Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

A Model of Software Testability Measurement

Jin-Cherng Lin, Szu-Wen Lin & Ian Ho
Dept. of Computer Science and Engineering
Tatung Institue of Technology
Taipei 10451, Taiwan, R.O.C
swlin061@ms3.hinet.net
Tel: (02)592-5252 ext 3295
Fax: (02) 755-1766

Abstract

Software testability is becoming an important factor to be considered during software development and assessment, especially for those critical softwares. This paper gives software testability, previously defined by Voas, a new model and measurement which is done before random black-box testing with repect to a particular input distribution. We also compared our measurement results with the one simulated according with Voas's model. It showed that our rough testability estimate provides enough information and will be used as guidelines for software development.

Keywords : software testability, error propagation, data state infection, location execution, domain/range ratio

1. Preliminary

It is believed that software industry is at a risk for a disaster or some kind, a disaster in which the blame will clearly lay to software, and all will be tarred with the brush used on the unlucky developer. Nowadays, many computer systems are used in critical applications such as spacecraft and defense systems. When lives and fortunes depend on software, software quality and its verification demand increased attention. But how can we access the acceptable safety and reliability of such critical software? When can we say that the software is reliable enough; testing can be stopped and the software can be released ?

In the past, much research on software testing has concentrated on methods for selecting effective sets of test data [Dem87], variously based on program specification, on program structure, or on hypotheses about likely faults. Testing tries to reveal software faults by executing the program and comparing the expected output with the one produced so that it can guarantee correctness. Much effort has been put in to answer the question: "what is the probability that the program will fail?" However, software testing can not show the absolute absence of failure unless it is exhaustive [DDH72]. Therefore, if testing is to be effective at all, then non-exhaustive testing must be performed in a manner that offers some degree of confidence that the code is reliable. Here, we ask another question: "what is the probability that the program will fail, if it is faulty?" We present another technique, software testability analysis, that will help developers make the assessment. We focus on some program characteristics that make faults hard to find with random black-box testing.

Assessing a program's testability, T, is a hard task, and much research has explored this problem [Voas92b, MMNP92, Hami87]. However, the analyses of testability in the past were done after testing, especially with random black-box testing. This paper suggests another analysis technique to gain the testability of a software by checking the source code instead of testing it. Doing so, it can not only reduce the effort and cost at testing step, but also give a direction for debugging because low testability implies that faults are more easily hidden during testing.

2. Related works

2.1. Fault-Failure model

Any assessment of testability must be based on a model of the fault/failure relationship. The following simple model of this relationship has been proposed in [More84, VM96]:

> Each textual location in a program is considered as a possible location of a fault. At a given location, a possible fault could result in a failure if and only if:
>
> 1. A fault must be executed.
> 2. The fault must affect the state of the program in a manner different than what the state of the program would have been had the fault not existed. This is termed as having an infection in the state.
> 3. The erroneous program state must propagate to an output state.

This model is very simplistic, because it imagines that faults at a single location; however, it can be used to define as practical approximation of testability.

If we confine the possibility of a program's failure to this model, the probability that a fault will turn into a failure will be the product of execution probability, infection probability and propagation probability. Furthermore, a lower bound on testability of a program is obtained as the least product above over all locations in the program.

2.2. PIE

PIE--propagation analysis, infection analysis, execution analysis--is a dynamic technique for statistically estimating the effects that a location of a program has on the program's computational behavior. PIE analysis collects information concerning the semantics of faults. It does not reveal the existence of faults, nor does the technique directly evaluate the ability of inputs to reveal the existence of faults. Instead, it identifies locations in a program where faults, if they do exist, are more likely to remain undetected during testing. The technique is based on fault/failure model and it estimates the probability of the three characteristics of a location:

1. the probability that the location is executed on inputs selected from the assumed input distribution of the software.
2. the probability that if a mutant exists at this location, it will adversely change the data states;
3. the probability that if the data state is adversely changed that that will propagate to the output.

