# Measures of testability as a basis for quality assurance

by Richard Bache and Monika Müllerburg

**Program testing is the most used technique for analytical quality assurance. A lot of time and effort is devoted to this task during the software lifecycle, and it would be useful to have a means for estimating this testing effort. Such estimates could be used, on one hand, for guiding construction and, on the other, to help organise the development process and testing. Thus the effort needed for testing is an important quality attribute of a program; we call it its testability. We argue that a relevant program characteristic contributing to testability is the number of test cases needed for satisfying a given test strategy. We show how this can be measured for glass (white) box testing strategies based on control flow. In this case, we can use structural measures defined on control flowgraphs which can be derived from the source code. In doing so, we bring together two well researched areas of software engineering: testing strategies and structural metrication.**

## 1 Introduction

In recent years a growing interest has developed in software quality assurance (QA) [1], since it is now recognised that rigorous guarantees of quality are necessary both to protect the user and to reduce the cost to the supplier incurred by high maintenance and reworks. Existing QA certification procedures, such as BS5750 [2], rely on monitoring the process via documented plans, procedures and checklists. It is now acknowledged [3–5] that procedural QA cannot be sufficient. It is also necessary to have measurable objectives of quality which the product must achieve.

The quality model proposed by McCall [6] suggests that high-level user-view attributes may be broken down into measurable properties of the software product. Researchers at CSSE and GMD support this view [7] and are attempting to define these measurable properties and investigate the impact these have on quality. Examples of such properties are *structural complexity* [8], *intra-system reuse* and *connectivity*. In this paper we look at another quality attribute, testability. We define testability w.r.t. control-flow-based test data selection strategies and show how this may be measured as a property of the code or low-level design. In doing so we bring together two well researched areas of software engineering: testing strategies and structural metrication.

## 2 Testability as a quality attribute

Quality assurance has a constructive and an analytic aspect: the constructive aspect considers the question of how to build a program of a certain quality, whereas the analytic deals with the question of how to check whether the required quality has been achieved.

The most used technique for analytical quality assurance is program testing, and much time and effort in the software development process is consumed by this task. Thus, it would be useful to have a means for estimating the effort needed for the testing phase. These estimates could be used, on the one hand, for guiding the construction by making the program easier to test, and on the other, as a help for organising the development process.

### *2.1 Program testing*

The purpose of testing is to uncover faults which have arisen in the construction stages. This is done by selecting a subset of all possible inputs to a program or subprogram and ensuring the outputs associated with given inputs are consistent with what the specification asserts. If the subset of inputs and outputs is found to be consistent with the specification, then confidence is increased in the correctness of the program. This subset of inputs is known as the *test data*.

In general, the more test data the greater the confidence will be in the correctness of the program. Of course, there is more to selecting test data than arbitrarily choosing a large sample of inputs. There are a number of well established test strategies [9, 10] which give rules to generate test data that will be effective in showing up faults where they exist.

There are two well known strategies for selecting test data: black box (or functional) testing and glass box (or white box or structural) testing. For black box testing, we derive the test data from the specification, analysing in particular the specified function and its partial functions, the domain and the subdomains, and the function's range. We use strategies such as equivalence testing, random testing, boundary testing and special value testing [11]. For glass box testing, we derive the test data from the program analysing the program paths, the data flow and the expressions and data values, i.e. the program states. We use strategies

such as statement, branch and path testing.

We may distinguish the following classes of test data selection strategies (or criteria) for glass box testing:

- **Control flow testing**; considering sequences of statements, such as 'execute each loop free path at least once'.
- **Data flow testing**; considering sequences of data usage (definition-reference or set-use sequences), such as 'execute each set-use sequence at least once'.
- **Predicate testing**; considering the values of Boolean subexpressions in predicates, such as 'execute each permutation of the values of the individual conditions in a decision'.
- **Computation testing**; considering the values of subexpressions in computations, such as 'let each expression in a computation statement assume at least two different values'.

A test strategy identifies certain items/features in the program and then finds sufficient data so that during the test run all of these items are executed. Examples of these *items* are program statements, arithmetic expressions or paths. So, for example, one strategy is to visit every statement in the program at least once.

