STRUCTURED TESTING: ANALYSIS AND EXTENSIONS

Arthur Henry Watson

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

TR-528-96

November 1996

Abstract

Structured testing, also known as basis path testing, is a methodology for software module testing based on the cyclomatic complexity measure of McCabe. In this dissertation, we analyze the theoretical properties of structured testing, describe the implementation of a system to support structured testing, empirically evaluate the error detection performance of structured testing, and extend the structured testing approach to cover integration testing.

We exhibit a class of programs with unbounded complexity for which the structured testing approach is both necessary and sufficient to ensure correctness, and place structured testing in a hierarchy with other structural testing criteria. We also discuss Weyuker's axioms for testing criteria, and show that a variant of structured testing in which only executable paths are considered satisfies those axioms.

We describe an automated system to support structured testing for the C language. We present a technique for assessing and improving a test suite with respect to basis path coverage, based on source code instrumentation and execution trace analysis.

We present an empirical study comparing the error detection effectiveness of structured testing, all-uses data flow coverage, and branch coverage. Structured testing outperformed branch coverage, was comparable to all-uses, and was more robust with respect to test set minimization than either branch coverage or all-uses.

We extend structured testing to support software integration testing. We present a mathematical model for integration testing, and derive an integration test sufficiency criterion from that model. We motivate the criterion from a practical perspective, and demonstrate its relationship to structured module testing. We extend our automated

system to assess test suites with respect to our integration test criterion, and suggest extensions of the core method to support incremental integration and object-oriented programming.

Acknowledgments

Much of the credit for this dissertation belongs to my family, whose consistent support and encouragement helped sustain me. My wife, Carolyn, encouraged me to put in the necessary time even when it encroached on our year as newlyweds. My brother, Richard, was always willing to put things in perspective. My parents, William and Christine, provided both moral and financial support over the years. Thank you all.

My advisor, Anne Rogers, helped improve both the details and the structure of this dissertation, and left a hopefully lasting impression on my writing style. Dick Lipton and Rich DeMillo helped out as readers, and Rich made significant contributions to the empirical work.

Tom McCabe literally provided the topic by inventing structured testing. Dave Hanson helped focus the research objectives and make the original outline. Bob Horgan provided encouragement and many helpful suggestions on an early draft. Issa Feghali, Keith Gallagher, Charles Butler, and Peter Wayner also read drafts and gave valuable feedback. Ken Steiglitz and Jack Gelfand provided support and research experience during the early phases of the degree program.

Financial support was provided by NSF Grant MIP87-05454 and the Department of Computer Science at Princeton University.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Structured testing is a software module testing technique based on the *cyclomatic complexity* measure of McCabe [37]. The key requirement of structured testing is that all decision outcomes must be exercised independently during testing [58]. The number of tests required for each software module is given by the cyclomatic complexity of that module.

Structured testing was first suggested twenty years ago [37], is included in some college curricula [49], has been used in a limited manual fashion in industry for over a decade, and even has a US national standard [38]. Prior to the work presented in this dissertation, however, structured testing has not been subjected to rigorous analysis, has not been automated, and has not been compared to other testing criteria either theoretically or empirically. In particular, the lack of automated support for structured testing has largely prevented it from being used outside of small projects.

This dissertation presents a detailed analysis of structured testing from a theoretical perspective, describes an implementation of a system that automates structured testing, and describes empirical results concerning the effectiveness of structured testing for detecting errors. It also describes extensions of the structured testing technique to support integration testing.

In this chapter, we review several other testing techniques and related work to put structured testing in context. We then describe structured testing and discuss several criticisms of structured testing from the literature. We then give an overview of this dissertation, indicating how we address these criticisms.

1.1 Common testing strategies

There are many testing techniques. Some are similar to structured testing and others are complementary. This section discusses several testing techniques to place structured testing in context and provide background information for the comparisons of structured testing with other techniques presented in Chapters 2 and 4.

There are two major classes of software testing strategies: *black box*, in which testing is done independently of the software implementation, and *white box* (or *glass box*), in which testing is guided by the implementation. Major black box testing strategies [1] include requirements-based testing, design-based testing, random testing, testing in a production environment, and multiple-version testing. Major white box testing strategies [1] include extremal value testing, mutation testing, and structural testing. Structured testing is a structural testing technique, so it falls into the white box category.

Adrion et al.'s survey [1] describes several techniques based on the above black box and white box strategies. We first discuss the black box techniques, which are the least similar to structured testing and therefore most likely to be used in combination with it.

In *boundary value testing* [40], the input data as determined from the requirements is partitioned into classes, and test data is selected both from the interior and the boundary of each class. In *functional testing* [24], the methods of boundary value testing are also applied to a program's design, which allows a greater depth of testing than that attained by analysis of the requirements alone, since what appears to be a single input domain in a requirements specification may be partitioned into several sub-domains in a design. For example, a specification may indicate that a daily backup of data files must be performed, while a design may indicate that full backups will be done on Sundays and incremental backups will be done on other days. A drawback of requirements and design–based techniques is that requirements and design specifications are often insufficiently detailed and accurate for use in testing.

In *random testing* [16], program inputs are generated randomly from the input domain. Duran and Ntafos [11] present an analysis of random testing that demonstrates good code coverage and error detection, although other studies (for example, [20]) find that random data typically gives poor code coverage. The effectiveness of random testing can vary depending on the application being tested and the distribution from which test cases are selected. And, although any partition testing strategy can be made at least as effective as random testing by making the number of test cases for each partition proportional to its size [5], random testing avoids the overhead of calculating partitions. Given an automated output checking procedure, it is often more cost-effective to generate many random tests than a small number of carefully constructed partition tests. However, if verifying program behavior is expensive, random testing is less attractive. Since random testing tends to avoid exceptional cases, it is most effective in combination with extremal value testing.

Testing in a production environment is similar to random testing in which the input distribution corresponds to the actual usage pattern of the program [53], so the same disadvantages of random testing also apply. However, this distribution tends to be more skewed away from exceptional cases than the fairly uniform input distributions typically used in random testing, so the code coverage properties are worse and as a result it is important to augment this strategy with other techniques [49].

Multiple version testing [4] concentrates more on detecting incorrect behavior than on generating test data. Several independent versions of software are developed, and each version is run on a test data set generated by some other technique. Data sets on which all versions agree are assumed to be processed correctly, and the program's behaviors on the remaining data sets are investigated. Multiple version testing is valuable for detecting implementation errors, but it is poor at detecting specification errors, since all versions developed from the same erroneous specification may be incorrect [49].

We now discuss white box testing techniques, which are more closely related to structured testing. In *extremal value testing* [1], program expressions are required to assume values both in the interior and on the boundary of their valid range. Typical examples are array indices and the operands of relational operators. Extremal value testing is closely related to boundary value testing, the difference being that in extremal value testing the test cases are developed from the implementation rather than the functional specification.

In *condition testing* [52], the truth assignments to operands in boolean expressions are required to satisfy one of a family of related criteria. The simplest criterion requires that each boolean operand assume both TRUE and FALSE values, which can typically be accomplished with just two tests for arbitrarily complex expressions, for example by setting all operands to TRUE in one test and to FALSE in another. The strictest condition testing criterion, known as *multiple condition coverage*, requires all possible boolean truth assignments to be exercised, which can require a number of tests that is exponential in the number of boolean operands. An interesting criterion of intermediate strength, known as *modified condition/decision coverage* [7], requires that each boolean operand independently affect the expression outcome during testing, which typically requires a number of tests proportional to the number of boolean operands. Chilenski and Miller showed that this criterion has a high probability of detecting errors in randomly generated boolean expressions [7], and their analysis depends only on the number of tests required. Weyuker et al. [63] point out that typical erroneous program expressions tend to have fewer points of failure than random expressions.

In *mutation testing* [9], a test set is developed to distinguish the program being tested from a large set of “mutant” programs which differ from the original program by small changes intended to model common programming errors. Examples of mutations include replacing one variable with another and replacing one relational operator with another. The basic idea behind mutation testing is that if two programs which differ by a common error are equivalent with respect to a test set, then that test set is insufficient to determine whether the original program contains that error. No particular method for constructing the candidate test data set is specified, although random data generation is most often used in practice [10]. One disadvantage of mutation testing is that many mutants (quadratic in the size of the program [23]) are generated, each of which is a separate program to be run on the proposed test data set. A corresponding advantage is that the test data selection process may be run automatically, since it only compares program behavior rather than assessing the correctness of that behavior. Once a small set of tests has been constructed that separates the original program from its mutants, the tester need only verify the correctness of the original program behavior on that small set [10].

There are several variants of the mutation testing approach that address efficiency concerns. Both *constrained mutation*, in which only a subset of typical mutant-generating operators are used, and *selective mutation*, in which a randomly selected subset of mutant programs are used, generate fewer mutants than standard mutation testing, and retain much of its error detection capability [29]. *Weak mutation testing* [25] allows the program state during execution rather than just the program output to be used to distinguish mutants, and has also been shown effective at detecting errors [43].

A more serious disadvantage of mutation testing is that mutants may be functionally equivalent to the original program, in which case no data set will distinguish them. For example, if a variable is never zero at a decision in which it is compared with zero using the “>” operator, the mutant in which that operator is replaced with the “>=” operator will be equivalent. Manual work is typically required to decide whether to continue attempting to distinguish mutants or consider them to be equivalent to the original program.

1.1.1 Structural testing

In *structural testing*, the test adequacy criterion can be expressed entirely in terms of the program execution trace through its control flow graph during testing. That is, any two tests that trace exactly the same path through the flow graph are equivalent under a structural testing strategy. The structural criterion may be *calculated* from non-structural information, however, for example from data flow analysis. The *all-paths* criterion, requiring one test through each of the typically infinite possible paths through the flow graph, is the strongest possible structural testing strategy in terms of error detection ability. Any weaknesses in error detection that apply to this strategy, such as insensitivity to boundary values of data, also apply to all structural strategies, and it is frequently used as a basis for comparison in theoretical analyses.

The simplest structural testing criterion is *statement testing*, in which every program statement must be executed. Although executing every statement is certainly desirable during testing, statement testing is widely considered inadequate [44]. *Line testing*, also known as *code coverage*, although not a true structural testing strategy, is often treated as synonymous with statement testing and is widely used in industry. Line testing requires that all (or a chosen percentage) of the source lines of code be exercised during testing. For assembly languages in which statements and lines are synonymous, this is a legitimate strategy. For modern languages, however, there is too much reliance on the source format rather than the implementation structure [36]. For example, an entire C program could be formatted as one line, in which case any single test would satisfy the line testing criterion.

In *branch testing*, every decision outcome must be realized during testing. Branch testing is considered the state of the practice in the industrial community, in the sense that when researchers propose a more elaborate testing technique they typically compare it to branch testing in their analysis, and subsumption of branch testing is typically considered necessary for a proposed structural testing technique.

Data flow testing [19] is a major category of structural testing in which tests are required to cover certain types of data interaction. A typical data flow criterion requires that all path segments between a variable being assigned a value and that variable's value being used (with no intervening assignments to that variable) be covered during testing. Clarke et al. [8] present a comparative analysis of many proposed variants. Data flow testing frequently gives much better detection of typical programming errors than branch testing [42, 64], but a substantial effort is required to calculate accurate data flow information for typical programming languages, and in fact the general case remains open [23, 45, 46]. One interesting approach [51] involves using the maximum data flow complexity of a single "pseudovariable" to calculate the required test sets directly from the control flow graph.

A weakness of data flow techniques arises from infeasible data associations. There are a vast number of data associations, which must be covered, and some may not be coverable due to computational constraints of the program. For example, consider the following code fragment:

```
j = i - 1;
if (i > 0)
    x = i;
else
    x = 0;
if (j > 0)
    y = 1/x;
else
    y = x;
```

There are four data associations involving variable `x`, since each of the two possible assignments to `x` in the first decision can be paired with each of the two possible uses of `x` in the second. However, only three of those associations are feasible, since the second decision can only be true when the first decision is true. Since it is undecidable whether a specific subpath may be covered, testers typically attempt to cover a fixed percentage of data associations in an attempt to compensate statistically for these uncoverable associations. Some researchers have noted a sharp discontinuity in error detection ability as coverage drops smoothly. Weyuker et al. [64] suggest that testers should only accept test sets that achieve 100% coverage of the feasible associations.

Frankl [13] notes that although most reasonable programs only have uncoverable statements and branches due to diagnostic and defensive programming techniques, typical program logic contains uncoverable data associations. Thus, uncoverable associations are more of an issue for data flow testing than many other testing methods. She gives an analysis of several data flow testing criteria restricted to realizable program paths, and shows that these criteria satisfy Weyuker's axioms for testing criteria [60]. We discuss those axioms in Chapter 2. She also describes *partial symbolic evaluation* of path expressions, a semi-automatic technique to help testers determine interactively whether associations are coverable.

1.2 Structured testing

Structured testing, the major topic of this dissertation, requires a basis set of paths through each module's control flow graph to be tested. The *cyclomatic complexity*,

$v(G)$, of a module, suggested by McCabe [37] and widely used as a measure of software complexity, is typically computed as $2+e-v$, where e and v are the number of edges and vertices in the control flow graph G, respectively. The cyclomatic complexity is the number of paths in each basis set [37] and is therefore the number of tests required for structured testing. Mathematically, a basis is a linearly independent set of paths that generates all possible paths by linear combination. The linear algebra is done in the vector space of flow graph edge incidence, and a path is projected onto a vector by counting the number of times the path flows through each edge. The rank of a set of tests is given by the rank of the set of associated vector projections, and the structured testing criterion is satisfied when this rank is maximized. A complete example of structured testing is given in Chapter 3, and more detailed discussion of the basic definitions is given in Appendix B.

Structured testing has several strengths. It implies that each decision outcome must be exercised during testing, which ensures a reasonable minimum level of effectiveness [37]. Since the number of tests required equals the cyclomatic complexity, it is easy to predict the number of tests required in advance and to measure the progress of testing. Structured testing concentrates testing effort on the most complex software, which also tends to be the most error-prone software [58]. This link between complexity and the number of tests is not shared by other major structural testing criteria [56]. A stronger property than the number of tests is the fact that the tests must generate all possible paths by linear combination, so that in an abstract mathematical sense the tests must be a representative subset rather than redundantly exercising certain areas of the module while neglecting others.

Structured testing has been criticized in the literature for the number of tests required, the fact that the required paths are not unique, and the fact that a proposed method for generating a set of paths often produces infeasible paths. This dissertation addressess these criticisms.

Prather criticizes structured testing first for requiring more tests than branch testing, then later for requiring slightly fewer tests than another strategy [48]. The critical issue is whether the tests required by structured testing are cost-effective compared to the tests required by other methods. The experimental results presented in Chapter 4 show that structured testing compares favorably to branch testing for error detection even when it does not require significantly more tests.

Tai [51] criticizes structured testing for not having "a theoretical basis for the selection of n test cases," where n is the cyclomatic complexity. A series of studies show correlations between cyclomatic complexity and the number of errors [18, 54, 55], which provides a partial justification for the number of test cases. However, the number of test cases is just a by-product of the true theoretical basis for structured testing, that generating all paths by linear combination is a desirable property of a test set. Chapter 2 presents a class of programs for which structured testing is both necessary and sufficient to ensure correctness.

Prather also criticizes structured testing because the set of required paths is not unique [48]. Most other structural testing techniques give a set of required coverage items, such as statements or data flow associations, which can be simply checked off during testing to verify satisfaction of the criterion. Chapter 3 describes a technique for automatically measuring satisfaction of structured testing, so that a checklist approach is not required. With automated support, the freedom to select among many alternative adequate sets of test paths is a strength of the method, since the criterion may often be satisfied while avoiding specific infeasible paths.

Evangelist [12] criticizes structured testing because the manual technique presented by McCabe [38] for generating test sets to satisfy the structured testing criterion relies on deriving test data for a set of specific paths, which is an undecidable problem. This is a weakness of that test generation technique rather than of structured testing itself. For example, that same technique also generates test sets to satisfy statement and branch coverage, which do not have undecidability problems in practice. However, the limitations of this test generation technique have limited the practical use of structured testing to relatively simple programs. Chapter 3 presents automated techniques that reduce and in many cases eliminate the reliance on testing specific paths.

1.3 Dissertation overview

This dissertation describes the automated support and extensions necessary to make structured testing a practical technique, as well as presenting both theoretical and empirical analysis that addresses published criticisms of structured testing.

In Chapter 2, we present a theoretical analysis of structured testing, placing it in a subsumption hierarchy with several criteria that have been studied extensively. We also give a characterization of structured testing as the perfect testing method for a specific infinite class of programs and show that the feasible variant of structured testing satisfies Weyuker's axioms [60, 62] for testing criteria.

In Chapter 3, we describe a system to automate structured testing by instrumenting C code, collecting trace data during program execution, and performing both control flow graph analysis and matrix algebra to calculate whether a set of tests has satisfied the structured testing criterion. We present an algorithm for generating a basis set of paths through a control flow graph, which the system uses to produce a minimal set of additional paths through each function that, if executed, would extend the original tests to satisfy the criterion. A commercial product [32] based on this work has extended usage of structured testing significantly.

In Chapter 4, we describe an empirical study comparing the error detection effectiveness of structured testing, all-uses data flow coverage, and branch coverage. The criteria were evaluated with respect to nine erroneous programs using randomly generated test data sets, both with and without minimizing the test sets to exclude tests that did not increase coverage with respect to each criterion. Structured testing and all-uses both consistently outperformed branch coverage. Without minimizing test sets, structured testing had approximately the same error detection effectiveness as all-uses while requiring fewer tests. With minimizing test sets, structured testing had better error detection effectiveness than all-uses while requiring slightly more tests. Since structured testing performed better than branch coverage and at least comparably to all-uses data flow coverage, this study provides empirical evidence that structured testing is effective at detecting errors.

In Chapter 5, we discuss extensions for integration testing. McCabe and Butler [39] proposed an extension of structured testing to integration testing, in which control complexity that does not affect procedure calls is removed and analysis is done on this reduced graph. They also applied the idea of testing decision outcomes independently, an important property of structured testing for individual modules, to entire programs. However, some details of their approach were vague, not much analysis was given, there was no discussion of automated support, and there was no provision for integrating systems in stages. We describe an alternative integration level extension of structured testing that is well-defined and amenable to analysis and automation. We present two metrics that generalize cyclomatic complexity to measure the amount of integration-related control structure in single functions and entire programs, and establish a precise mathematical relationship between the two. We present a detailed analysis of McCabe and Butler's work, and establish a mathematical inequality between their metrics and ours.

We also discuss practical issues concerning control coupling and infeasible paths in integration testing, and propose a realistic integration testing criterion. We present an efficient algorithm to select a bounded set of control flow paths that satisfies the criterion, and demonstrate that selecting a minimal set of such paths is NP-hard. We describe an automated system to support the integration testing criterion, similar to our (module level) automated structured testing system. We also propose a method for generalizing structural integration criteria from support of single-phase integration to supporting the more realistic incremental integration model, and use that method to generalize our structured integration testing criterion.

Chapter 6 gives conclusions and suggestions for future work.

Chapter 2

Analysis of Structured Testing

The structured testing criterion [38], derived from analysis of module flow graphs using linear algebra, requires a basis set of paths to be tested for each module. See Appendix B for discussion of the basic definitions of structured testing. We construct a simple infinite class of programs such that if the output is correct for a basis set of flow graph paths then the output will be correct for all inputs that cause the program to halt.

One testing criterion is said to *subsume* another when every data set satisfying the first criterion also satisfies the second. We choose a subclass of programs such that any criterion that does not subsume structured testing within that subclass is insufficient to ensure correctness. We then show that some other major testing criteria do not subsume structured testing for this subclass, including the common statement and branch coverage criteria as well as the p-uses, c-uses, all-uses, and all-du-paths criteria, which were introduced by Rapps and Weyuker [50] and developed for C programs by Horgan and London [23]. Note that this also shows that none of these criteria are universally better than structured testing at detecting errors. We then show an example of an incorrect program and a set of tests that satisfies the structured testing criterion, does not satisfy the p-uses or c-uses criterion, and fails to detect the error. We use this information to place structured testing in a hierarchy with these other criteria. Structured testing is thus theoretically incomparable with all standard data flow testing criteria both in terms of the subsumption hierarchy and in terms of error detection.

We also define the *feasible structured testing* criterion by restricting the structured testing criterion to the paths in the module flow graph that are actually executable. Frankl [13] defined the *feasible data flow criteria*, and showed that most of them satisfy Weyuker's original set of axioms [60]. We discuss these axioms, Weyuker's additional axioms [62], and an axiom proposed by Zweben and Gourlay [66], and show that the feasible structured testing criterion satisfies all of them.

A frequent criticism of structured testing is that it does not have as clear an intuitive foundation as many other testing methods. Examination of a class of programs for which it is the ideal testing method may provide some insight. By showing that a theoretical variant of structured testing satisfies Weyuker's axioms, we demonstrate that it has several intuitively desirable properties. Chapter 4 provides empirical evidence in support of structured testing.

2.1 A simple class of programs

The programs that we will analyze have one integer-valued variable, `d`. The computational statements consist of adding an integer-valued (positive or negative) constant to `d`. All decision statements (that is, branching of control flow, such as an `if` or `while` statement) are controlled by reading a decision value from the input. We may examine the value of `d` before and after (but not during) the execution of the program. A program in the class is considered *correct*, if the value of `d` is always unchanged after executing the program, regardless of execution path. Figure 1 shows a C program that belongs to the class. This program is not correct, since for example the input sequence TRUE, TRUE, FALSE gives an execution path which subtracts one from the original value of `d`.

```
extern int d;
main()
{
    while (ReadDecisionValue()) {
        d = d + 2;
        if (ReadDecisionValue())
            d = d + -3;
        else
            d = d + -1;
    }
}
```

Figure 1: A C program in the class

2.2 Testing programs in the class

Consider the following path through the program of Figure 1: TRUE, TRUE, TRUE, FALSE, FALSE. The program behaves correctly on this path (the value of `d` is preserved), and this path satisfies the branch coverage testing criterion that each decision must have all possible outcomes exercised. Thus, branch coverage is insufficient to test members of the class. We now show that this situation can never occur with structured testing.

Theorem 1 *Structured testing is sufficient for the class.*

Consider a program in the class and its flow graph. Assign each edge in the flow graph a weight as follows. If the destination node of the edge corresponds to a statement that adds a constant to `d`, the edge weight is the value of that constant. Otherwise, the edge weight is zero. Figure 2 shows the flow graph of the program of Figure 1 with the edge weights given.

Consider a member of the class of programs for which the structured testing criterion has been satisfied without any incorrect behavior being observed. This means that the program behavior was correct for a basis set of paths. Each path has a corresponding edge vector, in which the elements of the vector are the number of times each edge was executed along the path. Let the edge vectors corresponding to

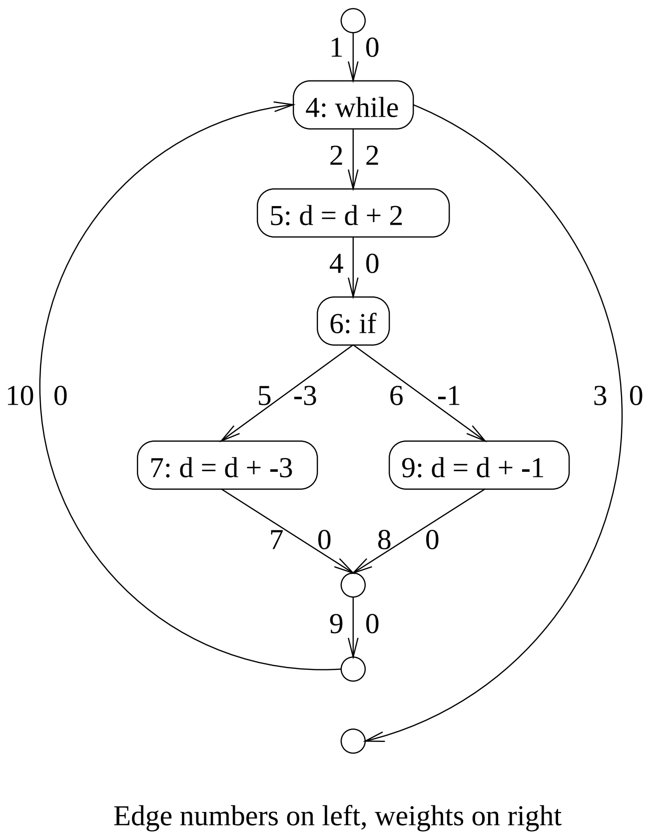


Figure 2: Flow graph

those basis paths be $v_1, \ldots, v_n$, and let the program edge weight vector be w. As each edge is traversed along a path, `d` is changed by the corresponding element of w. This means that the total change to `d` along a path is the inner product of that path's edge vector and w. Since each tested path produced correct behavior (`d` was unchanged), $v_i \cdot w = 0$ for each i. Consider any path through the program. Its corresponding edge vector, v, can be expressed as a linear combination of the edge vectors corresponding to the basis paths, that is, there exist $a_1, \ldots, a_n$ such that $v = \sum a_i v_i$. Then the output of the program when run through that path is equal to:

$$\begin{aligned} & v \cdot w \\ = \; & (\textstyle\sum a_i v_i) \cdot w \\ = \; & \textstyle\sum a_i (v_i \cdot w) \\ = \; & \textstyle\sum 0 \\ = \; & 0. \end{aligned}$$

Thus the program has correct behavior on all paths. □

The converse, that satisfying the structured testing criterion is necessary to test programs in the class, is also true in the following sense. Call a set of tests *structurally sufficient* for a program if, for each incorrect program in its class, at least one of the tests causes the incorrect program to exhibit incorrect behavior. The reason for this definition is that for any specific incorrect program, one test is sufficient to produce incorrect behavior. Structural sufficiency is a stronger condition that allows us to alter the functional statements (edge weights in the corresponding flow graph) in an attempt to exploit weaknesses in a structural testing strategy. We now show that structured testing is the weakest testing strategy that is structurally sufficient.

Theorem 2 *Any structurally sufficient testing criterion for the class subsumes structured testing.*

Consider the edge vectors $\{e_j\}$ for any set of tests that does not satisfy the structured testing criterion for a given program in the class. We will construct an edge weight vector w for which each test exhibits correct behavior, but which forms an incorrect program.

Let $\{v_1, \ldots, v_k\}$ be a basis for $\{e_j\}$, and extend this to $\{v_1, \ldots, v_n\}$, which is a basis for the edge space of the program graph. Since the tests do not satisfy the structured testing criterion, $n > k$. Let the matrix $V = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$ have the v_i as its rows. Observe that V has rank n, so we may select n linearly independent columns of V to form the square n-dimensional invertible matrix $U = \begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}$. Let r be an n-element column vector such that $r_n = 1$ and $\forall i \neq n, r_i = 0$, that is, $r = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{pmatrix}$. Let $x = U^{-1}r$. The entries of x are rational, and their denominators in lowest terms have a nonzero least common multiple g. Form w from x by multiplying by g and expanding to the same dimension as V, keeping the same column correspondence as between U and V, filling in the additional columns with 0. Then $Vw = U(gx) = (gr)$. This means that for each i, $v_i \cdot w = 0$, and thus also for each j, $e_j \cdot w = 0$, so the program exhibits correct behavior for each tested path. However, the program has incorrect behavior for the path corresponding to v_n, since $v_n \cdot w = g \neq 0$. Thus the initial set of tests is not structurally sufficient. □

As an example, consider the program structure shown in Figure 2, and the following set of four tests:

Test 1: TRUE, TRUE, TRUE, FALSE, FALSE.
Test 2: TRUE, TRUE, FALSE.
Test 3: TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE.
Test 4: TRUE, FALSE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, FALSE.