The following is the detailed definition for PIE analysis [Voas92b, VM95] : Let S denote a specification, P denote an implementation of S, x denote a program input, Δ denote the set of all possible inputs to P, D denote the probability distribution of Δ, l denote a program location in P. Then

> $\hat{\varepsilon}_{lPD}$ is the proportion of inputs (selected according to D) that causes location l to be executed.

Let $1 \le y \le z_l$, M_l represents a set of z_l mutants of location l :$\{m_{l1}, m_{l2}, \ldots m_{lz_l}\}$ (where $1 \le y \le z_l$) [Howd82].

> $\hat{\lambda}_{mlylpD}$ of mutant m_{ly} is the probability that the succeeding data state of location l is different than the succeeding data state that mutant m_{ly} creates.
>
> $\hat{\psi}_{alPD}$ for a simulated infection affecting variable a in the data state succeeding location l (where this data state is created by a randomly selected input x according to D) is the probability that P's output differs (from that would normally be produced) after execution is resumed using the simulated infection.

Let θ_l denote the testability of some location l, then

$$\theta_l = \varepsilon_{lPD} * \sigma\left[\left(\min_{my}\left(\lambda_{m_{ly}}\hat{}_{lPD}\right), \min_{a}\left(\hat{\psi}_{alPD}\right)\right)\right]$$

where

$$\sigma(a,b) = \begin{cases} a-(1-b) & \text{if } a-(1-b) > 0 \\ 0 & \text{otherwise} \end{cases}$$

2.3. PISCES

PISCES is the commercial software testability tool which is written in C++ and operates on programs written in C. It implements the PIE technique defined in [VMM91, Voas92b]. PISCES produces

testability estimates by creating an "instrumented" copy of the program and then compiling and executing the instrumented copy, which is about 10 times as large as the original source code, with inputs that are either supplied in a file or PISCES uses random distributions from which it generates inputs.

If PISCES is performed in its entirety, we will have:

$\hat{\varepsilon}_l$: the estimate of the probability that program location l is executed.

$\left\{\hat{\lambda}_{l,p1}, \hat{\lambda}_{l,p2}, \ldots\right\}$: the estimates of the probabilities, one estimate for each mutant in $\{p_1, p_2, \ldots\}$ at program location l, that given the program is executed, the mutant will adversely affect the program state

$\left\{\hat{\psi}_{l,a1}, \hat{\psi}_{l,a2}, \ldots\right\}$: the estimates of the probabilities, one estimate for each live variable in $\{a_1, a_2, \ldots\}$ at program location l, that given that the live variable in the program state following the program location is perturbed.

Then, a PISCES testability postprocessor inputs all probability estimates and allows the user several choices of how the testability will be displayed: either for a location, a module or for the entire program.

The value of this tool to software quality is two-fold: improved testing, and improved debugging [VMP92].

2.4. DRR

The domain/range ratio (DRR) of a specification denoted by $\alpha : \beta$ [VM93] is the ratio between the cardinality of the domain to the cardinality of the range. It is one of the important factors in determining whether the software is likely to hide faults. For example, f(x)=2x corresponds to a DRR of $\infty_I : \infty_I$ (the symbol ∞_I denoted the cardinality of integers) . The specification f(x) has only one possible input for x for any output f(x). It is a one-to-one function, which implies high testability. The function f(x)=sqr(x), however, mandates a loss of information because both x=5 and x=-5 produce the same internal state after execution f(x). It may be that the negative integer could signal a problem with the software, but f(x) will erase that information in the output. Previous research [VM91] has also suggested that many-to-one computable function tends to have lower testability since the phenomenon--called internal state collapse--that its internal state is not communicated in its output always occurs. By paying attention to the DRRs, we can potentially give a closer insight for the estimate of testabilities, especially for infection estimate.

