Semantic Metrics for Software Testability

Jeffrey M. Voas
Reliable Software Technologies Corporation, Arlington, Virginia

Keith W. Miller
Department of Computer Science, College of William and Mary, Williamsburg, Virginia

Software faults that infrequently affect output cause problems in most software and are dangerous in safety-critical systems. When a software fault causes frequent software failures, testing is likely to reveal the fault before the software is released; when the fault "hides" from testing, the hidden fault can cause disaster after the software is installed. During the design of safety-critical software, we can isolate certain subfunctions of the software that tend to hide faults. A simple metric, derivable from semantic information found in software specifications, indicates software subfunctions that tend to hide faults. The metric is the domain/range ratio (DRR): the ratio of the cardinality of the possible inputs to the cardinality of the possible outputs. By isolating modules that implement a high DRR function during design, we can produce programs that are less likely to hide faults during testing. The DRR is available early in the software lifecycle; when code has been produced, the potential for hidden faults can be further explored using empirical methods. Using the DRR during design and empirical methods during execution, we can better plan and implement strategies for enhancing testability. For certain specifications, testability considerations can help produce modules that require less additional testing when assumptions change about the distribution of inputs. Such modules are good candidates for software reuse.

1. INTRODUCTION

Testability of software is the tendency for failures to be observed during testing when faults are present [1]. Although the concept of testability generalizes to other testing strategies, we will focus here on the testability associated with random black box testing. Throughout this article, when we refer to testing, we specifically mean random black box testing.

Software has high testability if it tends to expose faults during testing, producing failures for many inputs that execute a fault. Software had low testability if it tends to hide faults during testing, producing almost no failures even through at least one fault is present. Software with low testability is dangerous in safety-critical systems because a well-hidden fault can surface after installation, perhaps causing disaster. This paper presents a design technique for isolating software with low testability.

The domain/range ratio (DRR) of a specification is the ratio between the cardinality of the domain to the cardinality of the range. We denote the DRR of a specification by $\alpha:\beta$, where α is the cardinality of the domain and β is the cardinality of the range. Generally, as a DRR increases, the testability of software decreases. When α is greater than β, previous research has suggested that faults are more likely to hide during testing than when $\alpha = \beta$ [2].

Note that the DRR as we have defined it depends only on the number of values in the domain and range, not on the relative probabilities that individual elements may appear in these sets. This simplifies the calculation of the DRR and makes it more likely that a designer will have a reasonable approximation of the DRR at design time. Although the input distribution to the software can change the DRR, the DRR is not as sensitive to distribution changes as, for example, other semantic metrics such as the probability of failure or reliability. Both these measures depend critically on accurate information about the input distribution.

The DRR is one factor in determining whether

Address correspondence to Jeffrey M. Voas, RST Corporation, Suite PH, 1001 N. Highland, Arlington, VA 22201.

0164-1212/93/$6.00

software is likely to hide faults, but it is not the only one. A different factor that cannot be predicted by the DRR is the frequency with which statements are executed. This is a limitation of the predicting power of the DRR.

When the DRR is high, the resulting software will have a greater potential to hide faults, although it may not be hiding them. First, a fault may cause total or near total failure (almost all inputs give incorrect output) despite a high DRR; in this case, testing is likely to reveal the fault. Second, the software might be correct, in which case there are no faults hiding. However, a lack of faults, is difficult to demonstrate using testing [3].

When the DRR is low, the resulting software is less likely to hide faults, but programmers can still include faults that are difficult to detect. No software engineering technique can completely insulate programs from programmer errors. However, the examples in a later section illustrate why programmers will be less inclined to include well-hidden faults when implementing a specification with a low DRR.

By using the DRR during design and validation, we can enhance the process of producing higher quality code in four ways. Each of these is discussed in more detail in later sections.

1. Early discovery of potential problems. Before coding begins we can focus attention on modules that are likely to hide faults. We can concentrate time and talent on the software that, by the nature of its specified purpose, will be difficult to test effectively.
2. Higher overall testability. Isolating modules that are less likely to expose faults during module testing allows testing and verification resources to be allocated more intelligently. This produces software that is less likely to contain unsafe hidden faults.
3. Greater reusability. The original unit testing for a reusable module is based on an input distribution. (The domain of a module is the set of values from which its inputs are drawn; the input distribution determines the probability that a particular input is selected during production runs.) When reused, the module may be exercised under a completely different distribution, invalidating any conclusions drawn from the original testing. Using the DRR does not eliminate this problem; however, by designing with the DRR in mind, submodules that are less sensitive to the change in distribution can be isolated and reused without additional testing.
4. Increasing testability with increased module output. If a specification indicates that the resulting software will tend to hide faults, we can reduce this risk by adding output parameters to critical modules. These output parameters are specifically designed to reveal faulty data states during testing.