The corresponding edge vectors are:

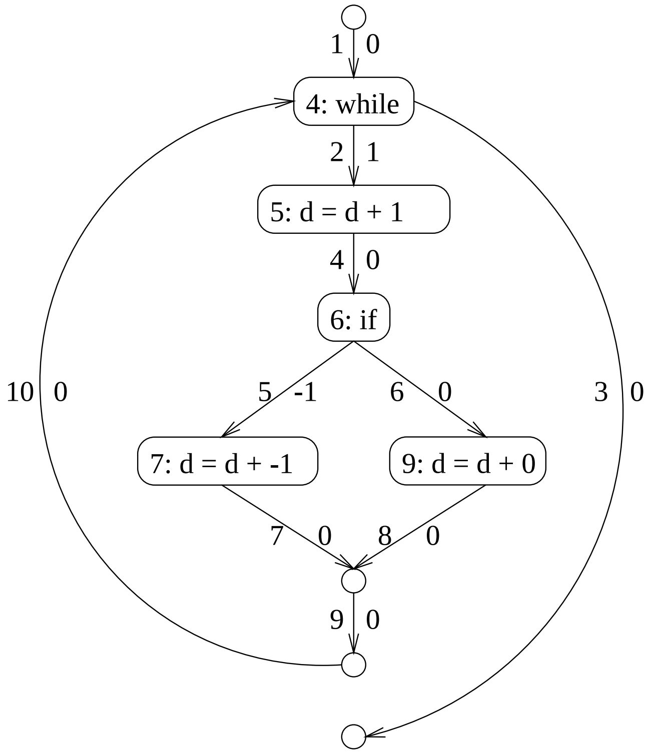
$$
\begin{aligned}
e_1 &= \begin{pmatrix} 1 & 2 & 1 & 2 & 1 & 1 & 1 & 1 & 2 & 2 \end{pmatrix}. \\
e_2 &= \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}. \\
e_3 &= \begin{pmatrix} 1 & 3 & 1 & 3 & 1 & 2 & 1 & 2 & 3 & 3 \end{pmatrix}. \\
e_4 &= \begin{pmatrix} 1 & 4 & 1 & 4 & 1 & 3 & 1 & 3 & 4 & 4 \end{pmatrix}.
\end{aligned}
$$

Observe that for the actual program of Figure 2, Tests 2, 3, and 4 all detect errors, although Test 1 does not. We will construct an alternative weight vector (program with the same control structure), which is incorrect but for which none of the four tests detect an error, following the proof of Theorem 2.

Note that $e_3 = 2e_1 - e_2$ and $e_4 = 3e_1 - 2e_2$, so $\{v_1 = e_1, v_2 = e_2\}$ is a basis for the given tests. An additional edge vector which extends our set to form a basis for the edge space of the graph is $v_3 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$, corresponding to the additional path FALSE. Thus we get

$$
V = \begin{bmatrix} 1 & 2 & 1 & 2 & 1 & 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.
$$

Columns 1, 2, and 5 form a basis for the column set of this matrix, so restricting V to those columns we get $U = \begin{bmatrix} 1 & 2 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}$. We select $r = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, Then $x = U^{-1}r = \begin{bmatrix} 0 & 0 & 1 \\ 1 & -1 & 0 \\ -1 & 2 & -1 \end{bmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$. Since 1, 0, and -1 are all integers, $g = 1$. Since the three columns of U correspond to columns 1, 2, and 5 of V, respectively, we get $w = \begin{pmatrix} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$. As a quick check, note that for each of the four original tests the first and fifth column have the same value (1), while for the new test the first and fifth columns have a different value (1 and 0, respectively). Thus if we use w as a weight vector for the same program structure as described in Figure 2, we have an incorrect program that has correct behavior for all the tests



Edge numbers on left, weights on right

Figure 3: Constructed flow graph

in the original set. Figure 3 shows the weighted flow graph for that program, and Figure 4 shows the corresponding C code.

2.3 Other criteria

First we review the definitions for various data flow testing criteria (all-c-uses, all-p-uses, and all-du-paths). We then show that the all-du-paths testing criterion does not subsume structured testing, and that structured testing does not subsume either the all-c-uses or the all-p-uses testing criterion. We then conclude that structured testing is independent of all the data flow testing criteria described by Horgan and London [23].

In discussions of data flow testing criteria, a *definition* is a statement assigning

```
extern int d;
main()
{
    d = d + 1;
    while (ReadDecisionValue()) {
        if (ReadDecisionValue())
            d = d + -1;
        else
            d = d + 0;
    }
}
```

Figure 4: Constructed C program

a value to a variable. A *c-use* is a computational statement that uses a variable's value. A *p-use* is a predicate statement that uses a variable's value, and all possible outcome statements of such predicates. The *all-c-uses* criterion specifies that at least one partial path be traversed from each definition of a variable to every c-use of that variable (with no intervening definitions of that variable). The *all-p-uses* criterion is the analogous criterion with p-uses. The *all-du-paths* criterion specifies that every simple partial path be traversed from each definition of a variable to every c-use and every p-use of that variable (with no intervening definitions of that variable).

Theorem 3 *All-du-paths does not subsume structured testing.*

Consider the subclass of our class of programs with graphs having the structure shown in Figure 5, in which the outermost control structure is a loop which begins with a non-predicate statement. Note that each edge of the graph uses and then defines the program variable `d`, since it increments `d` by a constant (which may be zero). To make sure that this data flow property is shared by the C code representation of the flow graph, we place the trivial assignment statement `d = d + 0` before each node (other than the first) that does not correspond to an explicit assignment statement. Also note that `d` does not appear in any predicate decisions — all predicates are driven by user input. Thus the du-paths correspond exactly to pairs of edges that can be executed in direct sequence. Thus we may satisfy the all-du-paths criterion with two tests regardless of the complexity of the graph. All du-paths that lie completely within the scope of the loop may be executed during a single test that executes the loop a sufficient number of times to go through all such paths. That first iterated test also covers the du-path from the entry edge to the first internal edge of the loop, and the du-path from the upward edge of the loop to the exit edge. The only remaining du-path is the one from the entry edge to the exit edge, and this may be covered by a second test that avoids the loop. If the body of the loop contains at least one decision, two tests cannot satisfy the structured testing criterion. □

Corollary 1 *There are arbitrarily complex programs for which all-du-paths coverage is not sufficient to guarantee correctness, but structured testing is.*

Since all-du-paths does not subsume structured testing, by Theorem 2 it is not sufficient to detect errors for programs in the class. Such programs can be arbitrarily complex. □

Figure 6 shows the program of Figure 4 with explicit assignment statements added. The following two tests satisfy the all-du-paths criterion, but do not satisfy the structured testing criterion or detect the error:

Test 1: 5:TRUE, 7:TRUE, 5:TRUE, 7:FALSE, 5:FALSE.
Test 2: 5:FALSE.

Theorem 4 *Structured testing does not subsume all-c-uses*

Figure 7 shows a simple C program for which structured testing does not subsume all-c-uses. The three paths (i => TRUE, j => FALSE), (i => FALSE, j => TRUE), and (i => FALSE, j => FALSE) form a basis for all paths and thus satisfy the structured testing criterion. However they do not cover the c-use (x = 0, y = 1/x), which exposes a division by zero error. □

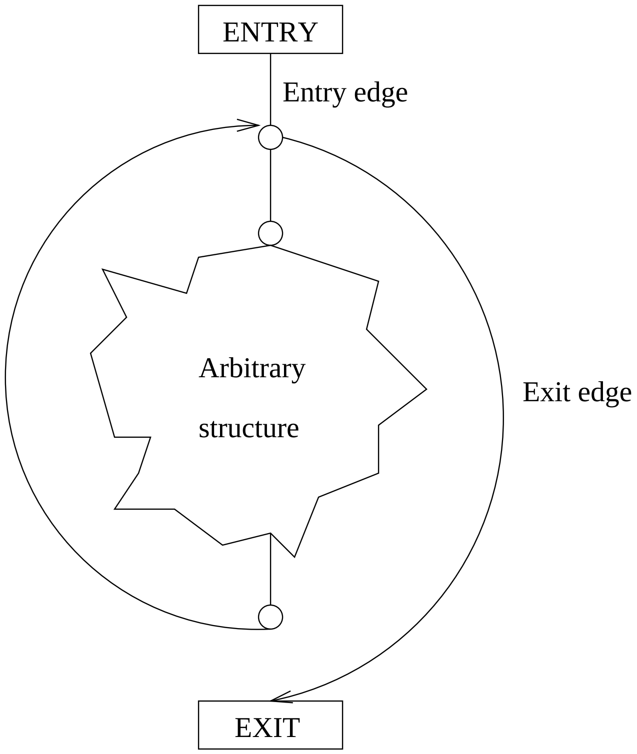


Figure 5: Subclass of graphs

```
extern int d;
main()
{
    d = d + 1;
    while (ReadDecisionValue()) {
        d = d + 0;
        if (ReadDecisionValue())
            d = d + -1;
        else
            d = d + 0;
        d = d + 0;
    }
     d = d + 0;
}
```

Figure 6: C program where two paths can satisfy all-uses

```
extern int i, j;
main()
{
    int x = 1, y = 1;
    if (i)
        x = 0;
    if (j)
        y = 1/x;
}
```

Figure 7: C program for c-uses

```
extern int i, j;
main()
{
    int x = 1, y = 1;
    if (i)
        x = 0;
    if (j)
        if (x >= 0)
            y = 1/x;
}
```

Figure 8: C program for p-uses

Theorem 5 *Structured testing does not subsume all-p-uses*

Figure 8 shows a simple C program for which structured testing does not subsume all-p-uses. The four paths (i => TRUE, j => FALSE), (i => FALSE, j => FALSE), (i => FALSE, j => TRUE, x >= 0 => FALSE), and (i => FALSE, j => TRUE, x >= 0 => TRUE) form a basis for all paths and thus satisfy the structured testing criterion. However they do not cover the p-use (x = 0, x >= 0 => TRUE), which exposes a division by zero error. □

This shows that structured testing is independent of the data flow testing criteria defined by Rapps and Weyuker [50], both in terms of subsumption and in terms of error detection. Since it subsumes branch coverage, structured testing can be placed with respect to the structural and data flow testing criteria in the diagram given by

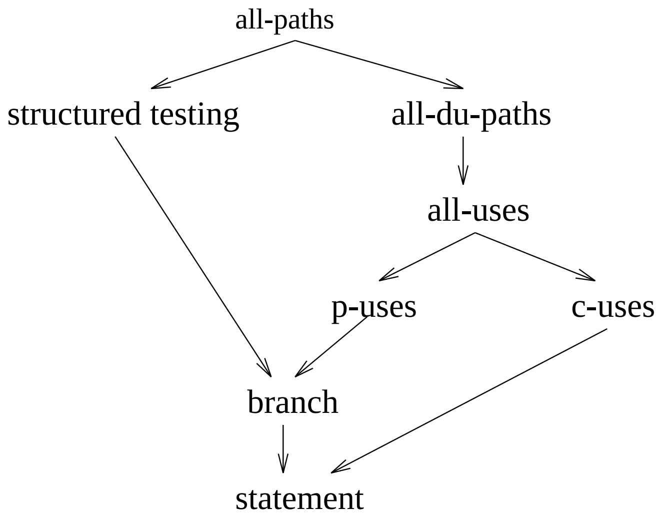


Figure 9: Test criteria hierarchy

Horgan and London [23]. Figure 9 shows the augmented diagram.

2.4 Feasible structured testing

In this section, we discuss the axiomatic approach to the evaluation of testing criteria, then discuss Weyuker's axioms and show that they are satisfied by the feasible structured testing criterion.

2.4.1 Axioms for testing criteria

Since there are so many different testing criteria, it is natural to investigate methods for preferring some over others. Theoretical analysis concerning the subsumption hierarchy finds most pairs of criteria to be incomparable. There are, of course, some exceptions, such as that most structural criteria subsume statement coverage and that all-du-paths data flow coverage subsumes all-uses data flow coverage, but a subsumption relationship is unlikely among the kinds of criteria that might be realistically competing for use on a particular project. Experiments and case studies take considerable effort to perform, so sufficient data may not be readily available for a particular candidate criterion. In addition, although experiments and case studies typically distinguish between criteria, the relevance of a specific empirical data set to a particular real project is unclear frequently. This leaves intuition as the deciding factor in many cases when evaluating testing criteria.

Weyuker's axioms [60] are an attempt to express a set of intuitively desirable properties of testing criteria in a systematic way, so that they can be applied in a uniform and objective manner to evaluate various criteria. While a complete axiomatization in the mathematical sense of any aspect of software development seems unlikely, testing is a good candidate area for a useful set of axioms in the "rule of thumb" sense. Testing has a clear goal: the detection of errors. Although there are many specific testing techniques, they can be classified into a fairly small number of well-understood categories, which can be considered when evaluating potential axioms. Finally, a testing criterion must have some minimal level of demonstrated value to be considered for application, so the axiomatization does not have to exclude all ridiculous potential criteria to be useful.

In contrast, a similar attempt by Weyuker [61] to axiomatize software complexity measurement was refuted by Cherniavsky and Smith [6], who exhibited a ridiculous complexity measure that satisfied all the axioms. The main problem is that software complexity measures are so diverse that any axiom system that excludes "ridiculous" measures seems likely to exclude legitimate ones as well. Aside from the nearly one hundred purely structural measures discussed by Zuse [65], there are measures incorporating operator and operand usage, comment density, length of identifiers, and a nearly limitless set of other potentially counterintuitive software attributes. Also, ridiculous complexity measures are less likely to be weeded out quickly than ridiculous testing criteria, because there is no immediate negative feedback to using a poor complexity measure. While the "ideal" testing method can be defined concisely as one that guarantees detection of all errors, there is no similar ideal for complexity measurement, since complexity measures attempt to quantify many different and often weakly correlated software attributes. Any measurable software attribute that correlates with development time, cost, errors, maintenance effort, comprehension accuracy, or any one of dozens of other significant software development factors, may be legitimately included in a complexity measure. Also, no single complexity measure could have a strong correlation with all such factors, since they are not even strongly correlated with each other.

Weyuker [62] proposed additional axioms to rule out ridiculous testing criteria, giving an example of such a criterion that satisfies the original axiom set. One of the revised axioms is that a testing criterion must require all executable statements to be executed. This requires, in a fairly heavy-handed way, a reasonable minimum level of effectiveness in any criterion that satisfies the revised set of axioms.

Zweben and Gourlay [66] questioned the lack of precision of the axioms and show how ambiguities affect their interpretation, and propose a strengthened version of one of the axioms. Parrish and Zweben [47] reduced the axioms to formal statements, showed dependency relationships between the axioms, explored the role of specifications in testing criteria, and proposed a revised set of seven independent axioms.

2.4.2 Satisfaction of Weyuker's axioms

In this subsection, we review Weyuker's axioms, define the feasible structured testing criterion, and apply the axioms to this criterion. The axioms [60, 62] are:

- Axiom 1, *applicability*: For each program, there is a (finite) test set that is adequate (satisfies the criterion).

- Axiom 2, *non-exhaustive applicability*: There is some program for which a non-exhaustive test set is adequate.

- Axiom 3, *monotonicity*: If a test set is adequate, then all supersets of it are also adequate.

- Axiom 4, *inadequate empty set*: The empty set is not an adequate test set for any program.

- Axiom 5, *anti-extensionality*: There are two functionally equivalent programs and a test set that is adequate for one but not the other.

- Axiom 6, *general multiple change*: There are two programs with the same "shape" and a test set that is adequate for one but not the other. An example of a shape-preserving transformation is replacing one conditional operator in a decision with another.

- Axiom 7, *antidecomposition*: There is a program and an adequate test set such that if a component of the program is made into a separate program and replaced in the original program by a call to the component, then the test set induced on the component is not adequate.

- Axiom 8, *anticomposition*: There are two programs such that the first contains a call to the second, and a test set that is adequate for the first and induces an adequate test set on the second, but that is not adequate for the program obtained by expanding the call to the second program in-line in the first.

- Axiom 9, *renaming*: Consistently renaming the variables in a program does not affect the adequacy of any test set.

- Axiom 10, *complexity*: For any integer $n > 0$, there is a program with a minimal adequate test set of size n.

- Axiom 11, *statement coverage*: For every program, every adequate test set covers all executable statements of the program.

Most white-box testing methods, including all those in Figure 9, fail to satisfy Axiom 1 (applicability) due to the possibility of unexecutable paths through programs [13]. While arguments have been made that programs can and should be modified to make various testing criteria apply during the testing process, it is generally conceded that the costs of such modifications are prohibitive. Since it is undecidable to determine which paths are executable, there is no clear practical way to adapt a criterion to satisfy this axiom. Frankl [13] defined the *feasible data flow testing* criteria by restricting the data flow testing criteria of Rapps and Weyuker [50] to the subset of the program that is executable for some input data, and showed that the criteria satisfied all of Weyuker's axioms [60]. This approach is of mainly theoretical interest, since it is undecidable to determine the executable subset of the program being tested, and thus to determine if a set of tests meets the feasible criterion. As Frankl noted, however, actual test sets consist only of executable paths, so testing according to a particular absolute criterion may be viewed as an attempt to approximate its feasible counterpart rather than to approximate the criterion itself.

Following Frankl's approach, we define the *feasible structured testing* criterion as requiring that we test a basis for all executable paths. The minimum number of tests required is the rank in the program edge space of all executable paths, so it is less than or equal to the cyclomatic complexity, the number of tests required by ordinary structured testing. Thus a finite number of tests is sufficient to test any program using this criterion, so applicability is satisfied. We now show that the other axioms are also satisfied, including one which is not satisfied by ordinary structured testing.

Theorem 6 *Feasible structured testing satisfies all of Weyuker's axioms.*

Axiom 1 is satisfied by feasible (but not ordinary) structured testing, as discussed above.

Axiom 2, that some program has a non-exhaustive test set, is a non-controversial but relatively weak axiom, since it essentially only excludes one specific testing criterion (exhaustive testing). Both ordinary and feasible structured testing satisfy this criterion, for example since a program with two sequential decisions requires only three out of four possible paths to be exercised to satisfy either criterion.

Axiom 3, that all supersets of an adequate test set are adequate, is a fairly strong axiom in the sense of being able to exclude a wide range of criteria. It is also clearly desirable. Error detection is the motivation for testing criteria, and although hidden state information can cause inconsistent error observation, it is reasonable to expect all supersets of an error-revealing test set to reveal at least the same errors. Both ordinary and feasible structured testing satisfy this criterion, since adding more tests to a set can never decrease its matrix rank.

Axiom 4, that the empty set can never be adequate, is stronger than Axiom 2. It is also obviously desirable, ruling out criteria that could let programs pass without any testing at all. Note that the implicit testing criteria of many software developers actually violate this axiom, at least at the level of individual modules, since their testing effort is only targeted at selected modules while others may not be executed until some rare condition occurs after the system has been delivered. Hence, this axiom underscores the point that even the best testing methods are only useful when they are actually used. Since there is always at least one feasible path to a program, both ordinary and feasible structured testing satisfy this criterion.

Axiom 5, that the adequacy of a test set cannot be determined by functionality alone, excludes all purely black-box testing criteria. In the context of white-box testing criteria, however it is quite weak, merely requiring that some non-trivial use be made of the software implementation. Both ordinary and feasible structured testing satisfy this criterion, since, for example, a program that adds one to its input could be either coded with

```
return input + 1;
```

requiring one test or

```
if (input % 2)
    return input + 1;
else
    return input + 1;
```

requiring two tests.

Axiom 6, that the adequacy of a test set cannot be determined by “shape” alone, is a fairly strong axiom, continuing in the spirit of Axiom 5 by requiring testing criteria to be sensitive to more details of the program’s implementation. Hybrids of black box testing techniques and white box complexity metrics tend to be excluded by this axiom, for example, criteria requiring merely that n different tests be executed, where n is the number of statements, the cyclomatic complexity, or a similar structural metric. The reason is that all programs with the same shape share a common value for each purely structural metric, and would therefore require the same number of tests. Both ordinary and feasible structured testing satisfy this criterion, for example, replacing `>` with `>=` in

```
if (i > 0)
    foo(i);
```

The test data set { i = 0, i = 1 } satisfies structured testing in the original by exercising both decision outcomes, whereas it only satisfies the "true" outcome in the changed version.

As an aside, consider a "structural" version of Axiom 6 with "test set" replaced by "set of execution paths," which would seem to require that something other than the execution trace be used to determine adequacy. Structured testing would not satisfy this somewhat contrived modified axiom, nor would statement, branch, or all-paths testing, since they all depend entirely on the flow graph structure and the execution trace. Surprisingly, however, the feasible variants of all of those criteria do satisfy the modified axiom, for example by replacing `>=` with `>` in

```
i = n * n;
if (i >= 0)
    foo(i);
else
    bar(i);
```

In the first case, only the "true" outcome of the decision is feasible, so one test suffices, for example "n = 1". In the second case, both decision outcomes are feasible, so a second test "n = 0" is required. The execution trace information is in effect augmented by the functional information that determines which paths are executable, providing enough flexibility to satisfy the axiom. This example indicates that the feasible variants of testing criteria may have unintuitive properties. The feasible variant of a criterion may be more likely to satisfy an existential axiom and less likely to satisfy a universal axiom than its ordinary version.

Axiom 7, that some program can be modularized so that an adequate test set for the original program induces an inadequate test set on one of the component modules, does not seem to be intuitive. It is the negation of the seemingly reasonable property that testing an entire program suffices to test each of its components. However, in practice it is certainly possible to do more thorough testing of an individual software component via stubs and drivers than could be done in integration. Taking a dual view of Axiom 7, it excludes criteria that are unable to test a module in isolation any more thoroughly than they can in integration. From that perspective, Axiom 7 seems more reasonable, at least for the feasible variants of criteria (since integration testing is usually limited in practice by feasibility constraints), although its value is still not clear. This axiom is not satisfied by ordinary structured testing, since in linear algebra when a set of vectors that span a space are projected onto a subspace the projections span the subspace. Feasible structured testing satisfies this axiom, as shown by the programs in Figure 10. The input set { i = 1, i = -1 } satisfies the feasible structured testing criterion for the original program, since it generates the only two feasible paths through the program. However, once the component is separated out from the main program, this same data set generates only one of the two independent feasible paths through the component, so the feasible structured testing criterion is not satisfied for the component.

Zweben and Gourlay [66] have shown Axiom 7 to be a trivial consequence of Axioms 1 and 4, and propose a strengthened version. The strengthened Axiom 7 states that neither of the components used to demonstrate satisfaction of the axiom can be unreachable. Zweben and Gourlay [66] show that the feasible variants of statement, branch, and all-paths criteria all fail to satisfy the strengthened axiom. However, feasible structured testing satisfies the strengthened axiom, and in fact satisfies the even stronger version in which all statements of the original program are reachable, as shown by the programs in Figure 11. The input set { i = 1, i = 0, i = -1 } satisfies the feasible structured testing criterion for the original program, since it generates the only three feasible paths through the program. However, once the component is separated out from the main program, this same data set generates only two of the three independent feasible paths through the component, so the feasible structured testing criterion is not satisfied for the component. Passing zero as the value of `i` to the component would give the remaining feasible path necessary to satisfy structured testing for the component.

Axiom 8, that a test set that is adequate for each component of a program can become inadequate when all subordinate components are expanded in-line, extends

```
original(int i)
{
    if (i < 0)
        i = -i;
    if (i < 0)
        print("Negative");
    else
        print("Nonnegative");
}

main(int i)
{
    if (i < 0)
        i = -i;
    component(i);
}

component(int i)
{
    if (i < 0)
        print("Negative");
    else
        print("Nonnegative");
}
```

Figure 10: Satisfaction of Axiom 7

```
original(int i)
{
    if (i == 0)
        i = 1;
    if (i > 0)
        print("Positive");
    if (i < 0)
        print("Negative");
}

main(int i)
{
    if (i == 0)
        i = 1;
    component(i);
}

component(int i)
{
    if (i > 0)
        print("Positive");
    if (i < 0)
        print("Negative");
}
```

Figure 11: Satisfaction of strengthened Axiom 7

```
main(int i)
{
    if (i > 0)
        print("Positive");
    component(i);
}

component(int i)
{
    if (i < 0)
        print("Negative");
    else
        print("Nonnegative");
}

expanded(int i)
{
    if (i > 0)
        print("Positive");
    if (i < 0)
        print("Negative");
    else
        print("Nonnegative");
}
```

Figure 12: Satisfaction of Axiom 8

the spirit of Axiom 5 by requiring that the interactions between different parts of the same program be taken into account. This reasonable and yet fairly strong requirement excludes simple coverage criteria such as statement and branch coverage, but not all-paths coverage [60]. This axiom is satisfied by both ordinary and feasible structured testing, for example by the programs shown in Figure 12. The input set { i = 1, i = -1 } satisfies both ordinary and feasible structured testing for both the main and component modules of the original program, since it generates the only two paths through each. However, once the component is expanded into the main program, this same data set only generates two of the three linearly independent feasible paths through the program, the remaining path being generated by the input $i = 0$.

Axiom 9, that renaming variables cannot affect the adequacy of any test set, is unusual in the sense that it attempts to limit the type of white-box information that a testing criterion can use. One could imagine a host of similar axioms, prohibiting the use of other textual features such as whitespace and comments in testing criteria to attempt to focus testing on the structural and computation aspects of the algorithms rather than the source code format. Although most commonly accepted testing criteria satisfy this axiom, and some absurd artificial criteria are excluded by it, it should still be considered controversial. For example, poor coding style such as variable names that are too short, too long, or just misleading may lead to a greater likelihood of error. Axiom 9 would exclude criteria (perhaps hybrids of common criteria) that attempt to provide more thorough testing for such software. This axiom is satisfied by both ordinary and feasible structured testing, since variable names have no impact on control flow.

Axiom 10, that every positive integer is the size of a minimal adequate test set for some program, is not as restrictive as it first appears. Since we can construct programs with exactly n possible sets of input data for any $n > 0$, this axiom does not artificially limit the number of tests required by a criterion for any program. Its primary effect is to exclude testing criteria that only require a bounded number of tests without regard to the program being tested. That is both strong and reasonable. A secondary effect is that the number of tests required by a criterion is not allowed to "skip" any values, for example any criterion that always requires an even number of tests is excluded. At first glance this seems like a frivolous restriction, but it is actually quite reasonable. We can construct programs with exactly n possible sets of input data in which each set of data produces a distinct output by executing a distinct statement. In that case, any criterion that subsumes statement coverage (see Axiom 11 below) and does not require duplicate tests will require exactly n tests for those programs. This axiom is satisfied by both ordinary and feasible structured testing, and for ordinary structured testing the "complexity" measure suggested by this axiom's name is the cyclomatic complexity.

Axiom 11, that criteria must require all executable statements to be executed, is a very strong criterion. It captures the common opinion that subsumption of statement coverage is a good indicator of whether a white box testing method is worth further investigation. While it does not exclude testing criteria with ridiculous frills, it does exclude all "purely ridiculous" criteria, since statement coverage has been widely used in practice with at least some degree of success. It seems a bit too restrictive for theoretical elegance, but since essentially all interesting white box criteria satisfy it, its inclusion probably does not reduce the value of the axioms as a set. This axiom is satisfied by both ordinary and feasible structured testing since structured testing subsumes branch coverage which subsumes statement coverage.

Thus feasible structured testing satisfies all the axioms. □

2.5 Conclusion

We have exhibited an infinite class of programs for which structured testing is the ideal testing strategy in the sense that satisfying the structured testing criterion is both necessary and sufficient to determine correctness of programs in the class. We have placed structured testing into a subsumption hierarchy with several other popular testing methods. We have also shown that the structured testing criterion, when restricted to executable paths, satisfies the axioms proposed by Weyuker [60, 62] for good testing criteria, and discussed the implications and some variations of those axioms.

Chapter 3

Automating Structured Testing

Although methods to generate a minimal sufficient set of flow paths to satisfy the structured testing criterion are known and have been automated [31], their use is frequently considered impractical due to the possibility of unrealizable generated paths and the difficulty of generating test data based on specific paths. Indeed, the primary presentation of structured testing [38] recommends a weaker, but more easily enforced approximation: satisfy branch coverage and run a number of tests equal to the cyclomatic complexity. As we will show in Chapter 4, full structured testing can detect errors significantly better than this approximation.

In this chapter, we present our technique for measuring structured testing progress automatically and generating a minimal set of additional test paths to complete structured testing after any arbitrary set of tests have been executed. The technique involves instrumenting code at the source level to write a control flow trace, then analyzing that trace along with the module control flow graphs to determine the current level of testedness, then calculating the minimal set of additional test paths.

The rest of this chapter proceeds as follows. First, we present a brief usage scenario to introduce our technique. Next, we discuss several restrictions on the programs we accept and give an overview of system architecture. Then, we present our techniques for `C` code instrumentation, trace data analysis, and generation of a minimal set of additional test paths. We then discuss efficiency issues, and present an extended example to demonstrate the technique. See Appendix B for the basic definitions of structured testing.

3.1 Usage scenario

Our technique for automating structured testing allows a minimal sufficient set of tests to satisfy the structured testing criterion to be built up either by trial-and-error with no test data generation at all, or rapidly with a minimal amount of test data generation by augmenting previously available data, including a black box test suite. This section gives an overview of the testing process using the automated techniques.