3. Software testability analysis

Software testability analysis measures the benefit produced by a software testing scheme to a particular program. There are several ways to define software testability [BS95]. For instance, it is regarded to as the ease with which some input selection criteria can be satisfied during testing. Thus, a program is said to have "low testability" if it is difficult to find inputs that can satisfy a particular structural coverage criterion (Ex: branch testing) for the given program. Another view of software testability defines it as a prediction of the probability that existing faults will be revealed during testing given an arbitrary input selection criterion C [Voas92b]. The former definition focuses on syntactic features, whereas the latter, semantic analysis. In either definition, software testability analysis result is a function of a (*program, input selection criteria*) pair [Voas94]. Therefore, a program may have varying T when presented with varying input selection criteria. Moreover, because T refers to a probability, testability is bounded in a closed interval [0,1].

3.1. Terminology

A ***failure*** is the event where a program's output is not compliment with the specification [BS95]. Using the definition in [Lap92], a failure occurs because the program enters an erroneous state. Hence, an ***error*** is a part of the state that can lead to a failure. A ***fault*** in a location is the cause of an error. Here, we define a ***location*** as a single instruction--an assignment, input/output statement, a <condition> part of *if* or *while* statement and procedure calls. As for the definition of "***software testability***," we use the one provided in the early 90's Voas [VMM91, Voas92b] : a prediction of the probability of software failure occurring if the software were to contain a fault, given that software execution is with respect to a particular

input distribution during random black-box testing. It consists of three estimates, similar to PIE [Voas92] at each location:

Execution estimate: The likelihood that the location is executed on inputs selected from the assumed input distribution of the software.

Infection estimate: The likelihood that if a mutant exits at certain location, it will change the data state.

Propagation estimate: The likelihood that if the data state has been changed, the change will propagate to the output.

For any test case to reveal a fault at a specified location, execution, infection, propagation must occur; without these three events occurring, the execution will not result in failure. Thus for a specific fault, the product of the estimates of these three events occurring yields an estimate of the probability of failure that would occur when this location contains a fault. This is the testability of that location. We then take the location with the lowest non-zero testability to be the testability overall program.

3.2. The model of testability measurement

We determine testability by the code's structure and semantics, and by an assumed input distribution. Thus two programs with the same function may have different testabilities Since a fault can lie anywhere in a program, we should take all places in the source code into consideration when estimating testability.

In the paper, we consider that the fault might exist at any location in the program. We have purposely limited the faults to single faults to avoid the explosion that occurs in the number of combinatorial changes that could be made at each location. Furthermore, the faults considered in our research are limited to faults of arithmetic expressions and predicates. For arithmetic expressions, the faults considered are limited to single changes to a location--this is similar to the mutations used in mutation testing [Howd82]. For assignment predicates, the faults include: (1) A wrong variable/constant substitution, (2) a variable substituted for a constant, (3) a constant substituted for a variable, (4) a wrong operator. For boolean predicates, the faults considered included: (1) wrong variable/constant substitution, (2) a wrong equality/inequality operator substitution, (3) exchanging *and* and *or*.

Estimate for "E"

Case 1 : For any sequential statement, E =1.

Case 2 : For any *if* or *while* statement, E is still 1.

Case 3 : For the statement in execution body of *if* or *while* :

if there is some clear domain/range ratio for the <control> part of *if* or *while*, we can estimate the execution rate as domain/range ratio or (1-domain/range ratio), depending on whether location l is in *yes* or *no* execution body of if or while.

if there isn't, then we give a weight of certain branch according to the graphic theorem: the number of branches on data flow diagram at location l

Since there might be a lot of paths from the program beginning to a specified location, and the program structure might be complicated--there might be nested branches, we conclude that the estimate for "E" as follows :

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

$E = \sum_{i=1}^{n} E_{p_i}$ where E_{p_i} is the execution rate of a specified location on the i-th path

$E_{p_i} = \prod_{j=0}^{k} e_j$ where e_0=1,

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

See appendix for detailed calculation.