A recent article by Freedman [4] gives a more complex development concerning testability. Freedman bases his definitions of testability on two related ideas: controllability and observability. He states, "an evaluation of expression procedure $F(E)$ is *controllable*, if, for any state s, the domain of values of the evaluation map *equals* the domain of values denoted by its output specification." (Emphases are Freedman's.) And he states, "F is observable if: $F(B_1,\cdots,B_n) \neq F(A_1,\cdots,A_n)$ **implies** $(B_1,\cdots,B_n) \neq (A_1,\cdots,A_n)$." Using our terminology, Freedman's controllability is a special term for a DRR of 1 and his observability characteristic formalizes the notion of a deterministic function without state memory between calls.

There are two main differences between Freedman's approach and ours:

1. We assume observability in our description. This simplifies the discussion considerably and still affords useful insights into design considerations.
2. Freedman bases his testability measures on extensions that would be required to make the code observable and controllable. Instead, we focus on the DRR exclusively. This is particularly useful during design, when extensions may be difficult to assess.

The remainder of the article is organized as follows. Section 2 describes in detail one way in which software hides faults. By recognizing the mechanics of hidden faults, we can isolate factors that lower testability. Section 3 demonstrates the relationship between lower testabilities and higher DRRs in a few simple functions. Section 4 shows how paying attention to the DRR during design benefits testing by suggesting a more proper balance of testing resources among various regions of a program. Section 5 discusses an automated system that estimates the testability of software after it has been implemented. Section 6 shows how our DRR design technique can facilitate software reuse.

2. INTERNAL STATE COLLAPSE

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 specifications that require a many-to-one computable function lose information that uniquely identifies the input given an output. Restated, many-to-one specifications mandate a loss of information during execution; one-to-one specifications do not.

Software that implements a many-to-one specification has information in its internal states that is not communicated in its output. This corresponds to a DRR of $d{:}r$, where $d > r$. We term this phenomenon internal state collapse. An internal state is a collection of all live variables along with their current values at some point during execution. A live variable is any variable that has the potential to affect the software's output. The program counter is always considered live.

When internal state collapse occurs, the lost information may have included evidence that internal states were incorrect. Since such evidence is not visible in the output, the probability of observing a failure during testing is reduced. As the probability of observing a failure decreases, the probability of hidden faults increases.

Any module that implements a subspecification with a DRR that is $\alpha{:}\beta$ where $\alpha > \beta$ must lose information about the input or about intermediate values of the computation. This loss suggests a lower module testability. Therefore, we can examine a DRR for a subspecification and know before implementation how much internal state collapse is required. This a priori information can be thought of as a rough approximation of the software's testability.

Each assignment of a value to a variable is a subfunction of the overall function being computed (meaning the module or program). Variables that are restricted to a small set of potential values will frequently suffer from the effects of internal state collapse; these variables warrant special consideration during validation. Internal state collapse is a discrete parallel to a rule of thumb in analog measurement: "the less accuracy in the output, the greater the chances of a mistake going unnoticed."

Integrated circuit design engineers have a notion similar to internal state collapse that they term "observability." Observability is the ability to view the value of a particular node that is embedded in a circuit [5]. When internal state collapse occurs in software, we have lost the ability to see information within the internal state of the program. So in this sense, internal state collapse in software is a parallel to lower observability in hardware.

Discussing the observability of integrated circuits, Berglund [6] 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 bit strings that can be output from the chip.

In Section 4.1.1, we advocate a software design strategy that parallels the hardware strategy of increased pin count for testing: during the design of safety-critical modules, the number of output parameters can be increased to reveal more internal information in the output. This additional internal information increases the probability of discovering faults that would otherwise be difficult to reveal during testing. This additional information is indicated by a reduced DRR.

Internal state collapse occurs in at least two ways. First, the number of live variables in an executing program decrease as execution continues, until finally the only live variables are the output variables. (That is, the only variables that are live when the final output values are produced are the variables holding the values.) When an erroneous live variable becomes dead, unless the variable transfers its "incorrectness" into another live variable before becoming dead, this incorrectness will be invisible during testing.

