Factors That Affect Software Testability

Jeffrey M. Voas
Systems Architecture Branch
Information Systems Division
Mail Stop 478
NASA Langley Research Center
Hampton, VA 23665
(804) 864-8136
jmvoas@phoebus.larc.nasa.gov

Abstract: *Software faults that infrequently affect software's output are dangerous. When a software fault causes frequent software failures, testing is likely to reveal the fault before the software is released; when the fault remains undetected during testing, it can cause disaster after the software is installed. A technique for predicting whether a particular piece of software is likely to reveal faults within itself during testing is found in [Voas91b]. A piece of software that is likely to reveal faults within itself during testing is said to have high testability. A piece of software that is not likely to reveal faults within itself during testing is said to have low testability. It is preferable to design software with higher testabilities from the outset, i.e., create software with as high of a degree of testability as possible to avoid the problems of having undetected faults that are associated with low testability.*

Information loss is a phenomenon that occurs during program execution that increases the likelihood that a fault will remain undetected. In this paper, I identify two broad classes of information loss, define them, and suggest ways of predicting the potential for information loss to occur. We do this in order to decrease the likelihood that faults will remain undetected during testing.

Index Terms: Testability, Domain/Range Ratio (DRR), random black-box testing, information loss, information hiding, software design, specification metric.

Jeffrey Voas is working as a National Research Council resident research associate at the National Aeronautics and Space Administration's Langley Research Center. His research interests include software testing, studying data state error propagation, debugging techniques, and design techniques for improving software testability. Voas received a BS in computer engineering from Tulane University and a MS and PhD in computer science from the College of William and Mary.

Factors That Affect Software Testability

1 Introduction

This paper exposes factors that I have observed which affect program testabilities. *Testability* of a program is a prediction of the tendency for failures to be observed during random black-box testing when faults **are** present [VOAS91b]. A program is said to have *high testability* if it tends to expose faults during random black-box testing, producing failures for most of the inputs that execute a fault. A program has *low testability* if it tends to protect faults from detection during random black-box testing, producing correct output for most inputs that execute a fault. In this paper, I purposely avoid a formal definition for fault because of the difficulty that occurs when trying to uniquely identifying faults, and instead use the intuitive notion of the term fault.

Random black-box testing is a software testing strategy in which inputs are chosen at random consistent with a particular input distribution; during this selection process, the program is treated as a black-box and is never viewed as the inputs are chosen. An *input distribution* is the distribution of probabilities that elements of the domain are selected. Once inputs are selected, the program is then executed on these inputs and the outputs are compared against the *correct* outputs.

Sensitivity analysis [VOAS91b] is a dynamic method that has been developed for predicting program testabilities. One characteristic of a program that must be predicted before sensitivity analysis is performed is whether the program is likely to *propagate* data state errors (if they are created) during execution. Propagation analysis [VOAS91b, VOAS91c] is a dynamic technique used for predicting this characteristic. If the results of propagation analysis suggest that the *cancellation* of data state errors is likely to occur if data state errors are created, then sensitivity analysis produces results predicting a lower testability than if cancellation of data state errors were unlikely to occur.

When all of the data state errors that are created during an execution are cancelled, program failure will not occur. If this occurs repeatedly, this produces an inflated confidence that the software is *correct.* It might seem desirable for a correct output to be produced regardless of how the program arrived at the correct output. This *is* the justification for fault-tolerant software. But for critical software, any undetected fault is undesirable, even if the data state error it produces is frequently cancelled. For critical software, we prefer correct output from correct programs, not correct output from incorrect programs. By the fact the program is incorrect, there exists at least one input on which program failure will occur, and by the fact the software is critical, the potential for a loss-of-life exists.

This paper presents empirical observations concerning a phenomenon that occurs during program execution; this phenomenon suggests the likelihood of data state error cancellation

occurring. The degree to which this phenomenon occurs can be quantified by static program analysis, inspection of a specification, or both. Note that this phenomenon can be quantified statically, which is far less expensive to perform than the dynamic propagation analysis. Thus through static program analysis or specification inspection, insight is acquired concerning the likelihood that data state error cancellation will occur. And this gives insight into whether faults will remain undetected during testing, i.e., program testability.

