INVESTIGATING THE PARTIAL RELATIONSHIP BETWEEN TESTABILITY AND THE DYNAMIC RANGE-TO-DOMAIN RATIO

Zuhoor A. Al-Khanjari[1]
Department of Computer Science,
College of Science,
Sultan Qaboos University,
PO BOX 36, Al-Khodh 123,
Muscat, Sultanate of Oman
Tel/fax: +968-515407
Email: zuhoor@squ.edu.om

Martin R. Woodward
Department of Computer Science,
University of Liverpool,
Chadwick Building, Peach Street,
Liverpool L69 7ZF, U.K.
Tel: +44-151-794-3676
Fax: +44-151-794-3715
Email: mrw@csc.liv.ac.uk

ABSTRACT:

The word 'testability' has been used variously in the software community to represent a number of different concepts such as how easy it is to test a program or how easy it is to achieve execution coverage of certain program components. Voas and colleagues have used the word to capture a slightly different notion, namely the ease with which faults, if present in a program, can be revealed by the testing process. The significance of this concept is twofold. First, if it is possible to measure or estimate testability, it can guide the tester in deciding where to focus the testing effort. Secondly, knowledge about what makes some programs more testable than others can guide the developer so that design-for-test features are built in to the software. The propagation, infection and execution (PIE) analysis technique has been proposed as a way of estimating the Voas notion of testability. Unfortunately, estimating testability via the PIE technique is a difficult and costly process. However, Voas has suggested a link with the metric, domain-to-range ratio (DRR).
This paper reviews the various testability concepts and summarises the PIE technique. A prototype tool developed by the authors to automate part of the PIE analysis is described and a method is proposed for dynamically determining the inverse of the domain-to-range ratio. This inverse ratio can be considered more natural in some sense and the idea of calculating its value from program execution leads to the possibility of automating its determination. Some experiments have been performed to investigate empirically whether there is a partial link between testability and this dynamic range-to-domain ratio (DRDR). Statistical tests have shown that for some programs and computational functions there is a strong relationship, but for others the relationship is weak.

Keywords: test and evaluation, testability, PIE technique, MSG-Infection tool, domain-to-range ratio, empirical study.

INTRODUCTION

Software testing, which represents the last stage in the software development process, is currently a very active research field. The intent of testing is ideally to uncover faults and increase confidence in the correctness of the tested code. Software testing is a very important approach in building reliable software systems. Clearly therefore, it is desirable that testing should be both as easy and as effective as possible. The word 'testability' has been used in various ways to reflect this desire. One particular interpretation, due to Voas *et al.* (1991), is to define testability as the ease with which faults, if present in a program, can be revealed by testing. Under this definition, programs with high testability reveal their faults easily, while those with low testability may contain faults that are very difficult to expose. To measure testability, in the sense just described, the propagation, infection and execution (PIE) analysis technique, has been proposed (Voas *et al.*, 1991).
Intriguing though this technique is, it is still a difficult and computationally expensive process. The details of which is given in Section 3. To obtain an indication of the testability of a program early in the software development process (i.e. from a specification or a design) and without actually performing the PIE analyses, Voas and Miller (1991) suggested use of a semantic metric, the domain-to-range ratio (DRR): the ratio of the cardinality of the possible inputs to the cardinality of the possible outputs. The aim of this paper is to provide further investigation of the partial relationship between testability and DRR. Others have previously explored the possible link between testability and static program measures.
The structure of the remainder of this paper is as follows. Section 2 surveys various uses of the testability concept in the literature of the software community. Section 3 provides a brief idea about

[1] Author to whom all correspondence should be addressed.

mutation technique. It helps in understanding the concept of the PIE technique. Section 4 gives a brief summary of the PIE analysis technique (Voas, 1992b). Section 5 considers the automation of testability measurement by introducing a prototype tool, MSG-Infection, which has been developed by the current authors for obtaining execution and infection estimates of C programs. A commercial tool called the PiSCES Software Testability Analysis Toolkit^{TM} (Friedman and Voas, 1995), which has been developed by the Reliable Software Technologies Corporation is also mentioned. Section 6 considers the DRR metric in more detail and suggests that its inverse, the range-to-domain ratio (RDR), is more 'natural' in this context. In addition, a scheme for calculating RDR dynamically from a program execution is proposed, leading therefore to the dynamic range-to-domain ratio (DRDR). Section 7 describes a series of experiments, which have been conducted to investigate the possible partial relationship between testability and the proposed metric DRDR. Section 8 draws some conclusions from the experiments and summarises the contribution of this paper.

THE TESTABILITY CONCEPT

The word 'testability' has been used with a variety of meanings. One of the earliest and probably most influential studies concerning quality attributes of software was performed by Boehm *et al.* (1976). In that study, testability was defined as the extent to which software "facilitates the establishment of verification criteria and supports evaluation of its performance". Boehm *et al.* linked testability with a number of lower level characteristics, namely: accountability, accessibility, communicativeness, self-descriptiveness and structuredness. Among the specific features suggested as promoting testability were: "support of intermediate output and echo-checking of inputs". In a later paper Boehm (1984) discussed testability in terms of eliminating vagueness in the statement of software requirements or in the specification of software.

The IEEE definition, as given in the *Glossary of Software Engineering Terminology*, reflects both the aspects of testability discussed by Boehm. The IEEE defines testability as: (1) "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met"; and (2) "the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met" (IEEE, 1990).

Interestingly in 1982 Weyuker defined the negative of testability (Weyuker, 1982). She focused on the absence of, or difficulties with, an output oracle. Software was deemed to be *non-testable* if either of the following two conditions occurred: (1) no output oracle exists; and (2) it is theoretically possible, but too difficult in practice, to determine output correctness.

In 1985 Howden provided a more formal approach to the definition of testability than any previous approaches (Howden, 1985). By further development of a test completeness criterion that he had identified in earlier work, Howden defined testability as the property that "a finite set of tests can be specified that will determine if the program that implements a function contains one of a specified set of faults". The quotation just given, summarising Howden's definition, is from an excellent survey on testing and verification by White (1987).

In 1991 Freedman discussed what he termed *domain testability* of software components (Freedman, 1991). He introduced two properties of software called observability and controllability. *Observability* is "the ease of determining if specified inputs affect the outputs"; *controllability* is "the ease of producing a specified output from a specified input" (Freedman, 1991). By measuring the semantic 'size' of extensions, or program modifications, that enable a component to achieve domain testability, Freedman was able to quantify both observability and controllability.

Wang *et al.* (1996) have also defined software testability metrics for observability and controllability, but by utilising the syntactic control structures in modules. Others have also focused on the internal control structure of software when discussing testability. For example, Bache and Müllerburg (1990) define a number of testability metrics for control-flow based testing strategies. In essence the metrics they proposed determine the minimum number of paths, and hence the minimum number of test cases, needed in order to satisfy particular coverage criteria. Bainbridge (1994) went on to show how the Bache and Müllerburg metrics could be calculated axiomatically.

Voas and colleagues have taken a radically different view of testability from most previous researchers (Voas *et al.*, 1991). They defined testability as the ease with which faults will manifest themselves as failures when the software undergoes the testing process. Software with high testability, according to this definition, exposes its faults easily, whereas software with low testability may contain faults which 'hide', i.e. are difficult to expose. Voas *et al.* went further by introducing an approach for measuring testability in terms of estimates from propagation, infection and execution (PIE) analyses of software. This PIE technique will be summarised in the next section. In the remainder of this paper, the word 'testability' is used entirely in the sense of Voas and colleagues.

Bertolino and Strigini (1996a) have rigorously investigated Voas's notion of testability and its use in dependability assessment. They modified Voas's definition and have given a more precise formulation of it. They then investigated the mathematics of using testability to estimate the probability of program correctness and the probability of failures. They have argued that high testability is not always a desirable program property and anyway, that program testability "can be more simply described in terms of assumptions on the probability distribution of the failure intensity of the program" (Bertolino and Strigini, 1996b). Despite this rather critical sentence, testability in all its guises, remains a notion that continues to intrigue many researchers and the intention of the current authors is to add to the debate.

