



Efficient Test Suite Management

cont ...

Prof. Durga Prasad Mohapatra
Professor
Dept. of CSE, NIT Rourkela

Risk-Based Prioritization

- It is a well defined process that prioritizes modules for testing.
- Uses risk analysis to highlight potential problem areas, whose failure have adverse consequences.
- Testers use this risk analysis to select the most crucial tests.

Risk-Based Prioritization cont ...

- This technique is used to prioritize the test cases based on *some potential problems* which may occur during the project. It uses:
 - **Probability of occurrence/fault likelihood:** It indicates the probability of occurrence of a problem.
 - **Severity of impact/failure impact:** If the problem has occurred, how much impact does it have on the software.

Risk-Based Prioritization cont ...

- Risk analysis uses these two components by first listing the potential problems and then, assigning a probability and severity value for each identified problem, as shown in next Table.
- By ranking the results in this table in the form of risk exposure, testers can identify the potential against which the software needs to be tested and executed first.

Risk-Based Prioritization cont ...

A risk analysis table consists of the following columns:

- **Problem ID:** A unique identifier to facilitate referring to a risk factor.
- **Potential problem:** Brief description of the problem.
- **Uncertainty factor:** It is the probability of occurrence of the problem. Probability values are on a scale of 1(low) to 10(high).
- **Severity of impact:** Severity values on a scale of 1(low) to 10(high).
- **Risk exposure:** Product of probability of occurrence and severity of impact.

Table I .A sample risk analysis table

Problem ID	Potential Problem	Uncertainty Factor	Risk Impact	Risk Exposure
P1	Specification ambiguity	2	3	6
P2	Interface Problem	5	6	30
P3	File corruption	6	4	24
P4	Databases not synchronized	8	7	56
P5	Unavailability of modules for integration	9	10	90



Inference from the table

The problems / modules given in the previous table can be prioritized in the order of P5, P4, P2, P3, P1, based on the risk exposure values.

Prioritization Based on Operational Profiles

- In this approach, the test planning is done based on the **operational profiles** of the important functions which are of use to the customer.
- An **operational profile** is a set of tasks performed by the system and their probabilities of occurrence.
- After estimating the operational profiles, testers decide the total number of test cases, keeping in view the costs and resource constraints.

Prioritization using Slices

- During regression testing, the modified program is executed on all existing regression test cases to check that it still works the same way as the original program, except where a change is expected.
- But re-running the test suite for every change in the software makes regression testing a time-consuming process.

Prioritization using Slices cont ...

- If we can find the portion of the software which has been affected with the change in software, then we can prioritize the test cases based on this information.
- This is called the **slicing technique**.

Execution Slice

- The set of statements executed under a test case is called the **execution slice** of the program.
- Please refer to the following program.
- Table 2 shows the test cases for the given program.

```

Begin
S1: read (basic, empid);
S2: gross = 0;
S3: if (basic > 5000 || empid > 0)
    {
S4:     da = (basic*30)/100;
S5:     gross = basic + da;
    }
S6: else
    {
S7:     da = (basic*15)/100;
S8:     gross = basic + da;
    }
S9: print (gross, empid);
End

```

Fig 1. Example program for execution

Table 2 Test cases

Test Case	Basic	Empid	Gross	Empid
T1	8000	100	10400	100
T2	2000	20	2300	20
T3	10000	0	13000	0

Prioritization using Slices cont ...

- T1 and T2 produce correct results.
- On the other hand, T3 produces an incorrect result.
- Syntactically, it is correct, but an employee with the empid '0' will not get any salary, even if his basic salary is read as input.
- So it has to be modified.

Prioritization using Slices cont ...

- Suppose S3 is modified as `[if(basic>5000 && empid>0)]`.
- So, for T1, T2, and T3, the program would be rerun to validate whether the change in S3 has introduced new bugs or not.
- But, if there is a change in S7, `[da = (basic*25)/100; instead of da = (basic*15)/100;]`, then only T2 will be rerun.

Prioritization using Slices cont ...

- So in the execution slice, we will have less number of statements.
- The execution slice is highlighted in the given code segment.