Internal state collapse can also occur when two different program states are presented to a particular subfunction in a program and that subfunction produces the same internal state in both cases. For example, assume that $f(x) = x^2$. Both $x = 5$ and $x = -5$ produce the same internal state after executing $x := f(x)$. It may be that the negative integer could signal a problem with the software, but $f(x)$ will erase that information in subsequent output. Thus, a live variable that contains evidence of a fault can be reassigned in such a way that the evidence disappears.

3. RELATIONSHIP BETWEEN TESTABILITY AND DRR

Internal state collapse is common in many of the built-in functions of modern programming languages. Functions such as **div**, **mod**, and **trunc** can have high DRRs. We now look at some common

functions and their degrees of internal state collapse. Table 1 contains a set of functions with their "generalized" testabilities and DRRs. A function classified as having low testability in Table 1 is likely to receive an "altered" incoming parameter (containing evidence of a fault) and produce output that hides the fault; a function classified as having high testability in Table 1 is likely to retain evidence of a fault. In Table 1, b is assumed to be a constant. The symbol ∞_I denotes the cardinality of the integers and ∞_R denotes the cardinality of the reals. 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. We use the symbol ∞^N to denote a domain of N values of size ∞ each.

The reasoning used for the generalizations in Table 1 follow.

1. If a has a negative value and this value has been changed from the original, yet the original value of a is also negative, this altered value will be invisible. Positive a values will not participate in internal state collapse.
2. This is a one-to-one function, and an altered value in a will be visible.
3. As the potential values of b decrease, the cardinality of the range of f decreases, and the value of a can more easily hide values.
4. The testability of f depends on the value of b; however, as the potential values of b increase, the cardinality of the range of f decreases and altered values of a will be less visible.
5. Since the fractional part of the real is discarded, any altered value that only affects the fractional part of the result is invisible.

Table 1. DRRs and Testabilities of Various Functions

	Function	Testability	DRR	Comment
1	$f(a) = \begin{cases} 0 & \text{if } a < 0 \\ a & \text{otherwise} \end{cases}$	low	$\infty_I : \infty_I/2$	a is integer
2	$f(a) = a + 1$	high	$\infty_I : \infty_I$	a is integer
3	$f(a) = a \bmod b$	low	$\infty_I : b$	testability decreases as b increases
4	$f(a) = a \text{ div } b$	low	$\infty_I : \infty_I/b$	testability decreases as b increases
5	$f(a) = \text{trunc}(a)$	low	$\infty_R : \infty_I$	a is real
6	$f(a) = \text{round}(a)$	low	$\infty_R : \infty_I$	a is real
7	$f(a) = \text{sqr}(a)$	high	$\infty_R : \infty_R$	a is real, $a \geq 0$
8	$f(a) = \text{sqrt}(a)$	high	$\infty_R : \infty_R$	a is real, $a \geq 0$
9	$f(a) = a/b$	high	$\infty_R : \infty_R$	a is real
10	$f(a) = a - 1$	high	$\infty_I : \infty_I$	a is integer
11	$f(a) = \text{even}(a)$	low	$\infty_I : 2$	a is integer

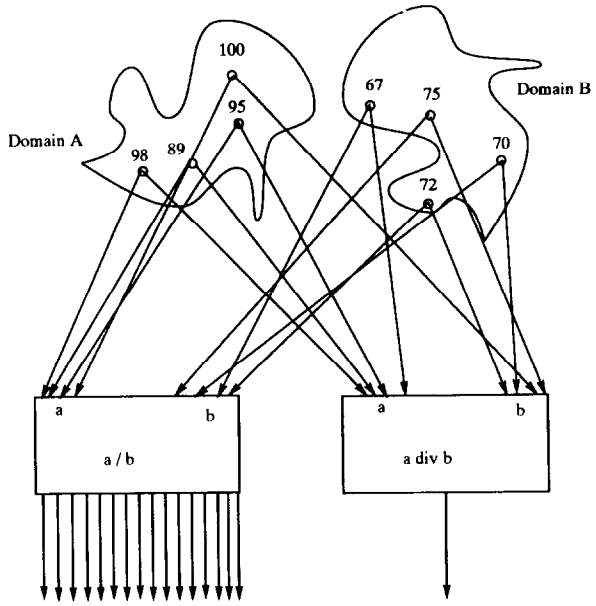

Figure 1. Div vs./.

6. Same argument as 7.
7. Since this is a one-to-one function, an incorrect a is visible.
8. Same argument as 9.
9. In real division, virtually any change in the value in a is visible. See Figure 1.[1]
10. Same argument as 2.
11. Any altered value of a that retains even parity is invisible.