MUTATION TECHNIQUE

Mutation testing is a powerful program testing technique used to generate good test data to uncover specific errors or classes of errors. The idea of mutation can be represented by simple changes made to a program, one at a time, e.g. changing the '-' operator to the '+' operator. In other words, the mutations represent different versions of the program, each of which differs from the original by a simple change. Test data should then be constructed in a way that distinguishes each mutant from the original program. This means that if the output of the mutant differs from the output of the original program then the mutant is termed *dead*. However if the outputs are the same, then the mutant is called *live* and needs further investigation. If it is the case that no test data could ever 'kill' a *live* mutant, then that mutant is called *equivalent* to the original program [DeMillo, Lipton and Sayward, 1978; Untch, Offutt and Harrold, 1993; Woodward, 1993].

Mutation testing, in the sense just described, which involves running the entire program till completion to get the result, is termed *strong* mutation testing. This is considered to be expensive and time consuming. As a result of this, *weak* mutation testing has been introduced. "Weak mutation requires that a mutant program component yields a different 'component outcome' on at least one execution" [Howden, 1982; Woodward, 1991].

THE PIE MODEL

This section summarises the propagation, infection and execution (PIE) model for measuring the sensitivity (or testability) of program locations. PIE is a white-box analysis technique based on the syntax and semantics of the code (Voas, 1992a; Hamlet and Voas, 1993). It makes predictions concerning future program behaviour by estimating the effect that input distribution, syntactic mutants and changed data values in data states have on current program behaviour (Voas and Miller, (1992a,1992b)).

The PIE assessment model implements the definition of testability promoted by Voas and colleagues (Voas and Miller, 1995; Voas, 1997; Voas and McGraw, 1998) by performing three independent dynamic analyses: execution, infection and propagation, which produce a set of estimates for each location of the given program. *Execution analysis* is the process for predicting the probability that a location is executed when inputs are selected according to a particular input distribution, *D* say. *Infection analysis* is the process for determining the probability that the succeeding data state of location *L* is different from the succeeding data state that a specific mutant creates, given that the original location and the mutant execute on a data state that would normally precede *L*. A data state is a collection of all variables and their associated values at some point during program execution. Once infection estimates are calculated for all mutants of the specified location, the minimum probability among all will be considered as the infection estimate of that location. Infection analysis is similar to weak mutation. *Propagation analysis* is the process concerned with determination of the probability that a forced change in an internal computational state causes a change in the program's output. Once again propagation estimates are calculated for all forced changes, the minimum probability among all will be considered as the propagation estimate of that location. Propagation analysis is similar to strong mutation. The three probability estimates can then be integrated to derive the sensitivity of each location and the overall testability of the program. The current authors have previously investigated the stability of the results from the PIE technique itself when some of the parameters involved in the measurement process are altered (Al-Khanjari and Woodward, 1998; Al-Khanjari *et al.,* 2002).

CURRENT TESTABILITY TOOLS

Automating the measurement of testability involves automating the three individual processes of sensitivity analysis: execution, infection and propagation analyses. This section introduces MSG-Infection, a prototype tool that has been developed by the current authors for automating infection and execution analyses of C programs. First, however, a commercial tool called PiSCES is briefly described.

The PiSCES tool

A commercial tool called the PiSCES Software Testability Analysis Toolkit^{TM}, that implements the PIE algorithms, has been developed by the Reliable Software Technologies Corporation of Sterling, Virginia (Friedman and Voas, 1995). It would appear that PiSCES is the only commercial software for testability determination; it evolved from various proof-of-concept prototypes (Voas *et al.*, 1993a). PiSCES is written in C++ for analysing C programs. It generates testability estimates by developing an instrumented copy of the original program which should then be compiled and executed. PiSCES performs its infection analysis by creating appropriate C programming language mutants.
The PiSCES Toolkit^{TM} is a combination of several individual tools or packages. One of the tools is the SafteyNet tool which incorporates propagation analysis to get an estimate for the fault tolerance of a program or indeed, for individual modules, functions, or even lines.

The MSG-Infection tool

A prototype tool called the MSG-Infection system, for determining both execution and infection analysis estimates, has been developed by the current authors. It uses a novel approach which is a modification of the mutant schemata generation (MSG) technique for mutation testing of programs. The mutant schemata generation approach is used to improve the performance of mutation analysis systems by generating one metaprogram, which contains all encoded mutants for a given program (Untch *et al.*, 1993; Untch, 1995). This approach based on program schema, which is a template. A partially interpreted program schema is defined by Baruch and Katz [Baruch and Katz, 1988]. The MSG-Infection system retains the spirit of the MSG approach by encoding a number of mutants of each location in one single modified version of the original program. It uses an existing MSG system for C programs (Flanagan, 1997) which has been adapted in order to develop the MSG-Infection system.

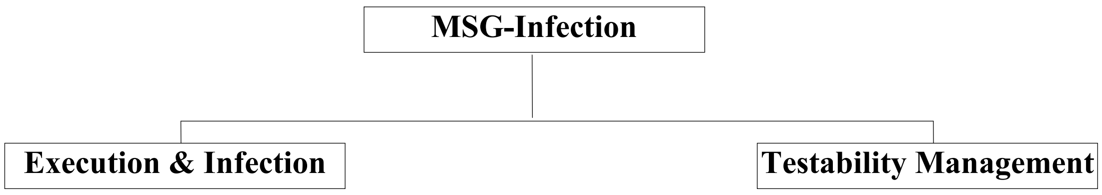


Figure 1. The MSG-Infection system and its components

The MSG-Infection system splits into two major subsystems: an Execution & Infection system and a Testability Management system (see Figure 1). The Execution & Infection system is, as has already been mentioned, a modification of an existing MSG system which develops a parse tree corresponding to the given C program and modifies it to encode a variety of mutants at each location. The parse tree, which is modified to construct the metamutant version of the given C program, has been further altered in the MSG-Infection system so that each location *L* of the original program now becomes, in essence, the following:

```
store pre-$L$;
execute $L$;
store post-$L$;
for each mutant $L_m$ of $L$
loop
    restore pre-$L$;
    execute $L_m$;
    compare post-$L_m$ with post-$L$;
end loop;
restore post-$L$;
```

In the above, 'pre-*L*' ('post-*L*') corresponds to the data state immediately before (after) location *L*. The result of each comparison of the post-L_m state with the post-*L* state is saved and used to determine the infection estimate of location *L*. The storing of the data state prior to a location is achieved by instrumenting with assignments to a special array which remains undisturbed by execution of the location or any of its mutants. The restore operation after the location can then use the array to recover the values that the variables had before the location. Note that, at present, only assignment statements are considered as locations by the MSG-Infection system.

The Testability Management system manages the process of running the test cases against the original locations of the program and the corresponding mutants. It generates a file which contains all the required infection and execution estimates for each location of the given program.

The MSG-Infection system currently incorporates four mutant classes: arithmetic operator replacement, variable replacement, constant replacement and statement deletion. Several important macros have been included in the MSG-Infection system to facilitate the compilation process and further macros can easily be added to the system. The system has been built in a highly modular fashion with loose coupling between the modules to make future development as easy as possible. The system is user-friendly, interacting with the user as necessary to build the modified version of the original program and complete the compilation and execution processes of this metamutant program with the given test cases.

THE DOMAIN-TO-RANGE RATIO AND ITS INVERSE

This section considers the domain-to-range ratio in more detail, suggests that its inverse the range-to-domain ratio is more natural and proposes a method for its dynamic determination of the latter.