Example

Begin

S1: read (basic, empid);

S2: gross=0;

S3: if(basic > 5000 || empid > 0)

{

S4: da = (basic*30)/100;

S5: gross = basic + da;

}

S6: else

{

S7: da = (basic*15)/100;

S8: gross = basic + da;

}

S9: print(gross, empid);

End

Dynamic Slice

The set of statements under a test case having an effect on the program output is called the **dynamic slice** of the program with respect to the out-put variables.

Example

Begin

S1: read (a,b);

S2: sum=0;

S2.1: I=0;

S3: if (a==0)

{

S4: print(b);

S5: sum+=b;

}

S6: else if(b==0)

{

S7: print(a);

S8: sum+=a;

}

S9: else

{

S10: sum=a+b+sum;

S10.1 I=25;

S10.2 print(I);

}

S11:endif

S12:print(sum);

End

Test Cases

Table 3: Test cases for the given program.

Test Case	a	b	sum
T1	0	4	4
T2	67	0	67
T3	23	23	46

Dynamic Slice cont ...

- T1, T2 and T3 will run correctly but if some modification is done in S10.1 [say $l = 50$], then this change will not affect the output variable.
- So, there is no need to rerun any of the test cases.
- On the other hand, if S10 is changed [say, $\text{sum} = a * b + \text{sum}$], then this change will affect the output variable 'sum'.
- So there is a need to rerun T3.
- The dynamic slice is highlighted in the code segment (S1, S2, S10, S12).

Begin

S1: read (a,b);

S2: sum=0;

S2.1: I=0;

S3: if (a==0)

{

S4: print(b);

S5: sum+=b;

}

S6: else if(b==0)

{

S7: print(a);

S8: sum+=a;

}

S9: else

{

S10: sum=a+b+sum;

S10.1 I=25;

S10.2 print(I);

}

S11:endif

S12:print(sum);

End

Relevant Slice

- The set of statements that were executed under a test case and did not affect the output, but have the potential to affect the output produced by a test case, is known as the **relevant slice** of the program.
- For example, consider the example given in previous figure.

Relevant Slice cont ...

- Statements S3 and S6 have the potential to affect the output, if modified.
- On the basis of relevant slices, we can prioritize the test cases.
- This technique is helpful for prioritizing the regression test suite which saves time and effort for regression testing.

Prioritization Based on Requirements

- This technique is used for prioritization of the system test cases.
- The system test cases also become too large in number, as this testing is performed on many grounds. Since system test cases are largely dependent on the requirements, **the requirements** can be analysed to prioritize the test cases.
- This technique does not consider all the requirements on the same level. Some requirements are more important and critical as compared to others, and these test cases having more weight are executed earlier.

PORT

- Hema srikanth et al. have proposed a requirements based test case prioritization technique.
- It is known as PORT (prioritization of requirements for test).
- They have taken the following four factors for analyzing and measuring the criticality of requirements.

Factors for analyzing & measuring criticality of requirements.

- **Customer-assigned priority of requirements:** Based on priority, the customer assigns a weight (on scale of 1 to 10) to each requirement.
- **Requirement volatility:** This is a rating based on the frequency of change of a requirement.
- **Developer-perceived implementation complexity:** The developer gives more weight to a requirement which he thinks is more difficult to implement.
- **Fault proneness of requirements:** This factor is identified based on the previous versions of system. If a requirement in an earlier version of the system has more bugs, i.e. it is error-prone, then this requirement in the current version is given more weight. This factor cannot be considered for a new software.

Prioritization Factor Value

- Based on these four factor values, a prioritization factor value (PFV) is computed as given below.

$$PFV_i = \sum (FV_{ij} \times FW_j)$$

where FV_{ij} = Factor value which is the value of factor j corresponding to requirement i , and FW_j = Factor weight which is the weight given to factor j .

- PFV is then used to produce a prioritized list of system test cases.