4. DESIGNING ACCORDING TO THE DRR

The DRR of a specification is fixed and cannot be altered without changing the specification. However, there are techniques that can be applied to produce software with higher overall testability based on the DRR. During design, we have "hands-on" control of the DRR that each subspecification will have. We can gain insights (before a subspecification is implemented) concerning the amount of testing needed to obtain a particular confidence that each module is correct [7]. The rule of thumb that guides this intuitive feeling is "the greater the DRR, the more testing that will be needed."

We consider two strategies for exploiting information about the DRR during design—isolating subfunctions with relatively high DRR's and increasing

[1] In Figure 1, there are four values for variable a and four values for variable b, for a total of 16 pairs of inputs. Note that for these 16 inputs, integer division gives 1 output and real division gives 16 outputs.

the DRR of individual functions by adding output parameters. To facilitate the discussion of these strategies, we introduce names for three categories of DRRs.

4.1 Three DRR Subspecification Classifications

In theory, each different DRR maps to a different degree of testability. But currently this mapping is unknown, so we simply categorize the DRRs of subspecifications into three groups. Although these three groups can be further subdivided, we will not do so here.

In the examples of the three groups that follow, we use x to represent an integer number in the infinity mathematical sense of integer numbers. Any computer implementation of the functions described would in practice have a finite, though very large, domain. The three groups are as follows:

1. Variable domain/fixed range (VDFR): A VDFR DRR's domain is over an infinite set of values and its range is over a finite set whose cardinality is independent of the domain. An example of a VDFR is seen in the function $f(x) = x \bmod 2$ for all x whose DRR is ∞:2.
2. Variable domain/variable range (VDVR): A VDVR DRR's range and domain are over two infinite sets of values. An example is the specification: $f(x) = x + 2$ for all x, whose DRR is ∞:∞.
3. Fixed domain/fixed range (FDFR): A FDFR DRR's range and domain are over two finite sets. An example is the specification $f(x) = x \bmod 2$ for $0 \leq x \leq 10$, whose DRR is 11:2.

In any computer implementation, the infinity of integers is really a large, finite number; however, the large number of discrete integers makes exhaustive testing so impractical as to render the actual size irrelevant. A similar argument holds for real numbers and strings. It is difficult to determine in general the point at which a finite set becomes so large that it is, in a practical sense, infinite. However, we have found that in most specific situations encountered in software testing the difference is clear.

We can now generalize these DRR groups from highest testability to lowest testability:

1. FDFR has the highest testability because exhaustive testing is theoretically possible and in some cases practical.
2. VDVR has the next highest testability.
3. VDFR has the lowest testability because this group represents the largest amount of internal state collapse of the three groups.

Therefore, we prefer to have as many FDFR DRR modules in an implementation as possible and as few VDFR DRR modules as possible. However, if we are forced to accept a large number of VDFR DRR modules, we at least have a priori knowledge of their tendency to hide faults during testing.

4.1.1 Transforming a VDFR DRR to a VDVR DRR by adding output parameters. Since VDFR DRRs suggest lower testabilities, we prefer, if possible, to limit the number of VDFR DRR subspecifications that will exist. During design and validation, we pay particular attention to modules that implement VDFR DRR subspecifications.

One method of lowering the number of VDFR DRR subspecifications is to apply a debugging technique during design to effectively transform modules implementing subspecifications of this classification to modules implementing VDVR DRR subspecifications. Although not always practical, it will often produce a program much simpler to test effectively.

A common approach to debugging (after a failure is observed) is to insert "write" statements to print internal information in order to locate the fault. Similarly for VDFR DRR subspecifications, we can alter the output requirements of these subspecifications to return more internal information, even if this additional information is not needed by the calling subspecification. Hoffman [8] calls for a similar notion, by inserting assertions to check internal information dynamically. This is similar to a technique used in integrated circuit design to increase observability: the pin count is increased to carry out internal information even though that information is not necessarily needed for the chip's performance [5]. In software, we can accomplish this by making particular local variables into out-parameters of the module. Thus, we increase the range of the module and check the validity of the outcoming internal information.

The added parameters can be used in at least three ways:

1. Added parameters can be used only during unit testing, and then removed. Care must be taken to ensure that the removal preserves the original specified behavior.
2. Added parameters can be present during integration testing as well. Again, these could be removed after the testing phase.
3. The added parameters can be permanent. In safety-critical systems, these parameters could participate in runtime consistency checks and

correctness assertions at various levels of abstraction.

The technique of exposing internal state information accomplishes important objectives when producing safety-critical software:

1. It forces those involved in the formalization of a specification to produce detailed information about the expected internal states of the computations.
2. It increases the cardinality of the range of a subspecification (this potentially transforms a VDFR DRR subspecification to a VDVR DRR subspecification), thus increasing the expected testability.

