Testability, Fault Size and the Domain-to-Range Ratio: An Eternal Triangle

Martin R. Woodward
University of Liverpool,
Computer Science Department,
Chadwick Building, Peach Street,
Liverpool L69 7ZF, U.K.
Tel: +44-151-794-3676
m.r.woodward@csc.liv.ac.uk

Zuhoor A. Al-Khanjari
Sultan Qaboos University,
Computer Science Department,
P.O. Box 36, Al-Khod, PC 123,
Sultanate of Oman.
Tel: +968-515400
zuhoor@squ.edu.om

ABSTRACT
A number of different concepts have been proposed that, loosely speaking, revolve around the notion of software testability. Indeed, the concept of testability itself has been interpreted in a variety of ways by the software community. One interpretation is concerned with the extent of the modifications a program component requires, in terms of its input and output variables, so that the entire behaviour of the component is observable and controllable. Another interpretation is the ease with which faults, if present in a program, can be revealed by the testing process and the propagation, infection and execution (PIE) model has been proposed as a method of estimating this. It has been suggested that this particular interpretation of testability might be linked with the metric domain-to-range ratio (DRR), i.e. the ratio of the cardinality of the set of all inputs (the domain) to the cardinality of the set of all outputs (the range). This paper reports work in progress exploring some of the connections between the concepts mentioned. In particular, a simple mathematical link is established between domain-to-range ratio and the observability and controllability aspects of testability. In addition, the PIE model is re-considered and a relationship with fault size is observed. This leads to the suggestion that it might be more straightforward to estimate PIE testability by an adaptation of traditional mutation analysis. The latter suggestion exemplifies the main goals of the work described here, namely to seek greater understanding of testability in general and, ultimately, to find easier ways of determining it.

Keywords
Testability, observability, controllability, domain-to-range ratio, fault size.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'00, Portland, Oregon.
Copyright 2000 ACM 1-58113-266-2/00/0008…$5.00.

1. INTRODUCTION
There are a number of concepts concerned with program faults and program testability that intuitively would seem to be related. The concept of testability itself has been subject to a number of different interpretations [1,2,3,4,6,17] which one might imagine bear some relationship to one another. One interpretation of testability, due to Freedman [6], is concerned with determining modifications to the inputs and outputs of a given program component in order to make the behaviour of the component both observable and controllable. A different interpretation of testability, due to Voas and colleagues [17,19], is concerned with determining the ease with which faults may be revealed, if there are any present in the program. This latter version of testability can be estimated using the so-called propagation, infection and execution (PIE) model [15], although the necessary analysis is sophisticated and expensive to perform. As a consequence there is interest in finding easier ways to calculate testability, or at least in getting some indication of it, without actually performing the PIE technique. It has been suggested that the concept of domain-to-range ratio (DRR) might fulfill this role [16]. One further concept to be considered here is semantic fault size [12].

This paper reports part of an ongoing project to explore the relationships between the concepts just mentioned. Although there are a variety of approaches to such an investigation that one might adopt, attention here is confined to consideration of the idealized model of programs as functions. Both DRR and semantic fault size necessitate taking such a functional view of software anyway and, by using this framework, a number of simple observations can be made. The next two sections briefly introduce the functional view of software and the notion of domain-to-range ratio that ensues. Subsequent sections consider the relationships between Freedman's testability (observability and controllability) and DRR, between fault size and Voas's testability (the PIE model) and between fault size and DRR. The paper finishes with a discussion of some other related work followed by some concluding remarks.

2. A FUNCTIONAL VIEW OF SOFTWARE
Every item of software at its most primitive level may be viewed as a function or mapping according to some specification, *S*, from a set of input values (its *domain, D*) to a set of output values (its *range, R*).

A program which implements specification S should also map from D to R. However, if a fault f exists in the program, there will be some subset of the domain, D_f say, on which the erroneous program $\mathbf{P}_f$ computes a faulty result. The set of faulty results, denoted R_f, may contain values both in R and outside R. The effect of $\mathbf{P}_f$ on values outside of D remains unspecified. See Figure 1. Note that the domain and the range can be considered for an entire program, an individual program component, a program path or simply a single program location.

