1-1992

The Complexity of Software Testing

J. Paul Myers Jr.
Trinity University, pmyers@trinity.edu

Repository Citation

Myers, J. P., Jr. (1992). The complexity of software testing. *Software Engineering Journal, 7*(1), 13-24. https://doi.org/10.1049/sej.1992.0002

The complexity of software testing

by J. Paul Myers, Jr.

The futility of using a general-purpose metric to characterise 'the' complexity of a program has recently been argued to support the design of specific metrics for the different stages of the software life-cycle. An analysis of the module testing activity is performed, providing evidence of the absurdity of all-purpose metrics, as well as a methodical means with which to measure *testing* complexity. Several standard metrics are seen to serve as component measures for the intricacies of testing. The methodology is applied to compare traditional and adaptive means of testing. It is shown that previous informal arguments asserting the superiority of adaptive methodologies are formally confirmed.

1 Introduction

The importance of affirming the accuracy and reliability of software need hardly be stated. With defence and economic systems increasingly dependent on computer software (not to mention dating systems!), our very lives and livelihood depend on software correctness to a significant degree. Therefore, it is crucial that our software perform correctly; but, with undecidability present, unfortunately we cannot rely on formal proofs of correctness alone. The much weaker but more pragmatic means of testing must be employed to at least instill confidence in a given software system, short of showing it to be provably correct. This testing enterprise will require certain resources, the measure of which we call the *complexity of testing*, to be reflected in the value of *a software metric for testing complexity*.

The notion of *algorithmic complexity* enjoys a revered status in computer science. What is thought of the actual implementation of algorithms, *programs?* Certainly computational complexity remains an issue, but now we turn from algorithmic analysis to the day-to-day issues surrounding their actual coding: language, structured versus unstructured, software life-cycle, faithfulness to the algorithm (correctness), testing, design differences etc. The field concerned with these issues is called *software complexity*. The main progress to date has been in the development of *software metrics*, intended to predict project cost, and to evaluate the quality and effectiveness of the design [1]. There is a plethora of these software metrics with mostly intuitive/logical/aesthetic motivation, with little hard empirical evidence to support their worthiness [2].

Although many authors acknowledge that the range of issues to be quantified is so extensive that no single metric could capture it all, new metrics are introduced nonetheless as 'all-purpose' measures of software complexity. Our focus here is more modest; we outline a schema for measuring testing complexity only, while shedding light on various established metrics.

1.1 Software testing methodologies: overview

The general *goal of software testing* is to affirm the quality of a program through systematic exercising of the code in a carefully controlled environment [3]. Here, the emphasis falls, for the most part, on *structural testing*, including a variation that might be called *adaptive* in its approach [4].

We assume that a program flowgraph F has a single *start vertex* and a single *stop vertex*, and that every vertex lies on at least one path from start to stop. This program flowgraph differs from a standard low-level flowchart, in that it focuses on decision (selection or loop) and branch (goto) statements as the graph vertices or nodes. In *conventional* structural testing, a finite number of paths are determined, whose successful execution by test inputs will cover the flowgraph with a certain degree of thoroughness, thereby imparting a level of confidence in the program correctness. Conventional structural testing methodology can be divided into four rather separate phases, performed in sequence:

- program graph construction.
- test path selection.
- test case generation.
- execution of the program on the test cases.

The first phase, *program graph construction*, is the most straightforward; it is a parsing problem that can be automated. We have only to 'annotate' the source code listing f to derive the underlying flowgraph $F = (V, E)$ as a

collection of vertices (V) and edges (E), in the graph-theoretic sense. As a component of this phase, the program to be tested is provided with 'instrumentation' (flags to monitor traversal of the edges in the flowgraph) to determine the exact path traversed on execution of a test input x. Such techniques are well understood and are commonly employed in structural testing methodology.

Given the program flowgraph, we then enter the second phase, *test path selection*, in which a finite set $P = \{p_i\}$ of program paths are chosen, with a view to satisfying one or more 'coverage' criteria. The most often cited criteria are

- ☐ *statement coverage (SC):* execute all statements in the graph (weak criterion).
- ☐ *node coverage (NC):* encounter all decision node entry points in the graph (weak criterion).
- ☐ *branch coverage (BC):* encounter all exit branches of each decision node in the graph.
- ☐ *multiple condition coverage (MCC):* achieve all possible combinations of simple Boolean conditions at each decision node in the graph.
- ☐ *path coverage (PC):* traverse all paths of the graph (strongest criterion).

Although extremely robust as a coverage criterion, path coverage is most difficult to achieve in practice. Simply stated, path coverage is the separate execution of all paths in the program flowgraph. In the presence of indefinite looping, however, the number of different paths is unbounded. To render this situation tractable, it is conventional practice to partition paths into equivalence classes, where two paths are considered equivalent if they differ by only their number of loop traversals. As we find later, loop-modified path coverage can be achieved in a finite number of tests.

Branch coverage has come to be regarded as the minimal standard of achievement in structural testing. Node and statement coverage are too weak; path coverage is stronger than branch coverage, but it may be explosively exponential in d (the number of decision nodes in the flowgraph). We show later that the number of tests required for branch coverage is always favourably bounded ($\leq d + 1$).

Once a coverage criterion has been selected, a set of program paths to provide that coverage must be chosen; the *test path selection phase*. There are numerous algorithmic methods (path generating functions) of selecting a set of paths for a given coverage criterion [1, 3, 5]. Thus, given an available set of paths through the program flowgraph sufficient to achieve the desired coverage, the next step is to acquire a set $X = \{x_i\}$ of test inputs, which, when executed by the program, causes traversal of those paths; the *test-case generation phase*.

This is the most difficult phase of structural testing; in fact, the selection of an input to traverse a preselected path is unsolvable in some cases [6]. There are two principal methods for test-case generation; symbolic execution and backward substitution [3, 5, 7–9] (these references may be consulted for examples of path selection and test-case generation algorithms). For our purposes, however, it is sufficient to say that all these conventional methods accomplish the test-case selection stage by the *preselection* of paths, i.e. all paths are selected before proceeding to the test-case generation stage of structural testing (Fig. 1). This is a critical difference between conventional and more recently developed adaptive strategies [10, 11], in which paths and test cases are derived together. We return to this issue later.

Only during the final phase, *execution of the program f* on the test inputs, do we finally attend to more traditional dynamic aspects of algorithm (program) execution. This is clearly a part of testing, since the program must be executed in order to compare the real output(s) to the ideal specified output(s). In adaptive testing, moreover, as we see later, it is the execution of an input that leads to the determination of the next input. Here a primary concern is computational complexity.

1.2 Software complexity measures: overview

The following summary of software metrics is necessarily brief. There is a tremendous abundance of metrics [12, 13], many of which are not relevant for our purposes. Instead we only mention a few metrics; those that are 'prominent' in the sense of being most often cited in the literature, and they play a role in our later developments.

First is the *McCabe metric* v (cyclomatic measure [14]). This purely structural measure is simply the circuit rank of the flowgraph, with vertices considered to consist of decision nodes, start node and stop node. For programs consisting of d binary decision nodes, we write McCabe's metric in the form normally encountered, $v(F) = d + 1$. The idea is that the presence of decisions lends complexity to a program. Whereas this observation is certainly incontestable, we note that the metric is not sensitive to nested structures versus those in sequence. On this basis, there have been many criticisms of v as an all-purpose complexity measure [15–17]. Nonetheless, the measure enjoys a position of prominence in the literature and we use it later.

Halstead developed a set of metrics [18] to define a 'software physics,' proceeding from the following information-theoretic counts:

λ = total number of operator and operand occurrences
= the program *length*.
η = total number of distinct operators and operands
= the program vocabulary *size*.

Of these, Halstead's length is the most interesting for later discussions.