Transforming a VDFR DRR to a VDVR DRR increases the testability of a module and increases the likelihood that the code is properly written. But this increase in testability has costs, including increased development effort, more testing effort for each test input, and performance degradation for increased parameter passage and runtime checks. When internal data states remain after unit testing, information hiding is diminished. Indeed, we have found that testability and information hiding often must be traded off during development. For safety-critical software, we contend that increasing testability may well be worth a reduction in the portability afforded by stricter information hiding.

An example of a transformation to decrease the DRR is given next. Although the example is small, it illustrates principles that generalize to more complex systems. The original VDFR specification is to be implemented with a module of the form:

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

local-parameters
z: integer
y: boolean

For this implementation, there are only two values for b and thus there are only two output values for x (assuming no global variables are modified in x). Now consider a modified declaration:

Module x(in-parameter a: real
out-parameter b: boolean
out-parameter z: integer
out-parameter y: boolean)

We have increased the cardinality of the range of x by forcing it to reveal its internal information; this information will be checked during validation of x using assertions about the correct values for z and y. The range of values that z can contain determines whether this modification has transformed the DRR of x from VDFR to VDVR. By facilitating assertions about the correct values of variables z and y, we increase the knowledge available about what should occur in x.

This example illustrates how testability concerns require us to compromise on issues long considered sacrosanct among software engineers. The technique of increasing the range of a specification violates a common design guideline: minimize coupling [9]. The additional parameters added are testing overhead because the calling module will not use the additional information received for producing its output. We are not discrediting the time-honored software engineering practice of minimizing coupling. However, when writing code, especially safety-critical code, there is a competing imperative: to enhance testability. Hidden information can help disguise faults during testing; exposing internal data state information helps uncover faults during testing.

4.2 Isolating High DRR Subfunctions

We have presented three classifications of DRRs. If, during design, we can isolate, as much as possible, the high DRRs inherent in VDFR modules, the testability of the overall program can be enhanced. We now illustrate how this design technique can be used to more appropriately balance testing resources during validation.

Consider the specification

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

where a, b and c are integers. Many different designs can be used to compute g, but we will concentrate on two: design 1 and design 2 (Figure 2).[2]

The DRR of specification g is ∞_I^3: $\infty_I + 2$. The DRRs of the subspecifications of designs 1 and 2 are shown in Table 2. Design 1 has two subspecifications, $f1$ and $f2$, that both have VDVR DRRs. Design 2 has five subspecifications: $f1$, $f3$, $f4$, $f5$, and $f6$. Subspecifications $f3$, $f4$, and $f5$ do not have

[2] In Figures 2 and 3, a thick arc represents large sets of values (too many to enumerate), and a thin arc represents a single value. A dotted box represents a FDFR DRR subspecification, a dashed box represents a VDVR DRR subspecification, and a solid box represents a VDFR DRR subspecification.