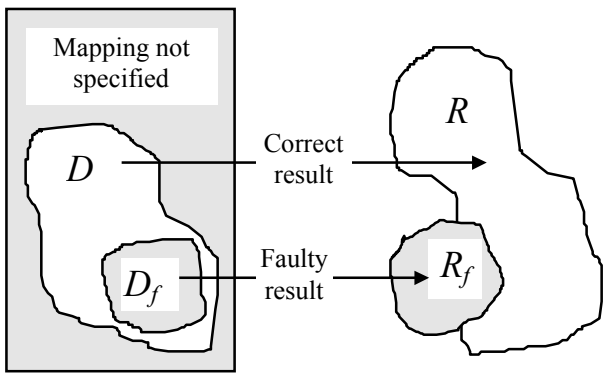

Figure 1. Functional view of a faulty program.

3. DOMAIN-TO-RANGE RATIO (DRR)

The domain-to-range ratio (DRR) has been proposed by Voas and Miller [16] as a specification metric. Put simply:

$$\text{Domain-to-Range Ratio} = \frac{|D|}{|R|} \qquad \textbf{(1)}$$

where $|D|$ is the cardinality of the domain of the specification and $|R|$ is the cardinality of the range. DRR can be determined for mathematical or computational functions. Consider, for example, the function $f(d) = d$ **mod** 2, where the input d is a member of the set of natural numbers not greater than 100, i.e. $D = \{1,2,3, \ldots, 100\}$. Clearly the function generates only two possible outputs, namely 0 when d is even and 1 when d is odd, so that $R = \{0,1\}$ and DRR = 100 / 2 = 50. Admittedly this is a simple example, but when a functional view of software is taken, as in the preceding section, it should, in principle, be possible to determine DRR for more realistic software components. Clearly difficulties arise when the domain, and more particularly the range, are infinite, although of course the corresponding sets for software must necessarily be finite, albeit very large in many cases.

Part of the rationale for the DRR metric is that it provides an approximate measure of information loss, i.e. the situation when two or more different values are used as input to a function and generate the same output. Voas *et al.* [18] have argued that this information loss may become manifest as 'internal data state collapse' when the specification is implemented as a program. Internal data state collapse occurs when two different data states are input to some sub-component in a program and yet that sub-component produces the same output state. As Voas and Miller [16] remark: "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". This connection between DRR and state collapse, or unlikelihood of fault exposure, is of course implying, in essence, a link between DRR and testability, as defined in the sense of Voas and colleagues. Indeed Voas *et al.* [18] conjectured that "the testability of a program is correlated with the domain-to-range ratio". More explicitly they hypothesized that: "as the DRR of the intended function increases, the testability of an implementation of that function decreases". In other words, high DRR is thought to lead to low testability and vice versa.

4. DOMAIN TESTABILITY AND DRR

Freedman [6] proposed what he termed 'domain testability' which involves use of the concepts of observability and controllability. Previously these latter two concepts had been used for assessing the testability of hardware components and Freedman showed how they could be used in the software context.

A software component is observable "if distinct outputs are generated from distinct inputs" [6] so that, if a test input is repeated, the output is the same. If the outputs are not the same, the component is dependent on hidden states not identified by the tester and Freedman calls this an 'input inconsistency'. Using a functional model (as introduced in Section 2) for component **P** mapping from domain D to range R, i.e. $\mathbf{P}: D \rightarrow R$, then *observability* can be characterised as:

$$\forall\ a, b \in D \bullet \mathbf{P}(a) \neq \mathbf{P}(b) \Rightarrow a \neq b \qquad \textbf{(2)}$$

A software component is controllable "if, given any desired output value, an extra input exists which 'forces' the component output to that value" [6]. If an output identifier is specified to be a certain range of values and there are particular instances of values that cannot be generated by any test input values, those are termed 'output inconsistencies'. Again using the functional model **P**: $D \rightarrow R$, then *controllability* can be characterised as:

$$\forall\ r \in R \bullet \exists\ d \in D \bullet \mathbf{P}(d) = r \qquad \textbf{(3)}$$