Fortunately, we can measure whether the test data are sufficient for satisfying the selected test strategy. We may measure *test coverage* (or *test effectiveness*), w.r.t. a given strategy, as

$$\frac{\text{number of required items executed}}{\text{number of required items in the program}}$$

In many cases a coverage of 1 (total coverage) may not be attainable, for instance, because of resource constraints.

Both the extent of coverage and the likelihood of faults being uncovered in a given program or subprogram, with a specific set of test data, depend on two factors:

- the nature of the test data,
- the properties of the program.

We shall examine the way in which attributes of the program can be measured so as to determine a property of the program which we call its *testability*.

## 2.2 *The flowgraph model*

Glass box testing strategies are defined either by the control flow, the data flow or the states of a program. Although all of these strategies are worthwhile and have individual merits, we shall concentrate on control flow strategies. This is because, in this case, we have a theory which we can apply to these strategies (see Section 3). The control flow can be described by the flowgraph model: an imperative language program can be modelled as a directed graph where the nodes represent statements (or parts of statements) in the program and edges represent the flow of control.

Examples of testing strategies based on control flow are as follows:

- execute every node at least once,
- pass through every edge at least once,

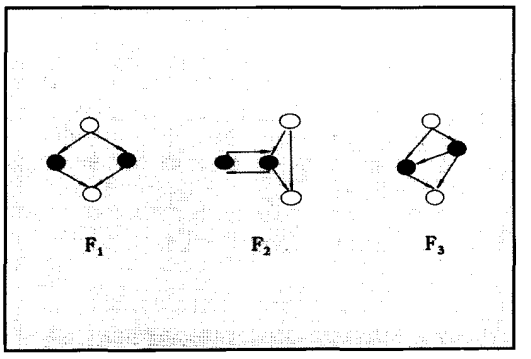


**Fig. 1. Three flowgraphs**

- travel every path at least once,
- go once around each loop.

## 2.3 *Measuring testability*

If we want to use measurement, we have to determine the quality attribute which we want to measure. Here, we want to measure the effort needed for testing a particular piece of software, and we call this property of the program its testability. The testability of a program is an important quality attribute, since it determines the effectiveness of testing, and so the likely correctness of the program. We argue that testability w.r.t. control flow testing strategies is of interest in two ways:

- In the design and implementation phases by choosing a design which maximises the testability we can improve the test coverage within limited resources. Of course, we have to be careful. If we used only control flow testability measures, the designer is likely to hide the complexity in the data.
- We may use testability measures for prediction. Thus, during the construction we could give an estimate of the time and effort needed in the testing stages. Since testing consumes much time and effort in the development process, testability measures could give a useful support for management.

In addition, we have to identify software characteristics which contribute to this attribute. In our case, the thesis is that the number of test cases (or test runs) needed for

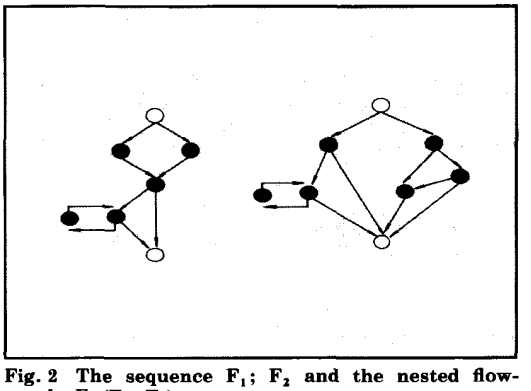

**Fig. 2 The sequence $F_1$; $F_2$ and the nested flowgraphs $F_1$ ($F_2$, $F_3$)**

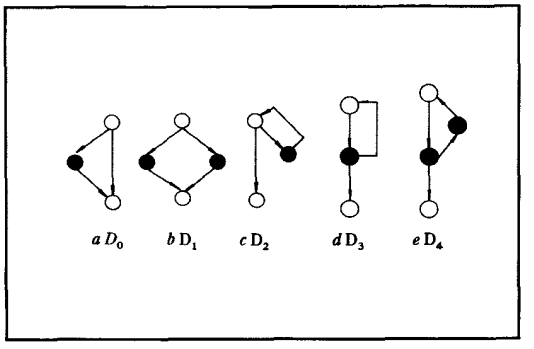


**Fig. 3**
*a* IF-THEN *b* IF-THEN ELSE *c* WHILE-DO *d* REPEAT-UNTIL *e* EXIT FROM MIDDLE

satisfying a test criterion is an important factor of the testing effort. Thus, we define testability for the control-flow-based test strategies as the number of paths (or test cases) required by the strategy. More precisely, we define testability as the minimum number of test cases to provide total test coverage, assuming that such coverage is possible. We should be aware that the program's property of testability is related to a particular test strategy, i.e. a statement that a program is testable is useless unless we also specify the particular test strategy.