I term this phenomenon "information loss." *Information loss* occurs when internal information computed by the program during execution is not communicated in the program's output. Information loss increases the potential for the cancellation of data state errors and this decreases software testability. As mentioned, information loss can be observed by both static program analysis and inspection of a specification. I divide information loss into two broad classes: implicit information loss and explicit information loss. Static program analysis is used to quantify the degree of explicit information loss, and specification inspection quantifies the degree of implicit information loss.

Explicit information loss occurs when variables are not validated either during execution (by a self-test) or at execution termination as output. The occurrence of explicit information loss can be observed using a technique such as static data flow analysis [Korel87]. Explicit information loss frequently occurs as a result of information hiding [Parnas72], however there are other factors that can contribute to it. Information hiding is a design philosophy that does not allow information to leave modules that could potentially be misused by other modules. Information hiding is a good design philosophy; however, it is not necessarily good for testability, because the data in the local variables is lost upon exiting a module. In Section 3.3, I propose a scheme where information hiding is kept as a part of the software design philosophy while its negative effect, explicit information loss, is lessened.

Implicit information loss occurs when two or more different incoming parameters are presented to a user-defined function or a built-in operator and produce the same outgoing parameter. An example is the integer division computation **a := a div 2**. In the computation **a := a + 1**, there is no implicit information loss. In these two examples, the potential for implicit information loss occurring is observed by statically analyzing the code. If a specification states that ten floating-point variables are input to an implementation, and 2 boolean variables contain the implementation's output, then we know that implicit information loss will occur in an implementation of this specification. Thus specifications may also hint at some degree of the implicit information loss that will occur if they are written with enough information concerning their domains and ranges.

2 Information Loss

I have proposed two broad classes of information loss. The following pseudo-code example contains both types of information loss and demonstrates how we can statically observe where these two types of information loss occur. For this example, I assume inputs **a** and **c** have effectively infinite domains, and **z** has an effectively infinite domain immediately before the statement **z := z mod 23** is executed.

```
Module x(in-parameter a : real, in-parameter c : real,
         out-parameter  b : boolean)
```

```
local-parameters
  z : integer
  y : boolean

Beginning of Body
    :
    :
z := z mod 23
    :
    :
b := f(a,c,y,z)
End of Body
```

With the assumption of effectively infinite domains for **a**, **c**, and **z**, module **x** suffers from both implicit information loss and explicit information loss. Explicit information loss occurs in **x** as a result of its 2 local variables whose values are not output nor passed out. Implicit information loss can be observed in several ways. The first way is the impossibility of taking **b**'s value at module termination and discovering the values of **a** and **c** that were originally passed in; infinitely many combinations of **a** and **c** map to a particular **b**. This potentially could have been observed from the specification of the module. The second way implicit information loss occurs is at the statement containing the **mod** operator; implicit information loss occurs because of the assumption that **z** has an effectively infinite domain. Many values of **z** map to a particular value in [0..22] after the computation.

2.1 Implicit Information Loss

Clues suggesting some degree of the implicit information loss that may occur during execution may be visible from the program's specification; I use a specification metric termed the "domain/range ratio" for suggesting a degree of implicit information loss [VOAS91a]. Recall that in the example we were also able to observe implicit information loss by code inspection. Therefore, a specification's domain/range ratio only suggests a portion of the implicit information loss that may occur; code inspection can give additional information concerning implicit information loss.

The *domain/range ratio* (DRR) of a specification is the ratio between the cardinality of the domain of the specification to the cardinality of the range of the specification. I denote a DRR by $\alpha : \beta$, where α is the cardinality of the domain, and β is the cardinality of the range. As previously stated, this ratio will not always be visible from a specification. After all, there are specifications whose ranges are not known until programs are written to implement the specifications. And if the program is incorrect, an incorrect DRR will probably be calculated.

DRRs roughly predict a degree of implicit information loss. Generally as the DRR increases for a specification, the potential for implicit information loss occurring within the implementation increases. When α is greater than β, previous research has suggested that faults are more likely to remain undetected (if any exist) during testing than when $\alpha = \beta$ [VOAS91a].