J. SYSTEMS SOFTWARE
1993; 20:207-216

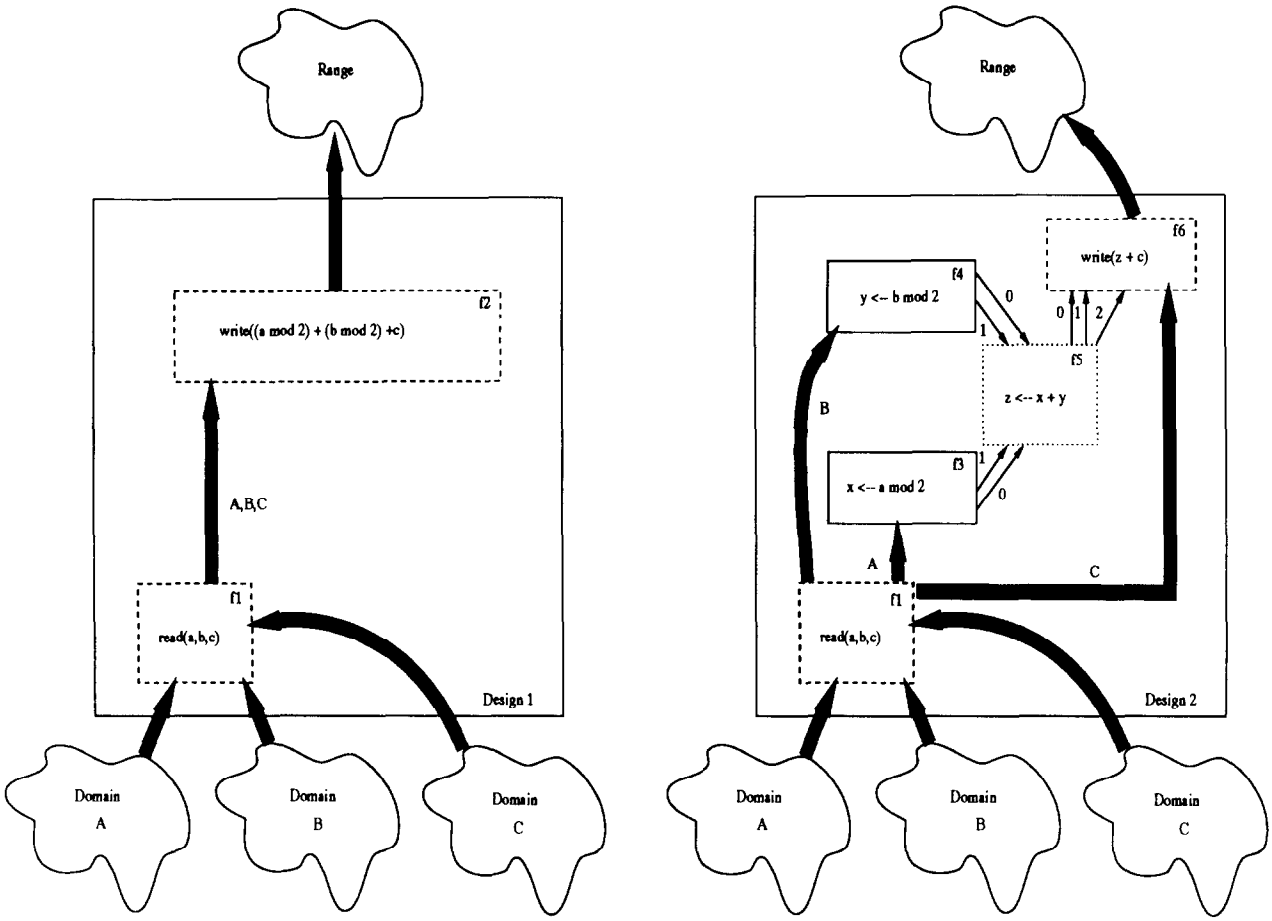

Figure 2. Designs of g: design 1 (left); design 2 (right).

Table 2. DRRs of the Subspecifications of Design 1 and Design 2

Subspecification	Classification	DRR
$f1$	VDVR	$\infty_I^3 : \infty_I^3$
$f2$	VDVR	$\infty_I^3 : \infty_I + 2$
$f3$	VDFR	$\infty_I : 2$
$f4$	VDFR	$\infty_I : 2$
$f5$	FDFR	4:3
$f6$	VDVR	$3\infty_I : \infty_I + 2$

VDVR DRRs. Instead, subspecifications $f3$ and $f4$ have VDFR DRRs and subspecification $f5$ has a FDFR DRR.

In design 2, we have taken g with a DRR of ∞_I^3: ∞_I and have decomposed it in such a manner as to isolate the subfunctions that create this high DRR: $f3$ and $f4$. This decomposition provides a priori information about where to concentrate testing (in $f3$ and $f4$) and where not to (in $f5$, since subspecification $f5$ can be exhaustively tested). Had we not separated out the subfunction computed in $f5$, then in whatever other design we created, we would effectively be retesting the subfunction needlessly.

5. EMPIRICAL METHODS FOR ESTIMATING CODE TESTABILITY

The DRR measures how much internal data state collapse must occur to fulfill a specification. This means that the DRR establishes an upper bound on the testability that can be expected from any software that correctly implements the specification. Given a specification for an entire program, the DRR does not help locate where in the particular piece of software the internal data state collapse occurs. Furthermore, in implementing software, the programmer can introduce data state collapse where it is not mandated by the specification. So although the DRR is a useful technique, it must be supplemented with other techniques to help identify specific software locations where faults (if they exist) are likely to hide from testing.

Research into testability has included the develop-

ment of an automated system that makes empirical estimates of the testability of software on a location-by-location basis. The system, called PIE (Propagation, Infection, and Execution), instruments code and does repeated code executions using test inputs from the assumed input distribution. Statistics from these executions are used to estimate three probabilities for each location in the code:

1. the probability that a code location is executed;
2. the probability that a syntactic fault at this location will adversely affect the succeeding data state;
3. the probability that an incorrect data state will eventually result in an incorrect output.