Finally, we have to know how we can calculate the measurement values, i.e. we have to define a function for computing these values. In this paper, we define such a function for measuring testability w.r.t. control-flow-oriented test strategies. The testability with respect to control-flow-based strategies can be determined by measuring the flow graph. This is a well researched area, and a common framework exists for defining *all* flow graph metrics.

## 3 Fenton-Whitty flowgraph theory

### *3.1 Sequencing and nesting of flowgraphs*

The Fenton-Whitty theory shows how a flowgraph may be split up uniquely into maximal subflowgraphs. The theory

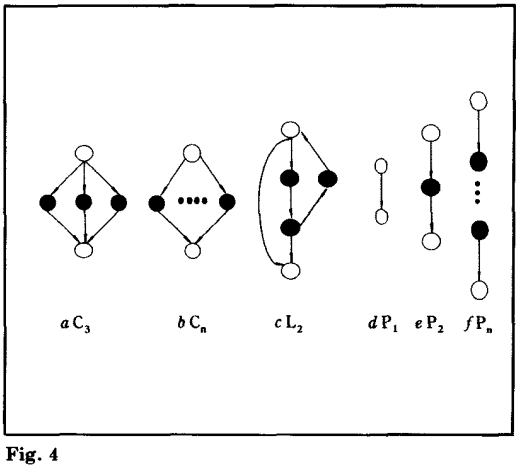


**Fig. 4**
*a* CASE-OF-3 *b* CASE-OF-n *c* 2-EXIT-LOOP *d e f* SEQUENCES

is mathematically rigorous and contains proven theorems, but here we give only an informal summary. (For a more formal description the reader should see References 12 and 13).

*Definition 1:* A flowgraph $(G, a, z)$ is a directed graph, $G$, with two distinguished nodes — the startnode $a$ and the stopnode $z$ — such that every node lies on a path from the startnode to the stopnode. The stopnode has outdegree zero. (The outdegree is the number of edges leaving a given node).

There are three types of nodes which occur in a flowgraph:

- **Selection nodes** which have an outdegree of two or more. These represent a branching of control based on the evaluation of an expression.
- **Procedure nodes** which have an outdegree of one and represent actions such as input, output and assignment as well as procedure calls.
- **The stopnode** which has outdegree zero and is unique.

The key concept in this theory is the decomposition of flowgraphs. The easiest way to explain this is by the opposite concept of composition. Two flowgraphs can be composed in one of two ways: sequencing and nesting. When two flowgraphs are sequenced (written $F_1; F_2$), the stopnode of the first is joined onto the startnode of the second. When one or more flowgraphs $F_1 \cdots F_n$ are nested onto a flowgraph $F$ (written $F(F_1 \cdots F_n)$), $n$ of the procedure nodes in $F$ are replaced by the flowgraphs $F_1$ to $F_n$. Figs. 1 and 2 give examples.

The operations of sequencing and nesting have a natural interpretation in the programming domain. In Pascal sequencing corresponds to joining two code segments with a semicolon. Nesting corresponds to replacing a single statement with a block delimited by BEGIN and END. The flowgraph which has two nodes and one edge, and corresponds to a program with just one statement, is called the *trivial flowgraph* or $P_1$. Nesting the trivial flowgraph onto any procedure node of any flowgraph leaves that flowgraph as it is.

$$F(P_1) = F$$

We define a family of flowgraphs $P_n$, where $n \geqslant 1$, as those flowgraphs with $n + 1$ nodes and $n$ edges which represent programs that are sequences of simple statements with no decisions. $P_1$ is a special case. We note that

$$F_1; \ldots; F_n = P_n(F_1, \ldots, F_n)$$

However we must treat sequencing separately from nesting to ensure unique decomposition.