Figure 1: There are four potential values for variable a and four potential values for variable b, for a total of 16 pairs of potential inputs. Notice that for these 16 inputs, integer division always produces the same output (1), and real division produces 16 unique outputs.

The granularity of a specification (or functional description) for which we can determine a DRR varies. For example, DRRs exist for unary operators, binary operators, complex expressions, subspecifications, or specifications. (By *subspecification*, I mean a specification for what will become a *module*.) In the example, the DRR of subspecification x is $(\infty_R)^2 : 2$ and $\infty_I : 23$ for the **mod** operator. In this paper, the symbol ∞_I denotes the cardinality of the integers, and ∞_R denotes the cardinality of the reals.

For certain specifications, the inputs can be found from the outputs by inverting the specification. For example, for an infinite domain, the specification $f(x) = 2x$ has only one possible input x for any output $f(x)$. Other specifications, for example $f(x) = \tan(x)$, can have many different x values that result in an identical $f(x)$; i.e., $\tan^{-1}(x)$ is not a one-to-one function. All inverted specifications that do not produce exactly one element of the domain for each element of the range lose information that uniquely identifies the input given an output. Restated, many-to-one specifications mandate a loss of information; one-to-one specifications do not. This is another way of viewing implicit information loss.

When implicit information loss occurs, you run a risk that the lost information may have included evidence of incorrect data states. Since such evidence is not visible in the output, the probability of observing a failure during testing is somewhat reduced. The degree to which it is reduced depends on whether the incorrect information is isolated to bits in the data state that are not lost and are eventually released as output. As the probability of observing a failure decreases, the probability of undetected faults existing increases.

Another researcher who has apparently come to a similar conclusion concerning the relationship between faults remaining undetected and the type of function containing the fault is Marick [Marick90]. While performing mutation testing experiments with boolean functions, Marick [Marick90] noted that faults in boolean functions (where the cardinality of the range is of course 2) were more apt to be undetected. Boolean functions have a great degree of

	Function	Implicit Information Loss	DRR	Comment
1	$f(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$	yes	$\infty_I : \infty_I/2$	a is integer
2	$f(a) = a + 1$	no	$\infty_I : \infty_I$	a is integer
3	$f(a) = a \bmod b$	yes	$\infty_I : b$	testability decreases as b decreases
4	$f(a) = a \text{ div } b$	yes	$\infty_I : \infty_I/b$	testability decreases as b increases, $b \neq 0$
5	$f(a) = \text{trunc}(a)$	yes	$\infty_R : \infty_I$	a is real
6	$f(a) = \text{round}(a)$	yes	$\infty_R : \infty_I$	a is real
7	$f(a) = \text{sqr}(a)$	no	$2 \cdot \infty_R : \infty_R$	a is real
8	$f(a) = \text{sqrt}(a)$	no	$\infty_R : \infty_R/2$	a is real
9	$f(a) = a/b$	no	$\infty_R : \infty_R$	a is real, $b \neq 0$
10	$f(a) = a - 1$	no	$\infty_I : \infty_I$	a is integer
11	$f(a) = \text{even(a)}$	yes	$\infty_I : 2$	a is integer
12	$f(a) = \sin(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
13	$f(a) = \tan(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
14	$f(a) = \cos(a)$	yes	$\infty_I : 360$	a is integer (degrees), $a \geq 0$
15	$f(a) = \text{odd(a)}$	yes	$\infty_I : 2$	a is integer
16	$f(a) = \text{not(a)}$	no	$1 : 1$	a is boolean

Table 1: DRRs and implicit information loss of various functions.

implicit information loss. This result compliments the idea that testability and the DRR are correlated. Additional evidence that correlation exists between implicit information loss and testability is currently being collected.

2.1.1 Correlating Implicit Information Loss and the DRR

Implicit information loss is common in many of the built-in operators of modern programming languages. Operators such as **div**, **mod**, and **trunc** have high DRRs.