Estimate for "I"

The estimate for "I" is much related with the number of tokens--operands and operators--in a location. If location l is an assignment, we only consider the right-side of the assignment because any incorrect variable substation on the left-side is hard to predict and control. Furthermore, we treat the constants as variables and their I_{l_opd}=1. The following is the calculation of I for location l :

- If location l is an input/output statement, we give every input/output variable a unique infection estimate. For example: if

the location is a statement like "read(a,b,c)", then it will have three infection estimates:

$$I_a = I_b = I_c = 1$$

Others :

$$I_l = \frac{1}{n}\left(\sum_{i=1}^{op} I_{l\_opr_j} + \sum_{j=1}^{od} I_{l\_opd_j}\right)$$

op : 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.

od : the number of the operands on the right-side of the assignment, boolean predicate, or in the <condition> part of *if* or *while* statements.

n : the number of tokens; op+od=n.

$I_{l\_opr_i}$: the infection weight, I_w, of the i-th operator or built-in function See Table 1 for I_w of some operators and built-in functions.

operator, built-in function	I_w
+,—,×,/	1
>,<,=,≦,≠,≧	DRR or 1
a mod b (b is a constant)	(b-1)/b
a div b (b is a constant)	([a/b]-1) / [a/b]
others (ex. trunc, round, div, mod, even,...)	1/2

Table 1. I_w for some operators and built-in fuctions

$$I_{l\_opd_j} : \begin{cases} \text{variable} & I_{l'} \\ \text{constant} & 1 \end{cases}$$

$I_{l'}$: the infection estimate at some location l', where the j-th operand is lastly defined.

Also see appendix for detailed calculation

Estimate for "P"

The estimate for P here is much different from the one defined by Voas. Voas considered all of the input variables and the variables which were assigned during the program. They were all treated as perturbed variables. According to the definition provided in [Voas92b], the propagation probability for location l is the minimum value among the probabilities got by perturbing all the perturbed variables. In our research, however, we only considered the variable *used* in the <condition> part of *if* or *while* statement or the definition of variable v at location l as the perturbed variable. We think that it will have much affect on the propagation analysis at location l .

Here we would like to define ***use*** or ***definition*** of variable v first :

A *use* of variable v is an instruction X in which this variable is referenced. A use can be in a test instruction (the <condition> part of *if* or *while* statement), on the right-side of an assignment instruction or an output instruction.

A *definition* of variable v is an instruction X which assigns a value to that variable. A definition can be an assignment instruction or input instruction.

Now the algorithm for estimating P is as follows :

1. For a specified location l , find the variable v, which is defined.
2. From location l', find location l', where v is used after it is executed at location l . And l', is the new specified location.
3. Get $I_{l'}$ of location

Repeat step 1, 2, 3 until we reach the output statement.

If we can't reach the output statement, then

$I_{l'}=0$

else $P_l = \prod I_{l'}$

See appendix for detailed calculation

Loop, while fepeat

We treat the whole loop as a single statement. We give the whole loop a testability score which is the smallest testability score among the ones for every statement in the body of the loop.

If the loop is finite, no matter how many times the loop will iterate, the testability score of the loop is estimated by running the loop body once because we can extend the loop like a series of sequential statements.

4. Performance discussion

We have implemented our new model of measurement to a simple program. The following is the source code of the simple program, and Table 2 is our measurement for the program's testability compared with those based

on Voas's model. Every location has two results. The first row of the result is gained by simulating Voas's algorithm, whereas the second row is our measurement based on our model.

```
Location No.
    1    read(a,b,c);
    4    if a <>0 then begin
    5         d:=b*b-4*a*c;
    6         if d<0 then
    7              x:=0
              else
    8              x:=(-b+trunc(sqrt(d)))
                   div (2*a)
              end
              else
    9         x:=-c div b;
    10        writeln(x);
```

From Table 2, we can find that the testability derived from our model is almost close to the one got by simulating Vaos's PIE model. Since a program's testability is depend on its input data distribution, our results are still accurate when comparing with the results based on Voas's model.

Please note that it doesn't make sense to say which model is better only according to the higher testability it gets because a program testability is related with its input data distribution and neither Voas' model nor our model proposed is exhaustive testing in Table 2.