### *3.2 Prime flowgraphs*

An important concept is that of a prime flowgraph.

*Definition 2:* A prime flowgraph is one which cannot be created by either sequencing or nesting other non-trivial flowgraphs.

There are an infinite number of primes, and it is possible to find a prime of arbitrary size. In language with unrestricted GOTO it is possible (although not advisable) to

create very large primes, but in GOTOless Pascal only a few prime flowgraphs can occur.

The prime flowgraphs correspond to the basic control constructs, some or all of which occur in every ALGOL-like language (see Figs. 3 and 4). The $P_n$s are not prime when $n > 1$, but are included for completeness.

Note that the standard definition of Pascal does not have an exit from middle loop, but this is present in S-ALGOL as the REPEAT-WHILE-DO construct. The 2 exit loop is present in ADA as LOOP-EXIT-WHEN.

### 3.3 *Decomposition*

The fundamental result of the Fenton-Whitty theory is that any flowgraph can be created from the repeated sequencing and nesting of primes and that this composition is unique. Conversely, any flowgraph can be decomposed uniquely into prime flowgraphs of which it can be made. An example is shown in Fig. 5. We can express this decomposition as a tree where the nodes of the tree are the primes and the edges represent the actions of sequencing and nesting. What is more, the decomposition tree is isomorphic to the flowgraph. It is possible to derive one from the other. Of course, in the case where a program has been developed using only 'structured' constructs, the decomposition tree is directly recoverable from the syntactic parse of the program. This is not true, however, in general, and where GOTO statements are used the program's prime decomposition can only be computed by appealing to the model such as described here.

### 3.4 *Hierarchical metrics*

In addition to providing an insight into the structure of a program, the decomposition tree can be used to define structural metrics. Since the tree is isomorphic to the flowgraph, in principle any metric which can be defined on the flowgraph can also be defined to the tree. However, it turns out that defining metrics on the tree is very straightforward. It is sufficient to define three function schemata, and the metrics can be calculated recursively on the tree. The following schema will permit the definition of all hierarchical metrics as defined by Prather [14]. This class contains most

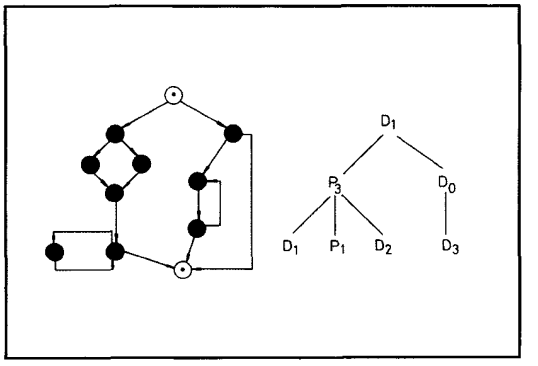


**Fig. 5 A flowgraph and its unique decomposition**

flowgraph-based metrics.

- For each prime $P$ define a metric value $\mu(P)$.
- For flowgraphs $F_1 \cdots F_n$ define a sequencing function $g$ such that

$$\mu(F_1; \ldots; F_n) = g(\mu(F_1), \ldots, \mu(F_n))$$

- For a prime $P$ and flowgraphs $F_1, \ldots, F_m$ define a nesting function $h$ such that

$$\mu(P(F_1, \ldots, F_m)) = h(P, \mu(F_1), \ldots, \mu(F_m))$$

Particular attention must be paid to defining values for primes. In some cases the values will be calculated from simple formulae derived from properties of the graph. In other cases each named prime will be given a specific value. In the latter case you must define values for all the primes you expect to encounter before measurement can begin.