Domain-to-range ratio (DRR)

The domain-to-range ratio (DRR) has been proposed by Voas and Miller (1991) as an approximate measure of implicit information loss. *Implicit information loss* occurs when two or more different values of a parameter are used as input to a built-in function or a user-defined function and generate the same output. Consider, for example, the function $f(a) = a$ **mod** 2. This function generates one of only two possible outputs for every input value of *a*, namely 0 when *a* is even and 1 when *a* is odd.

DRR is a specification or functional description metric that can be defined as the ratio d / r, where *d* is the cardinality of the domain of the specification and *r* is the cardinality of the range (Voas and Miller, 1991). It is a fundamental characteristic of both mathematical and computational functions and, despite the fact that it may not always be apparent from a software specification, it can often be derived from semantic information found in such a specification (Voas *et al.*, 1992; Voas *et al.*, 1993b). Although the DRR of a specification is fixed and can not be modified without changing the specification itself, the specification can be further decomposed into functions or modules, which can then be dealt with individually (Voas and Miller, 1991). It is possible to determine the DRR for unary operators, binary operators and more complex expressions. In this way it may be possible to determine the DRR at higher levels of granularity such as sub-specifications (i.e. specifications of modules), or even entire specifications (Voas, 1991). The ratio is static and is determined independently of the actual distribution of input values. Voas and Miller (1993b) note that DRR is related to Freedman's notion of controllability (Freedman, 1991). Indeed a controllable function can be considered as just the special case of a function *f* say, having DRR = 1, i.e. if the output values of *f* are different, then so are the corresponding inputs that generate those outputs. Because the domain-to-range ratio is available early in the software life cycle it could help in exploring for hidden faults using empirical methods when code is eventually produced (Voas and Miller, 1993a). Examples of DRRs for various functions, as originally given by Voas (though with corrections to the ranges of the circular functions 12, 13 and 14), are listed in Table 1 (Voas, 1991; Voas and Miller, 1993b). Continuing Voas's practice, ∞_I is used to represent the cardinality of the integers and ∞_R the cardinality of the reals.

Table 1. DRRs and testabilities of various functions as given by Voas

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* decreases
4. $f(a) = a \text{ div } b$	Low	$\infty_I : \infty_I / b$	testability decreases as *b* increases, *b* □ 0
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, *b* □ 0
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
12. $f(a) = \sin(a)$	Low	$\infty_I : \infty_R$	*a* is integer (degrees),
13. $f(a) = \tan(a)$	Low	$\infty_I : \infty_R$	*a* is integer (degrees),
14. $f(a) = \cos(a)$	Low	$\infty_I : \infty_R$	*a* is integer (degrees),
15. $f(a) = \text{odd}(a)$	Low	$\infty_I : 2$	*a* is integer
16. $f(a) = \text{not}(a)$	High	1 : 1	*a* is integer (0 or 1)

Although the DRR for a specification has its own intrinsic value, its significance is greatly enhanced by the prospect of a connection with the ease of exposing potential faults in a corresponding implementation. Voas *et al.* (1992) indicated that: "empirical observations suggest that a specification's DRR is related to its implementation's ability to hide faults". That is, high DRR implies that faults are more likely to remain hidden in an implementation. However, low DRR suggests that almost all faults in an implementation might be exposed during testing (Voas *et al.*, 1992). This means when DRDR is low the program becomes more testable and the errors become more easy to find and correct.

It has been argued by Voas *et al.* (1992) that this potential link between DRR and ease of fault exposure arises from data state error cancellation or simply from internal data state collapse. Data state error cancellation occurs if an infected variable has its erroneous value corrected or perhaps never used in subsequent computations. 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 (1993b) 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 likelihood 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.* (1992) conjectured that "the testability of a program is correlated with the domain/range ratio". More explicitly they stated 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. For each of the functions listed in Table 1, the testability as suggested by Voas is given in the second column of the table (Voas, 1991; Voas and Miller, 1993b).

Dynamic range-to-domain ratio (DRDR)

In order to assist with the investigation of the relationship between testability and domain-to-range ratio, which is the main focus of this paper, the current authors have adapted the ratio in two minor ways.

The first (trivial) adaptation is simply to invert the ratio so that it becomes the range-to-domain ratio (RDR), i.e. the range-to-domain ratio is r / d where r is the cardinality of the range and d is the cardinality of the domain. This has the advantage that if there is a link between the ratio and testability, it is likely to be of the form: high values in one lead to high values in the other and similarly low values in one lead to low values in the other. This is clearly more natural than dealing with a mathematical relationship based on reciprocals.

The second adaptation is to propose calculation of the range-to-domain ratio dynamically for individual program locations. In other words, the dynamic range-to-domain ratio (DRDR) is defined as the ratio of the cardinality of the range to the cardinality of the domain for a given program location when that program is executed with test data from a given input distribution. The principle advantage of calculating the ratio dynamically (i.e. from program execution) is that it opens up the possibility of its automatic determination during white-box testing of the code.

The remainder of this section illustrates the calculation of DRDR with an assignment statement as an example. Two possible methods are investigated: calculation using states and calculation using input/output tuples.

Calculation using states

Recall that it has been argued that there is a link between the domain-to-range ratio and 'internal state collapse' (Voas and Miller, 1993b). Hence it seems worth investigating whether counting unique data states both before and after some program location on all executions of that location during a run of the program might provide a measure of 'state collapse'.

Consider for example the assignment statement:

$$z = x * x + y * y \qquad (1)$$

with an input distribution such that: x, y = -1, 0, +1. Also assume that z is initialised to zero prior to this location. Then *pre-states*, the set of states involving the data values for variables x, y, z (given in angle brackets in that order) is as follows:

$$\textit{pre-states} = \{ \begin{array}{lll} <-1, -1, 0>, & <-1, 0, 0>, & <-1, 1, 0>, \\ <0, -1, 0>, & <0, 0, 0>, & <0, 1, 0>, \\ <1, -1, 0>, & <1, 0, 0>, & <1, 1, 0> \} \end{array}$$

The set of corresponding states after execution of the location, *post-states*, is:

$$\textit{post-states} = \{ \begin{array}{lll} <-1, -1, 2>, & <-1, 0, 1>, & <-1, 1, 2>, \end{array}$$

$$< 0, -1, 1 >, \quad < 0, 0, 0 >, \quad < 0, 1, 1 >,$$
$$< 1, -1, 2 >, \quad < 1, 0, 1 >, \quad < 1, 1, 2 > \}$$

Thus:

$$\text{DRDR} = \frac{|post - states|}{|pre - states|} = \frac{9}{9}$$

Since the number of unique pre-states is the same as the number of unique post states, this would seem to imply that there is no reduction in the number of unique states for this location under these circumstances. Clearly this contravenes expectations and indicates that, in general, if the inputs to a location are not altered, then there is *no* state collapse.

Calculation using input/output tuples

Since counting unique states before and after a program location does not lead to an appropriate way of determining DRDR, an alternative approach is required. The approach adopted in this paper is based on counting unique tuples of input operands for the domain of a location and unique tuples of output operands for the range.

At present, the only locations considered are assignment statements, as has already been mentioned, although in principle other types of location could be handled in a similar way. The range of an assignment location is the set of values assigned to the variable in question. The number of values that are unique then represents the cardinality of the range. Determining the domain and its cardinality is a little harder. The current authors propose ascertaining all the data objects that may be considered input operands of the location. Each execution of the location makes use of a tuple of values for these input operands. The number of unique input tuples will then be considered as the cardinality of the domain. The choice of the word 'operand' in the explanation just given is deliberate. The intention is to include constant data values that are inputs to a location as well as any variable data values. The reason for this is so that the assignment of a constant value to a variable may be considered to have input domain of cardinality one.

The net effect of this entire approach for calculating the cardinality of the domain of an assignment location may be summarised as follows: (1) if a location assigns a constant to a variable, then the domain of this location has cardinality 1; (2) if the location assigns one variable to another, then the cardinality of the domain will be the number of unique values of the input variable; (3) if the location assigns to a variable using a combination of variables and constants, then the cardinality of the domain will be the number of unique combinations of the input variables and constants; (4) if the assignment involves a built-in function, then the cardinality of the domain will be the number of unique tuples which are used as parameters to the built-in function.

