficult for us to make a judgment whether testability value of this program meets the requirements.

C. Results Summarization

<table>
<thead>
<tr>
<th>TABLE V APPLICATION RESULTS SUMMARIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantitative Application</td>
</tr>
<tr>
<td>The minimum number of test case for covering conditions and judgment paths is 5.</td>
</tr>
<tr>
<td>Regression intensity caused by changing the program: RTI_A &gt; RTI_B, RTI_C &gt; RTI_D.</td>
</tr>
<tr>
<td>Qualitative Application</td>
</tr>
<tr>
<td>The importance of code reviews: ICR_D &gt; ICR_C &gt; ICR_B.</td>
</tr>
<tr>
<td>The testing sequence of the program: D --&gt; B --&gt; A --&gt; C.</td>
</tr>
<tr>
<td>The probability of performance bottleneck sorts from great to little: A --&gt; B --&gt; C --&gt; D.</td>
</tr>
<tr>
<td>The importance of the statement sorts from big to small: A --&gt; D --&gt; C --&gt; B.</td>
</tr>
<tr>
<td>Path coverage sequence after optimizing the program: judgment path --&gt; conditions path --&gt; statement path.</td>
</tr>
</tbody>
</table>

IV. CONCLUSIONS

Towards the conflict in software testing between the limited resources and the software testing efficiency, this paper researches emphatically execution probability, infection probability, propagation probability and testability value of SPM model and put forward the corresponding application approach to the software testing. By combing with some procedure, it is proved that execution probability can help
testers determine the minimum number of test case, propagation probability can help testers judge the difficulty degree of regression testing and analysis the program performance bottleneck, and infection probability and testability value can help testers analyze the importance of the testing and optimize testing sequence. Thus, before the software testing, it is of guidance on the software testing to conduct the testability measurement of the tested software, which can be widely applied to the practice.

Although SPM model is a static analysis approach, it possesses specific algorithm. It is feasible to generate the corresponding tools or plug-ins of the testability of software. Furthermore, the testability measurement of software is developing in the direction of object-oriented. It will cost further research on how to generate the corresponding tools or plug-ins and how to transform the application approach this paper putting forward into the direction of object-oriented.

REFERENCES

[1] Shih, M.C., “Verification and measurement of software component testability”, San Jose State University, in press.