For example, to define the number of levels of nesting which is an important structural attribute

- ☐ $\mu(P_1) = 0$; $\mu(Q) = 1$ for any prime $Q \neq P_1$
- ☐ $\mu(F_1; \ldots; F_n) = max\{\mu(F_1), \ldots, \mu(F_n)\}$
- ☐ $\mu(P(F_1, \ldots, F_m)) = max\{\mu(F_1), \ldots, \mu(F_m)\} + 1$

This framework has been used to define structural metrics [15, 16] which capture either simple attributes, such as size

**Table 1 Metric values for primes**

| Test strategy | $\mu(P_1)$ | $\mu(D_0)$ | $\mu(D_1)$ | $\mu(C_n)$ | $\mu(D_2)$ | $\mu(D_3)$ | $\mu(D_4)$ | $\mu(L_2)$ |
|---|---|---|---|---|---|---|---|---|
| All paths testing | 1 | 2 | 2 | n | — | — | — | — |
| Visit-each-loop path testing | 1 | 2 | 2 | n | 2 | 2 | 2 | 4 |
| Simple path testing | 1 | 2 | 2 | n | 2 | 1 | 1 | 3 |
| Structured testing | 1 | 2 | 2 | n | 2 | 2 | 2 | 3 |
| Branch testing | 1 | 2 | 2 | n | 1 | 1 | 1 | 2 |
| Statement testing | 1 | 1 | 2 | n | 1 | 1 | 1 | 1 |

**Table 2 Sequencing function**

| | |
|---|---|
| All paths testing | $\prod_{i=1}^{n} \mu(F_i)$ |
| Visit-each-loop path testing | " |
| Simple path testing | " |
| Structured testing | $\sum_{i=1}^{n} \mu(F_i) - n + 1$ |
| Branch testing | $max(\mu(F_1), \ldots, \mu(F_n))$ |
| Statement testing | " |

or depth of nesting or metrics, which capture some cognitive structural ordering — the so-called complexity metrics. We will now see how it can be used to measure testability.

## 4 A family of testability measures

### 4.1 *Control flow testing strategies*

Control flow testing strategies consider sequences of statements, subpaths and (complete) paths. *Branch testing*, requiring every edge of the control flow graph to be executed at least once, is the most used strategy in practice, since it is easy to realise. However, it is not very satisfactory, in particular w.r.t. loops. On the other hand, the strongest control flow strategy — *all path testing* (requiring every path to be executed at least once) — in most cases is not possible because loops in general lead to an infinite number of paths. Other strategies have been invented to fill the gap between branch testing and all paths testing.

Formally, we will discuss the following set of control flow testing strategies:

- **all paths testing** requires each path to be executed at least once. This is, of course, not always possible, since a program containing a loop may have an infinite number of paths.
- **visit-each-loop paths testing** requires a subset of paths such that control flows both straight past the loop and around the loop at least once.
- **simple path testing** is a strategy defined by Prather [14], requiring the execution of each path which does not contain the same edge more than once.
- **structured testing** is a strategy defined by McCabe [17], requiring the execution of a set of linearly independent paths.
- **branch testing** requires that each edge in the flowgraph be visited at least once.
- **statement testing** requires that each node of the flowgraph be visited at least once.

### 4.2 *Defining the testability measures*

In Section 2, we defined testability for control-flow-based test strategies as the minimum number of test cases or paths to provide total test coverage, assuming that such coverage is possible. The metrics function calculating the metrics values on a flowgraph $F$ is called $\mu(F)$. We will show how we can measure testability for each of the above test strategies using the example of branch testing. (The simple path testing metric is derived in Reference 14).

According to the Fenton-Whitty theory, we can define the metrics by defining three function schemata, i.e. by defining the metrics values for primes, a sequencing function and a nesting function. First, we define a *metric value* for each prime $\mu(P)$, as in Table 1. We include here a basic set of primes which is, in general, sufficient for constructing programs. Nevertheless, if we find a program which cannot be represented by using only primes from this set, it is straightforward to extend the metric values to additional primes.

Here we define the metric values for the example of branch testing; the metric values for the other test strategies are defined in the same way. As defined above, branch testing requires that every edge of the control flow graph is covered at least once.

- For prime $P_1$ all edges lay on the only existing path, i.e. $\mu(P_1) = 1$.
- For primes $D_0$ and $D_1$ both existing paths have to be executed in order to cover all edges, i.e. $\mu(D_0) = \mu(D_1) = 2$.
- For prime $C_n$ we have to execute each path in order to cover all edges, and thus the metric value is $\mu(C_n) = n$.
- For primes $D_2$, $D_3$ and $D_4$ one path is sufficient to cover all edges, i.e. the metric value is $\mu(D_2) = \mu(D_3) = \mu(D_4) = 1$.
- For prime $L_2$ we need two paths for covering all edges, i.e. the metric value is $\mu(L_2) = 2$.