Measuring Effectiveness of a prioritized test suite

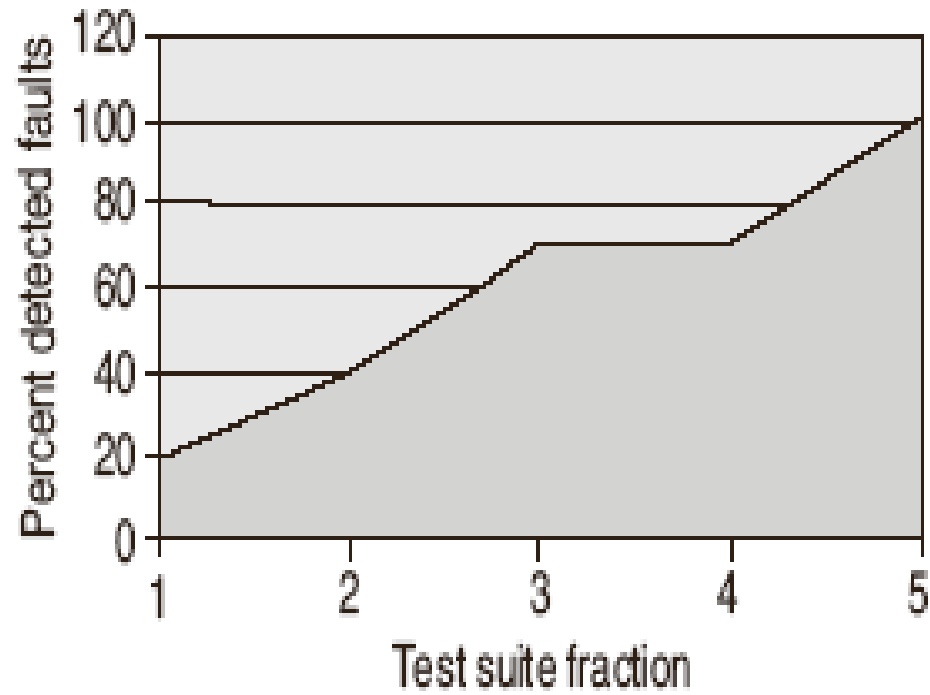
- Elbaum et al. developed **APFD** (average percentage of faults detection) metric that measures the weighted average of percentage of faults detected during the execution of a test suite.
- Its value ranges from 0 to 100, where a higher value means a faster fault-detection rate.

APFD Metric

APFD is a metric to detect how quickly a test suite identifies the faults. It is defined as follows:

$$APFD = 1 - ((TF_1 + TF_2 + \dots + TF_m) / nm) + 1/2n$$

where TF_i is the position of the first test in test suite T that exposes fault i , m is the total number of faults exposed in the system or module under T and n is the total number of test cases in T .




Example

Consider a program with 10 faults & test suite of 10 test cases, as shown in below table.

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1					x			X		
F2		X	X	X		X				
F3	X	X	X	x			x	X		
F4						X				x
F5	X		X		X	X		x		X
F6					X	x	x		X	
F7									X	
F8		x		X		x			X	X
F9	X									X
F10			X	x					X	X

- Let us consider the order of test suite as (T1,T2,T3,T4,T5,T6,T7,T8,T9,T10).
- Then calculate the APFD for this program as follows:

$$\begin{aligned} \text{APFD} &= 1 - (5+2+1+6+1+5+9+2+1+3)/(10*10) + \frac{1}{2 \times 10} \\ &= 0.65 + 0.05 \\ &= 0.7 \end{aligned}$$

- 
- All the bugs detected are not of the same level of severity.
 - One bug may be more critical as compared to others.
 - Moreover, the cost of executing the test cases also differs.
 - One test case may take more time as compared to others.
 - Thus, APFD does not consider the severity level of the bugs and the cost of executing the test cases in a test suite.

Cost-cognizant APFD

- Elbaum *et al.* modified their APFD metric and considered these two factors to form a new metric which is known as *cost-cognizant APFD* and denoted as **APFD_c**.
- In APFD_c, the total cost incurred in all the test cases is represented on x-axis and the total fault severity detected is taken on y-axis.
- Thus, it measures the unit of fault severity detected per unit test case cost.

Summary

- Discussed in detail the following types of test case prioritization techniques.
 - Risk-Based Prioritization
 - Prioritization Based on Operational Profiles
 - Prioritization using Relevant Slices
 - Prioritization Based on Requirements
- Discussed a metric, called **APFD**, to measure the effectiveness of the prioritization technique.
- Also, discussed a variance of APFD, called *cost-cognizant* APFD, which is denoted as **APFD_c**.



References

1. Naresh Chauhan, Software Testing: Principles and Practices, (Chapter – 12), Second Edition, Oxford University Press, 2016.



Thank you