To demonstrate how the calculations are performed, consider once again the assignment (1) with the same input distribution as given previously. The variable z constitutes the range and it achieves three unique values. The variables $< x, y >$ constitute the tuple of input operands and there are nine unique combinations of values. Therefore:

$$|R| = \text{card}\{ 0, 1, 2 \} = 3$$

$$|D| = \text{card}\{ < -1, -1 >, \quad < -1, 0 >, \quad < -1, 1 >,$$
$$< 0, -1 >, \quad < 0, 0 >, \quad < 0, 1 >,$$
$$< 1, -1 >, \quad < 1, 0 >, \quad < 1, 1 > \} = 9$$

Hence:

$$\text{DRDR} = \frac{|R|}{|D|} = \frac{3}{9} = 0.33$$

which is clearly a more sensible interpretation of events than that given in the previous section. As another example, consider:

$$z = 100 \qquad (2)$$

to demonstrate the calculation when assigning a constant to a variable. For all possible executions of this location:

$|R| = \text{card}\{100\} = 1$

$|D| = \text{card}\{100\} = 1$

Hence:

$$\text{DRDR} = \frac{|R|}{|D|} = \frac{1}{1} = 1$$

As far as implementation of an automated system to capture the DRDR for program locations is considered, all that is required is the insertion of statements before and after each location to record the appropriate values of data objects in some file or files. The files can then be processed on termination of program execution to determine uniqueness of the domains and ranges of each location and report the corresponding values of DRDR. This scheme could be generalised to sections of code, modules or even individual program paths once inputs and outputs are known.
It is worth noting that the required 'print' statements are not unlike simple assertions which have long been advocated (Stucki and Foshee, 1975) and can be incorporated into an instrumented version of the original program with little difficulty. Hence the DRDR metric is inexpensive and relatively easy to calculate. If it can generate estimates that relate to testabilities of the chosen locations without conducting the full PIE analysis, it will represent a significant saving of effort.

EMPIRICAL STUDIES

This section discusses the investigations, which have been undertaken to explore the link between infection and the metric, DRDR. First the goal, the hypothesis and the significance of the experiment are given. Then details are provided concerning the programs and test input distributions that have been used in the experiment. This is followed by a description of the experimental method. Finally, the results of the experiment are reported and analysed.

Goal of the experiment

The goal of this experiment is simply to investigate whether there is any relationship between infection and the DRDR metric. If there is a relationship, then, in addition, it would be useful to know whether the two quantities follow the same or opposite trend. Such a connection, if confirmed, could help in estimating the infection of programs without performing the actual calculations of the infection analysis which is part of the PIE technique.

Hypothesis of the experiment

The hypothesis of the experiment is that the DRDR score of program locations is correlated with the infection estimates of those locations. In other words it is conjectured that the higher the DRDR score, the higher the infection and vice versa.

Significance of the experiment

If the DRDR metric can give some indication concerning the infection of a program location, which is part of the sensitivity, this would have significant potential for savings on the effort, time and cost involved in conducting the analyses of the full PIE technique. Assuming that the DRDR metric is a reflection of infection in some fashion, then the DRDR metric could perhaps be used as a much simpler substitute for it and yet be useful in the same way that infection with conjunction with execution and propagation analyses are useful, namely for indicating those program locations where faults may 'hide' and further testing is desirable.

The programs used in the experiment

Eleven programs written in C were selected for the experiment. The programs may be thought of as composed of three pairs: (1) three non-artificial programs; (2) three programs that implement the functions of Table 1 in an unconnected fashion and the fourth program in this set represents the example program, which is concerned with checking reachability constraints, provided by Friedman and Voas [Friedman and Voas, 1995]; and (3) four programs that implement connected combinations of the same functions. For each of the eleven programs, Table 2 lists the name, purpose, number of locations and, in the final column, the number of test cases used and their input domains.

The first program (Average.c) determines the average of a set of numbers supplied as input data. The second program (Quadratic.c) determines whether a quadratic equation, whose coefficients are supplied as input data, has an integer solution. A deliberately erroneous version of this program has been used extensively in previous studies that showed it has very low overall testability (Voas *et al.*, 1991; Al-Khanjari and Woodward, 1998). The third program (Cubic.c) determines whether a cubic equation, whose coefficients are supplied as input data, has an integer solution.

The fourth program (Functions1.c) implements all the simple mathematical functions considered by Voas (1991) and also Voas and Miller (1993b) and repeated in Table 1 of this paper. In essence, each function is an assignment location and there is no connection between the assignments. The program requires a and b as test input – note b is not constant. The fifth program (Functions2.c) implements just the functions numbered 3, 4 and 9, i.e. a **mod** b and a **div** b where a, b are integer, and a / b where a is real. These three functions depend on two input parameters, but in this particular program b is kept fixed. Separate evaluations were performed with a range of values for b. The sixth program (Functions3.c) implements similar set of the simple mathematical functions used in (Functions1.c) program. The seventh program (Functions4.c) implements the example program provided by Friedman and Voas to check the reachability constraints.

The final four programs (Artificial1.c, Artificial2.c, Artificial3.c and Artificial4.c) are artificial in the sense that they were constructed specially for the experiment. They contain uses of several of the simple functions in connected combinations in the knowledge that these functions give a good spread for the scores of both infection and DRDR

Table 2. The programs selected for the experiment

Name	Purpose	Locations	Test cases
Average.c	Calculates the average of a set S of values; *next* identifies each value in turn.	4	100 cases card S $\in [2,16]$ $next \in [0,6000]$
Quadratic.c	Determines whether a quadratic equation has an integer root.	4	10000 cases $a, c \in [0,10]$, $b \in [1,1000]$
Cubic.c	Determines whether a cubic equation has an integer root.	9	1000 cases a, b $\in [-1000,1000]$
Functions1.c	Implementation of all 16 of the simple mathematical functions given in Table 1.	16	10000 cases $a, b \in [-10000,10000]$
Functions2.c	Implementation of functions: a **mod** b, a **div** b and a / b (b constant) from Table 1.	3	10000 cases $a \in [-10000,10000]$
Functions3.c	Implementation of some of the simple mathematical functions given in Table 1.	8	10000 cases $a \in [-1000,1000]$

Functions4.c	Implementation of the example program provided by Friedman and Voas to check the reachability constraints.	6	10000 cases $a, b \in [-100,100]$
Artificial1.c	Artificial program constructed from connected combinations of the functions in Table 1.	7	1000 cases $a, b \in [-10000,10000]$
Artificial2.c	Artificial program constructed from connected combinations of the functions in Table 1.	9	1000 cases $a, b \in [-10000,10000]$
Artificial3.c	Artificial program constructed from connected combinations of the functions in Table 1.	7	1000 cases $a, b \in [-1000,1000]$
Artificial4.c	Artificial program constructed from connected combinations of the functions in Table 1.	10	10000 cases $a, c \in [0,10]$ $b \in [1,1000]$

Experimental method

For the purposes of comparison between infection and DRDR scores, the authors have observed that the infection estimate is the most crucial component of the PIE testability for a given program location. Also as mentioned previously the MSG-Infection tool, which uses the mutant schemata approach, has been generated mainly to improve the mutation analysis and accordingly to improve the performance of the infection analysis. As a reflection the experiments have concentrated on infection analysis only. On the other hand, as authors observed that for some programs the infection analysis will be zero or close to zero. This means conducting such experiments will be waste of time. Because of that the purpose of the experiments was to find some concepts such as DRDR, which might give an influence of the expected infection estimate without conducting the real analysis.