Second, we define a *sequencing function* for flowgraphs $F_1, \ldots, F_n$, as in Table 2.

If we build a flowgraph $F$ by putting $n$ flowgraphs $F_1, \ldots, F_n$ in sequence, the paths of this new flowgraph $F$ are built by combining all the paths of all the single flowgraphs $F_i$. Thus, for computing the number of paths of the new flowgraph $F$ we have to multiply the number of paths of each single flowgraph $F_i$.

However, for branch testing we do not need all of these paths. We can cover all edges by executing paths which are combined from a path of each single graph. Thus, the sequencing function is

$$\mu(F_1; \ldots, F_n) = max(\mu(F_1, \ldots, F_n)).$$

Third, we define a *nesting function* for each prime, as in Table 3. This too can be extended for other primes.

We again show how to define the function for the example of branch testing. The nesting function depends on the prime.

- If we nest two flowgraphs $F_1$ and $F_2$ onto a prime $D_1$, the number of paths of the new flowgraph is the sum of the number of paths from the two nested flowgraphs. For branch testing, all these paths are required.

$$\mu(D_1(F_1, F_2)) = \mu(F_1) + \mu(F_2)$$

- If we nest $n$ flowgraphs $F_i$ onto a prime $C_n$, the number of paths of the new flowgraph is the sum of the number of paths from all the nested flowgraphs. For branch testing, all these paths are required.

$$\mu(C_n(F_1, \ldots, F_n)) = \sum_{i=1}^{n} \mu(F_i)$$

**Table 3 Nesting function**

| Test strategy | $\mu(D_1(F_1, F_2))$ | $\mu(C_n(F_1, \ldots, F_n))$ | $\mu(D_0(F))$ |
|---|---|---|---|
| All paths testing | $\mu(F_1) + \mu(F_2)$ | $\sum_{i=1}^{n} \mu(F_i)$ | $\mu(F) + 1$ |
| Visit-each-loop path testing | " | " | " |
| Simple path testing | " | " | " |
| Structured testing | " | " | " |
| Branch testing | " | " | " |
| Statement testing | " | " | $\mu(F)$ |

| Test strategy | $\mu(D_3 \& (F))$ | $\mu(D_3 \& (F))$ | $\mu(D_4(F_1, F_2))$ | $\mu(L_2(F_1, F_2))$ |
|---|---|---|---|---|
| All paths testing | — | — | — | — |
| Visit-each-loop path testing | $\mu(F) + 1$ | $\mu(F) + \mu(F)^2$ | $\mu(F_1) + \mu(F_1)^2\mu(F_2)$ | $1 + \mu(F_1) + \mu(F_1)\mu(F_2) + \mu(F_1)^2\mu(F_2)$ |
| Simple path testing | " | $\mu(F)$ | $\mu(F_1)$ | $1 + \mu(F_1) + \mu(F_1) \times \mu(F_2)$ |
| Structured testing | " | $\mu(F) + 1$ | $\mu(F_1) + \mu(F_2)$ | $\mu(F_1) + \mu(F_2) + 1$ |
| Branch testing | 1 | 1 | 1 | 2 |
| Statement testing | 1 | 1 | 1 | 1 |

- If we nest a flowgraph $F$ onto a prime $D_0$, the number of paths of the new flowgraph is determined by the number of paths of the nested flowgraph and the one path for the missing else part. For branch testing, all these paths are required.

$$\mu(D_0(F)) = \mu(F) + 1$$

- If we nest a flowgraph $F$ onto a prime $D_2$, $D_3$ or $D_4$, all edges can be covered by executing some path which enters the loop.

$$\mu(D_2(F)) = \mu(D_3(F)) = \mu(D_4(F)) = 1$$

- If we nest two flowgraphs $F_1$ and $F_2$ onto a prime $L_2$, branch testing requires two paths since there are two possible paths to reach the stop node. The two paths could be, for instance, the path directly leaving the loop and the other iterating the loop and leaving it at the second decision.

$$\mu(L_2(F_1, F_2)) = 2$$

### 4.3 *Computing the testability measures*

Now it is very easy to recursively compute the testability measures from the decomposition tree. We will show this for the example in Fig. 5. In this case, the function to be computed is

$$\mu(F) = \mu(D_1((D_1, P_1D_2), D_0(D_3)))$$

For computing the testability metric values, let us look at the examples of path testing, visit-each-loop path testing and branch testing.