Table 1 contains a set of functions with generalized degrees of implicit information loss and DRRs. A function classified as having a *yes* for implicit information loss in Table 1 is more likely to receive an altered incoming parameter and still produce identical output as if the original incoming parameter were used; a function classified as having *no* implicit information loss in Table 1 is one that if given an altered incoming parameter would produce altered output. A *yes* in Table 1 suggests data state error cancellation would occur; a *no* suggests data state error cancellation would not occur. In Table 1, all references to b assume it be a constant for simplicity. This way we only have to deal with the domain of a single input, instead of the domain of a 2-tuple input. The infinities in the table are mathematical entities, but for any computer environment they will represent the cardinality of fixed length number representations of finite size.

Instead of describing the generalizations made concerning implicit information loss for each element of Table 1, Figure 1 illustrates the relationship between implicit information loss and the DRR. In Figure 1, we have 16 (a,b) input pairs that are presented to 2 functions: one performs real division, the other performs integer division. For the real division function there are 16 unique outputs, and for the integer division function there is one output. This example shows how the differences in the DRRs of these two forms of division are correlated to different amounts of information loss.

2.2 Explicit Information Loss

Explicit information loss is not predicted by a DRR as implicit information loss is. Recall that explicit information loss is observed through code inspection, whereas the potential for implicit information loss can be predicted from functional descriptions or code inspection. Explicit information loss may also be observable from a design document depending on its level of detail. Explicit information loss is more dependent on how the software is designed, and less dependent on the specification's (input, output) pairs.

2.2.1 Observability

Integrated circuit design engineers have a notion similar to explicit information loss that they term "observability." *Observability* is the ability to view the value of a particular node that is embedded in a circuit [Markowitz88]. When explicit information loss occurs in software, you loose the ability to see information in the local variables. So in this sense, greater amounts of explicit information loss in software is a parallel to lower observability in circuits.

Discussing the observability of integrated circuits, [Berglund79] states that the principal obstacle in testing large-scale integrated circuits is the inaccessibility of the internal signals. One method used for increasing observability in integrated circuits design is to increase the pin count of a chip, allowing the extra pins to carry out additional internal signals that can be checked during testing. These output pins increase observability by increasing the range of potential bit strings from the chip. In Section 3.3, I propose applying a similar notion to increasing the pin count during software testing—increasing the amount of data state information that is checked during testing. Another method used for increasing observability is inserting internal probes to trap internal signals; Section 3.3 also proposes a similar technique by self-testing internal computations during execution.

3 Design Heuristics

Section 3 presents several strategies for lessening the effects of information loss. Section 3.1 describes benefits gained during program validation if specifications are decomposed in a manner that lessens the effect of implicit information loss at the specification level. Section 3.2 describes a way of lessening the effect of implicit information loss at the implementation level. Section 3.3 describes ways of lessening the effects of explicit information loss.

3.1 Specification Decomposition: Isolating Implicit Information Loss

Although the DRR of a specification is *fixed* and cannot be modified without changing the specification itself, there are ways of decomposing a specification that lessen the potential of data state error cancellation occurring across modules. During specification decomposition, you have hands-on control of the DRR of each subfunction. With this, you gain an intuitive feeling (before a subfunction is implemented) for the degree of testing needed for a particular confidence that a module is propagating data state errors. The rule-of-thumb that guides this intuitive feeling is: "the greater the DRR, the more testing needed to overcome the potential for data state error cancellation occurring."

Section 3.1 presents a benefit that can be gained for testing purposes by using a specification's DRR during design. During a design, a specification is decomposed in a manner such that the program's modules are designed to either have a high DRR or a low DRR. By isolating modules that are more likely to propagate incoming data state errors through them during program testing (low DRR), testing resources can be shifted during module testing to modules that are less likely to propagate incoming data state errors across them.

I am not suggesting that specification decomposition in this manner is always possible, but rather when possible, it can benefit those persons testing the program. By isolating higher amounts of implicit information loss, the benefit derived is knowing which sections of a program have a greater ability to cancel incoming data state errors before testing begins. This provides insight for where testing is more critically needed. This allows testers to shift testing resources from sections needing less effort to sections needing more.