For each assignment location in each of the chosen programs the DRDR scores and the infection analysis estimates were obtained. The number of test cases used for each program is listed in Table 2; each test case was constructed randomly using a uniform distribution within the input domains also listed in Table 2. Each C program together with its input data (test cases) was submitted to the MSG-Infection system and the infection estimate results were reported automatically. To determine the DRDR scores however, a special instrumented version of each C program was created, which, when executed, generated files containing the domains and the ranges for each individual location of the C program under test. A simple post-processor tool was then used to determine the unique domain and range tuples and hence the DRDR scores.

Experimental results

This section reports the results of the experiment. For each of the eleven programs selected for the experiment, the infection analysis estimates and corresponding DRDR scores for each program location are presented as pairs of graphs.

Figures 2 through 11 show infection estimates and corresponding DRDR scores for all locations of the programs Average.c, Quadratic.c, Cubic.c, Functions1.c, Functions3.c, Functions4.c, Artificial1.c, Artificial2.c, Artificial3.c and Artificial4.c respectively.

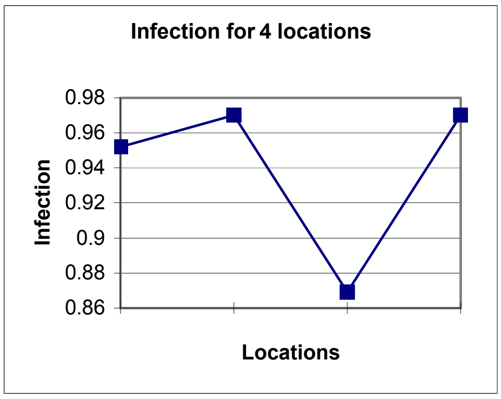


Figure 2(a). Infection for Average.c

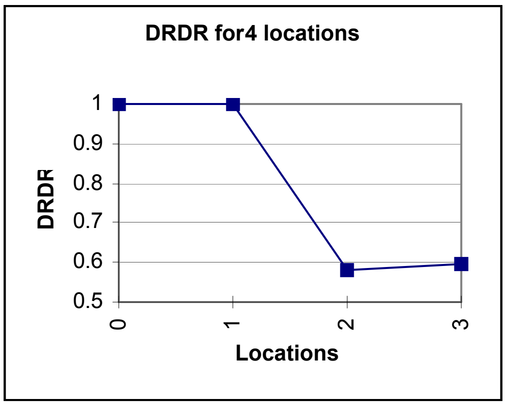


Figure 2(b). DRDR for Average.c

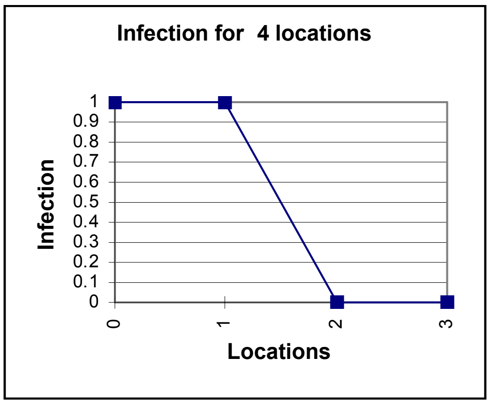


Figure 3(a). Infection for Quadratic.c

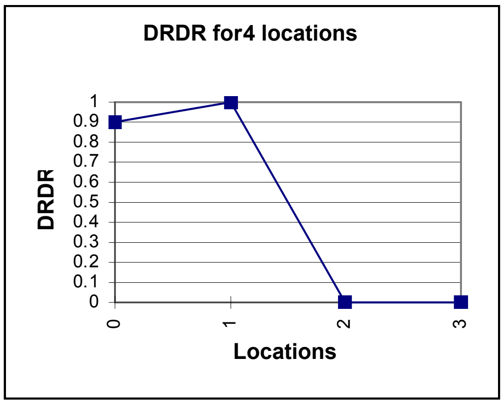


Figure 3(b). DRDR for Quadratic.c

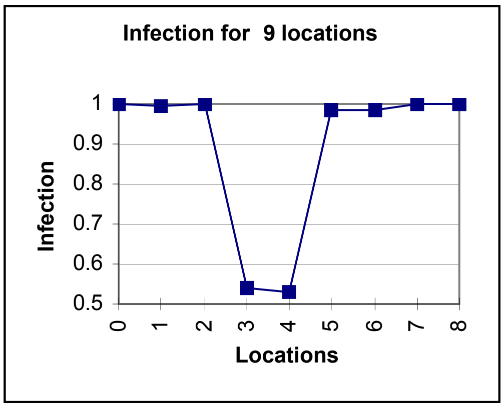


Figure 4(a). Infection for Cubic.c

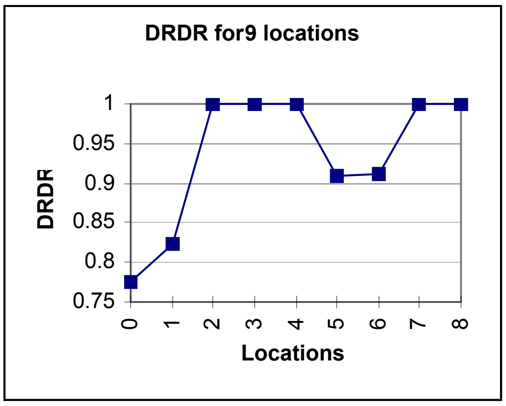


Figure 4(b). DRDR for Cubic.c

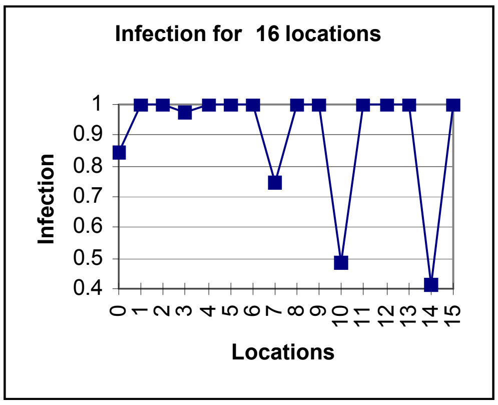


Figure 5(a). Infection for Functions1.c

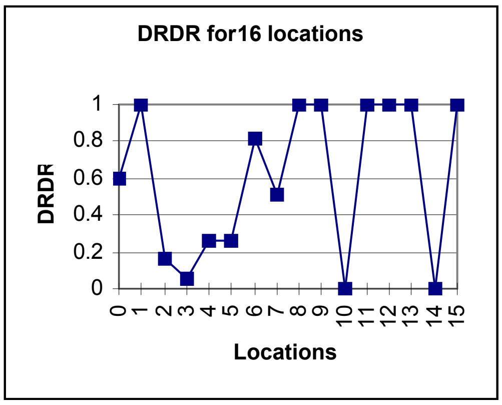
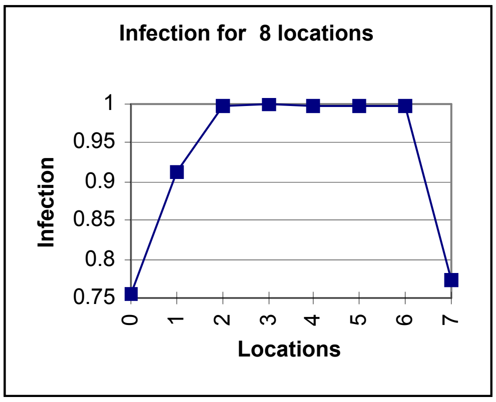