The first step is to prepare an instrumented executable program to test. This can be done using a shell script to invoke the necessary subordinate programs (and the compiler, since the instrumentation is performed at the source level), or each program can be invoked separately as described in Section 3.3. This phase also produces the control flow graphs and information about execution trace events, which are stored for future use.

Next, the tester runs any desired set of tests, usually based on the functional specification of the program being tested, and imports the trace data into the system for analysis. The system can report two different levels of detail. The high-level report shows the cyclomatic complexity and the number of independent tested paths for each module. When the numbers are equal, the structured testing criterion has been satisfied for the module. If the number of independent tested paths is zero, the module was not executed. A typical report line for a module is: "MODULE main VG[1] 3 TVG 2," indicating that two of the desired three independent tests have been executed for module "main." The low-level report shows a minimal sufficient set of paths in terms of node sequences (corresponding to the control flow graph output from the parsing phase) to complete structured testing for each module, such as: "MODULE main: 0 1 5 6 7." Although eliminating the manual cross-reference to the control flow graph would make the low-level report easier to use, deriving data for specific paths is inherently laborious. Therefore, it is best to exhaust the possibilities

[1]VG denotes v(G), the cyclomatic complexity, and TVG denotes tested v(G), the number of independent tested paths.

of the high-level output before examining individual paths.

Based on the high-level (testedness metrics) output, the tester identifies modules for which testing has been weak or nonexistent, and attempts to identify the corresponding areas of functionality and run additional tests in those areas. The tester may continue to iterate this process, isolate hard-to-reach modules with drivers and stubs, or even run some random tests if there is a way to automatically check the behavior of the software being tested. If the criterion is satisfied during this process, there is no need to examine the low-level (basis completion paths) output. If, however, the point of diminishing returns is reached without completing testing, the tester uses the low-level output to derive specific tests for just the paths required to complete testing. Once the new tests have been executed, the high level output can verify that the criterion has been satisfied. The situation in which some paths cannot be executed due to data constraints is illustrated in Section 3.8 and discussed in detail in Section 6.1.2.

3.2 Translation from source code to flow graph

All the analysis required for structured testing is performed on the module flow graph. We use the standard abstract definition of a module [27]: it must be a single-entry, single-exit graph in which the exit is reachable from all nodes that are reachable from the entry. A node in the flow graph corresponds to a basic block in the code. We require a sufficiently faithful translation that run-time trace information can be converted into paths in the flow graph. Two natural constraints arise from this requirement. First, non-modular constructs are not supported. For example, the standard C library function, `longjmp`, which dynamically terminates a set of current module activations, is not supported. Second, cases where the control flow is defined by the implementation rather than the language are not supported. For example, in the C expression, `(a && b)+(c && d)`, the language does not define whether the test to `a` or the test to `c` occurs first, so a given compiler can choose arbitrarily which test to evaluate first, perhaps based on optimization considerations.

We will use the C code in Figure 13 as a running example. The sample code shows

```
LINE  NODE
1     0     int main(int argc, char *argv[])
2           {
3     1         if (argc > 2)
4     2             while (argc--)
5     3,4,5             ;
6     6         return 0;
7           }
```

Figure 13: Source code

a source module, with source line and flow graph node numbers. Figure 14 shows its representation as a flow graph.

3.3 System architecture

The instrumentation is done at the source level, so that the instrumented code is portable and the system does not depend on any particular compiler. Also, the amount of processing performed by the instrumented code is minimal, which should facilitate the support of other languages using the same back end. The instrumented code writes a trace file and all analysis is done by the back end of the system using the trace file and the module flow graphs. Additionally, by analogy to most profiling tools, the system allows files to be instrumented individually and then linked with a small library of trace reporting functions. Indeed, the inserted code amounts to little more than calls to these functions. Figure 15 shows the overall system architecture.

The rest of this subsection gives an overview of the various processes and files of the system. The details of the instrumentation technique are described in subsequent sections. To illustrate the contents of the files, we use the `argc > 2` conditional expression from Figure 13. A completely instrumented version of the code from Figure 13 is shown in Figure 16. The `Trace Events and Source Locations` file written by the parser maps each interesting control flow event to both the control flow graph and the source code. For our example expression, the event file entry of:

```
E 40 48 1 2 5 main
```

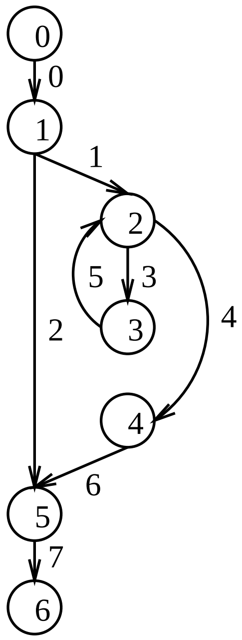


Figure 14: Flow graph

indicates that there is a conditional expression beginning at character 40 of the source file and ending just before character 48 of the source file, with associated control flow graph decision node 1, true outcome node 2, false outcome node 5, and that the expression is part of module `main`.

The language-specific `Event Converter` reads that file and writes the `Instrumentation Code and Locations` file, which gives the specific code to insert at each character location of the source file. For example, for the expression event described above, the two corresponding instrumentation code and locations entries are:

```
40 report_expression((
```

and:

```
48 )!=0,2,5)
```

The language-independent `Code Inserter` simply reads that file along with the source code and performs the insertion to produce the instrumented source file. The event converter is kept separate from the parser for flexibility. The parser can provide information about all potentially instrumentable areas of the source code, while the

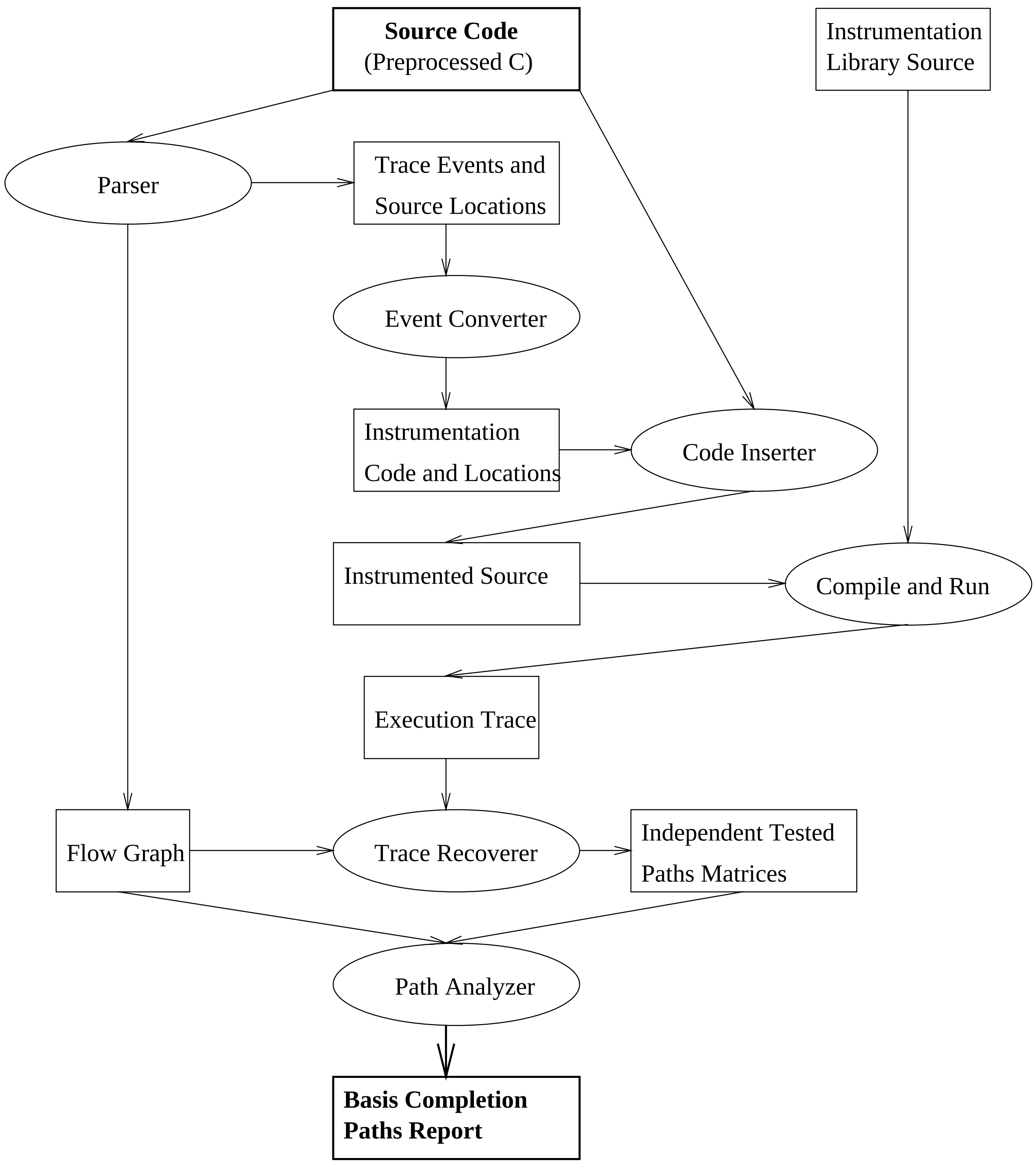


Figure 15: System architecture

```
int main(int argc, char *argv[])
{
    report_activation("main");
{
    if (report_expression((argc > 2)!=0,2,5))
        while (report_expression((argc--)!=0,3,4))
            ;
    return 0;
}
}
```

Figure 16: Instrumented code

event converter can process this file in different ways to implement different instrumentation strategies without requiring parser changes or even reparsing of the source code.

The `Instrumentation Library Source` compiled along with the instrumented source files, implements the functions to write the trace information, such as the `report_expression` function in our sample code. The `Trace Recoverer` program reads the trace file and the flow graph and writes the `Independent Tested Paths Matrices`, a file containing, for each module, the branch incidence vectors of a basis for the set of execution paths represented in the trace file. The number of vectors in a module's matrix is the tested rank. The `Path Analyzer` program reads this matrix and the flow graph file, determines a minimal set of paths to augment the tested paths to form a full basis set through each module (and hence complete structured testing), and produces the `Basis Completion Paths Report`, which is the primary system output.

While the architecture and the entire system implementation were done as part of this work, most of the mechanics such as parsing the source code and inserting the instrumentation code were straightforward. The source level tracing technique for C, although original, produces a generic control flow trace, and the inserted code is similar to Neuder's independently developed branch coverage instrumentation technique [41]. The functionality of the trace recoverer follows fairly naturally from the description of structured testing [37], since it projects each path through a module onto its associated vector and uses Gaussian elimination to select a linearly independent subset for the resultant matrix. It is a practical contribution, however, since by comparing the rank of the set of tested paths through each module with the module's cyclomatic complexity it determines whether the set of tests satisfies the structured testing criterion. Previously, structured testing could only be applied by deriving test cases from the control flow graph, since there was no known technique for determining whether an arbitrary set of tests satisfied the criterion. The path analyzer is the major original component of the system. It selects a minimal set of additional test paths to complete structured testing for each module.

3.3.1 Trace information

A list of node indices in dynamic execution order suffices to allow path recovery, where module entry nodes are annotated with their module identifier. For efficiency, only module entry nodes and targets of decision nodes are reported. These nodes are sufficient to guide a traversal of the flow graph to recover the full path, since any other node is the only successor from each of its predecessors. To simplify the mechanics of instrumenting the source code, we will report a decision target node only if it is encountered as a decision outcome, and omit it if it is encountered unconditionally.

3.4 Instrumentation of C source code

Instrumentation has been around for a long time in various forms. Many developers instrument sections of their code manually to recover useful information during testing and debugging. Most compilers support execution-time profiling at the object code level, and some support statement-count profiling. A few [14] support expression-count profiling. The major advantages of source-level instrumentation are that the instrumented code is portable across compilation environments and that the compiler may be chosen independently of the testing system. A source-level instrumentation system for data flow coverage [22] was used in the study presented in Chapter 4.

Table 1: Instrumentation library functions

Function	Arguments	Location
report_activation	Module name	Module entry
report_incidence	Node index	Case labels
report_expression	Decision value, True node index, False node index	"Wrapped around" decision expressions

We instrument all interesting events by inserting calls to a set of simple C functions (the "instrumentation library") that report the events. Each library function first ensures that the incidence reporting system is initialized, then reports a single event. The source code for this library is given in Appendix A. Table 1 lists the library functions, their arguments, and the location in the code where they are inserted.

Each module entry is instrumented by inserting a function call to report node 0 incidence and the module identifier. This is accomplished by placing the function call directly after the opening brace of the function, and nesting the original function body within braces as a single block statement. Instrumenting calls at the activation site rather than the call site allows the system to trace dynamic calls through function pointers with no special processing. The reason for the additional set of braces is that declarations must precede statements in C.

Each binary decision is projected onto integer 1 or 0 using the `!=` operator and then passed to a function that writes the trace information for the outcome node and returns the 1 or 0 value to preserve functionality.

3.4.1 Instrumentation of "switch"

For the multiway decision statement, `switch`, we use a more general strategy that does not depend upon evaluation of the decision expression. We instrument each statement that could be the target of a `switch` decision to report incidence of the corresponding node in the flow graph. We refer to such nodes as "type I."

```
LINE   NODE
1      0      int main(int argc, char *argv[])
2             {
3                 int i = atoi(argv[1]);
4      1,2        if (i > 4 && i < 10)
5      3              ;
6                 else
7      4,5            ;
8      6          switch (i % 5) {
9      7              case 1: ;
10     8              case 3: ;
11     9                  break;
12     10             case 4: ;
13     11         }
14     12         return 0;
15            }
```

Figure 17: Source code for switch example

As opposed to binary decision target nodes, which are reported only in a conditional context, type I nodes are reported whenever encountered. This distinction is important due to the possibility of "fall-through" from one case to the next. The path recovery algorithm must distinguish these two cases, so the translator annotates the module flow graph to distinguish node type. The pathological case where a node is a target of both a binary and a multiway decision:

```
switch (i) { case 0: if (j) case 1: ; }
```

is not supported by the system currently, although it could be added easily as a special case in the path recovery algorithm.

Figure 17 shows C source code for a module containing a `switch` statement, Figure 18 shows its representation as a flow graph, and Figure 19 shows the instrumented source code. This example also contains a short-circuit logical operator, which is handled by the core method as a binary decision (since it generates unambiguous control flow).

An alternative strategy for instrumenting switch statements does not require type I nodes, but complicates the instrumented code. The idea is to instrument just

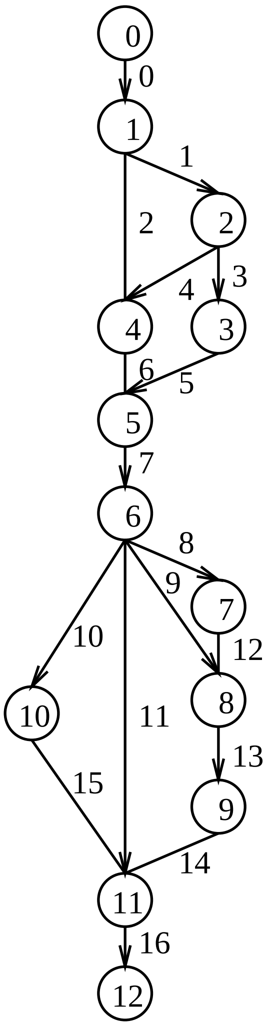


Figure 18: Flow graph for switch example

```
int main(int argc, char *argv[])
{
    report_activation("main");
{
    int i = atoi(argv[1]);
    if ((report_expression((i > 4)!=0,2,4)) &&
        (report_expression((i < 10)!=0,3,4)))
        ;
    else
        ;
{
    switch (i % 5) {
        case 1: {report_incidence(7);;}
        case 3: {report_incidence(8);;}
            break;
        case 4: {report_incidence(10);;}
    }
    report_incidence(11);
}
    return 0;
}
}
```

Figure 19: Instrumented code for switch example

the switch decision outcome edges, not the vertices that may be either targets of switch outcome edges or fall-through edges from a previous case. To do this, two additional mechanisms are required. First, if the switch statement does not contain a default label, a default label must be added during instrumentation to hold the instrumentation code for the default outcome. Second, gotos and labels must be used to "jump over" each case label and the associated instrumentation code, so that the instrumentation code is executed only as a result of the appropriate switch decision outcome and not as a result of fall-through.

3.5 Recovering path information from trace information

Given the trace information obtained by running the instrumented program, the system determines the spaces spanned by the tested paths. This calculation is done on a per-module basis. To perform the calculation the system must:

1. Recover path information from the trace file and separate it by module.
2. Build, for each module, a matrix that represents the executed paths.
3. Compute the rank of the matrices built in Step 2.

Only the first step requires further elaboration. With each module we associate a set of execution paths each given as edge incidence count vectors. When an entry event occurs we begin a new path for the relevant module. During processing of events corresponding to that activation, we traverse the flow graph based on trace information, adding to the path as each edge is traversed. When the activation is recovered completely, the path is added to the set for that module. Nested activations are handled recursively. The paths are kept as the rows of a matrix. Figure 20 shows the main path recovery algorithm, which recovers trace information for a single run of the instrumented program. This algorithm is executed iteratively until all trace information is processed, in order to recover path information for multiple executions of the instrumented program. Type I nodes are always required to match a trace

```
process_activation()
{
    set up for new path in module identified by entry event;
    set current node to module entry node;
    while (TRUE) {
        while (current node has exactly one edge out of it
                and the node along that edge is not type I)
            traverse the single edge out of current node;
        if (current node is exit) {
            add path to module path set;
            return;
        }
        if (no more events)
            return;
        get next event;
        if (entry event)
            process_activation();
        else
            traverse matching edge for event;
    }
}
```

Figure 20: Path recovery algorithm

event, even if the event is not required to select a decision outcome.

3.5.1 Path matrix example

Table 2 shows the input data (as command-line arguments) and the associated trace data (as instrumented node incidence) for three runs of the instrumented code shown in Figure 16. Table 3 shows the corresponding path matrix. Each column represents an edge in the flow graph, and each row represents a path. The rows are marked YES if they are linearly independent of all previous rows, NO otherwise. Note that the tested rank of the module is equal to the number of YES entries in the linear independence column, in this case two. Since the cyclomatic complexity, $v(G)$, for this module is three, we must test another linearly independent path to complete the structured testing of the module.

Table 2: Input data and corresponding trace data

PATH	INPUT DATA	TRACE DATA
1	a a	main:0 2 3 3 3 4
2	a a a	main:0 2 3 3 3 3 4
3	a a a a	main:0 2 3 3 3 3 3 4

Table 3: Path matrix

PATH	EDGE								INDEPENDENT
	0	1	2	3	4	5	6	7	
1	1	1	0	3	1	3	1	1	YES
2	1	1	0	4	1	4	1	1	YES
3	1	1	0	5	1	5	1	1	NO

3.6 Calculating minimal basis completion path set

To complete our basis set of paths, we must take care to generate a set of actual paths when we use linear algebra to fill the matrix to full rank. Therefore the system adopts the following approach: Begin with a complete basis set of actual paths and add one at a time only those which are linearly independent of the current matrix. This method adds exactly a minimal basis completion set of actual paths, which we report as the output of this phase. Figure 21 shows the basis completion algorithm.

```
complete_basis()
{
    fill matrix with tested paths using the path recovery algorithm;
    generate a (static) basis;
    for (each path in basis)
        if (path is linearly independent of matrix)
            add path as a new row of matrix and report it;
}
```

Figure 21: Basis completion algorithm

3.6.1 Generating a basis

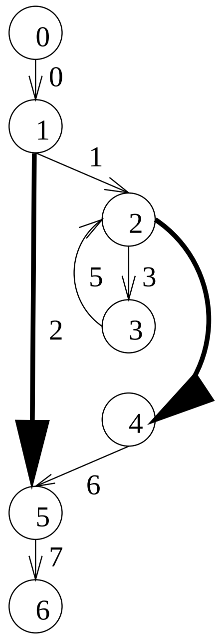
This section describes an algorithm to generate a minimal set of tests to perform structured testing, in the form of a basis set of paths through the control flow graph. A purely algebraic approach to this problem is inadequate, since many vectors in the associated vector space do not correspond to complete paths through the flow graph. For example, if `v1` is a vector corresponding to a path with one iteration of a `while` loop, and `v2` is a vector corresponding to a path with two iterations of the same loop, then `v2 - v1` is a vector that corresponds to the subset of edges on the paths that lie in the loop body, which is not itself a path. Thus, a purely linear-algebraic approach to generating a basis does not produce a useful result. McCabe [38] described a manual technique, known as the Baseline Method, to generate a basis set composed of actual paths from the flow graph, but the technique was not sufficiently detailed for rigorous analysis. The intent is to start with a single "baseline" path, then vary exactly one decision outcome to generate each successive path until all decision outcomes have been varied, at which time a basis of v(G) independent paths will have been generated. Our algorithm, given in Figure 22, is a provably correct variant of the Baseline Method.

In addition to a basis set of paths, which is a basis for the rows of the path/edge matrix in which all possible paths are represented, we can also consider a basis set of edges, which is a basis for all the columns of the same matrix. Since row rank equals column rank, the cyclomatic complexity is the number of edges in such a basis. Our approach is to select a basis set of edges, then use that set to generate each successive path. Consideration of the resulting path/edge matrix restricted to basis columns will show that the set of generated paths is a basis.

We first discuss the steps of our algorithm using the flow graph from Figure 14 as a running example, and then demonstrate the algorithm's correctness. There are two decision outcomes in our example graph. The algorithm identifies the shortest path (in terms of decision nodes) to the exit from these nodes and marks the corresponding decision edges as *non-basis* edges. The shortest paths in Step 1a are shortest in terms of the number of decision outcomes, with ties broken arbitrarily. These paths can be determined efficiently using breadth-first search. Figure 23 shows the example flow

```
    generate_basis()
    {
1       for each decision in the module {
1a          select an outcome with the shortest path
                    to the module exit, and mark it
                    as a non-basis outcome;
        }
2       generate the first path by taking the non-basis
                  outcome for each decision encountered;
3       while not all decision outcomes have been traversed {
3a          select a decision, d, that has had the non-basis
                    outcome traversed along some earlier path, p,
                    but has another outcome that has not yet
                    been traversed;
3b          generate a new path by following path p up to
                      decision d, traversing the new outcome,
                      and then following non-basis decision
                      outcomes to the module exit;
        }
    }
```

Figure 22: Basis path generation algorithm



Thick lines indicate non-basis edges

Figure 23: Flow graph with non-basis edges marked

graph with the `non-basis` edges (2 and 4) marked. We use the term `non-basis` because all other decision outcome edges, along with the module entry edge, form our column basis, and are referred to as `basis` edges. Step 2 of the algorithm generates the first path in the basis. It executes exactly one basis edge, the module entry. The remaining path edges are either not decision outcome edges or are non-basis edges identified in Step 1. For our example, the node sequence of this path is: `0 1 5 6`. For the second path in our example, Step 3a selects the decision at Node 1 along Path 1, and Step 3b traverses the new outcome Edge 1 and then follows the non-basis outcome Edge 4 to the module exit. The node sequence of this path is: `0 1 2 4 5 6`. For the third path, Step 3a selects the decision at Node 2 along Path 2, and Step 3b traverses the new outcome Edge 3 and then follows the non-basis outcome Edge 4 to the module exit. The node sequence of this path is: `0 1 2 3 2 4 5 6`. Since all decision outcomes have been traversed, the algorithm terminates.

To show that our algorithm generates a basis set of paths, we first show that the number of paths is equal to the cyclomatic complexity, and then show that the paths

Table 4: Example of independence of generated paths

PATH	EDGE		
	0	1	3
1	1	0	0
2	1	1	0
3	1	1	1

are all linearly independent. Since each successive path generated by the algorithm traverses exactly one new basis edge, the number of paths is equal to the number of basis edges. This number is equal to the cyclomatic complexity, since the cyclomatic complexity is one greater than the number of decision outcomes minus the number of decisions [57]. To see that the paths are all linearly independent, consider the path/edge matrix with the generated set of paths in order as the rows and the basis edges as the columns, with the columns ordered by the index of the path that first traversed each corresponding basis edge. Table 4 shows this matrix for our example. The matrix is then lower-triangular with all major diagonal entries equal to `1`, so the rows are linearly independent. Thus, our algorithm generates a basis set of paths.

3.6.2 Basis completion example

Table 5 shows the section of the path matrix corresponding to the basis generated by our algorithm for our example program in the previous section. The rows are marked YES if they are linearly independent of all previous rows including the rows of tested paths shown in Table 3, NO otherwise. Note that since the cyclomatic complexity of the sample module is three, there are three paths in the static basis. Since the tested paths from Table 3 have rank two, only one of these additional paths is linearly independent, and the completion set consists of that one path. One output of the system is the completion set as sequences of flow graph nodes, in this case the single path `main:0 1 5 6`. Given the new path, we derive manually the program input data to execute this path. The effort required to perform this derivation depends strongly on the code, and in this case it is easy — executing the program with no

Table 5: Basis completion matrix

PATH	EDGE								INDEPENDENT
	0	1	2	3	4	5	6	7	
1	1	0	1	0	0	0	0	0	YES
2	1	1	0	0	1	0	1	1	NO
3	1	1	0	1	1	1	1	1	NO

command-line arguments is sufficient.

3.7 Efficiency issues

In this section, we consider the amount of memory, disk space, and run time required by the techniques described in this chapter, as well as ways to make them more efficient. The major areas of concern are the matrix size, the trace file size, the path recovery run time, and the basis completion run time. This section gives asymptotic bounds. Actual time and space usage numbers for the extended example of Section 3.8 are shown in Table 10.

3.7.1 Matrix size

Recall that $v(G)$ denotes the cyclomatic complexity of the flow graph. The matrix size is proportional to the number of paths times the number of edges in the graph. Dependent rows can be detected and discarded immediately, giving a maximum matrix size of $v(G)$ times the number of edges. This can be reduced further by observing that the set of columns corresponding to flows out of decision nodes and the entry node span the column space of the graph, so columns corresponding to other edges may be omitted. Table 6 shows the matrix of Table 5 reduced in this manner. The system currently uses this optimization. Figure 24 shows the graphical representation of each path for this reduced matrix. There are between $v(G)$ and $2 * v(G) - 1$ edges in the reduced matrix, giving an upper bound of $2 * v(G) * v(G)$ on the matrix size. A lower bound is $v(G) * v(G)$ since the column rank must equal the row rank of

Table 6: Reduced basis completion matrix

PATH	EDGE				
	0	1	2	3	4
1	1	0	1	0	0
2	1	1	0	0	1
3	1	1	0	1	1

Table 7: Minimal basis completion matrix

PATH	EDGE		
PATH	0	1	3
1	1	0	0
2	1	1	0
3	1	1	1

$v(G)$ once a basis has been inserted. This bound can be attained using a technique developed by Ball and Larus [2]: First construct a spanning tree of an undirected version of the flow graph, then let the matrix rows correspond to the edges that are not in the tree. Table 7 shows a maximally reduced version of Table 5. Figure 25 shows the graphical representation of each path for this matrix.