As an example, consider a specification g:

$$g(a,b,c) = \begin{cases} c+2 & \text{if odd}(a) \text{ and odd}(b) \\ c+1 & \text{if odd}(a) \text{ or odd}(b) \\ c & \text{otherwise} \end{cases}$$

where a, b, and c are integers. Many different designs can be used to compute g, but I will concentrate on two designs are also shown in the table in Figure 2: Design 1 and Design 2 (In Figure 2 a thick arc represents large sets of values (too many to enumerate), and a thin arc represents a single value.) The DRR of g is $\infty_I^3 : \infty_I$. The DRRs of the subfunctions of Designs 1 and 2 are shown in Figure 2. Design 1 has two subfunctions, $f1$ and $f2$. In Design 2, I have taken g with its DRR of $\infty_I^3 : \infty_I$ and have decomposed it in such a manner as to isolate the subfunctions that create its high DRR: $f3$ and $f4$. This decomposition provides *a priori* information concerning where to concentrate testing (in $f3$ and $f4$) and where not to (in $f5$, since subfunction $f5$ can be exhaustively tested). Had subfunction $f5$ not been separated out, then in whatever other design this computation occurred, it would be needlessly retested.

The reader might ask why subfunctions $f3$ and $f4$ should receive additional testing. This is because if anything were to occur to the values of variables **a** and **b** before subfunctions $f3$ and $f4$ are executed (thus causing a data state error affecting these variables), it is likely that these subfunctions will cancel the data state error. We should test less in $f5$ and test more in $f3$ and $f4$. This shows how isolation according to module DRRs can benefit testing.

subfunction	classification	DRR
$f1$	VDVR	$\infty_I^3 : \infty_I^3$
$f2$	VDVR	$\infty_I^3 : \infty_I$
$f3$	VDFR	$\infty_I : 2$
$f4$	VDFR	$\infty_I : 2$
$f5$	FDFR	4:3
$f6$	VDVR	$3 \cdot \infty_I : \infty_I$

Figure 2: Design 1 (left); Design 2 (right).

3.2 Minimizing Variable Reuse: Lessening Implicit Information Loss

A method for decreasing the amount of implicit information loss that occurs at the operator level of granularity is *minimizing* the reuse of the variables. For instance, as we have already seen, a computation such as **a := sqr(a)** destroys the original value of **a**, and although you can take the square root after this computation and retrieve the absolute value that **a** had, you have lost the sign. Minimizing variable reuse is one attempt to decrease the amount of implicit information loss that is caused by built-in operators such as **sqr**.

Minimizing variable reuse requires either creating more complex expressions or declaring more variables. If the number of variables is increased, memory requirements are also increased during execution. If complex expressions are used, we lessen the testability because a single value represents what were previously many intermediate values. Although there is literature supporting programming languages based on few or no variables [Backus78], programs written in such languages will almost certainly suffer from low testabilities. Thus I advocate declaring more variables.

3.3 Increasing Out-Parameters: Lessening Explicit Information Loss

Consider the analogy where modules are integrated circuits and local variables are internal signals in integrated circuits. This analogy allows us to see how explicit information loss caused by local variables parallels the notion of low observability in integrated circuits. Since explicit information loss suggests lower testabilities, I prefer, when possible, to lessen the amount of explicit information loss that occurs during testing. And if limiting the amount of explicit information loss is not possible, I at least have the benefit of knowing where the modules with greater data state error cancellation potential are before validation begins.

One approach to limiting the amount of explicit information loss is to insert **write** statements to print internal information. This information must then be checked against the correct information. A second approach is increasing the amount of output that these subspecifications return by treating local variables as out-parameters. A third approach inserts self-tests (this is similar to the assertions suggested in [Shimeall91] for fault detection) that are executed to check internal information during computation. In this approach, messages concerning incorrect internal computations are subsequently produced.

These approaches produce the same end results, however in the processes employed to achieve these results they differ slightly. The end results of these approaches are:

1. Forcing those persons involved in the formalization of a specification to produce detailed information about the states of the internal computations. This should increase the likelihood that the code is written correctly.