Figure 5(b). DRDR for Functions1.c



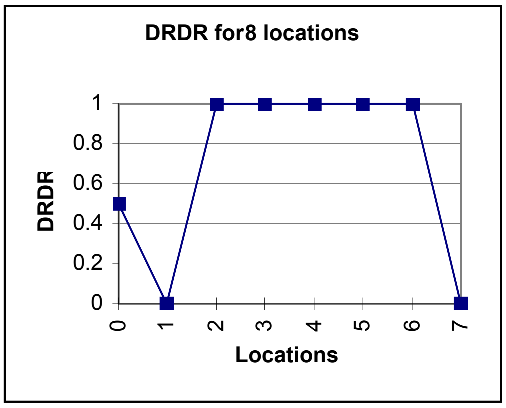
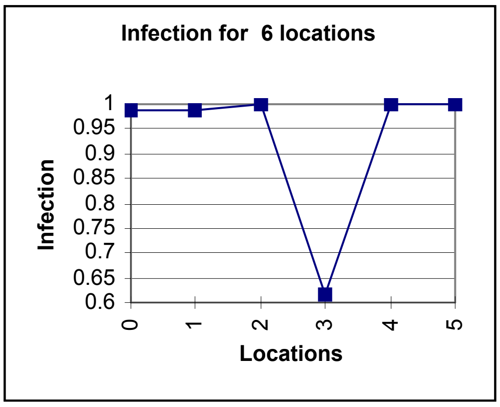


Figure 6(a). Infection for Functions3.c

Figure 6(b). DRDR for Functions3.c



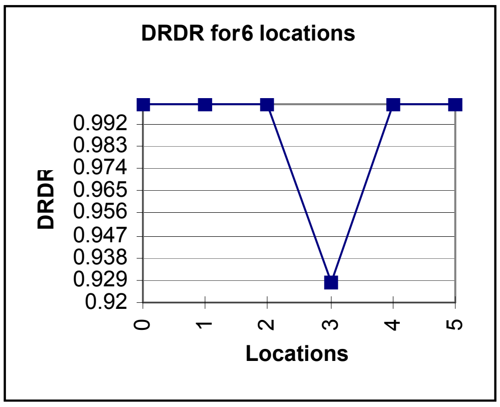


Figure 7(a). Infection for Functions4.c

Figure 7(b). DRDR for Functions4.c

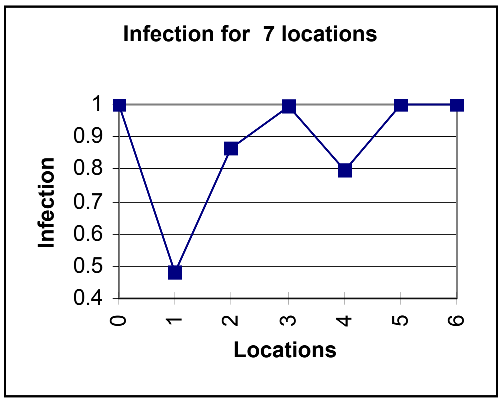


Figure 8(a). Infection for Artificial1.c

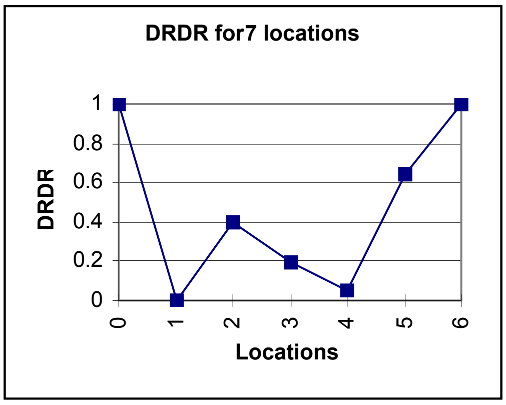


Figure 8(b). DRDR for Artificial1.c

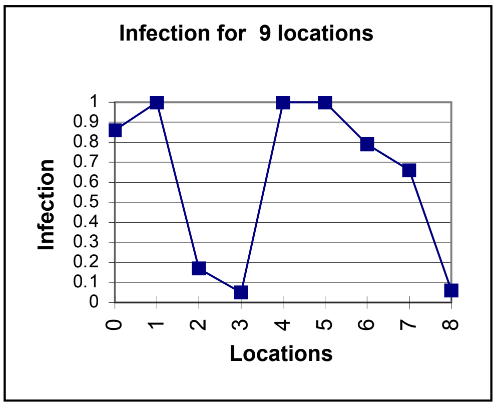


Figure 9(a). Infection for Artificial2.c

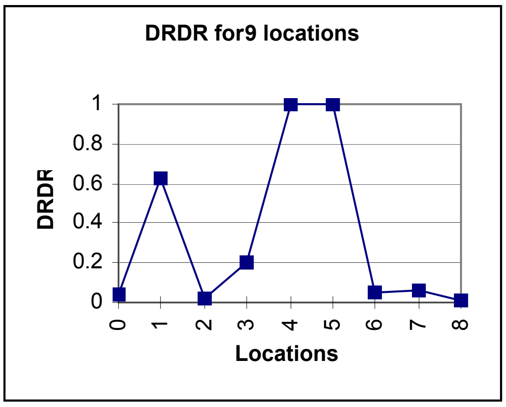


Figure 9(b). DRDR for Artificial2.c

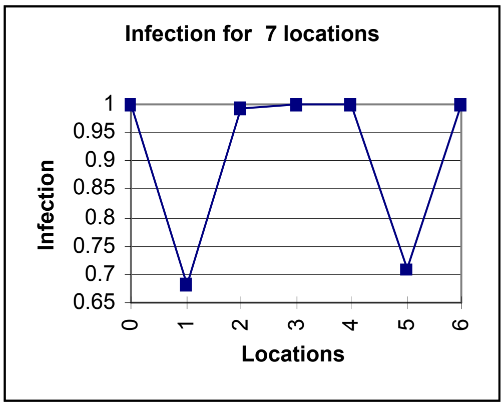


Figure 10(a). Infection for Artificial3.c

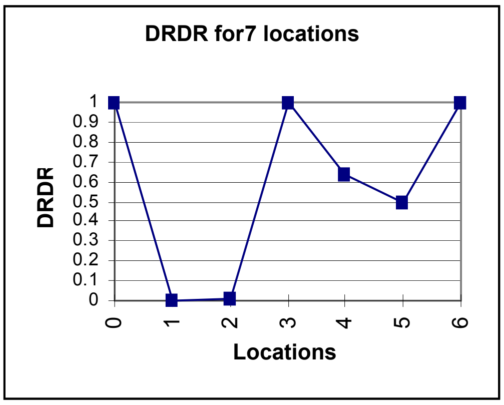
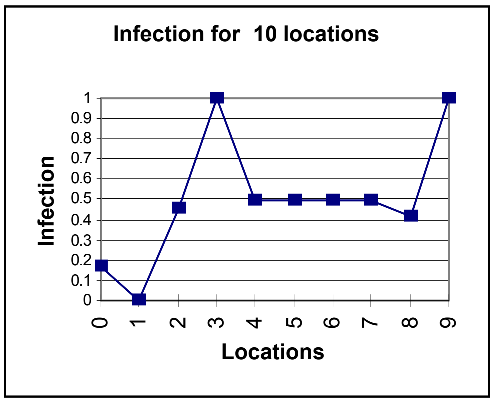


Figure 10(b). DRDR for Artificial3.c



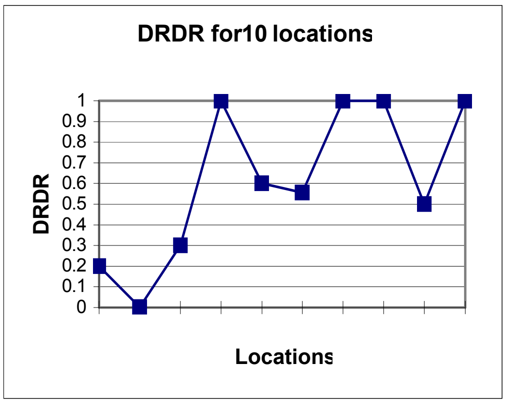
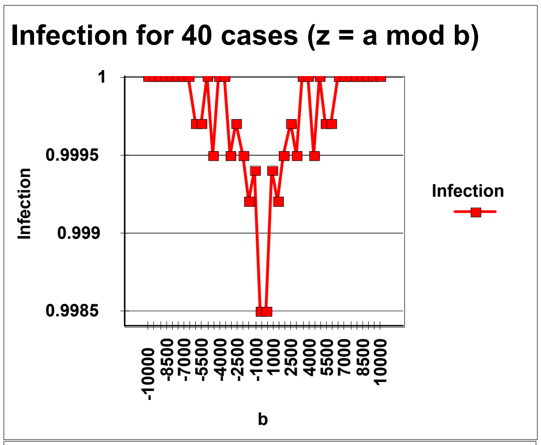


Figure 11(a). Infection for Artificial4.c **Figure 11(b).** DRDR for Artificial4.c

Finally infection estimates and corresponding DRDR scores for the remaining program Functions2.c, which is a special case, are presented with reference to its individual locations. This is because these locations represent different isolated functions involving a second parameter *b*, which has been considered as a constant when determining the infection and DRDR scores. The most interesting functions are *a* **mod** *b* and *a* **div** *b*. Results are reported for 40 different cases of the variable *b*, where the value of *b* varies from -10000 to +10000 in steps of 500.