Most functions and procedures are not *a priori* observable and controllable. The modifications required to achieve domain testability are called *extensions*. Observable extensions are achieved by introducing new input variables so that the component becomes observable, i.e. distinct outputs can only arise from distinct inputs. Freedman suggested that *Ob*, a measure of observability, could be obtained by taking log_2 of the product of the cardinalities of the types of the additional input variables. The cardinality of a type is the cardinality of the set of values represented by that type and taking logarithm to base 2 determines, in effect, the number of extra binary inputs. In other words, if there are n extra input variables and T_i represents the type of the ith input variable, then:

$$Ob = log_2(\ |T_1| \times\ \ldots \times |T_n|\ ) \qquad \textbf{(4)}$$

Controllable extensions are achieved by modifying outputs for the given component so that it becomes controllable, i.e. all claimed outputs are attainable with some input. In fact, although called an extension, Freedman makes it clear that controllability is achieved by an appropriate *reduction* of the range. Controllability, *Ct*, can be measured by taking log_2 of the product of the cardinalities of the types of the modified output variables, i.e. if there are m modified output variables and T_k represents the type of the kth output variable, then:

$$Ct = log_2(\ |T_1| \times\ \ldots \times |T_m|\ ) \qquad \textbf{(5)}$$

In order to consider the relationship between domain testability and domain-to-range ratio, let ΔD denote the extension to the domain D required to achieve an observable component. Similarly let ΔR denote the reduction to the range R required in order to achieve a controllable component. Let the domain and range of the component after modification with observable and controllable extensions in this way be denoted D' and R' respectively. Clearly:

$$D' = D \cup \Delta D \quad \text{and} \quad R' = R - \Delta R \tag{6}$$

Since by definition ΔD is an extension of D and has no overlap with it, the following can be deduced:

$$|D \cup \Delta D| = |D| + |\Delta D| \tag{7}$$

Similarly, since ΔR is wholly contained within R:

$$|R - \Delta R| = |R| - |\Delta R| \tag{8}$$

Then it follows that DRR', the domain-to-range ratio of the now extended and domain testable component can be written as:

$$DRR' = \frac{|D'|}{|R'|} = \frac{|D| + |\Delta D|}{|R| - |\Delta R|} \tag{9}$$

By rearranging terms and noting that $|D| / |R|$ corresponds to DRR, the domain-to-range of the original program component (without extensions), the formula becomes:

$$DRR' = DRR \times \left(1 + \frac{|\Delta D|}{|D|}\right) \Big/ \left(1 - \frac{|\Delta R|}{|R|}\right) \tag{10}$$

In words, this says that the domain-to-range ratio of a program component, after modification to make it domain testable, is the domain-to-range ratio of the component before modification multiplied by one plus the relative size of the domain extension and divided by one minus the relative size of the range reduction. Finally, note that since $|\Delta D|$ is 2^{Ob} and $|R'|$ is 2^{Ct}, formula (9) can be written:

$$DRR' = \left(|D| + 2^{Ob}\right) \Big/ 2^{Ct} \tag{11}$$

5. SEMANTIC FAULT SIZE

In an attempt to obtain greater understanding of program faults, Offutt and Hayes [12] drew a distinction between the syntactic and the semantic nature of faults. The syntactic nature can be described by the syntactical differences between the faulty program and the correct program. Indeed Offutt and Hayes went further and defined the *syntactic size* of a fault as "the number of statements or tokens that need to be changed to get a correct program". The semantic nature of a fault, on the other hand, results from the view that for some subset of the input domain a faulty computation takes place producing incorrect output. Corresponding to the syntactic size of a fault, Offutt and Hayes defined the *semantic size* of a fault as "the relative size of the subdomain of D for which the output mapping is incorrect". Referring to Figure 1 the following formula is obtained:

$$\text{Semantic fault size} = \frac{|D_f|}{|D|} \tag{12}$$

where $|D|$ is the cardinality of the entire input domain and $|D_f|$ is the cardinality of just that subpart which results in faulty output. It should be obvious that there is no reason why there should be a link between syntactic fault size and semantic fault size. Indeed it is perfectly possible to find situations where a syntactically small fault results in a very large semantic fault size, and vice versa.

5.1 Semantic Fault Size and Testability

Offutt and Hayes [12] suggested that semantic fault size is closely related to testability in the sense of Voas *et al*. [19]. That is, if a statement in the subject program has low testability, then any faults associated with that statement might be expected to have small semantic size and any statement containing a fault with large semantic size could be expected to exhibit high testability.