3.7.2 Trace file size

The size of the trace file is proportional to the number of module activations plus decision targets encountered during execution, which is unbounded in the size of the flow graph. However, the path recovery algorithm may be run incrementally. In the incremental case, the only unbounded behavior is proportional to the maximum cyclomatic complexity of a module times the deepest nested activation during execution, since each activation keeps a partial instrumented edge incidence count vector for the current path. The activation nesting level may be reduced by instrumenting subsets of modules individually and recovering paths through them separately after running on the same data. This technique can reduce the nesting level by a factor of

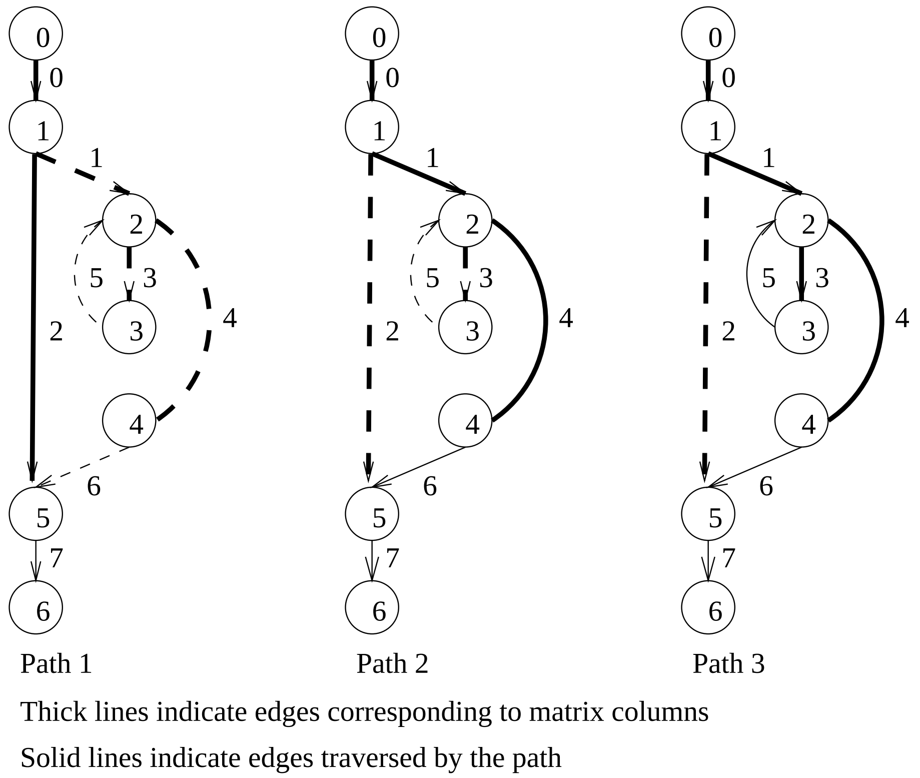


Figure 24: Graphs for reduced basis completion matrix

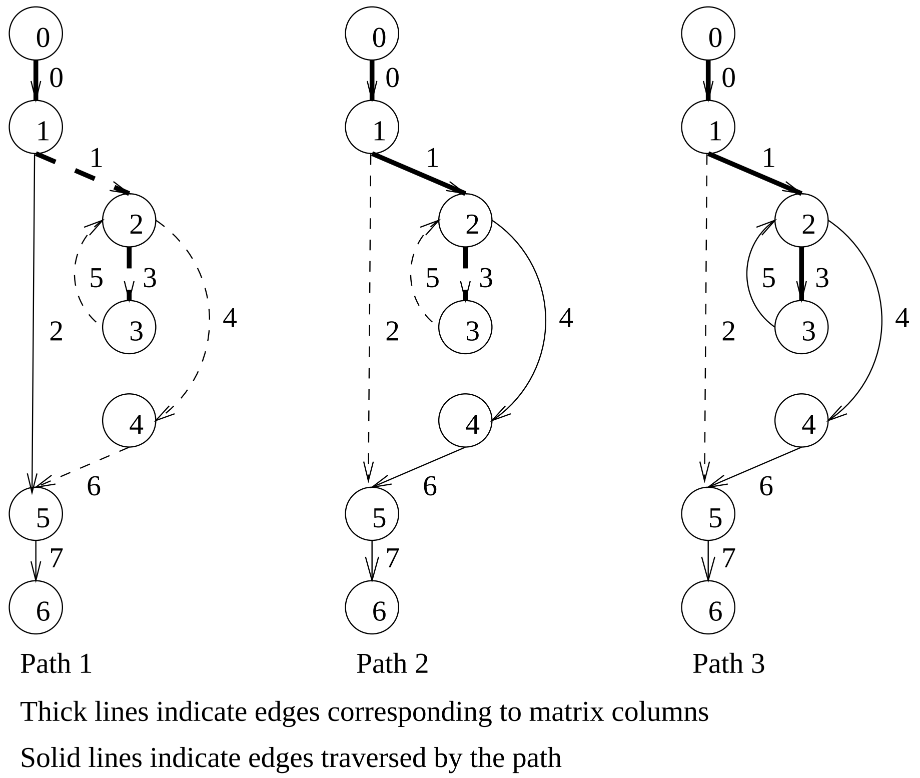


Figure 25: Graphs for minimal basis completion matrix

the number of modules being instrumented, and is especially useful if there are many indirectly recursive modules.

3.7.3 Path recovery run time

Entry events require locating the module by identifier (binary search). This can be improved to constant time by adding a link phase to use an index into an array of flow graphs as a module identifier. Decision events require traversing all edges between the last event and the decision node, which is bounded above by the number of edges. This can be improved by precalculating the next decision node for each decision outcome node in the graph. This can be done during a graph traversal, so it adds an overhead equal to the total number of edges in all instrumented modules. Then decision events take constant time and we get:

$$Time = O(edges + events).$$

This is asymptotically optimal, being linear in the number of events and the module sizes.

3.7.4 Basis completion run time

For each module there are some number of executed paths through the module, not bounded by the size of the graph. There are also the $v(G)$ basis paths. Each must be tested for linear independence against the matrix of size $2 * v(G) * v(G)$, and if the matrix is stored in row-reduced form this test can be done in time proportional to the size of the matrix. Thus we get a bound of:

$$Time = O((executed\_paths + v(G)) * v(G) * v(G)).$$

3.8 Extended example

We applied this system to the corrected, "fast" version of the common words program of Hanson [17]. The code for this program can be found in Appendix A. The program reads a list of words from standard input and writes a list of the most common words in the input and their frequencies to standard output, in decreasing order of frequency. A word is defined to be a sequence of upper and lower case letters of length at most 100, and the excess characters of longer words are discarded. For example, "don't" is considered two words, "don" and "t". The program optionally uses a command-line argument or an environment variable to determine the number of words to report.

As a black-box test suite we split the file "/usr/dict/words" (a list of about 25,000 words commonly used to check spelling) into 26 files, each consisting of lines beginning with a particular letter. During black-box testing we used the program as a pure filter, not overriding its default setting for number of words.

First we ran the black-box test suite in alphabetical order. Of the 26 files, only five (`a`, `b`, `d`, `e`, and `h`) added to the tested rank of the code. Thus, in terms of the structured testing criterion, these five files are of equivalent testing power to the original suite. In situations where it is desirable to limit the effort spent on testing, the system can be used in this way to extract a minimal subset of a large test suite without sacrificing independent path coverage.

After examining the set of untested basis paths produced by the system for each function, we constructed a white-box augmentation suite consisting of five tests. Three of the tests use the file `a` from the black box suite as program input, but vary other conditions: The first overrides the number of words via command-line argument, the second overrides the number of words via an environment variable, and the third alters the size of the internal hash table used by the program. The fourth test uses an input file with a 101-letter word, and the fifth uses an input file consisting of a single non-alphabetic character. Each of the five white-box tests increased the tested rank of at least one of the modules. Table 8 shows the complexity of each function, its tested rank after running the black box tests, and its final tested rank after running both the black box and white box tests.

The final tested rank for the `printwords` function is three less than its complexity due to three control dependencies between its decisions. Figure 26 shows the code for `printwords`. The first two dependencies are due to the double traversal of the hash table. The loops on Lines 7 and 12 are always executed the same number of times,

Table 8: Test coverage

MODULE	COMPLEXITY	BLACK BOX RANK	FINAL RANK
main	5	2	5
addword	6	6	6
getword	7	5	7
printwords	11	5	8

as are the loops on Lines 8 and 13. The third dependency is due to the extraneous test of `k-- > 0` on Line 19. This test is always true, since it is only executed if the equivalent `k > 0` test on Line 18 is true. Figure 27 shows the results of restructuring `printwords` to remove these dependencies. The first loop through the hash table on Lines 6 through 10 are moved to a new module `getmax`, and the extraneous test on Line 19 is replaced with an equivalent decrement of `k` in the initialization area of the loop on the next Line.

The restructured version of the common words program has no control dependencies. As a side benefit, restructuring the program lowered the complexity of `printwords` to below ten, a commonly used threshold for module acceptance [54]. We tested the restructured program using the black-box and white-box test suites constructed for the original. Of the black-box tests, only four (`a`, `b`, `d`, and `e`) increased rank. Test `h` from the black-box suite is unnecessary because the decision `wp->count > max` on Line 9 of Figure 26 has been moved to a different module from the decision `wp = list[i]` on Line 19, so the two decisions no longer need to be tested independently. Table 9 shows the complexity and tested rank information for the restructured program. Note that after running the black box and white box suites, full tested rank was attained for all modules, satisfying the structured testing criterion.

Table 10 shows the run time (including path recovery and all matrix calculations) and file sizes for the restructured code executing the full black-box test suite on a DEC 5100. Although the trace file was processed and removed after each test run, the total throughput is given for reference. The combination of large trace files and a small

```
printwords(k)
int k;
{
    int i, max;
    struct word *wp, **list, *q;
    max = 0;
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = wp->next)
            if (wp->count > max)
                max = wp->count;
    list = (struct word **) alloc(max + 1, sizeof wp);
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = q) {
            q = wp->next;
            wp->next = list[wp->count];
            list[wp->count] = wp;
            }
    for (i = max; i >= 0 && k > 0; i--)
        if ((wp = list[i]) && k-- > 0)
            for ( ; wp; wp = wp->next)
                printf("%d %s\n", wp->count, wp->word);
}
```

Figure 26: Original code for “printwords”

Table 9: Restructured test coverage

MODULE	COMPLEXITY	BLACK BOX RANK	FINAL RANK
main	5	2	5
addword	6	6	6
getword	7	5	7
printwords	7	4	7
getmax	4	3	4

```
int getmax()
{
    int i, max = 0;
    struct word *wp;
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = wp->next)
            if (wp->count > max)
                max = wp->count;
    return max;
}
```

```
printwords(k)
int k;
{
    int i, max;
    struct word *wp, **list, *q;
    max = getmax();
    list = (struct word **) alloc(max + 1, sizeof wp);
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = q) {
            q = wp->next;
            wp->next = list[wp->count];
            list[wp->count] = wp;
            }
    for (i = max; i >= 0 && k > 0; i--)
        if (wp = list[i])
            for ( k--; wp; wp = wp->next)
                printf("%d %s\n", wp->count, wp->word);
}
```

Figure 27: Restructured code for “printwords”

Table 10: Run time and file sizes for full test suite

Run time	Total trace throughput	Max trace size	Final matrix size
90 seconds	3.5 Mbytes	302 Kbytes	1,825 bytes

matrix file is due to the fact that long paths were being executed through functions with low complexity. Since functions with extremely long execution paths generate correspondingly long temporary trace files, having separate trace and analysis phases may be inappropriate. In that case, the conversion of trace data to matrix vectors and the linear algebra on those vectors can be done on-line while the instrumented code is running, eliminating the unboundedness of the trace file at the cost of slowing down the instrumented application. Further efficiency considerations are discussed in Section 6.1.3.

3.9 Conclusion

The practical impact of structured testing has been severely limited due to the difficulty involved in performing it manually. The automated approach presented in this chapter removes that obstacle and allows structured testing to be incorporated smoothly into the testing effort. The technique involves instrumenting code at the source level to write a control flow trace, then analyzing that trace along with the module control flow graphs to determine the space spanned by the tested paths, and then producing a minimal set of additional paths sufficient to complete structured testing. The source-level instrumentation method is portable, allowing use of this technique even on target architectures with little or no native code analysis support. Initial use shows the system can help supplement other testing techniques interactively to achieve the structured testing “basis path coverage” criterion without requiring wasteful duplication of effort on the part of the tester. The system described in this chapter also enables empirical evaluation of structured testing, and was used in the study of Chapter 4.

Chapter 4

Empirical Evaluation of Structured Testing

Placing testing criteria in a subsumption hierarchy does not necessarily determine their relative merit in practice. The important qualities of a testing criterion are effort and payoff, which can be approximated by the number of tests required to satisfy the criterion and the probability of detecting an error using the criterion, respectively. This chapter uses two metrics for each quality to compare structured testing with all-uses data flow coverage and branch coverage. To measure the structured testing and branch coverage criteria, we used the system described in Chapter 3. To measure the all-uses criterion, we used ATAC [22] version 3.3.13, treating all-uses as the union of block, decision, c-uses, and p-uses. We performed the comparison using multiple random trials on test programs with known errors, and therefore measured the relative power of the testing criteria themselves rather than the practical effectiveness of the specific tools and techniques in a production environment.

4.1 Comparison methodology

We selected nine test programs and random input distributions, described in detail in Section 4.3. The code for the test programs is given in Appendix A. For each program, we determined the maximum feasible coverage level for each of the testing criteria. We set the coverage targets for each experimental trial to these maximum feasible levels. Specific instances of infeasibility are noted in Section 4.3. Each trial consisted of executing random test cases until the maximum feasible coverage level was obtained. There was no fixed upper bound on the number of test cases generated for each trial. We executed a total of 100 independent trials for each program/criterion pair.

Four kinds of data were collected for each trial: number of test cases, number of test cases that increased coverage, whether an error was detected, and whether an error was detected by a test that increased coverage.

The number of test cases measures the total number of random tests required to satisfy the criterion. This approximates the effort necessary to select a set of tests that satisfies the criterion. In situations where verifying correct program behavior is cheap, it also approximates the effort necessary to apply the criterion for testing, since the criterion can be used as a stopping rule for testing while verifying program behavior on each test case.

The number of test cases that increased coverage measures the size of a non-redundant random set of tests that satisfies the criterion. This approximates testing effort needed in situations where verifying correct output is expensive, since the criterion can be used to verify program behavior on only the reduced set of test cases that increased coverage.

Whether an error was detected measures whether any random test detected an error before the criterion was satisfied. This approximates testing payoff in situations where verifying correct program behavior is cheap, by applying the criterion purely as a stopping rule for testing. One potential concern with this measure is that errors can often be detected merely by running a large number of random tests. Thus, a criterion that tends to be satisfied only by a large number of random tests may appear effective by this measure even if the tests that actually contribute to satisfying the criterion are unlikely to detect errors. Hutchins *et al.* [26] explored this effect in a study of moderate-size C programs with seeded errors and randomly selected human-generated test cases. The study showed that for both the DU coverage [26] (similar to all-uses) and edge coverage [26] (similar to branch coverage) criteria, coverage levels above 90% showed significantly better error detection than randomly chosen test sets of the same size. Since we require 100% feasible coverage, the measured error detection effectiveness of both all-uses and branch coverage is due primarily to the criteria themselves, rather than to incidental random tests. The same is true for structured testing, since, as reported later in this chapter, fewer random test sets were required to satisfy structured testing than all-uses for approximately the same error detection effectiveness.

Whether an error was detected by a test that increased coverage measures the error-detection effectiveness of a non-redundant random set of tests that satisfies the criterion. This approximates testing payoff in situations where verifying correct output is expensive, by verifying program behavior on only the reduced set of test cases that increased coverage.

In Section 4.4, we describe the relative performance of each of the testing criteria with respect to the above four data points, both averaged over all test programs and for each test program individually.

4.2 Experimental Design

We packaged each test program with a driver that takes a random number generator seed as input, generates a random test case from the input distribution using that seed, runs the test, and returns an error code to indicate whether the test detected an error. The error detection was done either by comparing program behavior with a corrected version, or by checking an assertion about program behavior. Since the error detection method was applied consistently across the testing criteria being evaluated, the experimental validity does not rely on the correctness of the error detection method. Nevertheless, significant effort went into deriving accurate error detection procedures.

To determine the maximum feasible coverage for each test program, we first ran a thousand random test cases from the experimental distribution, using the appropriate coverage tool for each criterion to measure the coverage. Using that level of coverage as a working hypothesis, we examined the code for the test program and detailed tool outputs (untested paths, all-uses, branches) to help prove that no further coverage was possible. Section 4.3 describes the results of this process for each program. In all cases the initial thousand tests were sufficient to attain full feasible coverage.

We used a high-level driver program to invoke the individual program test drivers, passing in a different random seed for each test case. The coverage type (structured, all-uses, or branch) was passed as a parameter. Each experiment consisted of 100 trials, each of which involved running the individual program test driver until the selected coverage type was satisfied. Figure 28 shows the flow diagram for the procedure used to capture the data described in Section 4.1 for each trial. The high-level driver invoked the coverage tool after each test case, comparing the coverage with the maximum feasible coverage target. The high-level driver gave the same random seed sequence to the trials for each type of coverage, even though the individual trials were of varying lengths. Invocation of the high-level driver to perform an experiment for each (test program, coverage criterion) pair was accomplished using shell scripts.

4.3 Test programs

The nine test programs used to compare the testing criteria are all C functions with "real" faults, except for one (triangle classification) with a published injected fault. The programs are: `Blackjack`, `Cobol`, `Comment`, `Find`, `Position`, `Sort`, `Stat`, `Strmatch`, and `Triangle`. Code for `Find` is shown in Figure 29, and code for the remaining programs may be found in Appendix A. Table 11 summarizes each program.

`Blackjack` plays a hand of blackjack and determines the outcome [38]. We translated it to C from the FORTRAN original for this experiment. It contains ten decision statements, with two loops created with goto statements. The fault is an incorrect decision. For the input distribution, cards are dealt uniformly and a simulated naive player makes randomized decisions according to simple rules. There is one exception to this distribution: each trial is started with two test cases (a "push," in which both the dealer and the player are dealt blackjack, and a five-card win for the player) that would take an excessive number of random tests to generate — neither test case

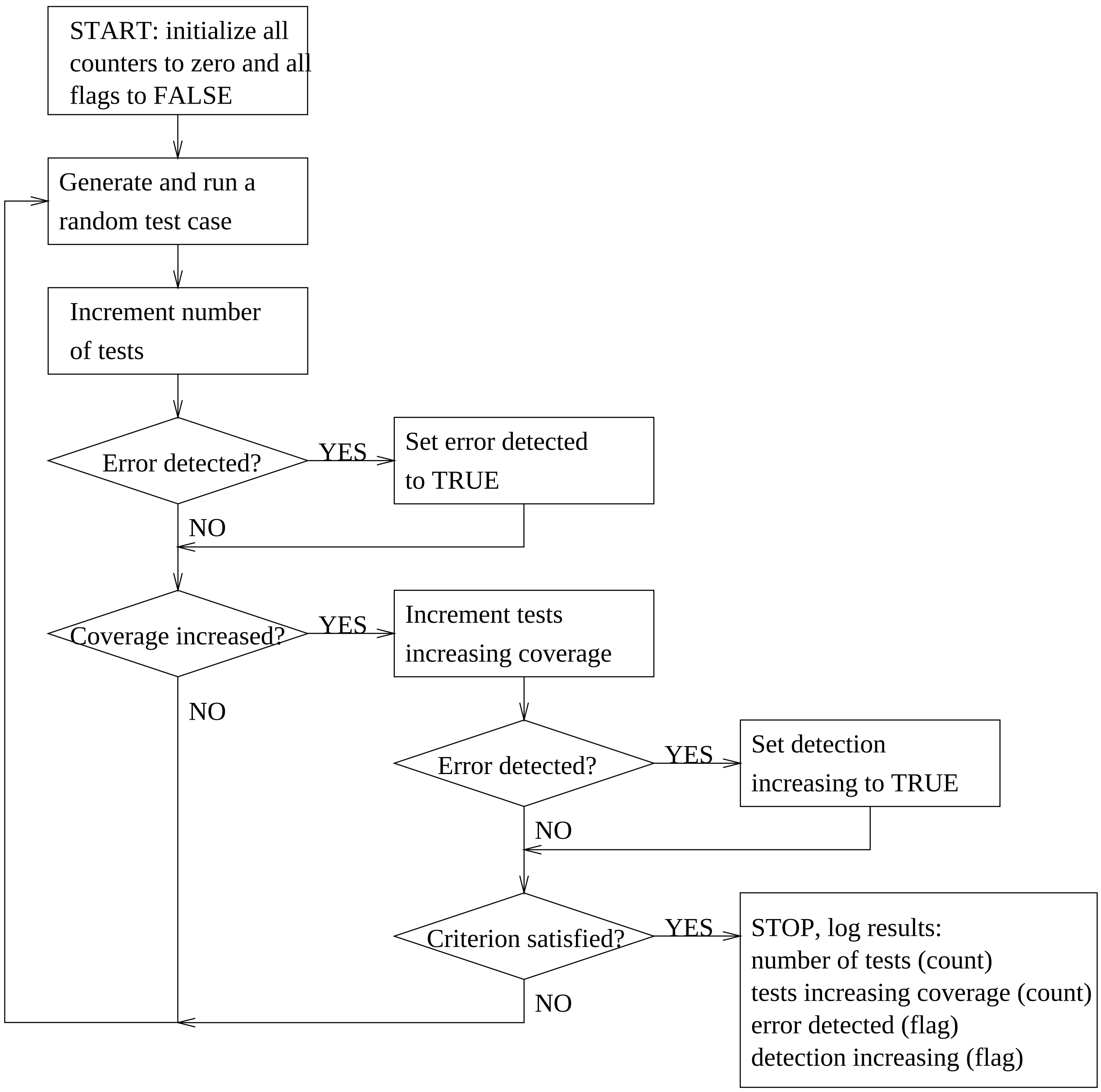


Figure 28: Test procedure for each trial

Table 11: Test programs

PROGRAM	DESCRIPTION	FAULT
Blackjack [38]	Plays blackjack	Incorrect decision
Cobol [33]	Formats COBOL strings	Missing decision
Comment [30]	Parses C comments	Extraneous decision
Find [28]	Permutes an array	Incorrect logic
Position [28]	Searches an array	Incorrect comparison
Sort [28]	Sorts	Missing assignment
Stat [28]	Computes statistics	Incorrect initialization
Strmatch [28]	Matches patterns	Incorrect logic
Triangle [9]	Classifies triangles	Missing predicate

involves the fault.

`Cobol` is from a version of the McCabe & Associates COBOL analysis product [33]. It formats a character string as a COBOL expression, and returns it as a vector of lines with continuation characters and extra quotation marks and spaces as necessary. It has two loops and ten decisions. The distribution is such that the input is a sequence of identifiers and COBOL strings (50% each) separated by one or two spaces (50%), alternating until either we choose to stop (75%) or the length exceeds 200 characters. The identifiers are uniformly between 1 and 30 characters long. The strings are uniformly between 0 and 130 characters long, enclosed in either single or double (50%) quotes. The characters inside the strings are letters (60%), spaces (20%), embedded (doubled) quotes (10%) or the non-active type of quote (10%). The fault is a missing decision.

`Comment` is from a version of the McCabe & Associates C analysis product [30]. It skips past C comments in the input stream, assuming the initial "/*" has already been processed. It has one loop and five decisions. The distribution consists of 20-character strings with the letters being alphabetic (50%), a '/' (40%), or a '*' (10%). The fault is an extraneous decision.

`Find`, the inner loop of quicksort [21], takes an array and an index, and permutes the array so that elements to the right of the index are greater than or equal to the indexed element, and elements to the left are smaller than or equal to the indexed element. The implementation studied was taken from Mathur and Wong [28]. It has three loops and five decisions. There are multiple faults involving incorrect logic, but Mathur and Wong do not specify the faults or the inputs that cause them to be detected. The distribution is that of Experiment A from Mathur and Wong [28]: array size uniformly `-5` to 10, array elements uniformly `-10` to 100, and if array size `>` 0 then index uniformly 1 to array size, otherwise index uniformly `-3` to 2.

`Position` [28] takes an array and a value, and sums the array elements until the sum is at least the value, returning the index of the element that caused the sum to reach or exceed the value, or 0 if no such element exists. It has one loop, and one decision that breaks out of the loop. The fault is an incorrect comparison operator and related calculation. The distribution is that of Experiment H [28]: array size = 5, array elements uniformly selected from { 0, 1, 2, 3, 4, 5, 6, `-5`, 8 }, value uniformly selected from { `-5`, 0, 1, 4, 8 }.

`Sort` [28] sorts an array in descending order. It has two loops and one decision. The fault is a missing assignment. The distribution is that of Experiment B [28]: array size uniformly `-1` to 10, array elements uniformly `-1` to 100.

`Stat` [28] computes the sum, minimum, and maximum of an array. It has two loops and two decisions. The fault is an incorrect initialization. The distribution is that of Experiment I [28]: array size = 5, array elements uniformly from `-20` to 20.

`Strmatch` [28] returns the position of the first occurrence of a pattern in some text, or 0 if the pattern does not occur. It has one loop and three decisions. There are multiple faults due to incorrect logic. Although Mathur and Wong do not specify the faults, they state that the faults might be detected if any of the following occurs: (1) the text has more than 10 characters, (2) the pattern has more than three characters, or (3) the text and pattern are both empty [28]. The distribution is that of Experiment E [28]: text length uniformly 0 to 12, pattern length uniformly 0 to 4, text elements and pattern elements both uniformly from { a, b, `#` }.

`Triangle`, from DeMillo *et al.* [9], classifies triangles based on the lengths of their sides. We translated it into C with just the harder to detect of the two faults intact, a missing decision predicate. It has no loops and five nested decisions. The distribution assigns the length of each side uniformly from 1 to 10, with the exception that each

Table 12: Total and feasible testable attributes

PROGRAM	Structured		All-uses		Branch	
	Total	Feasible	Total	Feasible	Total	Feasible
Blackjack	11	11	46	42	21	21
Cobol	13	12	146	136	25	25
Comment	7	4	31	31	13	13
Find	10	8	113	95	19	18
Position	3	3	17	17	5	5
Sort	4	4	37	36	7	7
Stat	5	3	45	44	9	9
Strmatch	5	4	53	50	9	9
Triangle	6	6	24	24	11	11

trial is started with a right triangle, which takes an excessive number of random tests to generate and does not involve the fault.

4.3.1 Feasibility constraints

Table 12 shows the total statically determined number and the maximum feasible number of structured testing paths, all-uses, and branches for each test program. Note that for only two of the test programs (`Position` and `Triangle`) were all attributes of each criterion feasible. We will use the `Find` program as an example of verifying the maximum feasible value of each criterion, since it has infeasible attributes for each criterion. Figure 29 shows the code for the `find` routine with selected line numbers marked. We first account for the one unexecutable branch in `find`, which is responsible for only 18 of the 19 branches being feasible.

Proposition 1 *In the decision on Line 8 of* `find`*,* `b` *is never evaluated as true.*

Note that due to the short-circuit `||` operator, `b` is only evaluated on Line 8 when `m < ns` is false. Since `b` is initialized to false, it is false the first time through the decision, which terminates the program if `b` is actually evaluated. Otherwise, at least one iteration of the outermost loop must be performed. In that case, consider the

```
void find(int n, int f)
{
    int m, ns, i, j, w;
    BOOLEAN b;
    b = false;
    m = 1;
    ns = n;
    while ((m < ns) || b) {
        if (!b) {
            i = m;
            j = ns;
        } else
            b = false;
        if (i > j) {
            if (f > j) {
                if (i > f)
                    m = ns;
                else
                    m = i;
            } else
                ns = j;
        } else {
            while (a[i] < a[f])
                i = i + 1;
            while (a[f] < a[j])
                j = j - 1;
            if (i <= j) {
                w = a[i];
                a[i] = a[j];
                a[j] = w;
                i = i + 1;
                j = j - 1;
            }
            b = true;
        }
    }
}
```

Figure 29: Code for `find`

first time that `b` is evaluated at Line 8. We know that `m < ns` must have been true on the previous iteration, but false on the current one, which means that the value of `m` or `ns` must have changed, which means that the decision at Line 14 must have been true on the previous iteration. Since `b` is always false at Line 14 and is not assigned in the true outcome of the decision at Line 14, it must therefore be false, which again terminates the program. □

The one infeasible branch is a linear control dependency, since its execution frequency is zero along every feasible path. This accounts for one infeasible structured testing path. We next identify the other linear dependency between decision outcomes. Together, the two dependencies are responsible for only 8 of the 10 structured testing paths being feasible.

Proposition 2 *The false outcome of the decision (*`!b`*) at Line 9 is executed the same number of times as the false outcome of the decision (*`i > j`*) at Line 14.*

The first time through the loop, `!b` is true at Line 9. The last time through the loop, `i > j` is true at Line 14 as shown in the proof of Proposition 1. At other times, `i > j` true at Line 14 on one iteration forces `!b` to be true on the next, since `b` is always false at Line 14 and only `i > j` false at Line 14 sets `b` to true. □

There are 18 infeasible all-uses. We present a representative c-use and p-use as examples. We first account for one of the three infeasible c-uses.

Proposition 3 *The c-use (*`m`*, def at Line 17, used at Line 10) is infeasible.*

Assume that `m` is set to `ns` at Line 17. Then, since neither is changed until the next loop test at Line 8, the test `m < ns` at Line 8 is false. Note that `b` is always false at Line 14 and also at Line 17, and is not changed between Line 17 and the next loop test at Line 8. As a result, the test of `b` at Line 8 is also false, and the loop exits without the possibility of execution reaching Line 10 for the c-use of `m`. □

Finally, we account for one of the 15 infeasible p-uses.