The number of tests required in a program may be taken as a measure of its complexity [1]. Needless to say, this measure is of specific importance to us, since it contributes to the determination of the difficulty of the path selection and test generation phases. Nor should we forget the metric lines-of-code, whose numeric value is just that. This extremely simple (and simple-minded) metric has the distrinction of correlating quite well with, and sometimes performing better than, more sophisticated measures that are more difficult to apply [19]. Reference 20 reports a correlation of 0.98 between lines of code and the McCabe metric over 26 programs). Surely longer programs are generally more complex, but the high correlation is testimony to a certain lack of sensitivity in conventional metrics. The schema developed here discriminates easily between a short complex program and a long but simple one.

1.3 Testing-specific measures

The two central ideas of this paper (indeed, central to software engineering) are testing and complexity. We now merge these two ideas into a view of measuring the complexity of a program with respect to the resources required to test it. The reasons for such specific measures of complexity are clear. The testing phase of the software life-cycle is extremely cost-intensive; 40% and more of the entire resources, from design through to maintenance, are often spent on testing [21]. Furthermore, testing is a multi-faceted, complex activity (Fig. 1).

These two reasons alone are sufficient to justify the need for specific measures of testing complexity. More general metrics may not be sufficiently sensitive to capture even substantial requirement differences between two programs. This insensitivity of general metrics is increasingly acknowledged. Weyuker [22] and Kearney *et al.* [23] decry the development of metrics without clear analysis of what is to be measured or how the metrics are to be used (e.g. for implementation, testing, understanding, modifying or maintaining). Indeed, Kafura and Reddy [24] report that, with respect to software maintenance, changes in components might have no effect on the values of certain general metrics, and they certainly are not designed to accurately measure the complexity of the entire and varied testing process.

Moreover, there are certain quirks arising in the testing process that are not handled appropriately by existing metrics. For example, there have been several improvements to the McCabe metric to ensure that nesting is punished more than sequencing decision nodes [17, 25]. For a general complexity measure, this is perhaps appropriate. Yet if, in testing, we opt for the path coverage criterion, we see that nesting requires a number of tests linear in the number d of decision nodes (in fact, equal to McCabe's metric). Whereas sequence requires an exponential number of tests. On the other hand, branch coverage in sequence might be accomplished with just two tests, whereas nesting requires all $d + 1$ tests. Thus, nesting is worse than sequence only with respect to other factors in the testing process; general metrics simply do not have this kind of sensitivity. We could also argue that d nodes nested is preferable to d in sequence in test-case generation. Both backward substitution and symbolic execution occur through all nodes with those nodes in sequence (hence over longer paths); but only some nodes, when nested, are used (hence shorter paths).

There is one final distinction. Many conventional general metrics are introduced with the intention of predicting effort required for the entire software life-cycle. By the time testing begins, the program already exists, i.e. we have more real information available than at an earlier stage in the development of the program. A metric for testing complexity should exploit this availability of the actual program. Thus, a measure of testing complexity should exploit both the dynamic aspects of computational complexity and the static features of standard software metrics.

2 The testing complexity schema T

In our development of a testing complexity measure (a 'parameterised metric'), we wish to state principles to be satisfied by any metric for testing complexity. We have noted that there is a certain 'relativity' involved here (e.g. whether nesting should be penalised); the effort needed to test a program is not a function of that program alone. We refer to the *testing complexity of program f with strategy s* as $T(f, s)$, where the testing strategy is a pair $s = (s_1, s_2)$ with (s_1 = coverage criteron) and (s_2 = testing algorithm). Sometimes, as is evident from the context, we find it convenient to identify s simply with either s_1 or s_2.

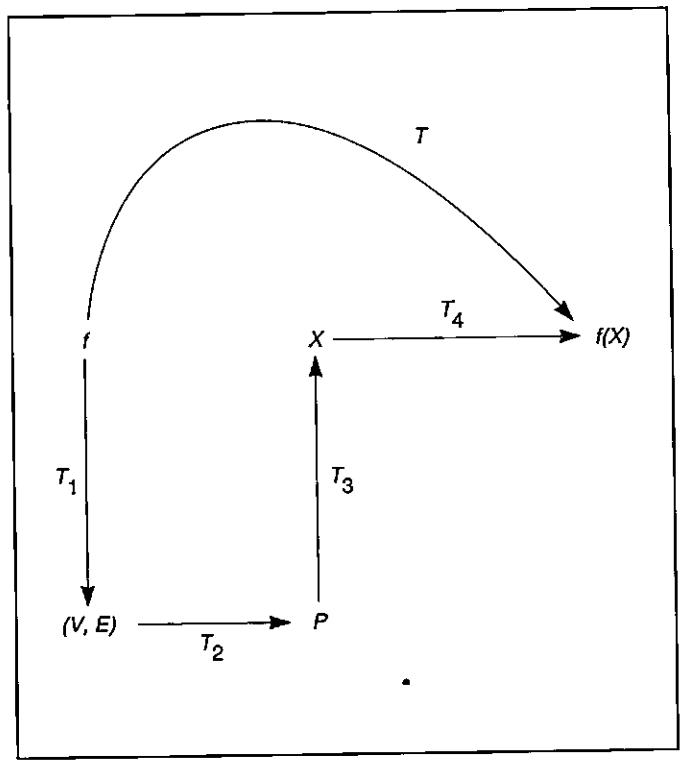

Fig. 1 Decomposition of testing complexity T: conventional strategies

2.1 Decomposition of T

Considering first the conventional testing strategies as outlined above, the paradigm for developing T is determined by a decomposition of the four phases of the testing process. Each of these stages has a certain complexity. Therefore, to each of these four phases, we assign a complexity measure $T_i (1 \leqslant i \leqslant 4)$, where f is the program, and P and X are the sets of test paths and inputs, respectively (Fig. 1).

We have previously discussed the testing stages and we now list the following parameters for T_i:

T_1 (graph construction)
- length of program f;
- d = number of decisions ($= v - 1$).

T_2 (path selection)
- d;
- nesting versus sequence;
- coverage criterion;
- path selection algorithm;
- number of Booleans (MCC).

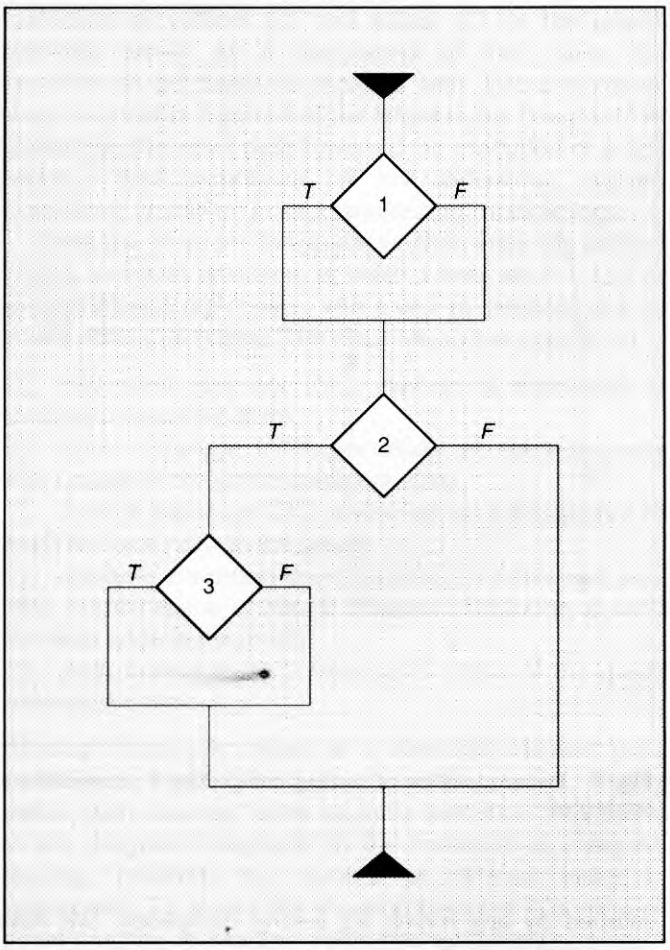

Fig. 2 Program g, illustrating P_{NC} and $P_{NC}^{\wedge}$

T_3 (test-case generation)
- $|P|$ = number of test paths;
- test generation algorithm;
- amount of backtracking (number of assignments/Booleans per path);
- $\tau(f)$ = computational complexity of f.