To explore this connection between semantic size and testability further, consider the propagation, infection and execution (PIE) model that provides the basis for testability estimation. Consider a program with a single fault f at one location in the program. According to the PIE model, the probability of failure under a particular input distribution, Pr[Failure], is a combination of the individual probabilities: (1) that the fault is executed (E=execution); (2) that execution of the fault causes corruption of the data state (I=infection); and (3) that the faulty data state propagates to the output (P=propagation). This can be written [7]:

$$\Pr[\text{Failure}] = \Pr[E] \times \Pr[I \mid E] \times \Pr[P \mid I] \tag{13}$$

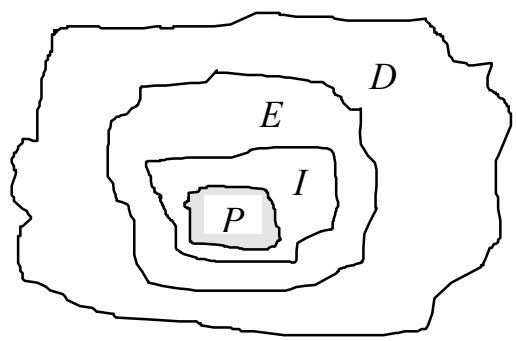

Figure 2. Input domain view of the PIE model.

Referring to Figure 2, where, as before, D represents the entire input domain of the subject program, there will be some subset E of D such that all test values in E cause the fault to be executed. Amongst those input values that cause fault execution, some will result in data state infection, as represented by the region I. Finally amongst those input values that cause data state infection, some will propagate the faulty state to the output, as represented by the region P. Assuming a uniform input distribution, the probabilities of formula (13) can be replaced by the ratios of the cardinalities of the relevant sets giving:

$$\Pr[\text{Failure}] = \frac{|E|}{|D|} \times \frac{|I|}{|E|} \times \frac{|P|}{|I|} = \frac{|P|}{|D|} \tag{14}$$

Since $|P|$ is the cardinality of the input set that causes execution of the fault f, resulting in data state infection which propagates to the output, this is none other than the cardinality of that region of the

input space previously labelled D_f. Hence formula (14) is just the semantic size of fault f.

In practice Voas [15] suggests estimating testability at a location by separate estimation processes for the three individual components of the model. Execution probability is estimated for some given input distribution by determining the proportion of cases that execute the location of interest. Infection probability is estimated by introducing mutations of the location and determining the proportion of input cases that give rise to a different state from the non-mutated location. Propagation probability is estimated by determining the proportion of cases for which perturbed data states at the location propagate and give rise to faulty results.

It would seem, as a result of the connection between semantic fault size and the testability of a location L (Testability$_L$), that an alternative testability estimation procedure could be based just on considering versions of the chosen program with location L mutated. Let M_L denote a representative such mutant of location L. The mutation change, provided it does not generate an equivalent mutant, can be regarded as a seeded fault that has a semantic size in the same way as naturally occurring faults. The smallest semantic size of such mutants, being a worst case, could provide an estimate for testability at the location L, i.e. the following approximation holds:

$$\text{Testability}_L \approx \min\,(\text{semantic size of } M_L) \qquad \textbf{(15)}$$

where the minimum is over an appropriate set of non-equivalent mutants of location L. In practice this would require using a traditional (strong) mutation testing tool such as Mothra [5] in a slightly different mode from normal. It requires establishing a large number of input test cases chosen randomly from the input domain and then determining for each mutant generated by the tool, the proportion of test cases that kill that mutant. This is different from normal usage where, once a mutant is killed with some test case, no further test cases are applied to that mutant. Offutt and Hayes [12] did adopt this procedure to estimate the semantic size of all mutants created by the same mutation operator in an attempt to measure the size of given fault types. What is being suggested here, however, is slightly different again, in that mutants resulting from different mutation operators are grouped and considered according to the program location affected. The aim is to determine the minimum semantic size of all mutations at a location. Although still an expensive process, this has the merit, superficially at least, of being considerably more straightforward than using separate estimation procedures for the three components of the PIE model.

A study by Rothermel *et al.* [14] used mutation analysis in an apparently similar way to assess the fault exposing potential of a test suite of test cases with respect to locations (statements). However, the aim in that work was to prioritize *individual* test cases based on their ability to kill mutants rather than, as here, to assess testability of a location by measuring the proportion of *all* test cases in the domain that kill mutants of that location.