These probabilities are estimated by using syntactic mutants (as in mutation testing [10]) and semantic mutants (altering the values of live variables) to mimic the action of faults. The assumption underlying this system is that these artificial "faults" will behave much like any real fault in the code; by tracking the effect of these simulated faults, we can predict the behavior of any real faults in the code.

The process of repeated executions is quadratic in the number of locations tested. Several experiments have been conducted with small modules [11] and with a handful of locations in larger software [12] using early versions of PIE. The results of these experiments have demonstrated that, at least for these examples, there is a high correlation between the testability estimate produced by PIE and the effect of a fault inserted at a location. A fully automated PIE system has only recently been completed [13], making possible more extensive experiments on large collections of modules. Detailed information about PIE and the theory that underlies it can be found elsewhere [14].

Testability information from the DRR can guide development during specification and design. Testability information from PIE helps focus attention on individual code locations during coding, testing, and maintenance. Throughout the life cycle, testability information can help avoid hidden faults through planning and can help uncover locations where faults can hide from testing. When faults are less likely to hide from testing, the software can be made more reliable. This reliability is especially important in safety-critical systems and in reusable software. The next section discusses reusable software in more detail.

6. REUSABILITY

Designing according to the DRR cannot only better balance resources during validation, but it can also increase the reusability of certain programs. A program has high reusability if it requires little extra testing when the input distribution changes. A program has low reusability if it requires a great deal of extra testing when the input distribution changes. Input distribution change is a major problem with reusable software: when software is used in different environments, the input assumptions that drive the initial testing may be inappropriate for the new situation in which the software is used. For VDFR DRR and VDVR DRR functions, this problem frequently requires additional testing; for FDFR functions, less additional testing may be required.

Ideally, we would like to have as many FDFR DRR subspecifications as possible, because they can potentially be exhaustively tested. If a module is exhaustively tested on all elements that can serve as input, it need never be retested when the input distribution changes. Because of their potential for reuse, we can design reusable modules by isolating FDFR DRR modules out of modules that were originally designed to be VDFR DRR modules.

When we can isolate FDFR DRR modules from VDFR DRR or VDVR DRR specifications, we define two different domains of the specification, a true domain and an application domain. The true domain is a fixed set of inputs that the reusable FDFR accepts. The application domain is a variable set of inputs that the original specification calls for. When the application and true domains differ for a particular problem being computed, a VDVR DRR subspecification can be created which maps inputs from the application domain to the true domain.

As an example of decomposition guided by the DRR, consider the specification $g(x) = \sin(x)$ where x is in whole degrees. For this specification, we consider the true domain to be [0..359] and the application domain, [−maxint..+ maxint]. Consider the two different designs of g in Figure 3: one isolates the true domain (left) and one does not (right). Subspecifications s3 (left) has a FDFR DRR, because there are a fixed set of integers. Subspecification s3 (right) has a VDFR DRR. Subspecification s3 (left) has greater reusability because s3 (right) requires additional testing; s3 (left) requires no additional testing once it has been tested on all 360 possible inputs. Regardless of how the application domain changes for g, the true domain remains fixed and s3 (left) can potentially be exhaustively tested. Had we not isolated s3 (left) to only accept its true domain, additional testing of s3 would be necessary whenever the application domain changed. Now if the application domain of g changes, only subspecification s2 needs additional testing.

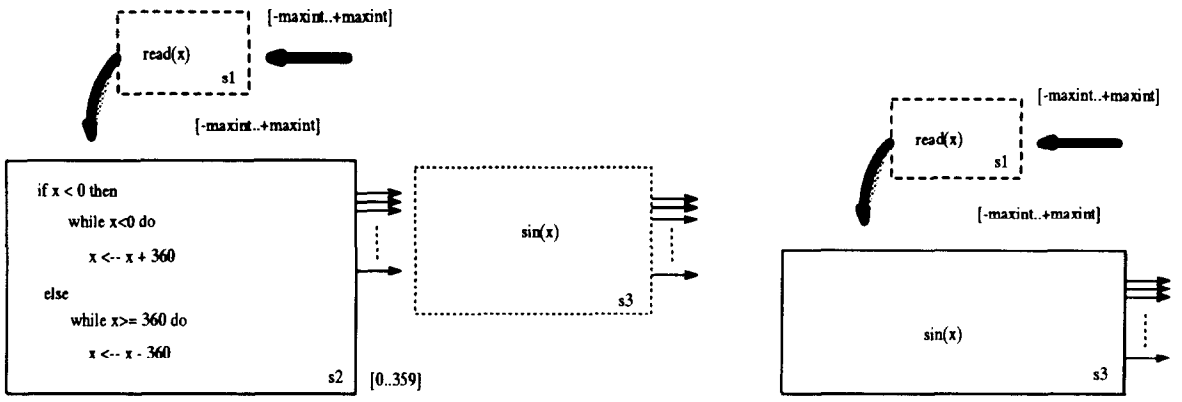

Figure 3. Design isolating true domain (left); design not isolating true domain (right).