- **All paths testing** is not possible because of loops.
- For **visit-each-loop testing**, we first use the nesting function for prime $D_1$ transforming the formula into

$$\mu(F) = \mu((D_1; P_1; D_2)) + \mu(D_0(D_3)).$$

Using the sequencing function and the nesting function for prime $D_0$, we get

$$\mu(F) = \mu(D_1)\mu(P_1)\mu(D_2) + \mu(D_3) + 1$$

Using the metric values for the primes and simple arithmetic, we get

$$\mu(F) = 2 \times 1 \times 2 + 2 + 1 = 7$$

- For **branch testing**, again we first use the nesting function for prime $D_1$ transforming the formula into

$$\mu(F) = \mu((D_1; P_1; D_2)) + \mu(D_0(D_3))$$

Using the sequencing function and the nesting function for prime $D_0$, we get

$$\mu(F) = max(\mu(D_1), \mu(P_1), \mu(D_2)) + \mu(D_3) + 1$$

Using the metric values for the primes and simple arithmetic, we get

$$\mu(F) = max(2, 1, 2) + 1 + 1 = 4$$

### 4.4 *Discussion of the results for testing*

The testability measures show very clearly that there are problems with some of the test strategies.

- Branch testing is an insufficient testing strategy for programs with loops because the number of required paths is reduced by a loop.
- Simple path testing is not satisfactory for loops with middle exits (see prime $D_4$). Since a complete loop traversal would require an edge to be traversed twice, the second part

of the loop must not be executed at all. In fact, this testability measure has been defined only for Dijkstra constructs.

On the other hand, we also easily recognise that exits from the body of loops (as in primes $D_4$ and $L_2$) may cause problems for testing because they may require very many paths.

## 5 Automation

The calculation of these metrics can be easily automated, both to save time and to avoid arithmetric errors which tend to be associated with a large volume of calculations. The QUALMS system devloped in the CSSE as part of Alvey project SE/69 [18] computes the decomposition tree for an arbitrary flowgraph using a fast algorithm [19]. The system also computes structural metrics defined on these trees. In order to input actual programs to QUALMS, 'front ends' are required to model programs as flowgraphs. Front ends currently exist for C, FORTRAN [20] and Kindra (a graphical design language devised by BT [21]) and are under development for Pascal and CORAL 66. All the testability measures can be calculated by QUALMS.

## 6 Conclusions and further work

Quality assurance by measurement of the product is our ultimate goal. To this end we have presented measures of one quality product is our ultimate goal. To this end we have presented measures of one quality attribute, control flow testability. Furthermore, these metrics are easily defined and can be computed from an arbitrary imperative language program.

Measurement of testability complements the measurement of test coverage as performed by tools such as TESTBED [22] and LOGISCOPE [23]. Such measures provide useful information for both management control and quality assurance.

An obvious pitfall in controlling control flow is that all of the problems, either in terms of complexity or testability, could be hidden in data. For this reason it is also necessary to develop testability measures for data flow, predicate and computation testing.

Other areas of future work are to extend the measurement of testability beyond the unit testing to integration testing and also extend it to nonimperative languages, such as PROLOG, where the flowgraph model does not apply.

## 7 Acknowledgment

The work described in this paper is carried out in part in project MUSE, which is supported by the EEC under the ESPRIT programme. We would like to thank Angelika Buth, Dr. Norman Fenton, Susanne Flacke, Hans-Ludwig Hausen and Frank Weber for many stimulating discussions and valuable comments.