Table 13: Average experimental data

Data	Structured	All-uses	Branch
Test cases	25	34	22
Test cases increasing	6.1	5.5	3.5
Percent detections	67%	68%	54%
Percent detections increasing	63%	57%	45%

Proposition 4 *The p-use (`m`, def at Line 17, used at Line 8, value TRUE) is infeasible.*

As with the above c-use, once `m` is set to `ns` at Line 17, both are unchanged until the test `m < ns` at Line 8, which must therefore be FALSE. □

4.4 Experimental Results

Table 13 lists the average data for all nine experiments. It shows for each criterion the number of test cases and test cases increasing coverage averaged over each trial, and the percentage of experimental runs detecting an error and detecting an error while increasing coverage. For both test effort measures, all three criteria were within a factor of two of each other. All-uses required the most random test cases to satisfy, with structured testing second, and branch coverage a close third. Structured testing had the most test cases increasing coverage, with all-uses second, and branch coverage third. For the percentage of trials detecting an error, all-uses had the highest percentage, with structured testing a close second, and branch coverage a distant third. For the percentage of trials that detected an error while increasing coverage, structured testing had the highest percentage, with all-uses second, and branch coverage third.

Figure 30 shows the total number of random tests necessary to satisfy each criterion for each program averaged over 100 trials. Figure 31 shows the total number of random tests increasing coverage for each criterion for each program averaged over 100 trials. Figure 32 shows the percentage of trials detecting an error for each criterion for each program. Figure 33 shows the percentage of trials detecting an error on

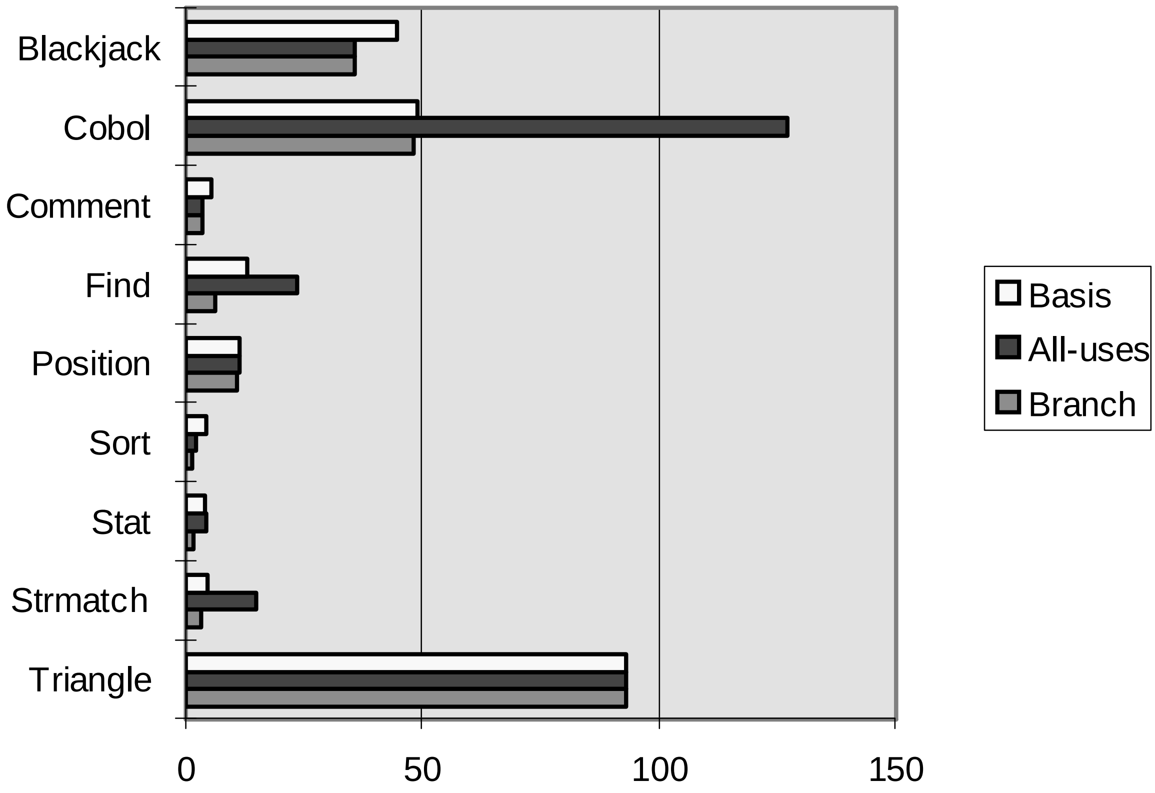


Figure 30: Total tests

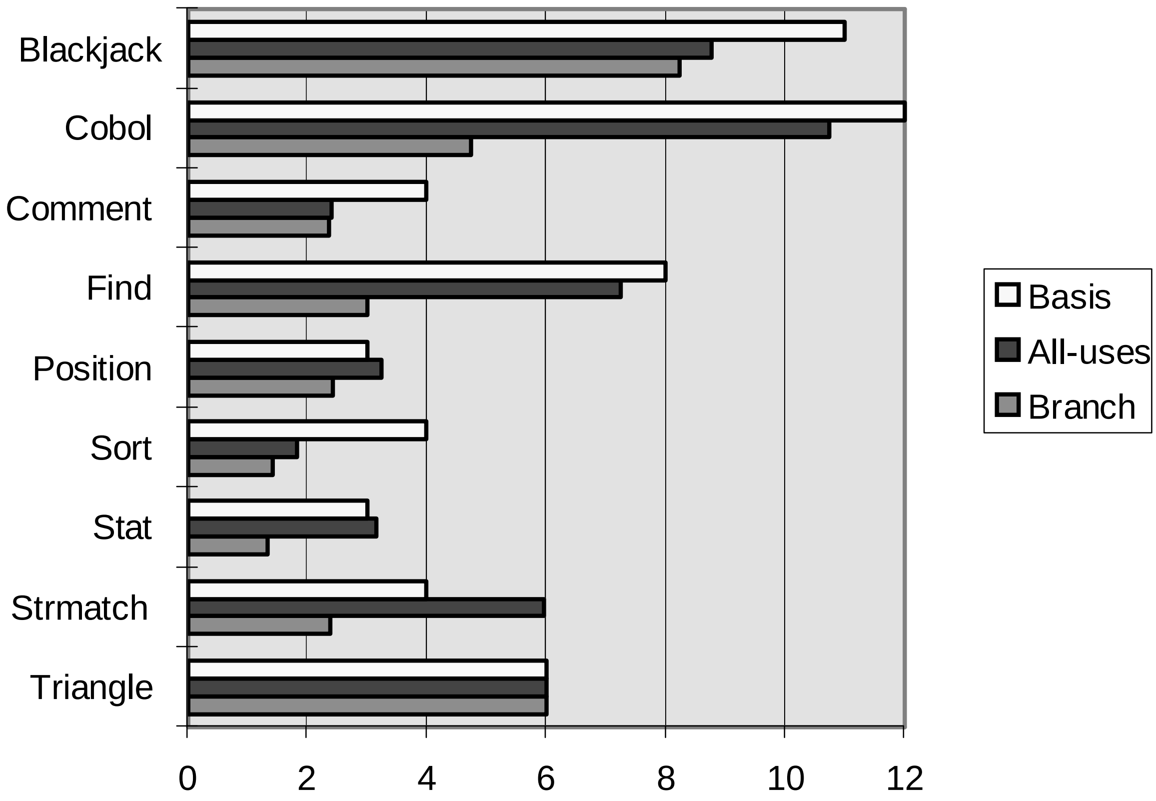


Figure 31: Tests increasing coverage

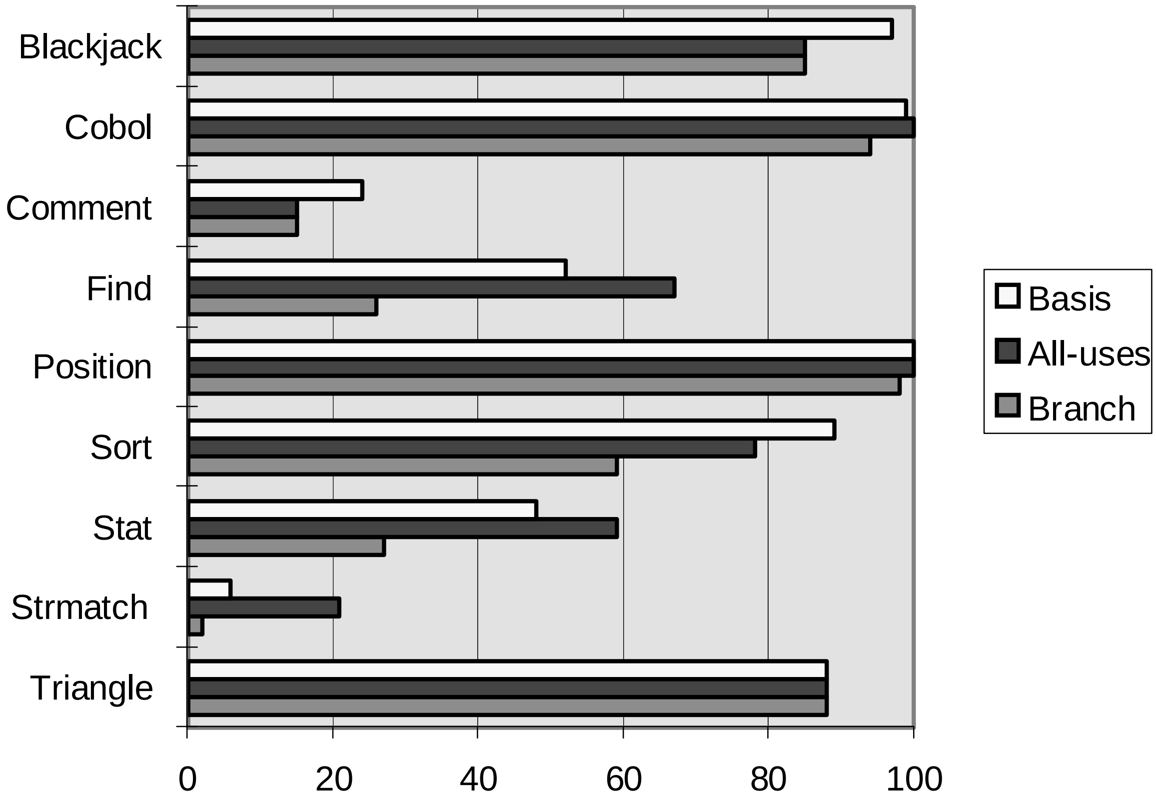


Figure 32: Total detections

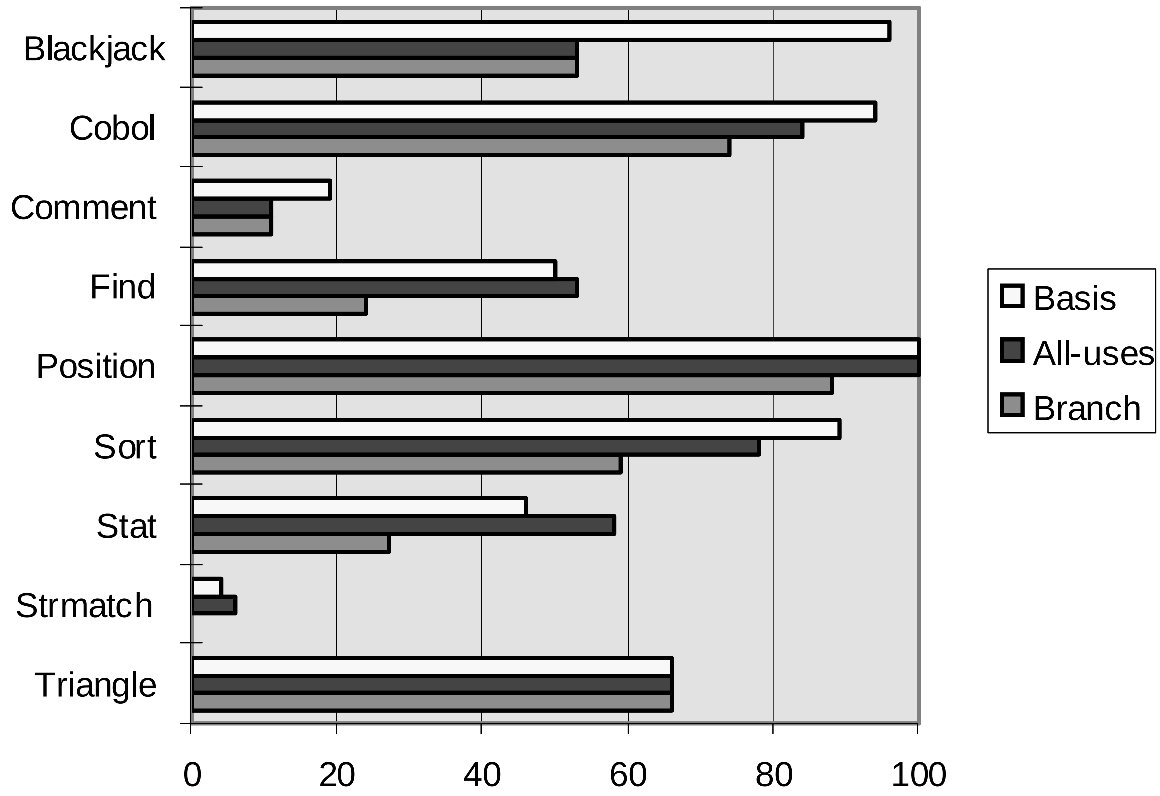


Figure 33: Detections increasing coverage

a test case that increased coverage for each criterion for each program.

4.4.1 Analysis of results

None of the criteria were effective at detecting errors in either `Strmatch` or `Comment`, with all criteria detecting an error in those programs fewer than 25% of the time even without minimizing the test sets. All criteria had exactly the same performance for `Triangle`, which makes sense since `Triangle` consists entirely of nested decisions and thus the structural coverage subsumption hierarchy collapses from all-paths to branch coverage. This leaves six of the original nine programs to provide meaningful differentiation between the criteria.

For total number of tests, the criteria were fairly close, except that the average number of tests to satisfy all-uses for `Cobol` was very high (127, compared to 49 for structured and 48 for branch), especially considering that the number of tests increasing coverage was less than for structured testing. Hence, there is no clear reason to prefer any of the criteria based on the number of random tests required for test suite construction.

For the number of tests increasing coverage, no criterion took more than 12 tests on average for any of the test programs. Structured testing took about half a test more on average than all-uses, which took about two tests more than branch coverage. This difference is only significant if running tests is extremely expensive, and in any case should be a subordinate factor to error detection effectiveness.

For the total probability of detecting errors, all-uses and structured testing were very close, with branch coverage significantly weaker for most tests. This indicates that for situations in which the coverage criterion is used as a stopping rule for random testing, all-uses and structured testing are of comparable value, while branch coverage is less powerful.

For the probability of detecting errors while increasing coverage, structured testing outperformed all-uses, which dramatically outperformed branch coverage. The most significant differentiator for structured testing was `Blackjack`, where it had an error detection probability of 96% as opposed to 53% for both all-uses and branch. This indicates that for situations in which test suites will be minimized with respect to the coverage criterion, structured testing may be preferable to all-uses, while branch coverage is much weaker.

An interesting pattern in the data appears when ranking the programs with respect to each testing criterion according to the probability of detecting errors. For total error detections, the ranks of each program differ by no more than one position across the three lists, except for `Triangle` which differs by two positions. This suggests that when used as a stopping rule for random testing, all three criteria may be sensitive to the same features of the programs being tested. There is a similar general trend in the data for error detections increasing coverage, although it is too weak to justify any conclusions.

Of particular interest is the excellent behavior of structured testing with respect to test suite minimization. Considering only the tests that increased coverage, the effort was reduced by 76%, while the probability of detecting errors was reduced by only 6% (as compared with 16% for all-uses and 17% for branch).

4.5 Conclusion

The results described in this chapter suggest that structured testing is a strong alternative to all-uses coverage, and that both are significantly stronger than branch coverage. For the experiments of this chapter, structured testing required 26% fewer random tests for test suite construction than all-uses coverage, and only 13% more than branch coverage. The probability of detecting an error with a test set minimized with respect to structured testing was 11% greater than with all-uses, and 40% greater than with branch coverage. Thus, structured testing is particularly appropriate for constructing test suites that will be minimized with respect to the coverage criterion.

Chapter 5

Structured Integration Testing

Effective integration testing is necessary for successful software development. Even if all software modules have been thoroughly tested in isolation, assembling the modules into a complete program tends to result in many errors, which can only be discovered by exercising the interfaces between modules [49]. Integration testing is also a major risk factor in software project management. Since errors found in integration testing can set a project back as far as the design phase, it is difficult to estimate the time required for integration testing. Since integration testing is one of the last steps before a system is put into production, any delay in integration testing translates directly into a delay in system delivery (or the delivery of a low-quality system). Many software development practices attempt to contain the risk of integration testing, including top-down development, hierarchical integration, incremental integration, and partial regression testing [49]. In order to be useful, an integration testing strategy must be compatible with these development practices.

Integration testing presents significantly greater difficulties than module testing. Module testing is typically done by the individual developer during the coding process, so the developer is free to limit module complexity, insert testing hooks, and derive test cases based on a solid understanding of the module's internal structure. None of these advantages are available to the typical integration tester. While module complexity may be easily limited and reduced if necessary, system complexity can be arbitrarily high. Integration testers may not have detailed experience with the code they are testing, and may not be authorized to make changes. If an error is found during module testing, the developer can fix the module locally and retest. It is often difficult to determine the cause of an error found during integration testing, or even whether it was introduced during design or coding.

While white-box testing is widely used at the level of individual source modules, most integration testing is done entirely using black-box techniques [20], such as equivalence partitioning and boundary value analysis [49]. Also, given that design documentation often contains specifications for individual modules, even the white-box module interface testing strategy can also be viewed as black-box design testing. Many software developers tend to assume that white-box integration testing is philosophically incompatible with modular development, since the implementation details of each module are encapsulated. However, the same arguments in favor of white-box testing at the module level apply equally well to integration. Two test cases that appear equivalent from the functional perspective may be handled by completely different code in the implementation. Also, errors tend to occur most frequently in code that is unlikely to be executed by black-box testing [49].

McCabe [38] originally suggested the direct application of structured testing for top-down integration. Beginning with the main routine, this technique requires that actual modules replace stubs one at a time, and that a basis set of control flow paths are tested through each module as it is added to the system. This approach has two serious weaknesses. First, it is only defined for the top-down integration strategy — there is no objective mathematical criterion that can determine whether a given test suite is sufficient to test a particular integrated system. Second, it relies on driving each module through a full basis set of control flow paths in an integration context, which is often impossible due to intermodule control coupling.

In this chapter, we extend the structured testing technique to the integration level in a way that avoids the above weaknesses. We give the basic technique, describe automated support, and then discuss extensions to support realistic integration strategies. This technique is based on the general approach, proposed by McCabe and Butler [39], that only control flow affecting module calling relationships needs to be considered for integration testing. In Section 5.4, we give a detailed analysis and comparison with McCabe and Butler's work.

5.1 A model for integration testing

In this subsection, we present a model for integration testing that is amenable to mathematical analysis. But first, we review the definitions of module and program. A *module* is a single-entry, single-exit directed graph in which all nodes are reachable from the entry and the exit is reachable from all nodes. A *program* is a set of modules in which some nodes, denoted *call nodes*, refer to other modules, with one of the modules denoted the *main* module.

Now we define the concept of paths and tests for programs. A *path* through a program is a collection of paths through the modules of the program in which the number of occurrences of the entry node of the module equals the number of occurrences of call nodes to that module, for all modules except the main module, and the number of occurrences of the entry node of the main module is one greater than the number of occurrences of call nodes to the main module. A *program path* through a module is the sum of the paths through that module induced by a path through the program. We require that each module appear on at least one program path to be considered part of a program. We will use the program structure in Figure 34, the details of which will be explained in the next section, as a running example. Consider the program path in which the loop body of module `A` is executed once, the decision of module `B` is taken to execute module `D` along one of its two possible paths, and the decision of module `C` is taken to execute module `D` along the other possible path. Then, the program path through `D` is the sum of both module paths through `D`. Since the other modules were executed only once, the module and program paths through them are the same.

A *test* is a program path in which any module may be considered to be the main module. The program path through Figure 34 described above is a test. Another test executes module `B`, taking the decision branch that does not call module `D`. The definition of a test is critical to the system. An immediate consequence of our definition is that a single execution of the program can contribute at most one program

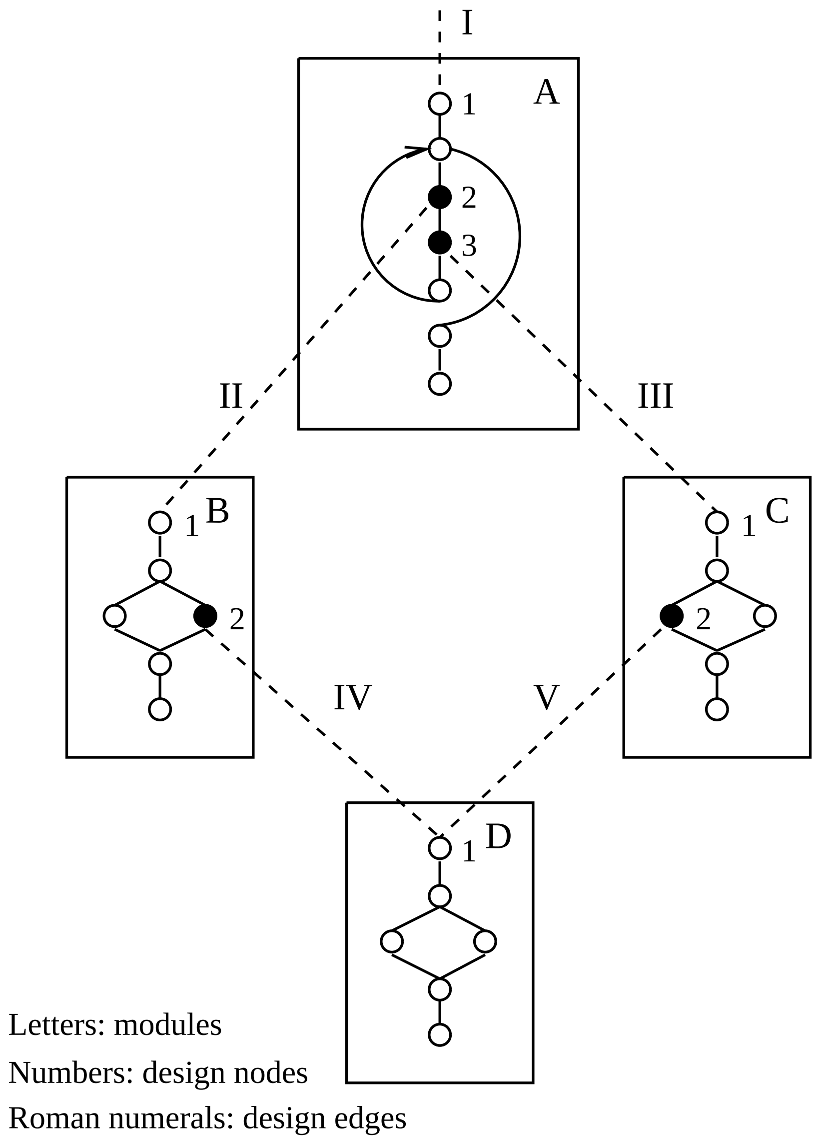


Figure 34: Example program

path through each module. Our test coverage criterion will be defined in terms of program paths, so this means that each test can contribute at most one unit of coverage to each module in the program being tested. This does not change our model of the program structure, it just means that the coverage vector corresponding to a test through a module is the sum of the coverage vectors corresponding to the individual activations of the module during a single execution of the program.

An important property of module-level structured testing is that the number of tests required increases with module complexity. Our definition of test preserves this property at the integration level, requiring $v(G)$ tests to execute a basis set of program paths through a module in any integration context. A basis set of *module* paths through an arbitrarily complex module, on the other hand, can be exercised with just one execution of a driver module that calls the given module iteratively. Although merging module paths to form program paths can decrease the tested rank of a module, it cannot increase it:

Theorem 7 *The rank in a module's edge space induced by a set of tests is never greater than the rank of the set of individual entry-to-exit paths through that module executed during those tests.*

Each test through the module is a linear combination, in fact the sum, of the individual module paths through the module during that test. Thus any edge vector formed from a linear combination of the tests is also a linear combination of the individual paths, and therefore cannot have greater rank.□

Our testing definition allows the usage of test drivers and in-place testing of subsystems. As an example, consider the case of testing a continuously running transaction processing system, where the top-level system is a loop around calls to routines that read a request and dispatch it to the appropriate subsystem for processing. The natural approach in this situation is to replace the single-loop routine with a test driver that performs the same function and makes each top-level call within the loop constitute a separate test. For more complex systems in which it would be difficult to augment a high-level routine with driver functionality, such as a menu-driven interface, tests may be initiated and terminated using an external signal mechanism.

A valuable attribute of our system is its flexibility to be used in many different integration environments without modification of the source code being tested.

5.1.1 Definitions

Let the *design nodes* of a module be its entry and call nodes. Let a *design matrix* of a module be a matrix in which rows correspond to tests and columns correspond to design nodes, and the entries are the number of times each design node occurred in each test. We define the *design complexity, dc,* of a module to be the maximum rank of its design matrix for all possible sets of tests, assuming that all decisions in the module are independent. Section 5.1.2 gives a method for calculating *dc*.

Consider the following representation of a program, as shown in Figure 34: each module is represented by a flow graph, each design node is connected to the corresponding module entry node by a special edge called a *design edge*, and an additional design edge is associated with the entry of the main module to model external activation. We define the *global design matrix* to be a matrix in which rows correspond to tests and columns correspond to design edges, and the entries are the number of times each design edge occurred in each test. We define the *global design complexity, gdc,* of a program to be the maximum rank of its global design matrix over all possible sets of tests, assuming that all decisions in the program are independent. Section 5.1.2 gives a method for calculating *gcd*.

For example, consider the program represented by Figure 34, with module `A` as the main module. Assume we run the following three tests, each one a complete program path:

1. Execute the loop in module `A` twice, the first time taking the left branch in all other modules, the second time taking the right branch in modules `B` and `C` and the left branch in module `D`.

2. Execute the loop in module `A` once, taking the left branch in all other modules.

3. Execute the loop in module `A` once, taking the right branch in modules `B` and `C` and the left branch in module `D`.

Test \ Design node	I	II	III	IV	V
1	1	2	2	1	1
2	1	1	1	0	1
3	1	1	1	1	0

Global design matrix
Rank = 3, gdc = 4

Test \ Design node	1	2	3
1	1	2	2
2	1	1	1
3	1	1	1

Design matrix for A
Rank = 2, dc = 2

Test \ Design node	1	2
1	2	1
2	1	0
3	1	1

Design matrix for B
Rank = 2, dc = 2

Test \ Design node	1	2
1	2	1
2	1	1
3	1	0

Design matrix for C
Rank = 2, dc = 2

Test \ Design node	1
1	2
2	1
3	1

Design matrix for D
Rank = 1, dc = 1

Figure 35: Design matrices

Figure 35 shows the global design matrix and the module design matrices for each module.

5.1.2 Calculation of dc and gdc

Recall that dc is defined to be the maximum rank of the module design matrix over any set of tests, a definition that is of no direct help in actually calculating dc. However, due to the following fact, we may use the techniques of module-level structured testing to calculate dc:

Theorem 8 *dc equals the rank of the design matrix for any basis set of module paths.*

Each node, and thus each design node, can be expressed as a linear combination of edges, so any set of tests that spans the edge space of the flow graph also spans its design node space. Thus any basis set of module paths has a design matrix of full (dc) rank. □

To calculate dc, we first use produce a basis set of module paths using the technique discussed in Chapter 3. Then, for each path, we assign an incidence count to each design node equal to the sum of the incidence counts of all edges out of that node. Finally, we arrange the vectors of design node incidence counts for each path into a design matrix and take its rank. Theorem 8 shows that this rank is dc.

Note that this method of calculating dc generalizes to a method of calculating a minimal set of module paths with rank dc. First, we generate a basis set of module paths and construct the associated design matrix. Then we perform row-reduction on this matrix, which will result in exactly dc nonzero rows by Theorem 8. The paths corresponding to those nonzero rows form a set of module paths with rank dc, and this set is minimal since there are exactly dc such paths. The next section discusses the application of this set of paths to integration testing. Section 5.2 includes further analysis of the design matrix.