It is noted in passing that since propagation analysis is akin to *strong* mutation testing, and infection analysis is akin to *weak* mutation testing, a similar distinction could be made for semantic fault size. This would entail re-labelling the existing notion of semantic fault size as *strong* semantic fault size since it can be considered as the proportion of the input domain that causes the fault to produce erroneous output. On the other hand, *weak* semantic fault size can be considered as the proportion of the input domain that merely results in an infected data state immediately after executing a fault, i.e.

$$\text{Weak semantic fault size} = \frac{|I|}{|D|} \qquad \textbf{(16)}$$

5.2 Semantic Fault Size and DRR

Since semantic fault size has been related to the testability concept, it is interesting to speculate whether it could be related to the domain-to-range ratio (DRR). However, since semantic fault size depends solely on the input domain, whereas DRR depends on both the domain and the range, there is unlikely to be a direct connection. What can be deduced is a relationship involving fault size, measured in terms of input and output, and DRR both for the correct program and also for a faulty version when executed over just that portion of the domain that exposes the fault. Suppose, as in Section 2, that $\mathbf{P}_f$ is an incorrect version of some program $\mathbf{P}$ containing a single fault f and that $\mathbf{P}_f$ maps D_f into R_f . Then 'strong' semantic fault size which is based purely on the input domain is given by formula (12), repeated here for clarity:

$$\text{Input semantic fault size} = \frac{|D_f|}{|D|} \qquad \textbf{(17)}$$

However, one could also define a fault size based purely on the output range in the following manner:

$$\text{Output semantic fault size} = \frac{|R_f|}{|R|} \qquad \textbf{(18)}$$

Then denoting DRR for the correct program $\mathbf{P}$ with input domain D by $DRR_{\mathbf{P}D}$ and for faulty program $\mathbf{P}_f$ with just the fault-exposing input domain D_f by $DRR_{\mathbf{P}_f D_f}$ the following is obtained:

$$DRR_{\mathbf{P}D} = \frac{|D|}{|R|} = \frac{|D|}{|D_f|} \times \frac{|D_f|}{|R_f|} \times \frac{|R_f|}{|R|} \qquad \textbf{(19)}$$

Rearranging:

$$DRR_{\mathbf{P}D} = DRR_{\mathbf{P}_f D_f} \times \frac{\text{output fault size}}{\text{input fault size}} \qquad \textbf{(20)}$$

This equation captures the (admittedly) rather limited connection between DRR and semantic fault size.

6. OTHER RELATED WORK

This section briefly mentions some of the most significant related work (besides that already cited) which is concerned with fault models, fault propagation and fault-based testing.

The PIE model bears some similarity to the RELAY model [13] in which a fault *originates* a potential failure that must then *transfer* through computations to produce a *state failure* and ultimately be revealed as an external *failure*. Morell [11] developed a theory of fault-based testing that placed emphasis on fault propagation and then used symbolic testing to explore its limitations. The work of Goradia [8] was also concerned with fault propagation and a technique known as 'dynamic impact analysis' was formulated to

determine the extent of the effect of program components on the program output for a specific test case. Hamlet and Voas [9] showed just how useful a PIE testability estimate could be when used in conjunction with conventional reliability testing to provide, via so-called 'squeeze play', a confidence bound for the correctness of a program. On a more cautionary note however, they also provided a stark critique of the assumptions underlying the PIE model.

7. CONCLUSIONS

Testability is an important attribute of software as far as the testing community is concerned since its measurement leads to the prospect of facilitating and improving the testing process. Unfortunately testability has various guises. Two distinct and significant interpretations are due to Freedman [6] and Voas *et al.* [19]. Freedman's notion of testability has two facets, observability and controllability, both of which can be measured by the extent of certain modifications to a program component. Voas's notion of testability can be estimated by the computationally expensive PIE technique and Voas himself has suggested a possible link with the rather simpler concept of domain-to-range ratio.

By taking a functional view of software, this paper has produced a succinct characterisation of controllability and observability and developed a simple mathematical relationship involving them and the domain-to-range ratio. Semantic fault size has also been considered and its relationship with Voas's testability has been explored. A consequence of this is the suggestion that testability of a program location could be estimated more straightforwardly by a small adaptation of the traditional strong mutation testing process, to find the minimum semantic size of all mutants at the location. Finally some refinements of semantic size have been introduced and their relationship with DRR has been considered.