T_4 (executing f on X)
- $|X|$ $(=|P|)$;
- $\tau(f)$.

Summarising these dependencies, we have

$$T(f, s) = T_1(f) + T_2(f, s_1, s_2) + T_3(f, P, s_2) + T_4(f, X).$$

We note how, at each stage, the different aspects of the problem come into focus under the decomposition of T, allowing resources to be apportioned among the testing activities. Now computing T via T_i is certainly not easy. It would therefore be advantageous to make use of standard, well accepted metrics in estimating T_i. We have argued that no standard metric can give a reasonable estimate of T, but they are of considerable use in approximating its components. In fact, one of our major achievements in describing a testing complexity theory is that it serves to lend a certain reasoned validation to some standard metrics, owing to the manner in which they become integrated into the theory. Our approach is to find an upper bound for $T(f, s)$ by finding bounds for each of T_i.

T_1: graph complexity

Computing the graph of a program f is quite similar to compiling that program and can, in fact, be done in parallel with the compilation in one pass [26]. The basic strategy is to push decision nodes onto a stack and pop them at the end of their scope, while inserting the instrumentation assignment statements, of virtually constant length, at each branch of the decision. T_1 is then determined by the length of f and particularly by d, the number of decisions in f. If we assume that d is also highly correlated to the length of f (indeed, we have seen that McCabe's $v(f) = d + 1$ is thus correlated), T_1 is seen to be proportional to d. The relationship is linear, since, whether nested or in sequence, each decision is pushed/popped exactly once.

Lemma 1

The graph complexity T_1 is proportional to McCabe's metric $T_1(f) = t_1 v(f)$.

T_2: path selection complexity

The number of paths to be selected depends critically on the coverage criterion chosen for the testing of f. This phase of the testing process also determines the size $|X|$ of the test set, thereby influencing the complexity of the test generation phase to follow.

We define, for a given program f, the maximum number P_α of paths (tests) required in order to attain α-coverage when only the *minimum* additional coverage is achieved with each new test (i.e. a worst-case scenario). Here α ranges over the previously defined coverage criteria: PC (loop-modified), MCC, BC, NC and SC. This maximum depends on the number d of decision nodes in f and is denoted by

$$P_\alpha(f) = \text{maximum number of tests to } \alpha\text{-cover } f.$$

We then define, for the class of programs with d decisions, $P_\alpha^{\wedge}(d) = max_f P_\alpha(f)$, f having d decisions. Thus, for all programs f with d decision nodes, we have $|P| \leqslant P_\alpha(f) \leqslant P_\alpha^{\wedge}(d)$, where $|P|$ is the number of test paths in a given coverage path set P. Thus, by counting the decisions in a program, $P_\alpha^{\wedge}(d)$ is an upper limit for the number of tests to provide α-coverage. Fewer tests may be required, i.e. it may be that even $|P| < P_\alpha(f)$, by a judicious selection of test paths, i.e. we might choose some input to achieve more than minimal additional coverage. For an example, we consider the program in Fig. 2.

Fig. 2 demonstrates that the three values may be distinct, e.g. for node coverage. Indeed, $P_{NC}(g) = 2$, since a maximum of two tests is required; $1(T$ or $F)2F$ and $1(T$ or $F)2T3(T$ or $F)$. However, since node coverage here can be achieved with only a single test $1T2T3T$, it could be that $|P| = 1$. We determine below that $P_{NC}^{\wedge}(3) = 3$.

We may agree with Shooman [1], however, that minimising the testing of a program is not a goal to inspire confidence in the adequacy of our test. In testing, unlike many other endeavours, we do not strive necessarily to find $|P|$ and $|X|$ significantly less than the upper bound $P_\alpha(f)$. Instead, we are likely to test somewhat beyond the maximum, and so lower are not all that important. Thus, for testing purposes, $P_\alpha(f)$ serves as a good estimate for

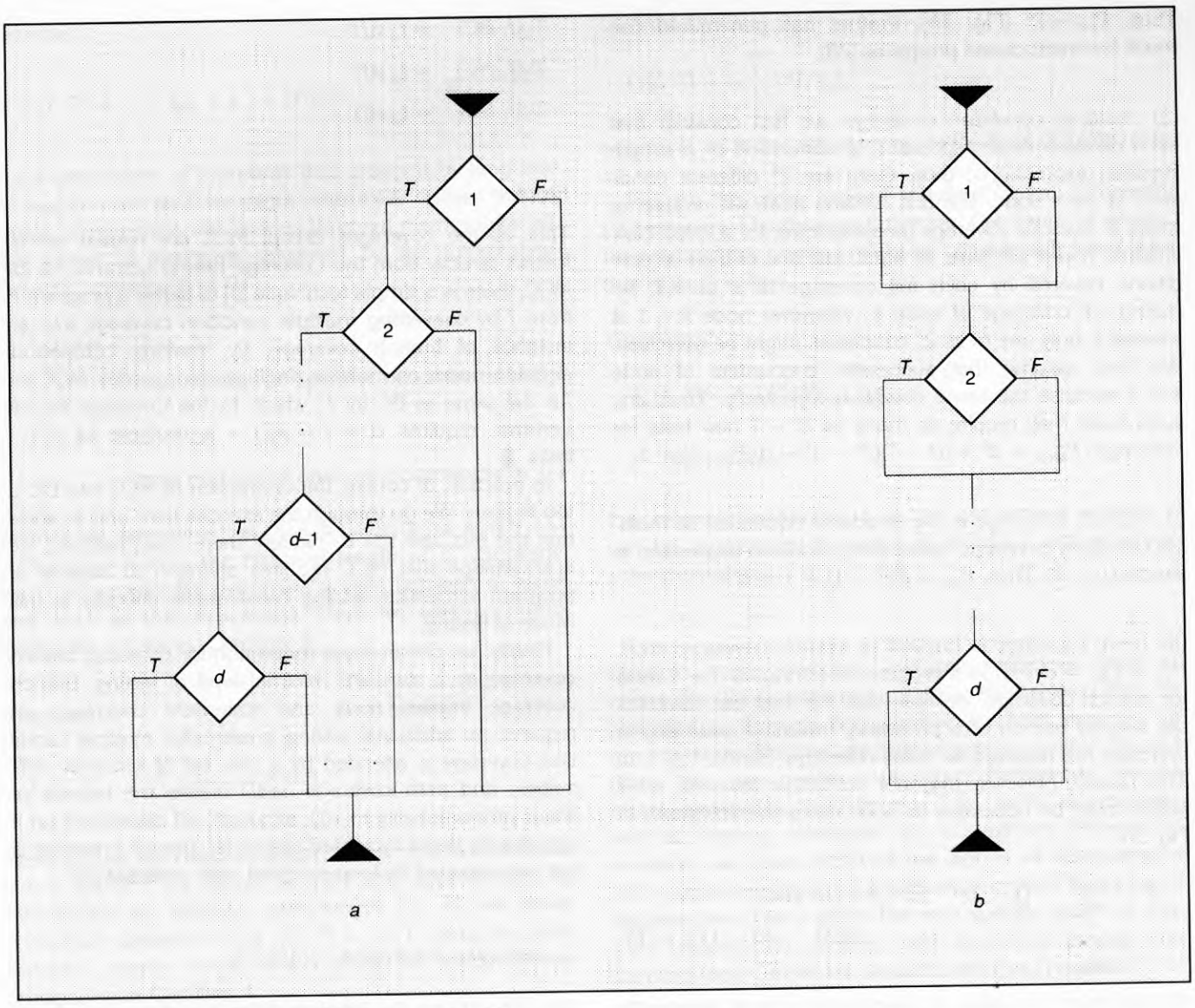

Fig. 3 Computation of P_{PC}: nesting versus sequence

a Total nesting
b Total sequencing

$|P|$, and hence for $|X|$ (the number of tests). These comments notwithstanding, it is reassuring that the maximum number of tests required to achieve coverage of a program is bounded in the 'worst' case; reassuring and useful in optimising one type of coverage versus another, given fixed resources for testing. These bounds are summarised in Lemma 2.