2. Increasing the cardinality of the range.

As an example of the third approach, consider inserting self-tests into the declaration given in Section 2:

```
Module x(in-parameter a : real, in-parameter c : real,
         out-parameter  b : boolean)

local-parameters
  z : integer
  y : boolean

Beginning of Body
    :
    :
z := z mod 23
self-test(z,ok)
if not(ok) then write('warning on z')
    :
    :
    :
y := expression
self-test(y,ok)
if not(ok) then write('warning on y')
    :
b := f(a,c,y,z)
End of Body
```

A self-test such as `self-test(z,ok)` may either state explicitly what value z should have for a given (a,c) pair, or it may give a range of tolerable values for z in terms of a particular (a,c) pairing. If a self-test fails, a warning is produced.

These three approaches simulate the idea previously mentioned that is used in integrated circuits—increasing the observability of internal signals [Berglund79, Markowitz88]. In these approaches, I am not discrediting the practice of information hiding during design. However, when writing software such as safety-critical software, there is a competing imperative: to enhance testability. Information that is not released encourages undetected faults, and increased output discourages undetected faults.

The downside to these approaches is that for the approaches to be beneficial, they all need additional specified information concerning the internal computations. Maybe the real message of this research is that until we make the effort to better specify what must occur, even at the intermediate computation level, testabilities will remain lower.

3.4 Combining Approaches

We have seen how different techniques can be used against various classifications of information loss. An even better methodology for achieving this goal is a combination of techniques, applied at both design and implementation phases. For example, combining the technique of releasing more internal information with the technique of minimizing variable reuse furthers the available information for validation. The limit to any combined approach, however, will be the ability to validate the additional information. After all, if the additional information can not be validated, then there is no reason to expose it.

4 Summary

Information loss is a phenomenon to be considered by those who gain confidence in the correctness of software through software testing. The suggestion that information loss and testability are related is important; it implies that the ability to gain confidence in the absence of faults from observing no failures may be limited for programs that implement functions that encourage information loss. Although discouraging on the surface, I feel that there are ways to lessen this limitation with prudent design and implementation techniques.

The unfortunate conclusion of Section 3 is that we must validate more internal information if we hope to increase software testability. To validate more internal information, we must have some way of checking this additional internal information. This requires that more information be specified in the specification or requirements phase. And for certain applications this information is rarely available.

It may be that a theoretical upper bound exists on the testability that can be achieved for a given (functional description, input distribution) pair. If we can change the functional description to include more internal information, we should be able to push the upper bound higher. Although the existence of an upper bound on testability is mentioned solely as conjecture, my research using sensitivity analysis and studying software's tendency to not reveal faults during testing suggests that such exists. I challenge software testing researchers to consider this conjecture.

5 Acknowledgement

This research has been supported by National Research Council NASA-Langley Resident Research Associateship.

References

[Backus78] J. Backus. Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Berglund79] Neil C. Berglund. Level-Sensitive Scan Design Tests Chips, Boards, System. *Electronics*, March 15 1979.

[Korel87] Bodgan Korel. The Program Dependence Graph in Static Program Testing. *Information Processing Letters*, January 1987.

[Marick90] Brian Marick. Two Experiments in Software Testing. Technical Report UIUCDCS-R-90-1644, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1990.

[Markowitz88] Michael C. Markowitz. High-Density ICs Need Design-For-Test Methods. *EDN*, 33(24), November 24 1988.

[Parnas72] David L. Parnas. On Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 14(1):221–227, April 1972.

[Shimeall91] Timothy J. Shimeall and Nancy G. Leveson. An Empirical Comparison of Software Fault Tolerance and Fault Elimination. *IEEE Transactions on Software Engineering*, 17(2):173–182, February 1991.

[Voas91a] J. Voas and K. Miller. Improving Software Reliability by Estimating the Fault Hiding Ability of a Program Before it is Written. In *Proceedings of the 9th Software Reliability Symposium*, Colorado Springs, CO, May 1991. Denver Section of the IEEE Reliability Society.

[Voas91b] J. Voas, L. Morell, and K. Miller. Predicting Where Faults Can Hide From Testing. *IEEE Software*, 8(2), March 1991.

[Voas91c] J. Voas. A Dynamic Failure Model for Estimating the Impact that a Program Location has on the Program. In *Proceedings of the 3rd European Software Engineering Conf.*, Milano, Italy, October 1991.