The authors recognise the desirability of validating the connections between the concepts as discussed here and this is the focus of ongoing work. Validation could take the form of empirical evidence, but could also consider a more analytical approach along the lines adopted by How Tai Wah [10] who has modelled software as finite functions to deduce theoretical results concerning fault coupling. In the meantime, this paper has made a limited start at putting together the various separate pieces of what might be considered a rather complex jigsaw of related concepts.

8. REFERENCES

[1] Bache, R. and Müllerburg, M., "Measures of testability as a basis for quality assurance", *Software Engineering Journal*, **5**(2), 86-92 (March 1990).

[2] Bainbridge, J., "Defining testability metrics axiomatically", *Software Testing, Verification and Reliability*, **4**(2), 63-80 (June 1994).

[3] Bertolino, A. and Strigini, L., "On the use of testability measures for dependability assessment", *IEEE Trans. on Soft. Eng.*, **22**(2), 97-108 (Feb. 1996).

[4] Boehm, B.W., Brown, J.R. and Lipow, M., "Quantitative evaluation of software quality", *Proc. 2nd Int. Conf. on Software Engineering*, San Francisco, U.S.A., IEEE Press, pp. 592-605 (Oct. 1976).

[5] DeMillo, R.A., Guindi, D.S., McCracken, W.M., Offutt, A.J. and King, K.N., "An extended overview of the Mothra software testing environment", *Proc. Second Workshop on Software Testing, Verification and Analysis*, Banff, Canada, IEEE Press, pp. 142-151 (July 1988).

[6] Freedman, R.S., "Testability of software components", *IEEE Trans. on Soft. Eng.*, **17**(6), 553-564 (June 1991).

[7] Friedman, M.A. and Voas, J.M., *Software Assessment: Reliability, Safety, Testability*, Wiley, New York, U.S.A. (1995).

[8] Goradia, T., "Dynamic impact analysis: a cost-effective technique to enforce error-propagation", *Proc. Int. Symp. on Software Testing and Analysis (ISSTA '93)*, Cambridge, MA, U.S.A., ACM Press, pp. 171-181 (June 1993).

[9] Hamlet, D. and Voas, J., "Faults on its sleeve: amplifying software reliability testing", *Proc. Int. Symp. on Software Testing and Analysis (ISSTA '93)*, Cambridge, MA, U.S.A., ACM Press, pp. 89-98 (June 1993).

[10] How Tai Wah, K.S., "A theoretical study of fault coupling", *Software Testing, Verification and Reliability*, **10**(1), 3-45 (March 2000).

[11] Morell, L.J., "A theory of fault-based testing", *IEEE Trans. on Soft. Eng.*, **16**(8), 844-857 (Aug. 1990).

[12] Offutt, A.J. and Hayes, J.H., "A semantic model of program faults", *Proc. Int. Symp. on Software Testing and Analysis (ISSTA '96)*, San Diego, U.S.A., ACM Press, pp. 195-200 (Jan. 1996).

[13] Richardson, D.J. and Thompson, M.C., "An analysis of test data selection criteria using the RELAY model of fault detection", *IEEE Trans. on Soft. Eng.*, **19**(6), 533-553 (June 1993).

[14] Rothermel, G., Untch, R.H., Chu, C. and Harrold, M.J., "Test case prioritization: an empirical study", *Proc. Int. Conf. on Software Maintenance (ICSM '99)*, Oxford, U.K., IEEE Press, pp. 179-188 (Aug. 1999).

[15] Voas, J.M., "PIE: a dynamic failure-based technique", *IEEE Trans. on Soft. Eng.*, **18**(8), 717-727 (Aug. 1992).

[16] Voas, J.M. and Miller, K.W., "Semantic metrics for software testability", *The Journal of Systems and Software*, **20**(3), 207-216 (March 1993).

[17] Voas, J.M. and Miller, K.W., "Software testability: the new verification", *IEEE Software*, **12**(3), 17-28 (May 1995).

[18] Voas, J.M., Miller, K.W. and Noonan, R., "Designing programs that do not hide data state errors during random black-box testing", *Proc. 5th Int. Conf. on Putting into Practice Methods and Tools for Information System Design*, Nantes, France (Sept. 1992).

[19] Voas, J.M., Morell, L.J. and Miller, K.W., "Predicting where faults can hide from testing", *IEEE Software*, **8**(2), 41-48 (March 1991).