Lemma 2 (coverage bound)

With the least additional coverage achieved with each new test, the number of tests to (PC, MCC, BC, NC, or SC) cover an arbitrary program with d decisions is bounded:

- $P_{PC}^{\wedge} = P_{PC}$, where $d + 1 \leqslant P_{PC} \leqslant 2^d$
- $P_{MCC}^{\wedge} = d(2^b - 1) + 1$ (b = maximum number of simple Boolean expressions in any decision)
- $P_{BC}^{\wedge} = d + 1$
- $P_{NC}^{\wedge} = d$
- $P_{SC}^{\wedge} = d + 1$

Proof

We assume full feasibility of all paths to avoid artificially low bounds.

(*1*) *path coverage:* for structured programs, we argue inductively following the inductive construction of the program. We note that P_{PC} is minimised with nesting, since then P_{PC} is only additively increased:

$$P_{PC}\ (\text{if } E \text{ then } S1 \text{ else } S2) = P_{PC}\ (S1) + P_{PC}\ (S2)$$

$$\text{and } P_{PC}\ (\text{while } E \text{ do } S) = P_{PC}\ (S) + 1.$$

A totally nested program then has the fewest paths as a function of d (Fig. 3*a*). Here $P_{PC} = d + 1$:

$$p_1 = 1F;\ p_j = 1T2T \cdots (j-1)TjF,\ 2 \leqslant j \leqslant d;$$

$$\text{and } p_{d+1} = 1T2T \cdots dT.$$

The maximum P_{PC}, on the other hand, occurs with total sequencing of decisions:

$$P_{PC}\ (S1; S2) = P_{PC}\ (S1) \cdot P_{PC}\ (S2),$$

resulting in Paige's 'weave' effect [27].

Thus, $P_{PC} = 2^d$ (Fig. 3*b*). Prather has generalised this result for unstructured programs [28].

(2) *multiple condition coverage:* we first consider that each decision node expression is composed of b simple Boolean expressions; thus, there are 2^b different conditions at each node. The first decison node will require, at most, 2^b tests for coverage (possibly fewer for a loop). Now children nodes will have, at worst, just one of their 2^b conditions covered by each full coverage of a parent, i.e. during full coverage of node k, whenever node $k+1$ is traversed, only one of its 2^b conditions might be exercised. We then assume that successive encounters of node $k+1$ exercise the same condition repeatedly. Thus, any such node may require as many as $2^b - 1$ new tests for coverage: $P^{\wedge}_{MCC} = 2^b + (d-1)(2^b - 1) = d(2^b - 1) + 1$.

(3) *branch coverage* is the structural equivalent of multiple condition coverage, where every Boolean expression is simple ($b = 1$). Thus, $P^{\wedge}_{BC} = d(2-1) + 1 = d + 1$.

(4) *node coverage* is implied by branch coverage; thus, $P^{\wedge}_{NC} \leqslant P^{\wedge}_{BC} = d + 1$. In a program requiring all $d+1$ tests for branch coverage, we note that the final test traverses the second branch of a previously traversed node and is therefore not required for node coverage. Hence, $P^{\wedge}_{NC} \leqslant$ d. This bound, however, may not be further lowered; all d paths could be necessary to node cover the flowgraph in Fig. 3*a*:

$$1F;\ 1T2T\cdots(j-1)F \quad (2 \leqslant j \leqslant d);\ \text{and}$$

$$1T2T\cdots(d-1)(T \text{ or } F).$$

(5) *statement coverage:* a program may have statements on all branches, requiring full branch coverage. ▮

The bound for multiple condition coverage can be improved for arbitrary programs, by acknowledging that not all decision nodes are governed by the same (maximum) number b of simple Boolean expressions. If d_k is the number of decision nodes with exactly k simple Boolean expressions, $d = d_1 + d_2 + \cdots + d_b$.

Corollary 1

$$|P|_{MCC} \leqslant 1 + \sum_{i=1}^{b} d_i(2^i - 1).$$

We may now relate the Coverage Bound Lemma to existing metrics.

Corollary 2

$T_2(f, s)$ is bounded by a function of the McCabe metric $v(f)$:

$$T_2(f, PC) = t_2 N_{PC}, \text{ where } v(f) \leqslant N_{PC} \leqslant 2^{v(f)-1}$$

$$T_2(f, MCC) = T_2(f', BC) = t'_2\, v(f') = t_2\, v(f).$$

$$T_2(f, BC) = t_2\, v(f)$$

$$T_2(f, NC) = t_2\, v(f)$$

$$T_2(f, SC) = t_2\, v(f)$$

Proof

That all the coverages except MCC are related to $v(f)$ follows directly from the Coverage Bound Lemma. As for MCC, there is a simple technique [5] to derive a program f' from f by converting multiple condition coverage into an instance of branch coverage, by rewriting compound-decision nodes as multiple single-decision nodes. MCC for f is the same as BC for f', which, by the Coverage Bound Lemma, requires $d' + 1 = v(f')$ - a multiple of $v(f)$ - tests. ▮

In practice, of course, the conversion of MCC into BC is too tedious. We go through the exercise here only to show that the McCabe $v(f)$ is 'hidden' there. Pragmatically, we acknowledge that MCC requires attention to aspects of program semantics, adding considerable difficulty to this mode of testing.

Finally, we obtain more motivation for choosing branch coverage as a standard minimal level of testing. Branch coverage implies node and statement coverage, yet requires no additional testing (maximally). Multiple condition coverage is attended by a new set of semantic difficulties; and path coverage easily breaks the bounds of linear proportionality to $v(f)$, although still dependent on it (yet despite these increased resources, branch coverage is not even assured by loop-modified path coverage [5]).

T_3: test-case generation complexity

We elaborate on the inherent difficulties discussed above. In our conventional paradigm, test-case generation has two components; path predicate determination and predicate satisfiability, where each path p_i has a conjunctive predicate $P_i = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, whose conjuncts C_j correspond to the decision nodes encountered along the path p_i.

By way of performing the first part, either symbolic execution or backward substitution are designed to cope with the complexity of the changing program state, as a means of determining the Boolean expression to be satisfied. Given the path p_k, for which we must find the path predicate P_k and test input x_k, we define

a_k = number of assignments in p_k.

b_k = number of simple Booleans in p_k.

With either symbolic execution or backward substitution, the effort to acquire P_k is proportional to $a_k + b_k$, since these methods manipulate the assignments and decision Booleans of the path.

In the worst solvable case, the Boolean satisfiability of P_k (the problem of finding x_k) can be regarded as NP-complete [29]. The satisfaction of P_k in this worst case is proportional to 2^{b_k}. Merging both components we find Lemma 3.

Lemma 3

$$T_3(f, P) = t_3 \sum_{i=1}^{|P|} [(a_i + b_i) + 2^{b_i}].$$

For a given class of programs (e.g. grouped by language or programmer etc.), we might empirically discern the following relationships with various Halstead metrics (after all, this phase of testing is governed by the information-theoretic parameters of f):

$$N_1(p_i) = k_1(a_i + b_i)$$

(total number of operator occurrences in p_i)

$$N_2(p_i) = k_2(a_i + b_i)$$

(total number of operand occurrences in p_i)

whence we determine that $(a_i + b_i) = [1/(k_1 + k_2)][N_1(p_i) + N_2(p_i)] = k'\lambda(p_i)$, the Halstead length of p_i. Similarly, we could empirically determine a ratio between b_i alone and $\lambda(p_i)$, so that $b_i = k\lambda(p_i)$. Thus, we obtain, consolidating the constants, Corollary 3.

Corollary 3

$$T_3(f, P) = t_3 \sum_{i=1}^{|P|} [\lambda(p_i) + 2^{k\lambda(p_i)}].$$

Here, we use the Halstead length metric relativised to individual paths. If we further simplify and approximate, by introducing an 'average' path length $\underline{\lambda}(f, P)$, we could empirically determine that $\underline{\lambda}(f, P) = c\lambda(f)$, using the pure Halstead length metric, which gives the parsimonious relationship in Corollary 4.