## 8 References

[1] BACHE, R.M., FENTON, N.E., TINKER, R., and WHITTY, R.W.: 'Tutorial on quality assurance'. Proc. 2nd IEE/BCS Conference Software Engineering 88, London, UK, 1988
[2] BS5750 British Standards Institute, 1987
[3] KAPOSI, A.A., and KITCHENHAM, B.: 'The architecture of system quality', *Softw. Eng. J.*, 1987, **2**, (1), pp. 2–8
[4] STOCKMAN, S.G.: 'A constructive approach to software quality assurance.' Proc. 1st European Conference on Software Quality, European Organization for Quality Control (EOQC), 1988
[5] FENTON, N.E., and KAPOSI, A.A.: 'Engineering theory of structure and measurement' *in* 'Measurement for quality control and assurance' (Elsevier, 1988)
[6] McCALL, J.A., RICHARDS, P.K., and WALTERS, G.F.: 'Factors in software quality.' Technical Report Vol I-III, RADC Reports ADA-049 014, 015, 055, National Technical Information Service, US Department of Commerce, 1977
[7] HAUSEN, H.L.: 'Generic modelling of software quality and productivity' in KITCHENHAM, B., and LITTLEWOOD, B. (Eds.): 'Measurement for quality control and assurance' (Elsevier, 1988), pp. 201–241
[8] FENTON, N.E.: 'Software measurement'. Technical Report, CSSE, South Bank Polytechnic, UK, 1988
[9] MÜLLERBURG, M.: 'On fundamentals of program testing'. Proc. 1st European Seminar on Software Quality, pp. 290–305, European Organisation for Quality Control (EOQC), 1988
[10] MÜLLERBURG, M., and LINNENKUGEL, U.: 'On the effectiveness of program testing'. Proc. 2nd IEE/BSC Conference Software Engineering 88, London, UK, 1988
[11] MYERS, G.J.: 'The art of software testing' (John Wiley & Sons, London, UK, 1979)
[12] FENTON, N.E., and KAPOSI, A.A.: 'Metrics and software structure', *Inf. Software. Technol.*, 1987, **29**, (6), pp. 301–320
[13] FENTON, N.E., WHITTY, R. W., and KAPOSI, A.A.: 'A generalised mathematical theory of structured programming', *Theor. Comput. Sci.* 1985, **36**, pp. 145–171
[14] PRATHER, R.E.: 'On hierarchical metrics', *Soft. Eng. J.*, 1987, **2**, (2), pp. 42–45
[15] BACHE, R. M.: 'Classification of flowgraph metrics'. Technical Report CSSE/015/2, CSSE, South Bank Polytechnic, UK, 1987
[16] BACHE, R.M.: 'Structural metricating within an axiomatic framework'. Technical Report BT/SBP/W021, CSSE, South Bank Polytechnic, UK, 1987, Alvey Project SE/69
[17] McCABE, T.J.: 'The structured testing technique' *in* McCabe, T.J. (Ed.) 'Structured testing' (IEEE, 1982)
[18] ELLIOTT, J.J., FENTON, N.E., MARKHAM, C., LINKMAN, S., and WHITTY, R.: (Eds.): Structure based software metrics' CSSE, 1988, Alvey Project SE/69, Collected papers
[19] BACHE, R.M., and WILSON, L.C.: 'Details of the implementation of the decomposition algorithm'. Technical Report BT/SBP/W039, CSSE, South Bank Polytechnic, UK, 1987, Alvey Project SE/69
[20] BANNER, N.J.: 'A tool to convert FORTRAN '77 programs to flowgraphs'. MSc Thesis, CSSE, South Bank Polytechnic, UK, 1988
[21] BACHE, R., and TINKER, R.: 'A rigorous approach to metrication: a field trial using Kindra'. Proc. 2nd IEE/BCS Conf. Software Engineering 88, London, UK, 1988, pp. 28–32
[22] Program Analysers Ltd.: TESTBED Description, 1989
[23] VERILOG: LOGISCOPE Technical Presentation, 1989.

CSSE papers may be obtained from Meg Russell, CSSE, South Bank Polytechnic, Borough Road, London, SE1 0AA, UK.

The paper was first received on 10th November 1988 and in revised form on 6th September 1989.

Richard Bache is with the Department of Mathematics, Glasgow College of Technology, Cowcaddens Road, Glasgow G4 0BA, UK; and Monika Müllerburg is with the Gesellschaft für Mathematik und Datenverarbeitung, Schloss Birlinghoven, D-5205 Sankt Angustin 1, Federal Republic of Germany.