As for the application of testability, other papers showed that testability can help us make decisions about debug testing [VMM91] and make predictions about reliability [VM92]. Furthermore, the measures of testability which is proposed in this paper can be used to draw inferences on program correctness because a program has passed a certain number of tests without failing--with a high value of testability and implies a high probability that the program is correct.

Location No.	Fault	E	I	P	T
5	b*b-5*a*c	0.9090	0.8264	0.9009	0.6767
		0.9090	0.9090	0.7664	0.6333
5	b*b-4*c*c	0.9090	0.7438	0.9009	0.6091
		0.9090	0.9173	0.7665	0.6391
5	c*c-4*a*c	0.9090	0.9083	0.9009	0.7438
		0.9090	0.9000	0.7847	0.6420
5	b*b+4*a*c	0.9090	0.8264	0.9009	0.6768
		0.9090	0.9090	0.7664	0.6333
8	x:=-a+trunc(sqrt(d)) div (2*a)	0.9014	0.9082	0.8186	0.6701
		0.9996	0.8106	0.8106	0.6558

Table 2 Testability Estimates

5. Concluding remarks

Software testability is becoming an important factor to consider during the software development and assessment. We contend that the preliminary results of our software testability analysis are sufficient to motivate additional research into quantify software testability analysis. Not only do we think that this technique may help assess critical systems but it is also acknowledged as a technique that should be further explored for its enormous impact on assessing ultra-reliable software.

Though our testability analysis is somewhat time-consuming, we have felt that our new model of measurement is worth to make a prediction for software testability without any testing. Our future research will focus on empirically exploring different strategies. We expect that the model of measurement will be implemented as a tool for further testability predictions.

Reference

[BS95] Antonia Bertolino, Lorenzo Strigini, "Using Testability Measures for Dependability Assessment," IEEE 17TH International Conference on Software Engineering, 1995.

[DDH72] O. J. Dahl, E. W. Dijsktra, and C. A. R. Hoare, "Structured Programming," Academic Press, 1972.

[Demi87] R.A. Demillo, et al., Software Testing and Evaluation, The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1987

[Hami87] Richard G. Hamilet, "Probable Correctness Theory," Information Processing Letters, pp. 17-25, April 1987

[Howd82] William E. Howden, "Weak Mutation Testing and Completeness of Test Sets," IEEE Trans. On Software Engineering, SE-8(4):371-379, July 1982.

[MMNP92] K. Miller, L. Morell, R. Noonan, S. Park, D. Nicol, B. Murrill, & J. Voas, "Estimating the Probability of Failure when Testing Reveals No Failures," IEEE Trans. on Softw. Eng., 18(1): pp. 33--44, Jan. 1992

[More84] Larry Joe Morell, "A Theory of Error-based Testing," Technical report TR-1395, University of Maryland, Department of Computer Science, April 1984.

[VM92] J. Voas & K. Miller, "Improving the Software Development Process Using Testability Research," In Proc. of the 3rd Int'l. Symp. on Softw. Reliability Engineering, pp. 114--121, October, 1992, RTP, NC, Publisher: IEEE Computer Society.

[VM93] J. Voas & K. Miller, "Semantic Metrics for Software Testability", The J. of Systems and Softw., Elsevier Science Publishers Ltd. 20: 207-216, March 1993

[VM95] J. Voas & K. Miller, "Software Testability: The New Verification," IEEE Software Appeared May 1995

[VM96] J. Voas & K. Miller, "Substituting Voas's Testability Measure for Musa's Fault Exposure Ratio," Proc. of the Int'l. Communication Conference, June 1996, Dallas, TX.

[VMM91] J. Voas, L. Morell, & K. Miller, "Predicting Where Faults Can Hide From Testing," IEEE Software, 8(2):41--47, March 1991.

[Voas92b] J. Voas, "PIE: A Dynamic Failure-Based Technique," IEEE Trans. on Software Engineer, 18(8): p.717--p.727, August 1992.

[Voas94] J. Voas, "Formal Testability Analysis," In the Encyclopedia of Softw. Engineering, John Wiley & Sons, pp. 517--518, 1994