Corollary 4

$$T_3(f, P) = t_3 |P| [\lambda(f) + 2^{c\lambda(f)}].$$

Thus, the complexity of test-case generation is determined by $|P|$, as we already of course knew, and by the Halstead length of f. Corollary 4 indicates this favourable relationship, but the formula is probably not adequate for practical estimates of T_3. The averaging and approximating are quite liberal and depend heavily on the homogeneity of the class of programs being averaged. In addition, even small errors in approximation will have a considerable effect on the size of the complexity of predicate satisfiability, since this task can be of exponential complexity. Nonetheless, the above results do place the Halstead metric into a proper perspective as to its relationship to the realm of testing, even if it is more accurate to use the formulas of Lemma 3 or of Corollary 3.

T_4: test execution complexity

We cannot add much here about $T_4(f, X)$ as so much has been said before, i.e. at T_4, our testing complexity is simply the classical computational complexity τ of f. Since we have the x_i at hand, we can be quite specific.

Lemma 4

$$T_4(f, X) = t_4 \sum_{i=1}^{|X|} c(f, x_i),$$

where $c(f, x_i)$ is the actual computation time of f executing x_i.

We use the more general standard 'order' complexity [30], relativised to the actual test set X at hand; of course, the result of this Lemma might be considerably lower than that of its Corollary.

Corollary 5

$$T_4(f, X) \leqslant t_4 |P| \tau(f)|_{max\, x \in X}.$$

2.2 Summary

It is certainly heartening to us that our general schema T can be represented in terms of standard well accepted software metrics:

$$T = T(v, \lambda, \tau)$$

(modulo approximations), and an auspicious state of affairs for those standard metrics themselves. For as accepted as they are for empirical/aesthetic reasons, we are not aware that they are mutually encompassed by, and fitting into, any previous theoretical framework. We set about developing the schema T in order to compare various testing strategies in a uniform framework. However, we have achieved the bonus of integrating a rather disparate group of previously unrelated metrics and showing their place within the very specific realm of software testing. By picking and choosing among the Lemmas and Corollaries, we summarise in Theorem 1.

Theorem 1

The complexity of testing $T(f, s)$ is a function of the McCabe metric (v), the Halstead length metric (λ) and the order of computational complexity (τ):

$$\begin{aligned} T(f, s) &= T(f, s, v, \lambda, \tau) \\ &= t_1 v(f) + t_2 S(v(f)) + t_3 S(v(f))[\lambda(f) \\ &\qquad + 2^{c\lambda(f)}] + t_4 S(v(f))\tau(f)_X \end{aligned}$$

or more simply as

$$\begin{aligned} T &= t_1 v + t_2 S(v) + S(v) + t_3 S(v)(\lambda + 2^{c\lambda}) \\ &\quad + t_3(\lambda + 2^{c\lambda}) + t_4 S(v)\tau_X. \end{aligned}$$

In these formulas, $S(v(f))$ treats $v(f)$ appropriately for the chosen strategy (coverage criterion) according to Corollary 2 of the Coverage Bound Lemma. We also note that $S(v(f))$ may serve as a replacement for the previously used $|P|$ $(=|X|)$. $\tau(f)_X$ (abbreviated τ_X) adjusts $\tau(f)$ to the input set X as either $c(f, X)$ or $|P|\tau(f)|_{max\, x \in X}$ as discussed above.

We now recall from the Coverage Bound Lemma that $S(v(f))$ is approximately equal to the McCabe metric for branch coverage, giving the most satisfying result in Corollary 6.

Corollary 6

For branch coverage, $T(f, s) = \nu[A + B(\lambda + 2^{C\lambda}) + D\tau]$, i.e. ν factors from the theorem with constants $A = t_1 + t_2$, $B = t_3$ and $D = t_4$.

Moreover, if we look carefully at other Lemmas and Corollaries, we may see T in all its computational details.

Corollary 7

$$T = \sum_{i=1}^{4} T_i \text{ decomposes as } T(f, s)$$

$$= t_1 d + t_2 S(d, b) + t_3 \sum_{i=1}^{|P|} [(a_i + b_i) + 2^b i] + t_4 \sum_{i=1}^{|X|} c(f, x_i).$$

It is worth emphasising that $T = \sum T_i$ is a schema for a comprehensive metric, rather than such a metric itself. This schema allows the construction of concrete metrics, depending on the nature of the programming-testing environment. A fine-tuned concrete metric, for example, can be developed by adjusting the coefficients t_i by empirical evidence, policy, organisation, environment or other concerns. For example, in a switching program involving little or no computation, t_3 could be zero or much less than in a numerical program involving extensive backtracking and interpreting of variables. t_4 would also be expected to be orders of magnitude less in a CRAY environment as opposed to a personal computer.

Finally, at this point, we could list a variety of theorems, comparing the testing complexity of various combinations of programs versus strategies. Program parameters can be varied (nested versus sequence, length, computational complexity etc.) to produce results such as 'if program f differs from f' in such and such, $T(f, \alpha) < T(f', \alpha)$'. Similarly, we could hold f constant and alter α. We turn to more significant issues in Section 3.

2.3 *Independence of T*

To motivate the development of the software complexity measure T, we noted that no single existing standard metric was sufficient to encompass the entire effort required in unit or module testing. Now that T has been generated, we show that, in fact, it is indeed independent of the standard metrics of software complexity: lines of code, McCabe's cyclomatic number v, Halstead's length λ, Prather's μ and order complexity (computational complexity) τ.

First, consider standard computational complexity. This is not, strictly speaking, a software metric; yet as it is a component of testing complexity T, it is considered. We can produce a trivial simple program f_1 with a single loop with complexity $\tau(2^n)$ or a mammoth million-decision, fully nested program f_2 of τ(constant). Now, for instance, branch coverage of f_1 occurs with a single input causing one loop traversal. Branch coverage is attained in f_2, however, with exactly 1 000 001 tests. Thus, $T(f_1, BC) < T(f_2, BC)$, but $\tau(f_1) > \tau(f_2)$. This discussion notwithstanding, certainly $T_4 \approx \tau(f)$ can be expected to dominate the other T_i s in those instances of a computationally complex program that must be tested extensively and with large inputs. This is an issue of domination rather than dependence; indeed, any T_i may dominate the others in certain classes of programs.

T is also logically independent from the simple metric lines-of-code. For example, straight-line code of 100 lines is simpler to test than a 20-line sequence of decisions with possible compound Boolean conditions. This example would also indicate the independence of T from Halstead's length. Of course, lengthy programs may indeed be more difficult to test than shorter ones; but, in principle, length and testing difficulty are not related. In motivating his metric μ, Prather [17] argues that the McCabe cyclomatic number can be misleading with respect to large activities, such as testing. Indeed, McCabe's number is independent of our amalgamation T. As a concrete example, consider $T(f_i PC)$, where f_1 has d fully nested decisions and f_2 has d sequenced decisions. All else being 'equal', or at least similar, $|X_1| = d + 1$; $|X_2| = 2^d$ and McCabe's number $v(f) = d + 1$.

Finally, the Prather μ penalises nesting and GOTOs. In testing, however, the GOTO may not cause additional problems and, in fact, can avoid additional decisions or compound Booleans to escape from deep nesting. The graphing of a GOTO edge is trivial; it is handled as straight-line code. Nesting, moreover, can indeed require additional test resources. For instance, with f_1 and f_2 above, $T(f_1, BC) > T(f_2, BC)$, since $|X_1| = d + 1$ and $|X_2| = 1$ is possible. On the other hand, $T(f_1, PC) < T(f_2, PC)$, where the weave effect of f_2 becomes deadly.

It should be mentioned that this logical independence of T from other existing software metrics is not surprising. T ranges over many parameters, from the testing strategy used to the structure of the program to be tested, to the computational complexity of the program. The other metrics rely on one or very few parameters; thus, some of these metrics are incorporated into the very fabric of T. With so much to work with, it is easy to produce situations in which T behaves contrary to less robust metrics.