Design complexity and global design complexity are related by the following formula:

Theorem 9 $gdc = 1 + \sum(dc - 1)$ *over all modules of a program.*

Omitting the entry design edge, the design edges of the program are naturally isomorphic to the call nodes of the program's modules. Thus for any program path, the entry for a design edge in the global design matrix is the same as the corresponding call node's entry in its module's design matrix. Also, the entry for each module entry node in its module's design matrix is equal to the sum of the entries for all the design edges that lead to that module in the global design matrix.

Since we assume all decisions are independent, the only dependencies between global design edges are those induced by the graph structure of each module, each of which can be expressed as a linear equation in terms of a single module's design nodes, or equivalently in terms of the individual design edges out of and the sum of the design edges into a single module.

Thus we may calculate *gdc*, by considering a global design matrix M of full rank, with the columns partitioned according to the modules from which the corresponding edges emanate. For each module A, let I be the set of columns corresponding to edges into A, O be the set of columns corresponding to edges out of A, and C be a column vector whose entries are the sum of the entries across the corresponding rows of M restricted to I. Note that C corresponds to the entry design node of A and O corresponds to the call design nodes of A. The entry design edge into the main module contributes 1. Each module A contributes an additional

$$rank(M\ restricted\ to\ (I \cup O)) - rank(M\ restricted\ to\ I).$$

Since the only linear dependence across I and O can be between the members of O and C (the sum of the elements of I), this equals

$$rank(M\ restricted\ to\ O\ augmented\ by\ C) - rank(C).$$

Since A is a member of the program it is called on some program path, so since M has full rank, C is a nonzero vector which therefore has rank 1. Thus we have

$$rank(module\ design\ matrix\ of\ A) - 1,$$

which is $dc - 1$. Summing over all modules we get

$$1 + \sum(dc - 1).$$

□

Given this theorem, we calculate *gdc* by first calculating *dc* for each module in the program, and then calculating $1 + \sum(dc - 1)$ over all modules in the program.

As an example, we will calculate *dc* for each module and *gdc* for the program in Figure 34. Since there is only one decision per module we will specify paths as a sequence of decision values. In the figure, consider TRUE to be the left branch of each decision and FALSE the right. Table 14 shows the basis paths as produced by the baseline method for each module, their corresponding design vectors, and *dc* calculated by taking the rank of those design vectors as a matrix. From those values of *dc*, we calculate $gdc = 1 + (2 - 1) + (2 - 1) + (2 - 1) + (1 - 1) = 4$.

Table 14: Calculation of dc

Module	Basis paths	Design vectors	dc
A	FALSE	1 0 0	2
	TRUE, FALSE	1 1 1	
B	TRUE	1 0	2
	FALSE	1 1	
C	FALSE	1 1	2
	TRUE	1 0	
D	TRUE	1	1
	FALSE	1	

5.1.3 An integration test criterion

An obvious, but impractical, integration testing criterion is: *perform a set of tests for which the global design matrix has rank gdc*. This criterion has the same conceptual appeal as McCabe and Butler's approach [39], in that it attempts to test all intermodule interfaces independently. Unfortunately, it also suffers from some of the same weaknesses, caused by intermodule control dependencies and the difficulty of deriving test data for any given path through an entire program.

Within a single module, non-structural control dependencies can be split systematically across modules (and often are, as described in Chapter 3), creating intermodule dependencies, or merely noted and taken into account during module testing. At the integration level, it is infeasible to recognize and control all intermodule control dependencies. Also, such dependencies are extremely sensitive to maintenance changes. For example, inserting a single call to a random number generator in one module can change the maximum attainable global design matrix rank of an arbitrarily complex system from 1 to its *gdc*. Consider a system in which one function generates boolean values, and all decision predicates in the system are calls to that function. If the function returns a constant value, then there is only one realizable program path, so the maximum attainable global design matrix rank is 1. If the function returns a random value, then each decision predicate in the program is independent, so the maximum attainable global design matrix rank is equal to the program's *gdc*.

To balance rigor and feasibility, we re-examine the integration testing process from a conceptual level. The primary strength of modular design is that it allows a module to be viewed without regard to its internal structure. In black-box testing the entire system is viewed without regard to its structure. While white-box testing requires that we view and test the internal structure of each module, still from the perspective of each module the rest of the modules of the system may be regarded as a black boxes. This is equivalent to stating that cross-module dependencies are irrelevant from the point of view of an integration test criterion, and logic need only be tested independently of other logic within the same module. Therefore, we propose the following minimal integration test criterion, called the *Structured Integration Criterion (SIC)*: perform a set of tests that causes each module's design matrix to have rank dc.

For module-level structured testing, the basis path generation algorithm discussed in Chapter 3 is an efficient way to generate a minimal set of tests that satisfies the structured testing criterion. Satisfaction of *SIC* requires at least *max dc* tests, since any test can contribute at most one row to each module's design matrix. It requires at most *gdc* tests, since a global design matrix of full rank implies that each module design matrix also has full rank. However, exact determination of the number of tests required to satisfy *SIC* is computationally intractable.

Theorem 10 *It is NP-hard to determine if max dc tests suffice to test dc rank in all modules of a program, even if all code is structured, acyclic, and non-recursive.*

We prove this by reduction from 3-SAT [15]. Given a 3-SAT instance with variables *u1,...,un* and clauses *c1,...,cm*, we construct a C program such that *max dc* tests suffice if and only if there is a truth assignment to *u1,...,un* that satisfies *c1,...,cm*.

Define one trivial module and one integer variable:

```
triv(){}
int var;
```

For each clause *ci* define three boolean variables and a function:

```
int cond_clause_i_1, cond_clause_i_2, cond_clause_i_3;
```

```
clause_i() {
    if (cond_clause_i_1) triv();
    if (cond_clause_i_2) triv();
    if (cond_clause_i_3) triv();
}
```

For each variable uj declare two boolean variables:

```
int cond_uj_1, cond_uj_2;
```

Create a main module with an n-way decision in which each branch corresponds to a variable and has 2 sequential decisions:

```
main() {
    read assignments to all program variables
    switch (var) {
        ...
    case j:
        if (cond_uj_1) {
            call clause_i() for all i s.t. uj appears in ci
        } else {
            call clause_i() for all i s.t. ~uj appears in ci
        }
        if (cond_uj_2) triv(); else triv();
    break;
        ...
    }
}
```

Due to the semantics of C we make the last label `default:` rather than `case n:`. The construction is now complete.

First, we show that the main routine has $dc = 3n$. The `call clause_i() for all i s.t. ...` pseudocode indicates a sequence of call statements to the specified clauses. Although if `uj` appears either only uncomplemented or only complemented one such sequence will be empty, at least one sequence for each branch will contain a call since each variable must appear in at least one clause. The fact that both outcomes of the test of `cond_uj_2` contain calls to `triv` ensures that each case-labeled branch contains at least one call. Thus, we can measure the contribution to dc independently for each case-labeled branch and add them to reach the value for main. Each case-labeled branch contains two sequential decisions, each of which contains a design node as at least one outcome, and therefore contributes its full cyclomatic complexity of three. The main routine thus has $dc = 3n$.

Assuming $n > 1$, $3n$ is therefore *max dc*, since the clause routines each have $dc = 4$ and the `triv` routine has $dc = 1$. Any $3n$ tests that have full design rank for the main routine are split into three tests for each main branch (variable) `j`, two of which set `cond_uj_1` to one value, and one of which sets `cond_uj_1` to the other value. Taking `cond_uj_1` true twice and false once corresponds to a truth assignment of TRUE to uj, and taking `cond_uj_1` true once and false twice corresponds to a truth assignment of FALSE to uj.

Thus each variable contributes two paths to functions corresponding to clauses in which a literal corresponding to it is true, and one path to functions corresponding to clauses in which a literal corresponding to it is false.

If all literals in a clause are false, then exactly three paths will go through the corresponding clause function, which are insufficient to cover its dc of four. If any literal in a clause is true, then at least four paths will go through the corresponding clause function, which is sufficient to cover its dc of four.

Thus the correspondence is exact, and since only linear expansion in the input was required, finding the size of a minimal integration test set is NP-hard. □

Despite the theoretical complexity of our proposed structured integration criterion, it has several properties that make it potentially attractive for practical application. It can be verified by an automated system, and a small (but not necessarily minimal) set of additional tests to satisfy it can be generated automatically and specified entirely in terms of paths within individual modules. This means that there is a great deal of freedom left to the tester to generate test data for each additional test. Generating data for a fully-specified program path is sufficiently difficult to be impossible to do by hand for any real system, while generating data to go through a particular path in an isolated module of a system is frequently not difficult. The developer of a particular module, for example, may be able to suggest system input data that causes a module to be exercised. Even so, a trial-and-error approach is probably the method of choice in selecting integration test data, since the integration tester may lack the in-depth experience necessary to perform systematic data derivation. Our automated system makes this approach practical by providing efficient feedback about the suitability of each test.

5.1.4 Realizing gdc

Although the *gdc* of a system is frequently unrealizable, the baseline method can be extended to generate *gdc* globally design-independent tests for a system, assuming no control dependencies and static function call resolution. The absence of control dependencies allows us to generate tests based purely on structural information. Static function call resolution allows us to trace control flow across modules. We assume these conditions for the rest of this section.

To extend the baseline method for generating system-level paths, we must consider iteration and recursion, which introduce complications not present in the single-module case. In particular, we may not restrict our attention to single entry-to-exit paths within individual modules. For iteration, we may generate an entry-to-exit path in one module that induces several entry-to-exit paths in another. We must deal with these composite paths in a way that ensures the independence of the resulting tests. Recursion has an even greater problem, since when we generate a test that includes a (directly or indirectly) recursive call, we must augment that test with a way to unwind the recursion and exit the program, again ensuring the independence of the resulting tests.

Our algorithm to generate *gdc* globally design-independent test paths through a program proceeds in three stages. First, for each module we select a *baseline* test that does not repeat any node within that module and only goes through baseline tests in other modules. We choose the baseline test through the main module of a program

```
calculate_baselines()
{
    let S = set of all modules
    assign infinite weight to all modules
    while (S nonempty) {
        select element m of S with least weight shortest entry-exit path
        make that shortest path the baseline for m
        assign weight of m = cost of the baseline
        remove m from S
        update all edge weights to reflect new weight for m
    }
}
```

Figure 36: Program baseline calculation

to be a shortest program path in terms of total number of edges. Next, we generate $1 + \sum(v - 1)$ tests that form a basis for the space of all flow graph edge vectors in the program, by varying exactly one decision outcome in one module from previous values for each test. Finally, we project these tests onto a global design matrix and perform row reduction to get *gdc* basis tests.

To select the baselines, we use the following data structures: Maintain a *weight* for each module. The final weight of a module is the length of the shortest baseline path through the module when the lengths of the paths induced on called modules are also considered. Within each module, we assign a weight to each edge as follows: if the source node of the edge is a call node, the weight of the edge is one plus the weight of the called module, otherwise the weight of the edge is one. All "shortest" paths will be calculated in terms of the sum of the edge weights on the paths. Figure 36 contains the code for calculating the baseline paths. The loop always terminates with S empty, since at each iteration one element is removed from S. The weight of the module removed from S at each iteration is less than ∞, since otherwise the contents of S would form an unconditionally infinite loop. Such a loop implies that none of those modules could appear on a program path, contradicting our assumption that the modules belong to the program. The baseline path never repeats a node, since it is a shortest weighted path with all edge weights positive.

To construct the (non-reduced) independent tests from the program baselines, we build a set of tests, initially containing the single test formed by beginning with the root module and inducing the baseline path through each called module. Then while some test T in the set contains a decision with an outcome not exercised on any test in the set, we form a new test T' based on T. In program execution order, T' is identical to T until the first occurrence of the decision in question, which is altered to have the new outcome in only the current activation. Then a shortest path within that module to any node that appears on some test in the set is followed, and that test is followed to the exit node of the module, after which control returns and test T is followed until the exit from the program. The baseline path is taken for any induced calls introduced by the new outcome, including recursive calls to the altered module. Since exactly one new decision outcome (edge vector) is given a nonzero value at each step, we end up with $1 + \sum(v - 1)$ tests that form a basis for the module edge space of the program.

To construct the reduced independent tests, we observe that since node incidence and thus design node incidence may be expressed as a linear combination of edge incidence, a subset of the non-reduced tests suffices. We construct the global design vectors corresponding to each of the non-reduced tests, and row-reduce the resultant global design matrix to recover a global design basis. As desired, this yields a basis set of *gdc* tests for the program.

As an example, we will calculate a basis set of *gdc* tests for the program in Figure 34. Figure 37(a) shows the baselines and weights calculated in Stage 1. Figure 37(b) shows the non-reduced independent tests calculated in Stage 2. Figure 37(c) shows the global design matrix for the tests of Figure 37(b). The first four tests form a basis for the matrix of Figure 37(c), and therefore these are the four basis tests for the program. Recall that we calculated $gdc = 4$ for this program in Section 5.1.2.

Note that the final weight of each module during the baseline selection algorithm is the minimum total number of edges required to traverse from the module's entry to exit, obeying all call/return relationships within the program. Thus, the baseline test through the main module of a program is a shortest program path in terms of total number of edges in the associated path through the program.

Module	Baseline path	Weight
A	FALSE	3
B	TRUE	4
C	FALSE	4
D	TRUE	4

(a) Stage 1 — Compute baselines

Test	Path
1	A FALSE
2	A TRUE, B TRUE, C FALSE, A FALSE
3	A TRUE, B FALSE, D TRUE, C FALSE, A FALSE
4	A TRUE, B TRUE, C TRUE, D TRUE, A FALSE
5	A TRUE, B FALSE, D FALSE, C FALSE, A FALSE

(b) Stage 2 — Non-reduced independent tests

Test	I	II	III	IV	V
1	1	0	0	0	0
2	1	1	1	0	0
3	1	1	1	1	0
4	1	1	1	0	1
5	1	1	1	1	0

(c) Stage 3 — Global design matrix

Figure 37: Example gdc calculation

5.2 Automation

In this section, we describe our implementation of the technique described in Section 5.1. The system is similar to the module-level implementation described in Chapter 3, including control flow analysis and source code instrumentation. We describe the system architecture and the source code instrumentation technique, and discuss efficiency issues.

5.2.1 Architecture

The system architecture is independent of any particular compiler, and minimizes language-dependent processing. It is very efficient both in terms of run-time overhead of instrumented code and the time and space required to perform analysis. All code instrumentation is done at the source level, and source files may be instrumented separately and then compiled together as a program along with the instrumentation library without any special link-time processing. We parse source code to determine the source code locations at which code could be inserted to instrument each design node, then analyze the flow graphs to determine a basis set of design nodes, then actually instrument just the basis design nodes. We keep a basis design node incidence vector for each module at run-time, and have library functions to clear all vectors and write all vectors, with the default behavior being to clear on program entry and write on program exit. The modules link their vectors into a global list the first time they are called, as suggested by Weinberger [59] in the context of statement-count profiling. No dynamic memory management is required at run-time of the instrumented code.

Figure 38 shows a data flow diagram of the system architecture. The `Parser` reads the preprocessed `Source Code`, and writes an abstract `Flow Graph` and a `Design Nodes and Source Locations` file. This file contains a list of design nodes for each module in the source file, each with a set of byte offsets (keys) into the source file that are sufficient to instrument it. The `Design Reducer` reads the flow graph and the design nodes file and produces a `Trace Events and Source Locations` file, which contains the trace event information for exactly a basis set of design nodes (including the entry node) for each module. The `Event Converter` reads the trace events file

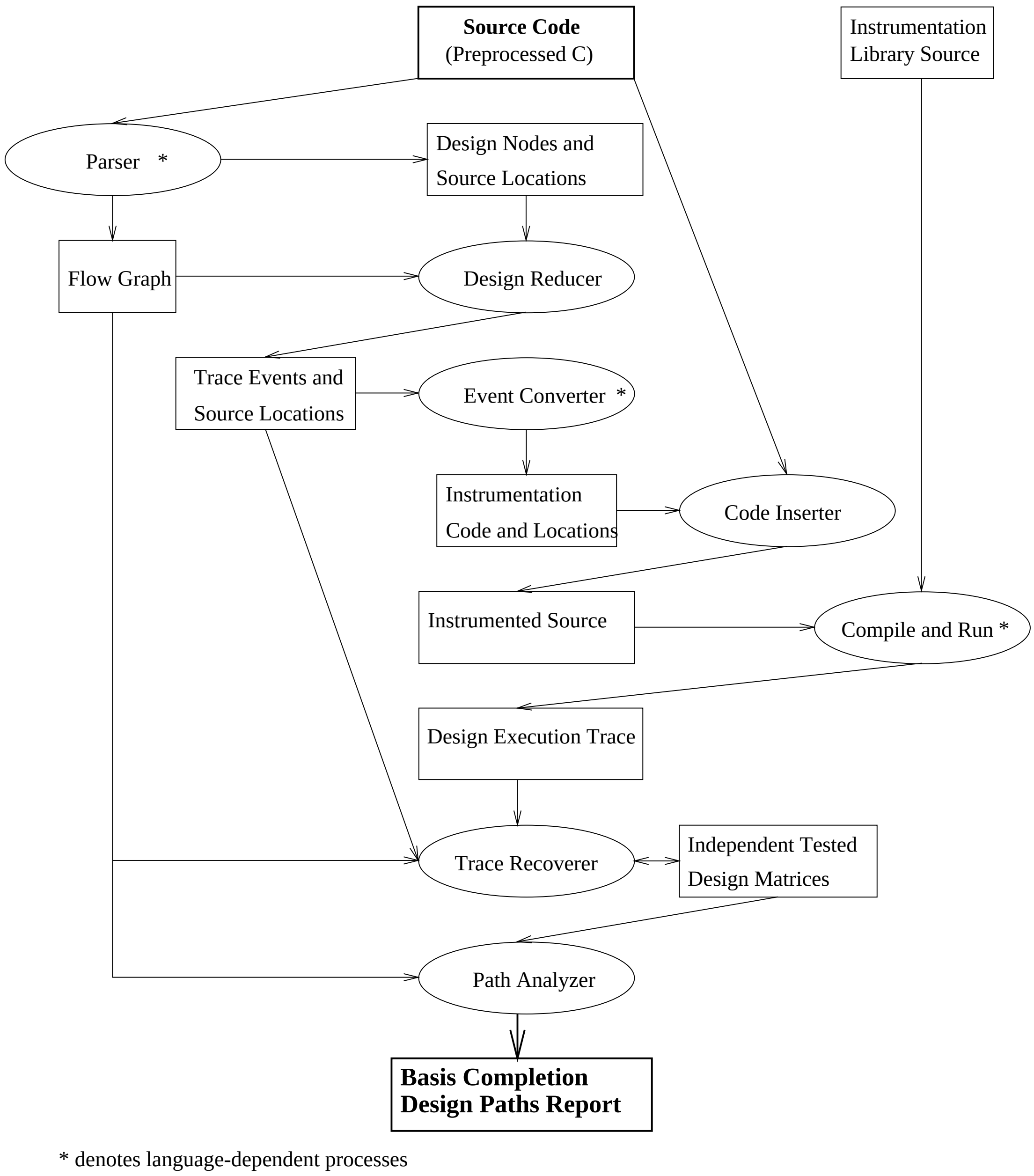


Figure 38: System architecture

and writes the `Instrumentation Code and Locations` file. This file contains code fragments to insert into the source code to instrument the design trace events, each with a byte offset into the source file to specify where to insert the fragment. The `Code Inserter` reads the source file and the code fragments and inserts the fragments to produce the `Instrumented Source` file. The user compiles a set of instrumented source files, linking with the `Instrumentation Library Source`. When the user runs the instrumented code, it produces a `Design Execution Trace` file containing basis design incidence count vectors for each module for each test. The `Trace Recoverer` reads the design trace file and the trace event file and writes an `Independent Tested Design Matrices` file, which contains a basis for all tested design vectors for each module. It can also be used to update the data in a design matrices file to reflect the results of additional tests. The `Path Analyzer` reads the design matrices and the flow graphs, and produces the `Basis Completion Design Paths Report`, a minimal set of paths for each module to complete a basis set of design tests.

5.2.2 Selecting basis design nodes

Given a flow graph with all the design nodes marked, we select a basis set of design nodes using the algorithm shown in Figure 39. Since we require that the entry node always be a member of the basis for instrumentation purposes, we place its column first in the design matrix so that it will always be nonzero after the column reduction step. The `Design Reducer` process performs this reduction, and eliminates all non-basis design nodes from the event file. We perform all subsequent matrix calculations using the basis design matrix, which has exactly `dc` columns.

5.2.3 Source instrumentation for C

As an example of C source instrumentation, we use the `main` function of Hanson's common words program [17], the complete code for which is listed in Appendix A. Figure 40 shows the code for `main`, and Figure 41 shows its flow graph with the basis design nodes shown in black and the other design nodes shown in grey. Observe that `main` has cyclomatic complexity five, seven design nodes, and has design complexity

```
select_design_basis()
{
    calculate a basis set of flow graph paths, P;
    form a design matrix, M, from P;
    column-reduce M;
    mark nodes corresponding to non-zero columns of M;
}
```

Figure 39: Basis design node selection

```
main(argc, argv)
int argc;
char *argv[];
{
    int i, k = 22;
    char *p, *getenv(), buf[MAXWORD];

    progname = argv[0];
    if (argc > 1)
        k = atoi(argv[1]);
    else if (p = getenv("PAGESIZE"))
        k = atoi(p);

    for (i = 0; i <= HASHSIZE; i++)
        hashtable[i] = NULL;

    while (getword(buf, MAXWORD) != EOF)
        addword(buf);
    printwords(k);
}
```

Figure 40: Code for main

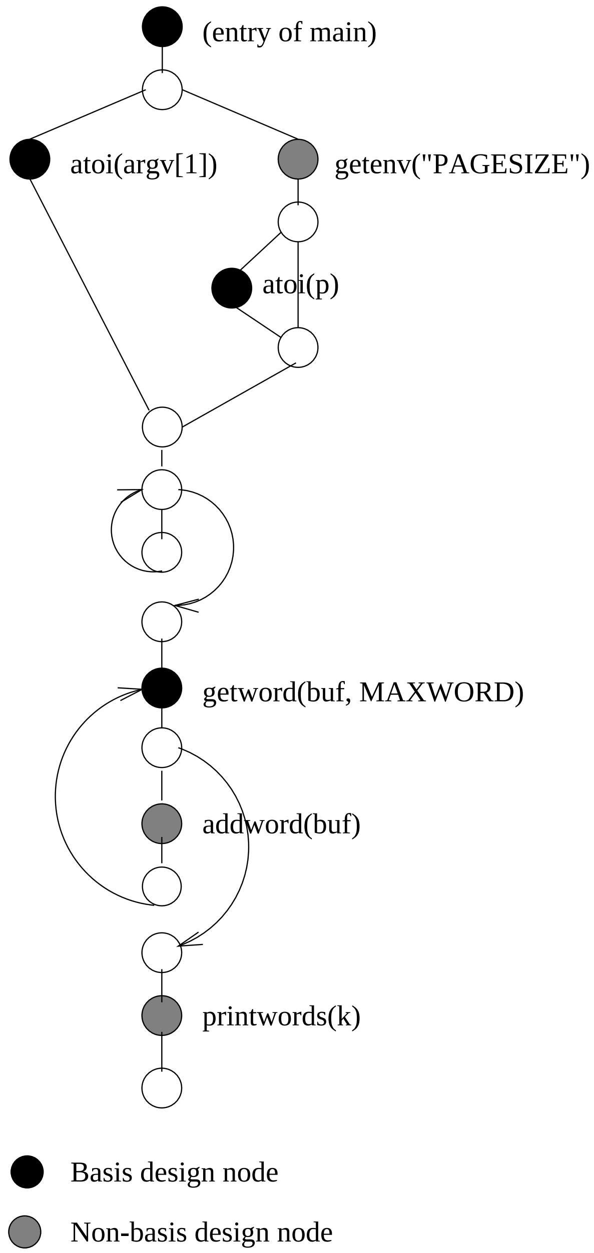


Figure 41: Graph for main

four.

Figure 42 shows the instrumented code for `main`, with the code inserted by the system in italics.

The source code for the C instrumentation library may be found in Appendix A. The `inst_link_module` function, which is called by the inserted code, puts the module's `inst_info` structure on a list to be written to the trace file when the program terminates, using the standard C library function `atexit`. The actual writing of trace data is performed by the `inst_dump_counts` function. One other function, `inst_reset_counts`, allows the coverage information for all modules to be reset to zero. The `inst_reset_counts` and `inst_dump_counts` functions are globally accessible so that users can run more than one test with a single execution, allowing more fine-grained control over the testing process.

5.2.4 Performance

We evaluated the system performance with respect to three criteria: space overhead, execution time time overhead, and trace file size. Since the trace file consists of execution counts for the module entry nodes and a subset of the call nodes, rather than the sequence of control flow branches produced by the system described in Chapter 3, the execution time overhead and trace file size are much less, suggesting that performance should be measured relative to an existing execution count profiler. We chose the call graph edge count facility provided by the $\widehat{\text{l}}$cc C compiler [14] with the `-Wf-C` option as a basis for comparison, and compare our system's performance in each of the three areas with this facility on a DEC 5100 computer.

The space overhead is the sum over all modules of $12 * dc$, which is less than *12 * total cyclomatic complexity*, plus the text taken up by the inserted code and the instrumentation library routines. Asymptotically, this is:

$$\textit{Space overhead} = O(\textit{total cyclomatic complexity}).$$

We used the code for our system's C parser as a benchmark, measuring object size with the `size -A` command for uninstrumented code, code instrumented with our

```
main(argc, argv)
int argc;
char *argv[];
{
    static struct {
        char *next;
        int cCounts;
        char *szMod;
        int rgCounts[4];
    } inst_info;
    if (!inst_info.next) {
        inst_info.cCounts = 4;
        inst_info.szMod = "main";
        inst_link_module(&inst_info);
    }
    inst_info.rgCounts[0]++;
{
    int i, k = 22;
    char *p, *getenv(), buf[MAXWORD];

    progname = argv[0];
    if (argc > 1)
        k = (inst_info.rgCounts[1]++, atoi(argv[1])  );
    else if (p = getenv("PAGESIZE"))
        k = (inst_info.rgCounts[2]++, atoi(p) ) ;

    for (i = 0; i <= HASHSIZE; i++)
        hashtable[i] = NULL;

    while ( (inst_info.rgCounts[3]++, getword(buf, MAXWORD ) ) != EOF)
        addword(buf);
    printwords(k);
}
}
```

Figure 42: Instrumented code for main

system, and code instrumented with `lcc`. Our system added 23% to the object size, and `lcc` added 29%.

The time overhead is essentially a test and two increments for each function call. Asymptotically, this is:

$$\textit{Time overhead} = O(\textit{total executed function calls}).$$

We used the fully-instrumented common words program [17] running on input from `/usr/dict/words` as a benchmark, measuring time in total "clocks" as given by the C library function `clock` called at the start and end of the program's main routine. We ran the same test for uninstrumented code, code instrumented with our system, and code instrumented with `lcc`, running each test 100 times to compensate for the effects of system load on measured performance. Our system added 8% to the run time, and `lcc` added 9%.

The trace file size is roughly proportional to the total number of basis design nodes in the system, which is bounded by the total cyclomatic complexity. Asymptotically, we get:

$$\textit{Trace file size} = O(\textit{total cyclomatic complexity}).$$

Since uninstrumented code does not produce a trace file, a percentage overhead measure is not meaningful. For the common words program with input from `/usr/dict/words`, the trace file produced by our system was about half as large as the trace file produced by `lcc`.

5.3 Extensions

In this section we extend the core integration testing method to support realistic integration strategies. Real systems require an extended method for several reasons.

First, systems are typically designed, developed, and tested as hierarchies of subsystems. Once the components of a subsystem have been tested in subsystem integration, it is wasteful and impractical to conduct full system integration tests to verify that exact same interaction. Hierarchical system design limits each stage of development to a manageable effort — the corresponding stages of testing should be similarly limited. Hierarchical design is most effective when the coupling between sibling components decreases as the component size in modules grows.

Second, systems or subsystems are often developed incrementally — skeleton functionality is achieved early on, and stub modules or subsystems are replaced with fully functional ones as development proceeds. Most real systems are too complex for one-shot integration to be effective — the system would fail in so many places at once that the debugging/retesting effort would be impractical [49]. As each new system component is added, integration testing must be performed to preserve the functionality of the skeleton system. It is impractical to perform a full regression test of system integration as each component is added — the integration testing of a new component should be limited to its interaction with the existing system. An extreme variation on the same problem occurs during system maintenance. Typically an extremely small number of software changes are made in proportion to the system size, after which the changed modules are re-integrated into the system. It is frequently impractical to run a full system regression test after a minor software change, yet it could be disastrous to omit re-integration testing entirely — the re-integration test effort should concentrate on the interaction between the changed code and the stable code.