This kind of decomposition may not always be easy to identify or practical to implement. However, if the differences between the true and application domains are readily discernible from the specification, this decomposition can provide a significant benefit in enhancing the potential for reuse. Reusable modules that have been extensively tested and verified can, in turn, facilitate software safety.

7. SUMMARY

The DRR semantic metric exposes an issue that should be addressed by those who seek to improve software quality. The DRR is available early in the software life cycle, during specification and design. The suggestion that the DRR and testability are related is provocative; it implies that we are limited in our ability to gain confidence in the absence of faults for certain classes of specifications that can not be exhaustively tested nor proven correct.

A high DRR for a specification is a warning that implementations will suffer from low testability. Although the overall DRR of a specification is fixed, we feel that through prudent design techniques at the subspecification level, we can minimize the effect of the overall DRR by isolating high DRR functions and by more appropriately allocating testing the verification resources during development.

Different DRRs require different amounts of effort during validation. Through hands-on control of a subspecification's DRR, we gain insight into

1. when to test versus when to verify modules, and
2. if testing is chosen, how much testing to perform on a module.

By paying attention to the DRR, we can potentially design modules with greater reusability.

Once code has been produced, more information about testability can be gathered using dynamic experimentation. The PIE system automates the process of this experimentation, giving a testability estimate for each code location.

The idea that decreased accuracy can hide mistakes is not new. However, software development has not exploited this simple idea in any systematic way. By paying more attention to the DRR during design, we can increase the efficacy of testing and complementary validation techniques.

ACKNOWLEDGMENTS

This paper was first presented at the 1992 Oregon Workshop on Software Metrics.

This work was supported in part by a National Research Council NASA-Langley Resident Research Associateship (J. M. V.) and NASA grant NAG-1-884 (K. W. M.).

We thank Sal Bavuso in the Information Systems Division of NASA Langley Research Center for providing references on the hardware notion of observability.

REFERENCES

1. J. Voas, L. Morell, and K. Miller, Predicting Where Faults Can Hide From Testing, *IEEE Software* 8 (1991), pp. 41–48.
2. 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*, Denver Section of the IEEE Reliability Society, Colorado Springs, Colorado, 1991.
3. O. J. Dahl, E. W. Dijsktra, and C. A. R. Hoare, *Structured Programming*, Academic Press, 1972, New York.
4. R. S. Freedman, Testability of Software Components, *IEEE Trans. Software Eng.*, SE-17, 553–564 (1991).
5. Michael C. Markowitz, High-Density ICs Need Design-For-Testing Methods, *EDN* 33 (1988).
6. N. C. Berglund, Level-Sensitive Scan Design Tests Chips, Boards, Systems, *Electronics* (March 15, 1979).

7. J. Voas and L. J. Morell, Applying sensitivity analysis estimates to a minimum failure probability for software testing, in *Proceedings of the 8th Pacific Northwest Software Quality Conference*, Portland, Oregon, pp. 362–371.
8. D. M. Hoffman, Hardware testing and software IC's, in *Proceedings of the Pacific Northwest Software Quality Conference*, PNSQC Inc., Portland, Oregon, 1989, pp. 234–244.
9. G. J. Myers, *Reliable Software Through Composite Design*, Petrocelli/Charter, New York, 1975.
10. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, *IEEE Comp.* 11, 34–41 (1978).
11. J. Voas and K. Miller, The Revealing Power of a Test Case, *J. Software Test. Verficat. Reliabil.* 2, 25–42 (1992).
12. T. J. Shimeall, CONFLICT Specification. Technical Report NPSCS-91-001, Computer Science Department, Naval Postgraduate School, Monterey, California, 1990.
13. J. Voas and L. J. Morell, Propagation and infection analysis (PIA) applied to debugging, in *Proceedings of the* 1990 *IEEE Southeastcon*, IEEE Region 6, New Orleans, Louisiana, 1990, pp. 379–383.
14. J. Voas. PIE: A Dynamic Failure-Based Technique, *IEEE Trans. Software Eng.* 18, 717–727 (1992).