3 Adaptive software testing

That there are serious shortcomings to the conventional testing strategies discussed above is apparent in the phase of test-case generation. There pre-selection of paths occurs without regard to their feasibility (i.e. the existence of some inputs causing execution of those paths). That some paths will prove infeasible requires repetition of at least some of the path-selection phase and thus, wasted effort. However, we are then confronted with a compound path predicate that must be satisfied in order to execute the chosen path governed by that predicate. In general, this is, at best, an NP-complete problem [31]. With the necessity of having sometimes to find appropriate inputs to functions, this satisfiability problem can become undecidable [6].

3.1 *Adaptive approaches*

Therefore, the practical difficulties associated with the conventional approach to testing are considerable. A new more 'adaptive' (the term is first used in References 4 and

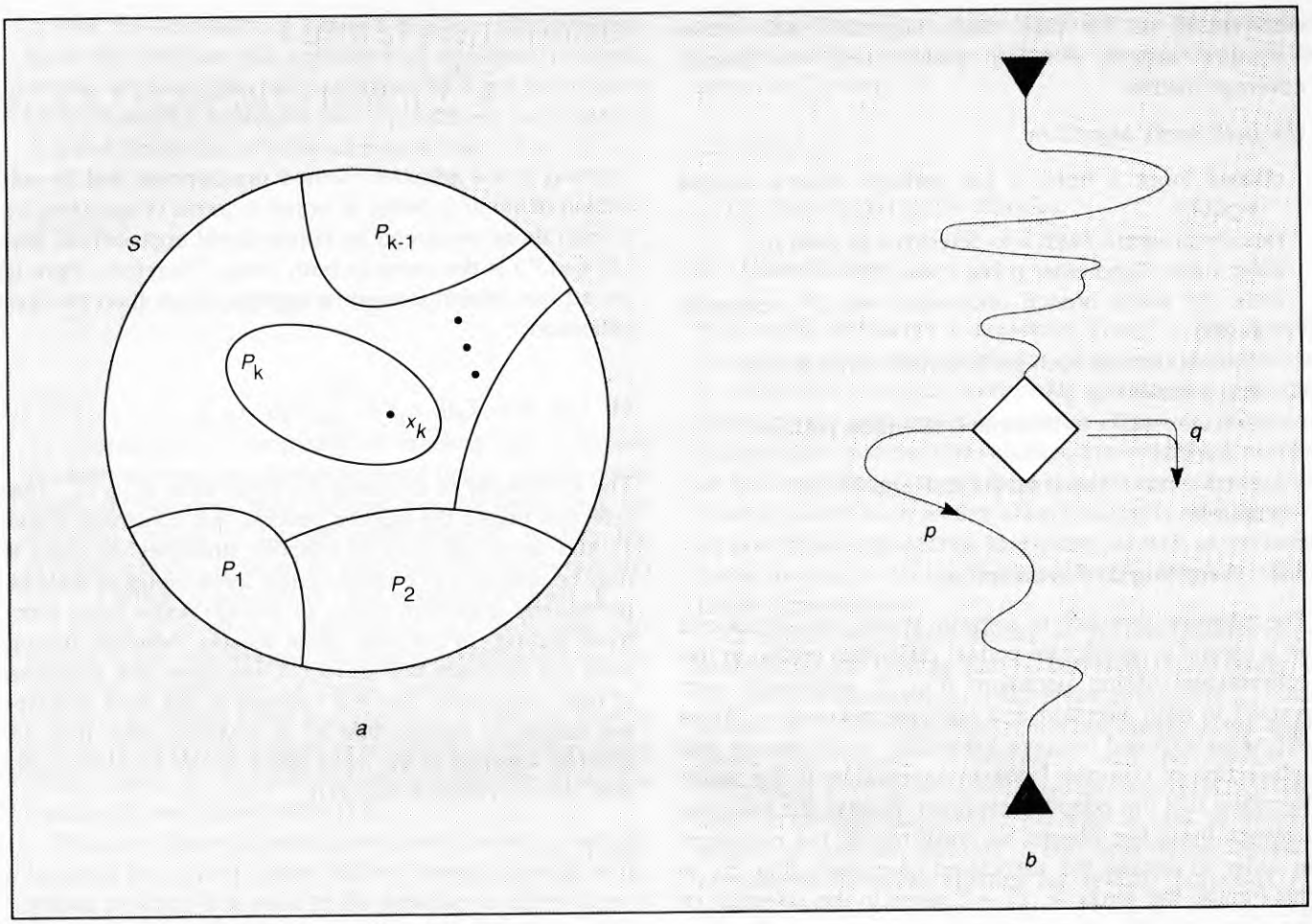

Fig. 4 Kundu and path prefix adaptive strategies

a Kundu: using diagonalisation strategy to find input x_k
b Path prefix: reversible prefix q of path p

11) methodology has begun to appear as a means of overcoming these difficulties. Before proceeding, we mention that we do not expect to fully solve the problems. After all, if we are 'treading' in the theoretically perilous 'waters' of NP-completeness and undecidability, there is no hope that a change of approach will settle everything. Thus, pragmatically we acknowledge that the adaptive approach is presented as an improvement of varying extent over the conventional.

Considering the issues in Section 2, we notice that several of the difficulties arise because the covering set of full paths is *preselected*, i.e. all of the path selection phase is carried out before the test-case generation phase. However, we have seen that this ideal fails because of infeasibility when, midway into test-case generation, we are forced to 'return to the drawing board' (i.e. select new paths). Thus, whether we like it or not, there is an intrinsic interplay between these two phases. The fundamental distinction of the adaptive approaches is that they not only acknowledge this interplay, they exploit it.

In adaptive testing, we once again make use of the program flowgraph F. However, the idea here is to add just one new test path (and hence, one new input test) at a time, using previous paths (inputs) as a guide to the selection of subsequent paths (inputs), according to some inductive strategy. It may be that we choose an initial input x_1 at random, thus determining a corresponding path p_1. Then, supposing that the pairs $\{x_1, p_1\}$, $\{x_2, p_2\}$, ..., $\{x_{k-1}, p_{k-1}\}$ have already been chosen, we have a particular strategy for using these data to influence the selection of x_k or p_k. The nature of this strategy serves to distinguish one adaptive method from another.

These important ideas were first developed by Kundu [10]. In using a kind of *diagonalization strategy*, he notes that each of the paths $p_1, p_2, \ldots, p_{k-1}$ is associated with corresponding conjunctive predicates $P_1, P_2, \ldots, P_{k-1}$, as before. These, in turn, describe non-overlapping subdomains of the input space S, as shown in Fig. 4*a*.

We try to find an input x_k, for which $x_k \in \neg(P_1 \cup P_2 \cup \cdots \cup P_{k-1})$, thus assuring that the corresponding path p_k is indeed distinct from all those chosen so far. If $P_i = C_1 \wedge C_2 \wedge \cdots \wedge C_n$, as before, we have only to ensure that x_k violates at least one of the conjuncts in each of the conjunctive predicates P_i $(1 \leqslant i \leqslant k-1)$. Certainly, this is a novel approach; but, we note that the diagonalisation strategy is not designed with any specific measure of testing thoroughness in mind.

In the new adaptive branch-coverage methodology, the *path prefix strategy*, a deliberate attempt is made to utilise the *best* of the previously traversed paths at each selection of a new input x. For each available path p, we determine its *reversible* prefix q to be the minimal initial portion of p to a decision node whose branches are not yet fully covered (Fig. 4*b*). Therefore, if, at any stage in the strategy, $p_1, p_2, \ldots, p_{k-1}$ are the previously executed paths, input x_k is found in order to cause reversal of the *shortest* reversible prefix q among all the p_i (cf. Path Prefix BINGO [11]). The mechanical nature of this strategy is

summarised as the path prefix algorithm (with stores X = input vectors, P = path history and M = branch-coverage matrix).

The path prefix algorithm

choose input x from S {or perhaps choose several inputs};
execute program f with x to determine its path p;
enter x into X and enter p into P and M;
while (M: some branch uncovered) with (P: reversible prefix)
  choose shortest such prefix q {with reversal q'};
  find x traversing q';
  execute f with x to determine extension path p {of q'};
  enter x into X and enter p into P and M
output X.