Figures 12 and 13 show infection estimates and corresponding DRDR scores for the *a* **mod** *b* and *a* **div** *b* functions of the Functions2.c program respectively, for the 40 different cases of the variable *b*.



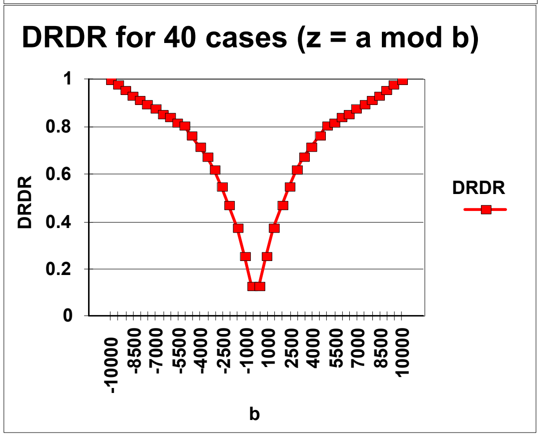
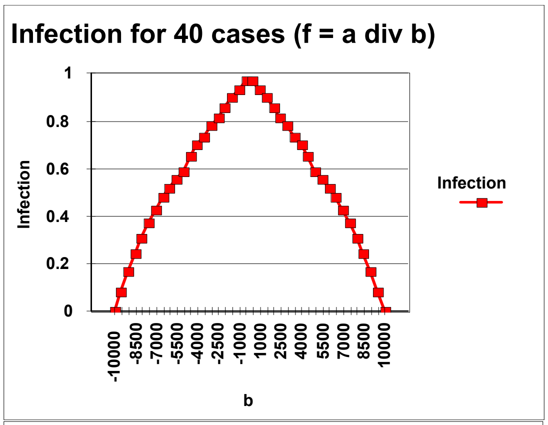


Figure 12(a). Infection for Functions2.c **Figure 12(b).** DRDR for Functions2.c



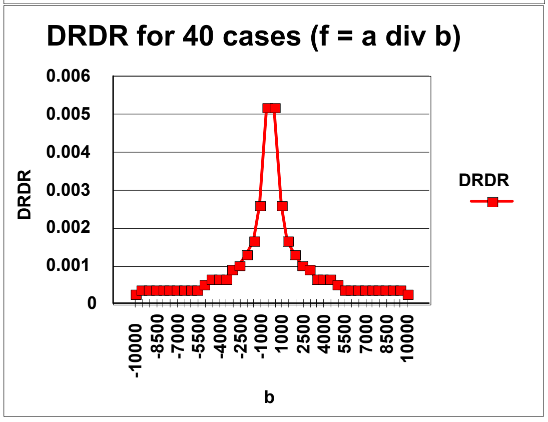


Figure 13(a). Infection for Functions2.c **Figure 13(b).** DRDR for Functions2.c

Analysis of results

Viewing the figures, one can see that there are often similarities between the infection and the DRDR graphs for a given program. They frequently take the same shape even though the scales may be different.
Since the figures suggest that a correlation between infection and DRDR is plausible, statistical analysis of the corresponding values has been conducted. The correlation coefficients for the selected programs are given in Table 3.

Table 3. Correlation (r^2) between infection and DRDR for the selected programs

Program	Functions	r^2	Confidence	Relationship
Average.c	All	0.2700	75%	Weak positive linear relationship
Quadratic.c	All	0.9946	95%	Strong positive linear relationship
Cubic.c	All	0.1742	50%	Weak positive linear relationship
Functions1.c	All	0.3446	95%	Weak positive linear relationship
Functions2.c	*a* **mod** *b*	0.6859	95%	Substantial positive linear relationship
Functions2.c	*a* **div** *b*	0.4013	95%	Weak positive linear relationship
Functions3.c	All	0.5764	95%	Substantial positive linear relationship
Functions4.c	All	0.9982	99%	Strong positive linear relationship
Artificial1.c	All	0.4852	90%	Weak positive linear relationship
Artificial2.c	All	0.3980	90%	Weak positive linear relationship
Artificial3.c	All	0.3051	80%	Weak positive linear relationship
Artificial4.c	All	0.6561	95%	Substantial positive linear relationship

From Table 3, the percentages of the correlation that explain the variations in the infection using the DRDR scores are about 27%, 99%, 17%, 34%, 69%, 40%, 58%, 100%, 49%, 40%, 31% and 66% of the selected programs respectively. As further evidence, the student's t-distribution test, with the confidence level given in Table3 for the selected programs, has been used to check the null hypothesis (that there is no relationship between the variables: infection and DRDR), against the alternative hypothesis (that there is a relationship between the suggested variables). The results of the tests were to reject the null hypothesis and accept the alternative hypothesis. This means that there is a relationship between infection and DRDR. An indication of the strengths of the relationship for each of the selected programs, or particular function within a program, is also given in Table 3.

SUMMARY AND CONCLUSIONS

This paper has summarised and reviewed some of the many interpretations of the concept of 'testability' before focusing on the particular definition introduced by Voas and colleagues. The propagation, infection and execution (PIE) method of determining Voas's notion of testability of program locations has been briefly explained and a prototype system called MSG-Infection developed by the authors for automating infection and execution analyses has been introduced. MSG-Infection is novel in being an adaptation of an existing Mutant Schemata Generation (MSG) system.

Since it has been suggested that there is a possible link between testability and the so-called domain-to-range ratio (DRR) metric, an attempt has been made to investigate partial relationship between infection and DRDR empirically. The authors have argued that the infection component of the PIE testability estimate can be considered the most crucial component for this investigation. To assist this investigation the authors have proposed two minor changes: (1) inverting the metric, to make the potential partial relationship direct rather than reciprocal, and (2) measuring it dynamically, to help automate its determination. The resultant metric, the dynamic range-to-domain ratio (DRDR), has been determined for the assignment locations in a number of programs. The infection estimates have also been determined for the same locations using the MSG-Infection system.

The experiment utilised a mixture of artificial and non-artificial programs. The artificially constructed programs were designed to incorporate, either in isolation or in various combinations, a number of simple mathematical functions studied theoretically by Voas. Although in most cases the trends shown by both infection and DRDR appear similar, when analysed statistically the correlation shows variation from a strong linear relationship to a weak linear relationship. The strongest correlation observed, somewhat surprisingly, was for the non-artificial program Quadratic.c which determines whether a quadratic equation has integral solutions. The next strongest correlation observed, perhaps less surprisingly, was for the function *a* **mod** *b*. The graph of infection values as *b* varies provides pleasing confirmation of the expectation that testability decreases as *b* decreases and the corresponding graph of DRDR provides dramatic evidence of the same trend, even though the numeric values are on different scales. However, the other programs demonstrate much weaker correlation between infection and DRDR.The tentative conclusion is that, whilst it is intuitively tempting to suppose there might be a link, it would appear that any such link is far less direct than one might hope. Hence, although this study has shed some light on the partial relationship between testability and the domain-to-range ratio, further research is certainly required.