Finally, it is impractical to do all module testing in isolation, due to the excessive overhead of writing test drivers and stub functions. A practical test strategy should support module testing in a partial integration context.

In this section we first describe our approach to incremental integration using statement coverage as a simple example. We then discuss the generalization of our proposed integration testing strategy to support integration of groups of modules, and discuss how common incremental integration strategies are handled by this technique. We also discuss the application of this technique to object-oriented systems.

5.3.1 Generalizing a simple integration strategy

As a preliminary example, consider a generalization of statement coverage to support incremental integration. To test interactions among modules at the statement coverage level, the clear choice is to cover all module call statements. Generalizing this to test interactions among components at a step of incremental integration, the criterion is to cover all module call statements that call from one component into a different component. This gives a subset of statement coverage in all modules.

To make a complete testing strategy, we require that at the unit level each statement is covered within each module, and then at each integration step all module call statements that cross component boundaries are covered. Given integration stages with good cohesive partitioning properties, this limits the testing effort to a small fraction of the effort to cover each statement of the system at each phase. Although the situation with structured integration testing is more complex, the same underlying generalization technique applies.

5.3.2 Groups and hierarchical integration

One extension to the basic method presented in Section 5.1 is sufficient to handle hierarchical testing, incremental development, and testing in a partially integrated system. We define a *group* recursively to be either a single module or a set of groups. We say that a module *belongs* to a group if either the group is a single-module group with the module as its only member, or the group is a non-single-module group and the module belongs to one of the group's members. We give a general method for integration testing a group, then show how to model the various development strategies outlined above using groups. For the case in which the group being tested consists entirely of single-module groups, this method is equivalent to the core method.

A single-module group is *integrated* if the module has been tested through a spanning set of module paths in some context, using the module-level structured testing techniques. A non-single-module group G is integrated if each of its subgroups is integrated and full rank in the group design matrix for each module that belongs to G has been attained through testing in the full group context. The group design matrix is analogous to the design matrix of the core method:

If M contains a call to a module that belongs to an immediate subgroup of G to which M does not belong, let the *group design nodes* for M be the entry node and all such call nodes to modules across subgroup boundaries. If no calls from M cross subgroup boundaries, M has no group design nodes. The *group design matrix* of M is a matrix in which rows correspond to tests and columns correspond to group design nodes, and the entries are the number of times each group design node occurred in each test.

Some minor anomalies are introduced by recursion and calls through function pointers. The core method treats a recursive call of a module by itself as a design node, so that all calls are treated uniformly. Since a module is always in the same groups as itself, the extended method will never treat an explicit self-recursive call as a design node. A minor amendment to the core method to exclude self-recursive calls from the design node set suffices, and we assume this amendment from now on. For example, only the entry node of the following module is now considered to be a design node, not the recursive call:

```
void recurse(int i)
{
    printf("%d\n", i);
    if (i > 0)
        recurse(i-1);
}
```

This is merely a technical detail in the coverage assessment process, which does not require any changes to the software being tested and has a negligible effect on the testing effort.

Dynamic calls are more problematic. Since we cannot in general determine to which groups a dynamic call may belong, we treat all dynamic calls as calls across group boundaries. For example, the call through the function pointer `pfn` in the following module is always treated as a design node, so both paths through the module must be tested at each integration stage regardless of the tests performed at previous stages:

```
void dynamic(void (*pfn)(int), int i)
{
    if (i > 0)
        (*pfn)(i);
    else
        (*pfn)(-i);
}
```

This may lead to extraneous retesting in cases where the dynamic calls remain within group boundaries. In this case, however, we deem it better to err on the conservative side. In some cases, for example menu tables or callback systems, it may be clear to which group a particular dynamic call belongs. In that case, we may include the dynamic reference explicitly as a group member.

Observe that if a module in a group contains calls only to other modules in the same group, and that group has been integrated, then no coverage is required for that module during higher-level integration (except incidental coverage required by calls to it from modules in other groups). Thus, as desired, to integrate a previously-integrated group with itself requires no additional tests. At the other end of the scale, we have the following result:

Proposition 5 *Assuming statically resolved calls, exactly the same sets of tests are sufficient to integrate a program using the core method as to integrate it as a group consisting entirely of single-module groups.*

For a module containing at least one call, the group design nodes are exactly the same as the design nodes, so the criteria are equivalent. For a module containing no calls, the core method indicates that it must be entered at least once by making its design matrix consist of exactly its entry node. While the group method has no explicit coverage requirement for such a module, a call to it must appear in the design matrix of some other module, and thus it must be entered to satisfy that module's coverage criterion. □

5.3.3 Application of groups

We now show how the real-system testing problems fit into the group integration paradigm.

Hierarchical integration is naturally accommodated by the group model. Each component of the design has an associated group, and a high-level system's group consists of the groups of its sub-systems.

Incremental development and re-integration testing also fit easily into the group integration model. Each new or changed module becomes a single-module group, and the entire rest of the system becomes a single group.

We also account for multi-module unit testing. The criterion states that the module-level structured testing criterion must be satisfied for each module "in some context." That context can be at the unit or subsystem level. If it proves difficult to satisfy the module testing criterion completely for certain modules in the minimal desired integration context, these few modules may be taken aside for isolation testing without the overhead of testing all modules in isolation.

5.3.4 Extensions for object-oriented systems

The polymorphic nature of object-oriented programming requires extensions to traditional testing techniques. What looks like a simple function call (or data reference) may cause any of a potentially large set of functions to be called depending on the run-time state of the program. Although function pointers in C present similar problems, polymorphism is so pervasive in object-oriented programs that failure to address it in a testing strategy is unacceptable. The ideas in this section have been published [34, 35], but have not been implemented in an automated system. The extension techniques do not use any special properties of the structured integration testing strategy, so they may be applied to other strategies such as statement testing as well.

The key issue in our object-oriented integration testing extension is managing the *implicit complexity* due to polymorphism. At any polymorphic call site, we can view the implicit complexity as a multiway decision with a different explicit function call

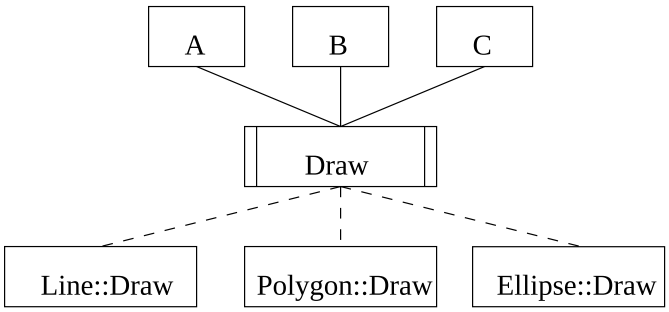


Figure 43: Polymorphism example

at each alternative. There are three major approaches to this implicit complexity: optimistic, pessimistic, and balanced. In the optimistic approach, we ignore implicit complexity entirely, trusting the level of abstraction of the language and in effect only testing the integration between the module and an arbitrary representative of the possible modules to which the polymorphic call statement can be resolved. Such optimism seems unwarranted for typical systems, since the inheritance structure only makes sure that the potential call sites share superficial properties such as name and number of parameters. In the pessimistic approach, we make no use of the abstraction of the language, testing calls to each potential call resolution independently as if the implicit control flow were explicit. This generates an overwhelming number of tests, and seems impractical for real systems. Figure 43 shows the structure of a simple object-oriented program, which we use to illustrate the three approaches [58]. Each of the modules `A`, `B`, and `C` call the polymorphic function `Draw`, which can be resolved dynamically to any of the modules `Line::Draw`, `Polygon::Draw`, or `Ellipse::Draw`. Figure 44 shows the optimistic approach, in which exercising each module and each call site is sufficient. Figure 45 shows the pessimistic approach, in which each possible resolution must be exercised from each call site.

The balanced approach, which we propose, is a compromise that makes some use of the abstraction level of the language. Note that each polymorphic call statement has a corresponding set of possible resolution modules. We wish to gain reasonable confidence that these modules are of equivalent integration testing value, in the sense that any member of the set is of equivalent value in testing a polymorphic call.

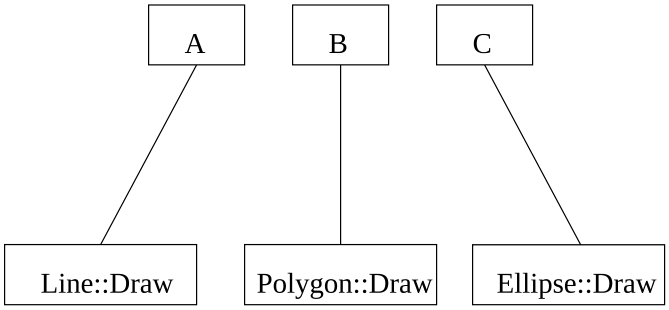


Figure 44: Optimistic approach to testing polymorphism

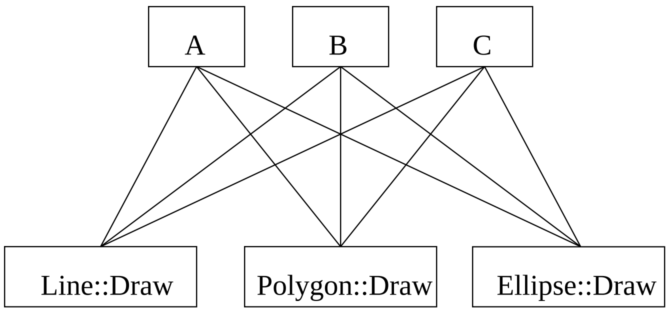


Figure 45: Pessimistic approach to testing polymorphism

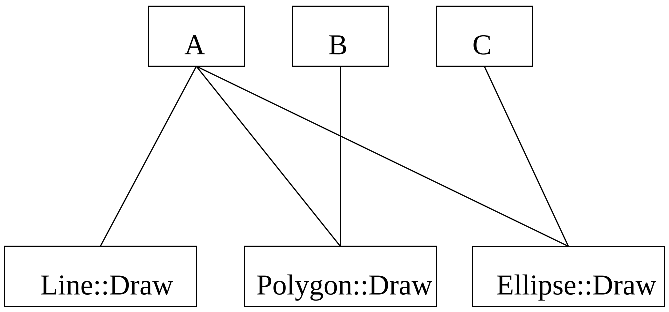


Figure 46: Balanced approach to testing polymorphism

Once we have that confidence, we can follow the optimistic strategy and test as if polymorphism were not used. Figure 46 shows the balanced approach. To gain that reasonable confidence at minimal cost, we feel that it is sufficient for each resolution set to test resolutions to each module in the set from the same call site, which provides evidence that they have equivalent integration testing value. The single module used as the call site for the purposes of establishing the resolution set as an equivalence class need not even be part of the application — it could be a test driver module, in which case the entire integration testing process using the integrated application proceeds as in the optimistic approach. Extending the view of the resolution set as an equivalence class, we could apply transitivity to remove the requirement that a single call site be used to establish equivalence as long as every module in the set is connected to every other one by a chain of being tested from the same call site.

5.4 Comparison with McCabe and Butler's work

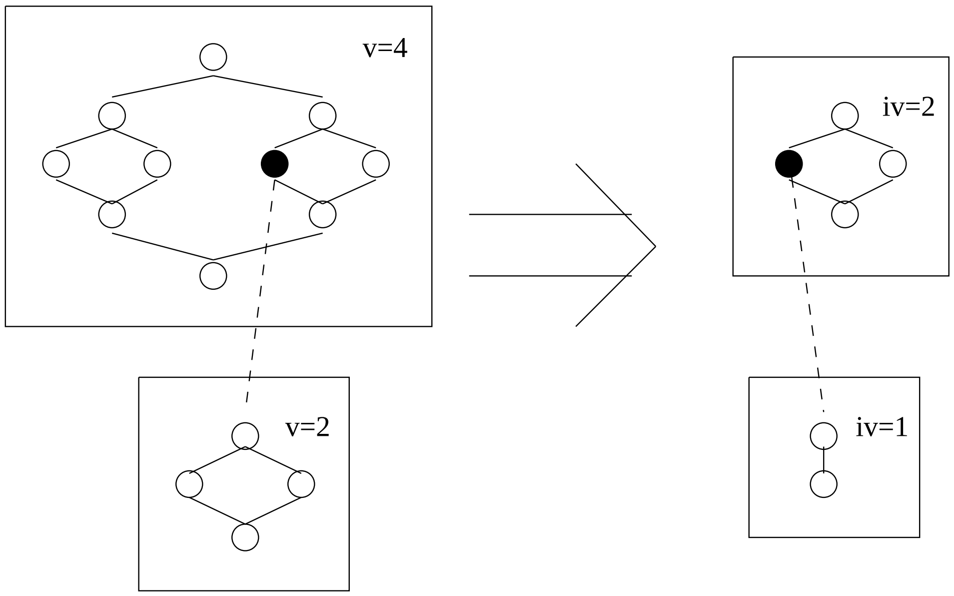
We now give a detailed analysis of McCabe and Butler's techniques [39], and compare them to our method. We first describe their work, including the iv and $S1$ metrics that are conceptually similar to our dc and gdc. We then examine several areas in which our dc improves on their iv. The most striking of these are cases in which the design reduction technique breaks down on extremely unstructured code — for typical code the differences tend to be relatively minor. We then show how iv may be used to approximate dc, in the sense that McCabe and Butler's manual techniques [39] can be used to derive test cases that satisfy our testing criteria and to approximate integration effort from a design.

5.4.1 Description of McCabe and Butler's technique

McCabe and Butler [39] loosely define a *subtree* as the structure chart counterpart of a path through a module flow graph, and their stated integration testing goal is to test a basis set of subtrees in the sense that structured testing at the module level tests a basis set of paths. Given a module flow graph, they define a *design reduced* graph by a set of local restructuring rules intended to eliminate control complexity that does not affect the sequence of module call ("black dot") nodes. The precise rules are shown in Figure 48. They define the *module design complexity, iv(G),* of a module as the cyclomatic complexity of the corresponding design-reduced graph, and the *integration complexity, S1,* as $1 + \sum(iv(G) - 1)$ over all modules in the program. The integration strategy is to generate $S1$ tests manually from the program structure using an inter-module extension of the baseline method [38] on the design-reduced module graphs, and then perform those tests. Figure 47 shows the graphs of a program with two modules, its design-reduced graphs, and its design metrics.

One of the primary suggested uses of $S1$ is to reduce the scheduling uncertainty inherent in integration testing by estimating integration testing effort before coding even begins. McCabe and Butler [39] present a technique for calculating approximate iv and hence $S1$ metrics based on the standard structure chart symbols (conditional call and iterated call), allowing the eventual number of integration tests to be estimated during the design phase.

Although conceptually appealing, this method still has several weaknesses. First, the integration tests must be generated from scratch — no method is given to assess and augment an existing integration test set. This is a much greater weakness in an integration testing strategy than in a module testing strategy, since it is impractical to derive manually the test data to exercise a particular execution path through a real system. Second, the mathematical foundations are not precise. For example, $S1$ is not the cardinality of "a basis set of subtrees" for any reasonable interpretation of the



(a) Before design reduction

(b) After design reduction

Black nodes and dashed lines denote procedure calls

Figure 47: Design reduction example for program with S1 = 2

definition of subtree. One consequence is that there is no natural way to remedy the first weakness. Third, intermodule control coupling can render the criterion unattainable, although this is less of a problem than in McCabe's earlier technique [38] due to the simplifying influence of design reduction. Fourth, the test generation technique is presented only through an example without iteration or recursion, with no attempt to address the general case. Our technique shares none of those weaknesses.

5.4.2 Analysis of iv

Recall that the *iv* design complexity measure [39] is defined as the cyclomatic complexity of the "design-reduced" flow graph, which is calculated through iterative application of four local restructuring rules to the flow graph. Figure 48 shows the design reduction rules [39]. Care must be taken to apply the loop rule only in situations where it preserves reachability from the entry to each node and from each node to the exit. Figure 49 shows a case where the loop rule fails to apply due to global reachability considerations, despite matching the local rule template. Reachability is trivially preserved by the other rules. No rule can increase cyclomatic complexity, since each removes at least as many edges as nodes from the graph. The main loop of the reduction method terminates after at most a linear number of iterations in the size of the graph, since each reduction removes either an edge or a node from the graph.

The *iv* design complexity measure has several weaknesses, mostly due to the local nature of its reduction rules, which make them inapplicable to many types of unstructured code.

First, using *iv*, modules without call nodes can have unbounded complexity. In contrast, modules without call nodes have bounded complexity using dc ($dc = 1$).

Proposition 6 *There are acyclic graphs, with no call nodes, that have arbitrarily high iv.*

Figure 50 shows an acyclic flow graph with no call nodes. None of the design reduction rules apply, so this graph has $iv = v = 4$. An arbitrarily complex graph may be formed

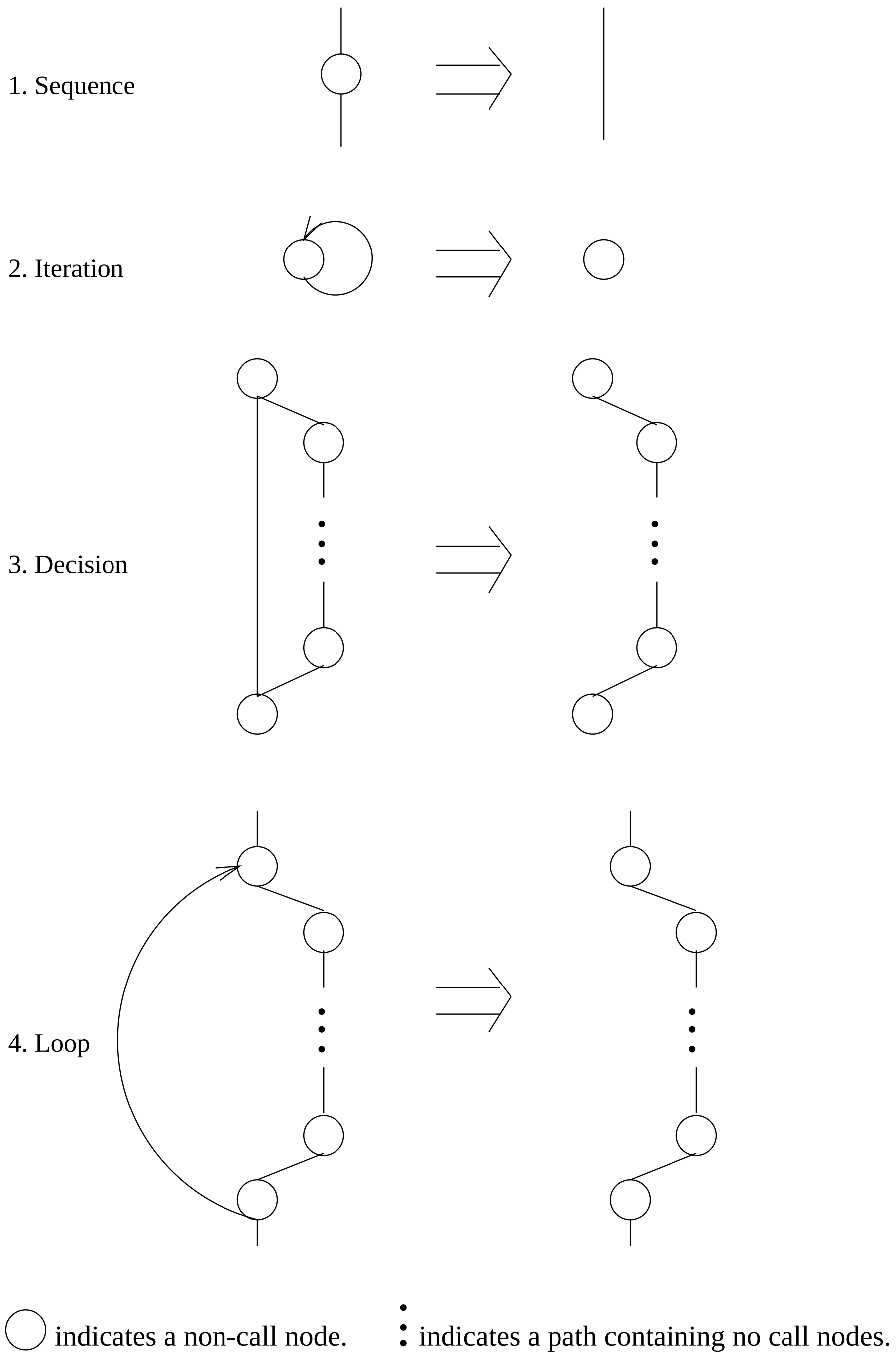


Figure 48: Design reduction rules

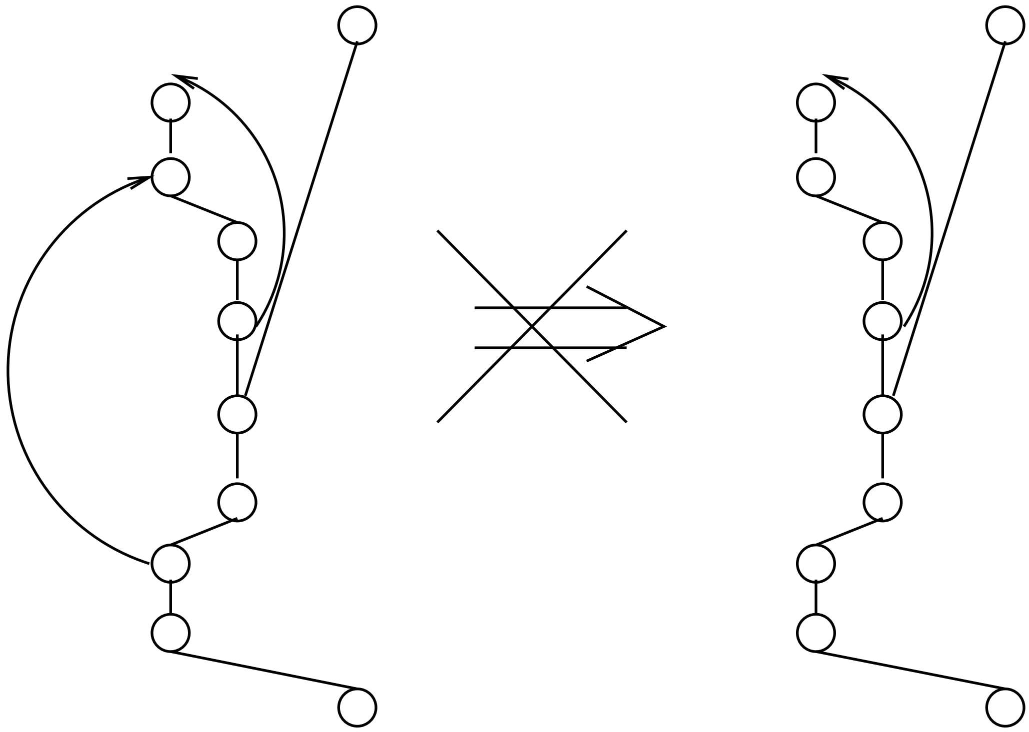


Figure 49: Exception to loop rule

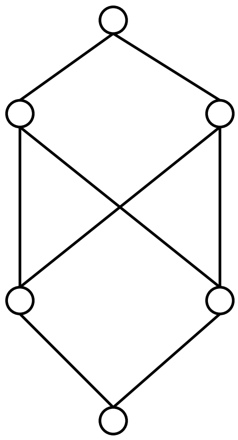


Figure 50: Graph with excess *iv*

by connecting this graph as components in sequence, and the rules still do not apply. □

On the other hand, design reduction does much better on typical graphs:

Proposition 7 *Structured graphs with no call nodes have* $iv = 1$*.*

The design reduction rules mirror the basic structured programming constructs for code without function calls, and may be applied from the deepest nesting level outward to reduce a structured graph to a single edge connecting the entry to the exit. □

Second, *iv* is not well-defined. In contrast, *dc* is well-defined, it is the rank of a matrix determined by the module flow graph.

Proposition 8 *There are graphs for which the iv measure depends on the order of application of the reduction rules.*

Figure 51 shows a flow graph with $dc = 3$ for which *iv* is not well-defined. Applying the loop version of Rule 4 to eliminate edge `A` leaves a graph that cannot be reduced further, implying that $iv = 4$. If instead we apply the loop version of Rule 4 to eliminate edge `B`, we may apply the decision version of Rule 4 to eliminate edge `C`, reaching the reduced graph shown in Figure 52, implying that $iv = 3$. □

Even for structured acyclic graphs, *iv* can be twice as high as *dc*.

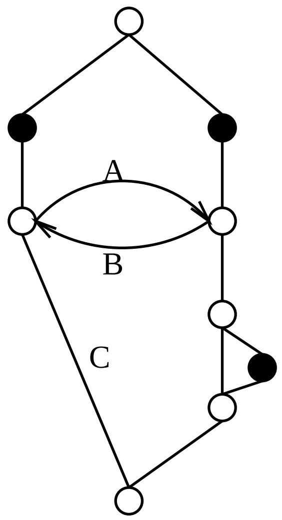


Figure 51: Graph without well-defined iv

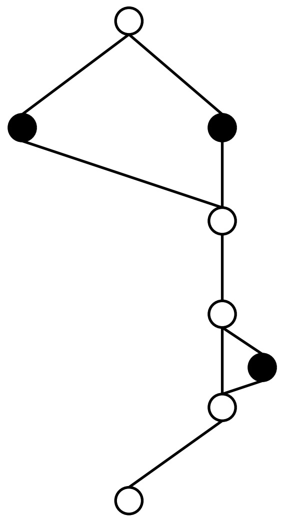


Figure 52: Alternate reduction

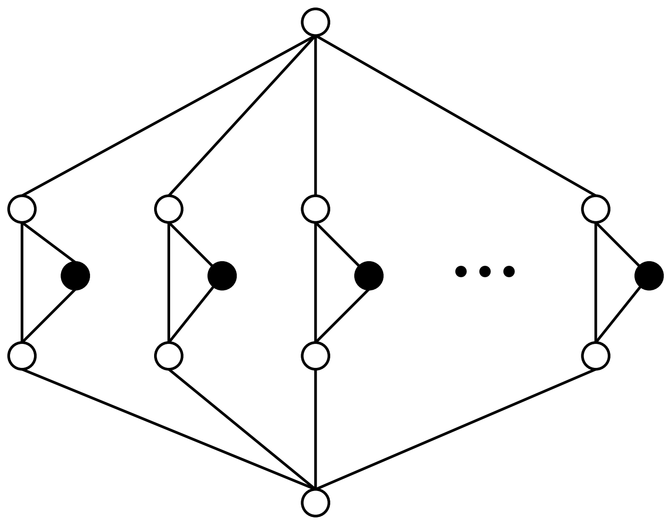


Figure 53: Graph with $iv = 2 * dc - 2$

Proposition 9 *There are arbitrarily large graphs for which $iv = 2 * dc - 2$.*

Figure 53 shows such a class of flow graphs. Let the number of branches of the initial decision construct be n. Since no reduction rules apply, this graph has $iv = v = 2n$. To calculate dc, observe that each of the n call nodes and the entry may appear independently, so $dc = n + 1$. Thus $iv = 2 * dc - 2$ for any such graph. □

Whenever there is a discrepancy between iv and dc, iv is the more conservative. A major strength of iv, which we will discuss in detail in Section 5.4.3, is that design reduction preserves dc.