3.2 Adapting to the adaptive

The adaptive approach to software testing was introduced as a means to ameliorate certain difficulties arising in the conventional testing paradigm (Fig. 1), particularly with respect to path selection and test-case generation. These difficulties included frequent infeasibility in the former and satisfaction of complex Boolean expressions in the latter. Recalling that the adaptive approach iterates the interplay between these two phases, we must modify the paradigm in order to discuss the associated complexity (Fig. 5). In this Figure, the arrow $(V, E) \rightarrow X$ refers to the selection of the first input(s) whose execution generates the first path(s):

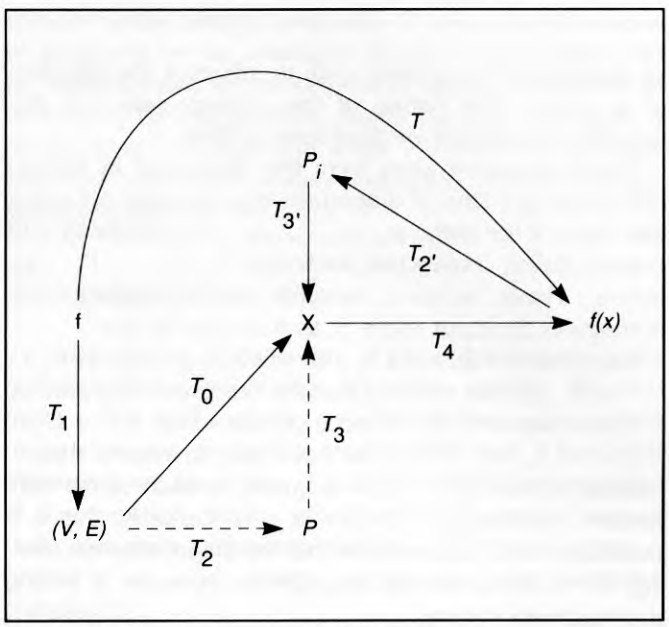

Fig. 5 Decomposition of T: adaptive strategies
$P_0 = \varnothing$
$P_k = P_{k-1} \cup \{p_k\}$
Dashed lines are conventional T_2 and T_3.

(1) $X \rightarrow f(x) \rightarrow P_i$. Then P is used to generate an additional input:
(2) $P_i \rightarrow x_{i+1}$. By (1), this serves to enlarge P_i.

This process is iterated until coverage is obtained under the desired criterion. Now, a complexity T_j is associated with each step, so that

$$(3)\ T(f, s) = T_1(f) + T_0 + T_4(f, x_1) + \sum_{i=1}^{|P|-1} [T_2'(f, x_i) + T_3'(f, P_i, s_2) + T_4(f, x_{i+1})].$$

Nothing in the adaptive method presupposes that its collection of inputs is better or worse, in terms of executing by f, than those produced by conventional approaches; and $|X|$ $(=|P|)$ is the same in both cases. Therefore; there is no *ad hoc* reason to assume anything other than the simplification;

$$(4)\ T_4(f, X) = T_4(f, x_1) + \sum_{i=1}^{|P|-1} T_4(f, x_{i+1}).$$

The complexity of choosing the initial input x_1 is T_0. This selection begins the looping process that constructs X and P, and so is part only of adaptive strategies. At times it may be desirable to choose certain initial inputs to traverse particularly important paths, to provide some initial functional testing, or for any other reason. Adaptive testing does not preclude this (in fact, it welcomes any instances of user interaction); but it is contrary to the spirit of adaptive testing to *require* that x_1 is anything other than an arbitrary element of the input space S. We thus conclude, that, since no effort is required,

$$(5)\ T_0 = 0.$$

3.3 Adaptive versus conventional testing

We may now compare the complexities of performing conventional versus adaptive testing (renamed to T_C and T_A, respectively). Based on the observations (4) and (5) above, T consolidates in the adaptive case:

$$(6)\ T_A(f, s) = T_1(f) + \sum_{i=1}^{|P|-1} [T_2'(f, x_i) + T_3'(f, P_i, s_2)] + T_4(f, X).$$

From earlier discussions, we recall that

$$(7)\ T_C(f, s) = T_1(f) + T_2(f, s) + T_3(f, P, s_2) + T_4(f, X).$$

Clearly, the comparison is reduced to the comparison of the middle term of T_A with the middle two terms of T_C, and the others are simply cancelled.

Lemma 5

$$\sum_{i=1}^{|P|-1} T_2'(f, x_i) < T_2(f, s).$$

Proof:

$T_2(f, s)$ measures the process of developing, from (V, E), a full set P of $|P|$ paths to assure the desired coverages, and path-generating functions can be rather involved. $T_2'(f, x_i)$, on the other hand, measures the mere chronicling of $|P| - 1$ paths as f executes on x_i. The instrumentation to monitor this was inserted during the construction of (V, E); and so virtually no resources are required at this step. ∎

That the inequality of Lemma 5 is so strong derives from the fact that the conventional strategies are path-driven, requiring the prior selection of a set of covering paths. Adaptive strategies are input-driven, necessitating only the monitoring of paths as they occur.

Lemma 6

$$\sum_{i=1}^{|P|-1} T_3(f, P_i, s_2) \leqslant T_3(f, P, s_2).$$

Proof

T_3 measures the complexity of creating and satisfying Boolean expressions governing entire paths; adaptive strategies are no worse than this and may be better (e.g. reliance on *prefixes* by the path prefix strategy). Anyway, this process is done $|P|$ times by conventional means but only $|P| - 1$ times by adaptive (x_1 was obtained for free). ▮

Lemma 6 is already established, but informally supported by the troublesome disregard of infeasibility by conventional preselection of paths before constructing X, resulting in wasted effort and requiring additional work at this stage. Although subject to empirical verification, infeasibility should be less problematic for adaptive strategies, resulting in less wasted effort [11].

Thus, our theory of testing complexity serves us well. By focusing the light it sheds on the various aspects of the testing process, it is seen to be sensitive to differences in the fundamental approaches to structural testing presented here. Thus, a highlight of the theory is stated in Theorem 2.

Theorem 2

$T_A(f, s) < T_C(f, s)$: given a program to be tested according to a given coverage criterion, an adaptive strategy requires less effort than a conventional strategy.

This is a stronger result than initially anticipated. After all, one motivation in developing the theory of testing complexity was to validate the speculation that the path prefix strategy is preferable to other strategies. Our result is even more general. Indeed, as the path prefix strategy is adaptive, we may simply write as in Corollary 8.

Corollary 8

$T(f, \text{Path Prefix}) < T(f, \text{conventional})$.

How gratifying that a major goal proves to be a mere corollary. Our only other example of an adaptive strategy, as the term is new to the literature, is that of Kundu. Certainly, we have Corollary 9.

Corollary 9

$T(f, \text{Kundu}) < T(f, \text{conventional})$.

But what about the Kundu strategy compared to that of the path prefix? The Kundu method is not crouched in particularly algorithmic terms, as is the case with the path prefix algorithm. Thus, a comparison must be conducted in rather informal terms, whereby we conclude [5] as shown in Corollary 10.

Corollary 10

$T(f, \text{Path Prefix}) \leqslant T(f, \text{Kundu})$.

4 Conclusion

This study represents a reasoned theory of the specific complexity of the various phases of the testing endeavour, in which the minimal complexity of adaptive strategies (including the path prefix) is provable. Moreover, the theory incorporates, and thereby validates theoretically, a number of well established and often used complexity metrics. However, work such as this always suggests new things to try and novel approaches to explore, as well as inevitable 'loose threads' to tie up. Below we briefly discuss possible future explorations.