ACKNOWLEDGEMENTS

Zuhoor Al-Khanjari gratefully acknowledges the support of Sultan Qaboos University and the government of the Sultanate of Oman. Both authors thank Steven Flanagan for permission to use his MSG system.

REFERENCES

Al-Khanjari, Z.A. and Woodward, M.R. (1998) "Investigations into the PIE Testability Technique", **Proceedings of the 4th International Conference on Achieving Quality in Software (AQuIS '98)**, Venice, Italy, April 1998, IEI-CNR, pp 25-34.

Al-Khanjari, Z.A., Woodward, M.R. and Ramadhan, H. (2002) "Critical Analysis of the PIE Testability Technique", **Software Quality Journa**l, December, Vol. 10, No. 4, pp 97-108, Kluwer Academic Publishers.

Bache, R. and Müllerburg, M. (1990) "Measures of testability as a basis for quality assurance", **Software Engineering Journal**, Vol. 5, No. 2, pp 86-92.

Bainbridge, J. (1994) "Defining testability metrics axiomatically", **Software Testing, Verification and Reliability**, Vol. 4, No. 2, pp 63-80.

Baruch, O. and Katz, S. (1988) "Partially Interpreted Schemas for CPS Programming". **Science of Computer Programming**, Vol. 10, No. 1, pp 1-18.

Bertolino, A. and Strigini, L. (1996a) "On the use of testability measures for dependability assessment", **IEEE Transactions on Software Engineering**, Vol. 22, No. 2, pp 97-108.

Bertolino, A. and Strigini, L. (1996b) "Acceptance criteria for critical software based on testability estimates and test results", **Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP '96),** Vienna, Austria, 1996, Springer, Berlin, Germany, pp. 83-94.

Boehm, B.W. (1984) "Verifying and validating software requirements and design specifications**",** **IEEE Software**, Vol. 1, No. 1, pp 75-88.

Boehm, B.W., Brown, J.R. and Lipow, M. (1976) "Quantitative evaluation of software quality", **Proceedings of the Second International Conference on Software Engineering**, San Francisco, U.S.A., 1976, IEEE Computer Society Press, Los Alamitos, California, U.S.A., pp 592-605.

DeMillo, R.A., Lipton, R.J. and Sayward, F.G. (1978) "Hints on test data selection: help for the practicing programmer", **IEEE Computer**, Vol. 11, No. 4, pp 33-41.

Flanagan, S.J. (1997) Mutation Testing Using Mutant Schemata, BSc dissertation, Computer Science Department, University of Liverpool, U.K.

Freedman, R.S. (1991) "Testability of software components", **IEEE Transactions on Software Engineering**, Vol. 17, No. 6, pp 553-564.

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

Hamlet, D. and Voas, J. (1993) "Faults on its sleeve: amplifying software reliability testing", **Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA '93)**, Cambridge, Massachusetts, U.S.A., June 1993, ACM SIGSOFT Software Engineering Notes, Vol. 18, No. 3, pp 89-98.

Howden, W.E. (1982) "Weak mutation testing and completeness of program test sets", **IEEE Transactions on Software Engineering**, Vol. 8, No. 4, pp 371-379.

Howden, W.E. (1985) "The theory and practice of functional testing", **IEEE Software**, Vol. 2, No. 5, pp 6-17.

IEEE (1990) **Glossary of Software Engineering Terminology**, IEEE Computer Society Press, Los Alamitos, California, U.S.A.

Stucki, L.G. and Foshee, G.L. (1975) "New assertion concepts for self-metric software validation", **ACM SIGPLAN** Notices, Vol. 10, No. 6, pp 59-71.

Untch, R.H. (1995) Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method, PhD thesis, Department of Computer Science, Clemson University, South Carolina, U.S.A.

Untch, R.H., Offutt, A.J. and Harrold, M.J. (1993) "Mutation analysis using mutant schemata", **Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA '93)**, Cambridge, Massachusetts, U.S.A., June 1993, ACM SIGSOFT Software Engineering Notes, Vol. 18, No. 3, pp 139-148.

Voas, J.M. (1991) "Factors that affect software testability", **Proceedings of the 9th Pacific Northwest Software Quality Conference**, Portland, Oregon, U.S.A., pp 235-247.

Voas, J.M. (1992a) "Dynamic testing complexity metric", **Software Quality Journal**, Vol. 1, No. 2, pp 101-114.

Voas, J.M. (1992b) "PIE: a dynamic failure-based technique", **IEEE Transactions on Software Engi**neering, Vol. 18, No. 8, pp 717-727.

Voas, J.M. (1997) "Software testability measurement for intelligent assertion placement", **Software Quality Journal**, Vol. 6, No. 4, pp 327-335.

Voas, J.M. and McGraw, G. (1998) **Software Fault Injection: Inoculating Programs Against Errors**, Wiley, New York, U.S.A.

Voas, J.M. and Miller, K.W. (1991) "Improving software reliability by estimating the fault hiding ability of a program before it is written", **Proceedings of the 9th Software Reliability Symposium**, Denver Section of the IEEE Reliability Society, Colorado Springs, U.S.A., 1991.

Voas, J.M. and Miller, K.W. (1992a) "The revealing power of a test case", **Software Testing, Verification and Reliability**, Vol. 2, No. 1, pp 25-42.

Voas, J.M. and Miller, K.W. (1992b) "Improving the software development process using testability research", **Proceedings of the 3rd International Symposium on Software Reliability Engineering**, Research Triangle Park, North Carolina, U.S.A., 1992, pp 114-121.

Voas, J.M. and Miller, K.W. (1993a) "Applying a dynamic testability technique to debugging certain classes of software faults", **Software Quality Journal**, Vol. 2, No. 1, pp 61-75.

Voas, J.M. and Miller, K.W. (1993b) "Semantic metrics for software testability", **The Journal of Systems and Software**, Vol. 20, No. 3, pp 207-216.

Voas, J.M. and Miller, K.W. (1995) "Software testability: the new verification", **IEEE Software**, Vol. 12, No. 3, pp 17-28.

Voas, J.M., Morell, L.J. and Miller, K.W. (1991) "Predicting where faults can hide from testing", **IEEE Software**, Vol. 8, No. 2, pp 41-48.

Voas, J.M., Miller, K.W. and Noonan, R. (1992) "Designing programs that do not hide data state errors during random black-box testing", **Proceedings of the 5th International Conference on Putting into Practice Methods and Tools for Information System Design,** Nantes, France, 1992.

Voas, J.M., Miller, K.W. and Payne, J.E. (1993a) "Software testing and its application to avionic software", **Proceedings of Computers in Aerospace 9**, CA. Publisher: AIAA, San Diego, U.S.A., 1993.

Voas, J.M., Miller, K.W. and Payne, J.E. (1993b) "An empirical comparison of a dynamic software testability metric to static cyclomatic complexity", **Proceedings of the 18th Annual Software Engineering Workshop, 1993**, NASA-Goddard Software Engineering Laboratory Series Report 93-003.

Wang, Y., King, G., Staples, G., Ross, M. and Court, I. (1996) "Towards a metric of software testability", **Proceedings of the 5th Software Quality Conference**, Dudhope Castle, University of Abertay Dundee Business School, Dundee, Scotland, U.K., July 1996, pp 234-241.

Weyuker, E.J. (1982) "On testing non-testable programs", **The Computer Journal**, Vol. 25, No. 4, pp 465-470.

White, L.J. (1987) "Software testing and verification", **Advances in Computers**, Vol. 26, pp 335-391.

Woodward, M.R. (1991) “Concerning Ordered Mutation Testing of Relational Operators”, **Journal of Software Testing, Verification & Reliability,** Vol. 1, No. 3, pp 35-40.

Woodward, M.R. (1993) “Mutation Testing-its Origin and Evaluation”, **Journal of Information and Software Technology**, Vol. 35**,** No. 3, pp 163-169.