Of course, large-scale studies on the practicalities of the testing schema T would also be beneficial. Such issues, as the practical dominance relationships among the T_i, correlations of T with other metrics, the fine-tuning of the coefficients t_i to classes of programs, and the difficulty of computing T, could be established and explored. We hope that the present study leads the way, at last, to avoiding single all-purpose metrics. Then theoretical research remains to be done; metrics for specific purposes. For example, the decomposition of T allows an even broader look at software life-cycle resource requirements. Thus, T can be 'added' to any effort metrics to predict design and development costs for a software project. Likewise, T can be augmented by measures of later stages in the project life-cycle; then the complexity of deriving a formal specification g from a problem could be integrated into T, as could measures such as

- T_5 = complexity of determining $f(X) = g(X)$.
- T_6 = complexity of debugging f.
- T_7 = cost of program modification.
- T_8 = cost of program maintenance.

For example, T_5 would measure the functional aspect of testing; there are significant issues involved here: formal versus informal specifications, complexity of specifications, how to determine $g(X)$, and so on. T_6 would obviously depend on the number of errors in f, integrating a variety of metrics reported in Reference 1 to estimate the number of errors.

Some notions of 'average' would be useful in the theory. To motivate this assertion, we recall that the Coverage Bound Lemma gives widely varying results for the same number d of decision nodes (depending, for example, on the nesting/sequence of those decisions). Or we note that the Halstead length metric is best used in measuring test-case generation complexity when restricted to paths and partial paths, rather than for total programs, as is customary. Therefore, readily computable attributes, such as an 'index of nesting' or 'average path length,' would be welcome within the theory. We are not aware of any such notions, even for structured programs; for example, d/P_{PC} is hardly suitable as an 'average number of decisions per

path'; try it for decisions in sequence. There would doubtless be a relationship between an index of nesting and average path length acceptable by the theory, but these have yet to be introduced. Indeed, Shooman [1] lists as an open question (p. 269):

'If the value v(G) is a lower bound and P_{PC} an upper bound on the number of program tests, how close are these bounds in practice?'

Another area of future interest is the study of the sensitivity of T to structured versus unstructured programs and to program restructuring [15]. On the one hand, this could give further practical evidence as to the preferred status of structured programming or at least illuminate those cases where limited nonstructuring is acceptable. On the other hand, this study might indicate instances when restructuring is desirable, as opposed to when it is not.

Finally, attention has been focused [28, 32] on Prather's work [17] on axiomatic software complexity measure, resulting in a new family of 'hierarchical' metrics, based on the underlying hierarchical structure of programs. As this work appears to be quite significant and promising, attention should be given to the possible integration of the testing-specific complexity schema T within these new concepts. Activities might include necessary modifications to T to incorporate it into the new setting, or applications of T to the classes of program structures, forming a sort of 'catalogue' of program constructs with their testing complexity.

Using such analyses as reported here, software designers have a tool for predicting the amount of resources required to conduct testing, at least relative to some previously tested and utilised software from an individual programmer, a team or a 'shop'. Most of the software tools necessary for such analyses are also already available.

5 References

[1] SHOOMAN, M.L.: 'Software engineering' (McGraw-Hill, New York, 1983)

[2] BEIZER, B.: 'Software system testing and quality assurance' (Van Nostrand Reinhold, New York, 1984)

[3] PRATHER, R.E.: 'Theory of program testing—an overview', *Bell Syst. Tech. J.*, 1983, **62**, (10), (part 2), pp. 3070–3105

[4] MYERS, J.P. JR.: 'Adaptive approaches to structural software testing'. Proc. Fifteenth Annual ACM Computer Science Conference, 1987, p. 432

[5] MYERS, J.P. JR.: 'Software testing: a new methodology and a theory of complexity'. PhD. Dissertation, University of Denver, 1986

[6] HAUSEN, H-L.: 'Comments on practical constraints on software validation techniques' *in* HAUSEN, H-L. (Ed.): 'Software validation' (Elsevier, Amsterdam, 1984)

[7] CLARKE, L.A.: 'A system to generate test data and symbolically execute programs,' *IEEE Trans.*, 1976, **SE-2**, (3), pp. 215–22

[8] CLARKE, L.A.: 'Automatic test data selection techniques'. INFOTECH State of the Art Report, Software Testing, 1979, pp. 43–63

[9] CLARKE, L.A., and RICHARDSON, D.J.: 'Symbolic evaluation methods for program analysis' *in* MUCHNICK, and JONES, (Eds.): 'Program flow analysis' (Prentice-Hall, Englewood Cliffs, New Jersey, 1981) pp. 264–300

[10] KUNDU, S.: 'SETAR—a new approach to test case generation'. INFOTECH State of the Art Report, Software Testing, 1979, pp. 163–186

[11] PRATHER, R.E., and MYERS, J.P. JR.: 'The path prefix software testing strategy', *IEEE Trans.*, 1987, **SE-13**, (7), pp. 761–766

[12] BELADY, L.A.: 'On software complexity'. Proc. IEEE Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, 1979, pp. 90–94

[13] COOK, M.L.: 'Software metrics: an introduction and annotated bibliography', *SIGSOFT Soft. Eng. Notes*, 1982, **7**, (2)

[14] MCCABE, T.J.: 'A complexity measure', *IEEE Trans.*, 1976, **SE-2**, (4), pp. 308–319

[15] EVANGELIST, W.M.: 'Software complexity metric sensitivity to program structuring rules', *J. Syst. Soft.*, 1983, **3**, pp. 231–243

[16] HANSEN, W.J.: 'Measurement of program complexity by the pair (cyclomatic number, operator count)', *SIGPLAN Not.*, 1978, **13**, 3

[17] PRATHER, R.E.: 'An axiomatic theory of software complexity measure', *Comput. J.*, 1984, **27**, (4), pp. 340–347

[18] HALSTEAD, M.H.: 'Elements of software science' (Elsevier North-Holland, 1977)

[19] KAFURA, D., and CANNING, J.: 'A validation of software metrics using many metrics and many resources.' Report TR-85-6, Virginia Tech. Computer Science, 1985

[20] WOODWARD, M.R., HENNELL, M.A., and HEDLEY, D.: 'A measure of control flow complexity in program text', *IEEE Trans.*, 1979, **SE-5**, (1), p. 45

[21] PRESSMAN, R.S.: 'Software engineering: a practitioner's approach' (McGraw-Hill, New York, 1982)

[22] WEYUKER, E.J.: 'Evaluating software complexity measures', *IEEE Trans.*, 1988, **SE-14**, (9), pp. 1357–1365

[23] KEARNEY, J.K., SEDLMEYER, R.L., THOMPSON, W.B., GRAY, M.A., and ADLER, M.A.: 'Software complexity measurement', *Commun. ACM*, 1986, **29**, (11), pp. 1044–1050

[24] KAFURA, D., and REDDY, G.R.: 'The use of software complexity metrics in software maintenance', *IEEE Trans.*, 1987, **SE-13**, (3), pp. 335–343

[25] MYERS, G.J.: 'An extension to the cyclomatic measure of program complexity', *SIGPLAN Not.*, 1977, pp. 61–64

[26] MYERS, J.P. JR.: 'The use of flowgraph generators in software testing'. MSc. Thesis, University of Denver, 1983

[27] PAIGE, M.R.: 'An analytical approach to software testing'. Proc. IEEE Computer Software and Applications Conference, COMPSAC 78, Chicago, 1978, pp. 527–531

[28] PRATHER, R.E.: 'On hierarchical software metrics', *Softw. Eng. J.*, 1987, **2**, (2), pp. 42–45

[29] MACHTEY, M., and YOUNG, P.: 'An introduction to the general theory of algorithms' (Elsevier North-Holland, New York, 1978)

[30] WILF, H.S.: 'Algorithms and complexity' (Prentice-Hall, Englewood Cliffs, New Jersey, 1986)

[31] GAREY, M.R., and JOHNSON, D.S.: 'Computers and intractability' (W.H. Freeman, San Francisco, 1979)

[32] FENTON, N.E., and WHITTY, R.W.: 'Axiomatic approach to software metrication through program decomposition', *Comput. J.*, 1986, **29**, (4), pp. 329–339

The author is with the Department of Computer Science, Trinity University, San Antonio, Texas 78212, USA.

The paper was first received on 18th March and in revised form on 9th July 1991.