Theorem 11 *None of the design reduction operations affect dc.*

Let G be the flow graph, E be the edge removed, A be the edge's input vertex, and B be the edge's output vertex. For the sequence and iteration rules the result is trivial. For the conditional rule, let W be a path from A to B that does not intersect E or any call nodes. Then any path P through E has the same call vector as the path P' through $G \setminus \{E\}$ where E is replaced by W. Thus the set of call vectors is unchanged by the rule, so dc is unchanged. For the loop rule, let W be a path from B to A that does not intersect E or any call nodes. We may consider a path P on which E occurs exactly once without loss of generality, since each path through E may be expressed as a linear combination of such paths and paths that avoid E. Decompose P into $I + E + T$, where I is the initial path segment from the entry to A and T is the terminal path segment from B to the exit. Since the graph must maintain the module reachability properties for the reduction rule to apply, there are path segments N and X that do not intersect E such that N connects the entry to B and X connects A to the exit. Figure 54 illustrates the relationship between A, B, E, W, I, T, N, and X. Consider the following paths, none of which intersect E:

$$\begin{aligned} P1 &= I + X \\ P2 &= N + T \\ P3 &= N + W + X \end{aligned}$$

Let V be the vector in $G \setminus \{E\}$ corresponding to P1 + P2 - P3. Then

$$\begin{aligned} V &= P1 + P2 - P3 \\ &= I + X + N + T - (N + W + X) \\ &= I - W + T \end{aligned}$$

which has the same call vector as $I + E + T = P$, since neither W nor E contain any call nodes. Thus the set of call vectors is unchanged by the rule, so *dc* is unchanged. □

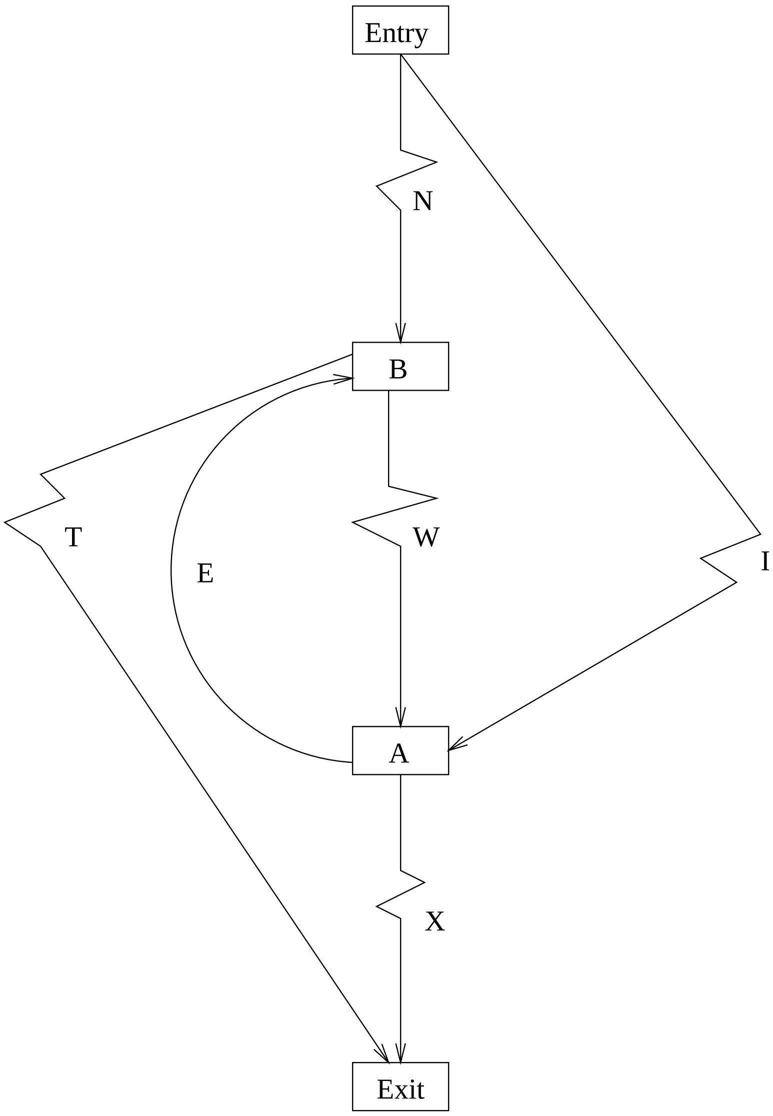
Corollary 2 $iv \geq dc$. □

Corollary 3 $S1 \geq gdc$.

$S1 = 1 + \sum(iv - 1) \geq 1 + \sum(dc - 1) = gdc$. □

5.4.3 Approximations

For typical modules, *iv* is close to *dc*. Since $iv \geq dc$, this means that *iv* may be used as a good upper bound to estimate *dc* and thus estimate the integration effort for each module. Additionally, McCabe and Butler's techniques [39] are well-suited to manual application — in the absence of automated tools it is often easier to deal



E: the single edge removed by the loop reduction rule

W: the white-dot path (no call nodes) required for the rule to apply

All other paths are arbitrary (may have call nodes)

Figure 54: The loop reduction rule does not affect dc

with the graphical design reduction rules than the matrix methods of our technique, especially for moderately large modules. To derive a sufficient set of integration paths through a particular module, we may start with the design-reduced graph instead of the original graph, since the design reduction rules do not affect *dc*. Thus, we may use the technique for deriving module-level integration paths proposed by McCabe and Butler [39] (the unit paths through the design-reduced graph) as a safe approximation to our technique.

A major contribution of McCabe and Butler's [39] is the technique for estimating integration effort based on a structured design, before the coding phase of development begins. Such an approximation is of critical importance to project scheduling, since it is frequently impractical to expand a project schedule during the coding phase. The technique is to assign a design complexity to each box on the structure chart equal to one plus the number of design predicate symbols (decision and iteration) shown on call lines coming out of the box. This is used as a lower bound for the module design complexity of the code that eventually implements the box. This same number also bounds *dc*.

We extend this design-level approximation technique to encompass group integration. First, annotate the boxes with group membership information. Then, for each box, count the number of predicate symbols that annotate lines that cross group boundaries, and add one if there are any. This gives a lower bound for the group design complexity of the code that will implement the box. For a leveled integration strategy, the same chart may be used to calculate the group design complexity for each module at each level of integration, for a lower bound on the total integration effort for all phases of the project.

5.5 Conclusion

We have extended the structured testing approach to support integration testing, and described an implementation to automate our technique. We proposed an approach to generalize integration testing techniques to support realistic incremental integration strategies, and discussed the application of this approach to our technique. We also gave a detailed analysis and comparison with related work by McCabe and Butler [39].

Chapter 6

Conclusions

Although many years have passed since structured testing was first proposed, very little has been published about it, and it is often criticized. This dissertation has examined structured testing in detail, while clarifying and extending it as necessary to overcome frequent criticisms.

One criticism of structured testing questions whether testing for correct behavior on a set of paths that generates all possible paths by linear combination is an effective shortcut for testing all paths. In Chapter 2, we explored the intuition behind structured testing in two qualitatively different ways. First, we exhibited an infinite class of programs for which structured testing is the ideal testing strategy, and placed structured testing in a subsumption hierarchy with several other major testing criteria. Then, we showed that structured testing, when restricted to executable paths, satisfies Weyuker's proposed axioms [60, 62] for good testing criteria. In Chapter 4 we presented empirical results indicating that structured testing performed significantly better at detecting errors than the widely used branch coverage criterion, and at least comparably to the widely studied all-uses data flow coverage criterion. However, as expected for a purely structural criterion, structured testing was far from perfect, failing to detect an error in at least one experimental trial for all but one of the test programs.

Another criticism is that structured testing requires excessive effort to perform, since the criterion must be applied manually. In Chapter 3, we described a method of automating structured testing that eliminates most of the manual work. The automated system allows structured testing to be applied as a supplement to other testing techniques rather than as a separate activity.

A final weakness of structured testing is that it is directed solely towards single-module testing, and does not address the larger problem of integration testing. In Chapter 5, building upon the work of McCabe and Butler [39], we described a mathematically rigorous extension of structured testing to support integration testing. Our integration testing criterion is sufficiently flexible to adapt naturally to several incremental approaches to developing and testing large systems. We also described an efficient method of automating this criterion.

During our analysis, we found that the mathematical foundations of structured testing are sound, and that typical criticisms can be addressed satisfactorily. Our empirical work indicates that structured testing is effective at detecting errors compared to some commonly used techniques. In addition, the structured testing approach can be applied to integration testing without any artificial restrictions on the integration process. We hope that the work presented in this dissertation will lead to structured testing becoming a well-understood member of the growing set of techniques available to testing practitioners for use on real projects.

6.1 Future work

There are several opportunities for future work in structured testing. One obvious area is adding automated support for other languages and language features. Another is dependency analysis, in which the specific control flow dependencies that prevent the structured testing criterion from being satisfied are isolated and displayed. Finally, moving from a linear control flow trace file to an architecture in which path vectors are kept locally for each module and written atomically using a caching mechanism can support multithreaded programs, while giving substantial performance improvements.

6.1.1 Other language features

One area for future work is in automating structured testing for other languages. While most languages have some features that provide interesting instrumentation challenges, the most notable feature is exception handling. Languages such as Ada and PL/1 offer significant translation problems within our simplified control flow model, and even giving a reasonable definition of cyclomatic complexity in the presence of implicit control flow is difficult. Handling ambiguous control is a more promising area. As it occurs in C (for example, `(i && j) + (k && l)`), cyclomatic complexity remains well-defined. Therefore it seems likely that with an appropriate representation of ambiguous flow graphs, the path recovery algorithm could be extended to at least calculate the tested rank for such graphs as an aid to structured testing. Alternatively, analysis of the object code could resolve any ambiguities.

6.1.2 Dependency analysis

If all possible data has been run and the tested rank remains less than the cyclomatic complexity, then there is some data dependency that constrains the path space in the flow graph. McCabe suggests that in this case the module can and should be restructured to remove the dependency, or modularized so the dependency falls across a module boundary. The effort of such restructuring and the risk of introducing errors tend to make it impractical on real projects. Simply identifying such dependencies to set goals for tested rank is a more realistic objective. Developing tools to assist users in isolating such dependencies during maintenance would be worthwhile. Since it is unreasonable to expect all possible data to be run, the dependency information must be built up heuristically from the observed behavior of the system.

It is possible to express observed control flow dependencies as linear equations in terms of the number of times each branch was executed. An example dependency is “branch 3 has been executed twice as often as branch 5 on every path that has been executed.” These equations can be calculated using a two-step process. First, select a column basis for all possible paths through the graph, which can be done by starting with all branch columns, generating a basis set of paths, and performing column reduction. This column reduction produces (and eliminates from further consideration) all "trivial" dependencies due to the conservation of flow equations for the flow graph. Second, perform a column reduction in the independent tested paths matrix restricted to the basis columns from step one. This produces a representation for each "non-trivial" dependency, which is unique up to the choice of column basis in the first step. However, the representation may not be minimal in terms of the number of nonzero co-efficients in the dependency equation, for example a single unexecuted column that is not in the selected column basis will have its dependency information represented in terms of the column basis using the appropriate conservation of flow equation. Efficient calculation of the minimal representations of dependency equations is an open problem.

6.1.3 Multitasking support and performance improvements

Tracing multitasking applications requires special effort, since events from different threads are interleaved. Although tagging each event with a thread identifier or maintaining a separate trace file for each thread solves this problem in theory, it requires significant interaction with the underlying execution environment. Also, multitasking applications often have "busy-wait" loops. Tracing such loops would cause vast numbers of events to be written to the trace file while the thread waited for some external event. We are exploring an alternate front-end architecture within the context of McCabe's commercial version of the system to solve these problems. Early results indicate that the run-time overhead of this alternate architecture can be dramatically lower even for single-threaded systems.

The basic idea is to write trace information for each module invocation atomically rather than writing trace events as they happen. Since we only need the number of times each branch has been executed along each activation to produce the independent tested paths matrix for the path analyzer, we store this information rather than the complete path traces. The simplest way to do this is to keep a local array of branch execution counters within each module, set and increment the counters at the same instrumentation points as the original system, and then transmit the array to the instrumentation library just before the module returns to be written to the trace file. To ensure that each activation's "trace" information is atomic, the instrumentation library must provide exclusive access to the trace file, and no explicit thread identification is needed. Note that this can give a significant reduction in trace volume, since for example going through a loop a million times generates the number one million as one element of the activation's trace vector rather than writing one million copies of the loop's test into the trace file. However, if an instrumented module is called within the loop, each iteration still generates an activation vector for that called module. To reduce the trace volume in this case, we can keep a cache of trace vectors for each instrumented module. The prototype uses write-through least-frequently-used caches with size equal to each module's number of branches. In preliminary testing, the trace files were reduced to essentially constant size by this technique, suggesting that the same few path vectors repeated many times tend to make up nearly all of each module's contribution to the complete execution trace.

Instrumentation at the source level for the above architecture is complicated by the necessity of transmitting an activation's branch vector immediately prior to the module's return. Simply instrumenting return statements is inadequate, since the returned expression may contain control flow events, for example:

```
return s ? strlen(s) : 0;
```

One possible approach is to declare a temporary variable of the same type as the function's return value and restructure the return statement to use this temporary value, for example:

```
{ char *temp; temp = s ? strlen(s) : 0; return temp; }
```

The branch vector can then be transmitted before the new return statement. Another approach, which may be more language-independent since it does not rely on data type analysis, is to traverse each flow graph at parse time and generate a table indicating which branches immediately precede the module exit. The branch vector can then be transmitted as part of incrementing the counter for the last branch before the return, rather than instrumenting the actual site of the return.

Bibliography

[1] Adrion, W., Branstad, M., and Cherniavsky, J. Validation, verification, and testing of computer software. *ACM Computing Surveys 14*, 2 (June 1982), 159–189.

[2] Ball, T., and Larus, J. R. Optimally profiling and tracing programs. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Albuquerque, NM, Jan. 1992), pp. 59–70.

[3] Berge, C. *Graphs and Hypergraphs.* North-Holland, Amsterdam, The Netherlands, 1973.

[4] Brilliant, S., Knight, J., and Leveson, N. The consistent comparison problem in n-version software. *ACM Software Engineering Notes 12*, 1 (Jan. 1987), 29–34.

[5] Chen, T., and Yu, Y. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering 20*, 12 (Dec. 1994), 977–980.

[6] Cherniavsky, J., and Smith, C. On Weyuker's axioms for software complexity measures. *IEEE Transactions on Software Engineering 17*, 6 (June 1991), 636–638.

[7] Chilenski, J., and Miller, S. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal 9*, 5 (Sept. 1994), 193–200.

[8] Clarke, L., Podgurski, A., Richardson, D., and Zeil, S. A comparison of data flow path selection criteria. In *Proceedings of the Eighth International Conference on Software Engineering* (August 1985), pp. 244–251.

[9] DeMillo, R., Lipton, R., and Sayward, F. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (Apr. 1978), 34–41.

[10] DeMillo, R. A. Progress toward automated software testing. In *Proceedings of the Fourteenth International Conference on Software Engineering* (1991), pp. 180–183.

[11] Duran, J., and Ntafos, S. A report on random testing. In *Proceedings of the Fifth International Conference on Software Engineering* (San Diego, CA, March 1981), pp. 179–183.

[12] Evangelist, M. An analysis of control flow complexity. In *Proceedings of the Computer Software and Applications Conference* (1984), pp. 388–396.

[13] Frankl, P. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD thesis, New York University, 1987.

[14] Fraser, C. W., and Hanson, D. R. A retargetable compiler for ANSI C. *SIGPLAN Notices 26*, 10 (Oct. 1991), 29–43.

[15] Garey, M., and Johnson, D. *Computers and Intractability.* Freeman, New York, 1979.

[16] Girard, E., and Rault, J. A programming technique for software reliability. In *Proceedings of the 1973 IEEE Symposium on Computer Software Reliability* (1973), pp. 44–50.

[17] Hanson, D. R. Printing common words. *Communications of the ACM 30*, 7 (July 1987), 594–599.

[18] Heimann, D. The link between complexity and defects in software - an analysis. In *Proceedings of IEEE 31st Annual Spring Reliability Seminar* (April 1993).

[19] Herman, P. A data flow analysis approach to program testing. *The Australian Computer Journal 8*, 3 (Nov. 1976), 92–96.

[20] Hetzel, W. C. *The Complete Guide to Software Testing.* QED Information Sciences, Wellesley, MA, 1984, ch. 6.

[21] Hoare, C. Quicksort. *Computer Journal 5*, 1 (1962), 10–15.

[22] Horgan, J., London, S., and Lyu, M. Achieving software quality with testing coverage measures. *IEEE Computer* (Sept. 1994), 60–69.

[23] Horgan, J. R., and London, S. Data flow coverage and the C language. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Oct. 1991), pp. 87–97.

[24] Howden, W. Functional program testing. *IEEE Transactions on Software Engineering 6*, 2 (Mar. 1980), 162–169.

[25] Howden, W. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering 8*, 4 (July 1982), 371–379.

[26] Hutchins, M. et al. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering* (Sorrento, Italy, May 1994), pp. 191–200.

[27] Ledgard, H. F., and Marcotty, M. A genealogy of control structures. *Communications of the ACM 18*, 11 (Nov. 1975), 629–639.

[28] Mathur, A., and Wong, W. Comparing the error detection effectiveness of mutation and data flow testing: an empirical study. *SERC-TR-146-P* (Sept. 1993).

[29] Mathur, A., and Wong, W. An empirical comparison of mutation and data flow-based test adequacy criteria. *SERC-TR-135-P* (Mar. 1993).

[30] McCabe & Associates. *C parser (unpublished source code).* Columbia, MD, 1990.

[31] McCabe & Associates. *Analysis of Complexity Tool User's Manual.* Columbia, MD, 1991.

[32] McCabe & Associates. *McCabe Instrumentation Tool User's Manual.* Columbia, MD, 1993.

[33] McCabe & Associates. *COBOL parser (unpublished source code).* Columbia, MD, 1995.

[34] McCabe, T., Dreyer, L., Dunn, A., and Watson, A. Testing an object-oriented application. *Journal of the Quality Assurance Institute 8*, 4 (Oct. 1994), 21–27.

[35] McCabe, T., and Watson, A. Combining comprehension and testing in object-oriented development. *Object Magazine 4*, 1 (March-April 1994), 63–66.

[36] McCabe, T., and Watson, A. Software complexity. *CrossTalk: The Journal of Defense Software Engineering 7*, 12 (Dec. 1994), 5–9.

[37] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering 2*, 4 (Dec. 1976), 308–320.

[38] McCabe, T. J. *NBS Special Publication 500-99, Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.* U.S. Government Printing Office, Washington, D.C., Dec. 1982.

[39] McCabe, T. J., and Butler, C. W. Design complexity measurement and testing. *Communications of the ACM 32*, 12 (Dec. 1989), 1415–1425.

[40] Myers, G. *The Art of Software Testing.* Wiley, 1979.

[41] Neuder, D. L. A test verification tool for C and C++ programs. *Hewlett-Packard Journal 42*, 2 (Apr. 1991), 83–92.

[42] Ntafos, S. An evaluation of required element testing strategies. In *Proceedings of the Seventh International Conference on Software Engineering* (Mar. 1984), pp. 250–256.

[43] Offutt, A., and Lee, S. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering 20*, 5 (May 1994), 337–344.

[44] Omar, A., and Mohammed, F. A survey of software functional testing methods. *Software Engineering Notes 14*, 2 (Apr. 1991), 62–70.

[45] Ostrand, T. J., and Weyuker, E. J. Data flow-based test adequacy analysis for languages with pointers. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Oct. 1991), pp. 74–86.

[46] Pande, H., Ryder, B., and Landi, W. Interprocedural def-use associations in C programs. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Oct. 1991), pp. 139–153.

[47] Parrish, A., and Zweben, S. Analysis and refinement of software test data adequacy properties. *IEEE Transactions on Software Engineering 17*, 6 (June 1991), 565–581.

[48] Prather, R. Theory of program testing - an overview. *The Bell System Technical Journal 62*, 10 (Dec. 1983), 3073–3105.

[49] Pressman, R. S. *Software Engineering A Practitioner's Approach 3rd Edition.* McGraw-Hill, New York, NY, 1992, ch. 18,19.

[50] Rapps, S., and Weyuker, E. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering 11*, 9 (Apr. 1985), 367–375.

[51] Tai, K.-C. A program complexity metric based on data flow information in control graphs. In *Proceedings of the Seventh International Conference on Software Engineering* (Mar. 1984), pp. 239–248.

[52] Tai, K. C. What to do beyond branch testing? *Software Engineering Notes 14*, 2 (Apr. 1989), 58–61.

[53] Thayer, R., Lipow, M., and Nelson, E. *Software Reliability*. North-Holland, 1978.

[54] Walsh, T. J. A reliability study using a complexity measure. In *AFIPS Conference Proceedings* (New York, NY, 1979), AFIPS Press, pp. 761–768.

[55] Ward, W. T. Software defect prevention using McCabe's complexity metric. *Hewlett-Packard Journal* (Apr. 1989), 64–69.

[56] Watson, A. Why basis path testing? *McCabe & Associates Outlook* (Nov. 1994), 6–7.

[57] Watson, A. McCabe complexity in software development. *Systems Development Management 34-40-35* (1995).

[58] Watson, A., and McCabe, T. *NIST Special Publication 500-235, Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*. U.S. Department of Commerce/National Institute of Standards and Technology, Washington, D.C., 1996.

[59] Weinberger, P. J. Cheap dynamic instruction counting. *Bell System Technical Journal 63*, 8 (Oct. 1984), 1815–1826.

[60] Weyuker, E. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering 12*, 12 (Dec. 1986), 1128–1138.

[61] Weyuker, E. Evaluating software complexity measures. *IEEE Transactions on Software Engineering 14*, 9 (Sept. 1988), 1357–1365.

[62] Weyuker, E. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM 31* (June 1988), 668–675.

[63] WEYUKER, E., GORADIA, T., AND SINGH, A. Automatically generating test data from a Boolean specification. *IEEE Transactions on Software Engineering 20*, 5 (May 1994), 353–363.

[64] WEYUKER, E., WEISS, S., AND HAMLET, R. Comparison of program testing strategies. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Oct. 1991), pp. 1–10.

[65] ZUSE, H. *Software Complexity: Measures and Methods.* de Gruyter, Berlin, 1990.

[66] ZWEBEN, S., AND GOURLAY, J. On the adequacy of Weyuker's test data adequacy axioms. *IEEE Transactions on Software Engineering 15*, 4 (Apr. 1989), 496–501.

Appendix A

Code for Test Programs and Libraries

This Appendix contains the source code for the functions used in the experiments described in Chapter 4, except for the `find` function that is shown in Figure 29 as part of the discussion in Chapter 4. It also contains the source code for the common words program [17] used in the examples in Chapters 3 and 5. It also contains the source code for the instrumentation libraries used in Chapters 3 and 5.

A.1 Code for "blackjack" ("hand()")

```
extern int cards[52];
extern int i;      /* card index */
extern int dshow;  /* hack, card dealer is showing */
int win, p, count; /* so checking for bug
                    * doesn't affect all-uses coverage */

int
hand()
{/* return win */
    int d, pace, dace;
    int k;    /* 1 for hit, zero for not hit */
```

```
    p = 0;
    d = 0;
    pace = 0;
    dace = 0;
    win = 0;
/* win will be 0 if dealer wins, 1 if player wins, 2 if a push */
    hit(&p, &pace);
    hit(&d, &dace);
    hit(&p, &pace);
    hit(&d, &dace);
    count = 0;
    printf("DEALER SHOWS ---  %d\n", cards[i-1]);
    dshow = cards[i-1];
    printf("PLAYER = %d, NO OF ACES - %d\n", p, pace);
    if (p == 21) {
        printf("PLAYER HAS BLACKJACK\n");
        win = 1;
    } else {
        count = 2;
L11:
        check_for_hit(&k, p, count);
        if (k == 1) {
            hit(&p,&pace);
            count += 1;
            printf("PLAYER = %d, NO OF ACES - %d\n", p, pace);
            if (p > 21) {
                printf("PLAYER BUSTS - DEALER WINS\n");
                goto L13;
            }
            goto L11;
        }
    }
/* Handle blackjack situations, case when dealer has blackjack */
    if (d == 21) {
        printf("DEALER HAS BJ\n");
        if (win == 1) {
            printf("------ PUSH\n");
            win = 2;
            goto L13;
        } else {
```

```
            printf("DEALER AUTOMATICALLY WINS\n");
            goto L13;
        }
    } else {
/* case where dealer doesn't have blackjack:
 * check for player blackjack or five card hand
 */
        check_for_bug();
        if (p == 21 || count >= 5) {
            printf("PLAYER AUTOMATICALLY WINS\n");
            win = 1;
            goto L13;
        }
    }
    printf("DEALER HAS %d\n", d);
L12:
    if (d <= 16) {
        hit(&d,&dace);
        if (d > 21) {
            printf("DEALER BUSTS - PLAYER WINS\n");
            win = 1;
            goto L13;
        }
        goto L12;
    }
    printf(" PLAYER = %d  DEALER = %d\n", p, d);
    if (p > d) {
        printf("PLAYER WINS\n");
        win = 1;
    } else
        printf("DEALER WINS\n");
L13:
    return win;
}
```

A.2 Code for "cobol" ("make_cobol_string()")

```
static char separators[] = "\n\t .();,=\"\'"; /* poss. separators */
```

```
char **
make_cobol_string(szString)
char *szString;
{/* Given szString, return a vector of strings that are the lines of
  * the COBOL version of the string (add spaces, cont chars, quotes)
  */
  int iLen;        /* length of section of line we expect to write */
  int iMaxLen;     /* maximum num characters we can fit on a line */
  int bInString;   /* flag: TRUE = In string, FALSE = not in string */
  int iStrLen;     /* the length of the passed in string */
  char *pStartPos; /* the starting point of next section to write */
  char cHold;      /* temp to save char at end of segment to write */
  char cQuote;     /* delimiting quote character found on a string */
  char *bufTmp;    /* temporary buffer for output lines */
  char **vecRV;    /* vector of output lines to return */
  char szStartQuote[2]; /* proper quote for next line, also as flag */
  char szContinue[2]; /* continue char for next line, also as flag */

  bufTmp = buf_get();
  vecRV = BuildVec((char *)0);
  bInString = 0;
  iLen = 0;
  iMaxLen = 61;
  szStartQuote[0] = szStartQuote[1] = '\0';
  szContinue[0] = ' '; szContinue[1] = '\0';
  pStartPos = szString;
  while( *pStartPos )
  {
    iStrLen = strlen( pStartPos );
    if( iStrLen > ( iMaxLen - !!szStartQuote[0] ))
    {
      iLen = 1;
      while( iLen <= ( iMaxLen - !!szStartQuote[0] ) )
      {
        if(pStartPos[iLen-1]=='\'' || pStartPos[iLen-1]=='\"')
        {
          if( !bInString )
          {
            cQuote = pStartPos[iLen-1];
```

```
          bInString = 1;
        }
        else if (pStartPos[iLen-1] == cQuote)
        {
          if( pStartPos[iLen] != pStartPos[iLen-1] )
            bInString = 0;
          else
            if( iLen < (iMaxLen - !!szStartQuote[0]))
              iLen++;
        }
      }
      iLen++;
    }
    iLen--;
    /* write current line -- change if user-specified columns */
    cHold = pStartPos[iLen];
    pStartPos[iLen] = '\0';
    sprintf( bufTmp, "      %s    %s%s",
             szContinue, szStartQuote, pStartPos );
    AppendToVec(bufTmp, &vecRV);
    pStartPos[iLen] = cHold;
    szContinue[0] = ' ';
    szStartQuote[0] = '\0';
    if (bInString)
    {
      szContinue[0] = '-'; /* continue string */
      szStartQuote[0] = cQuote; /* use proper quote */
      bInString = cHold != cQuote; /* hack embedded quotes */
    }
    else
      if (!strchr(separators, pStartPos[iLen-1]) &&
          !strchr(separators, pStartPos[iLen]))
        szContinue[0] = '-'; /* continue non-string */
  }
  else
  {/* short line (or remainder) so just write it out */
    sprintf( bufTmp, "      %s    %s%s",
             szContinue, szStartQuote, pStartPos);
    AppendToVec(bufTmp, &vecRV);
    iLen = iStrLen;
```

```
    }
    pStartPos += iLen;
  }
  buf_put(bufTmp);
  return vecRV;
}
```

A.3 Code for “comment” (“eatcomment()”)

```
eatcomment()
{
    /* having seen the initial slashstar of a comment, skip over past
     * the terminal starslash
     * Was the following lex pattern in the old parser:
     */
#if 0
     "/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"
#endif

    int c; /* next character of input */
    int iState; /* 0 => need starslash, 1 => need slash, 2 => done */

    iState = 0;
    while (iState != 2) {
        c = input();
        if (iState == 0 && c == '*')
            iState = 1;
        else
            if (iState == 1 && c == '/')
                iState = 2;
            else
                iState = 0;
        if (!c)       /* Hack - Allow EOF to terminate comments */
            iState = 2;
    }
}
```

A.4 Code for "position" ("position()")

```
int a[20];
int size = 5;    /* array is [1..5] */
int max;
int theirpos;

int
position()
{
    int pos, sum, i;
    sum = 0;
    pos = 0;
    for (i = 1; i <= 5; i++) {
        sum = sum + a[i];
        if (sum > max)
        {
            pos = i - 1;
            break;
        }
    }
    return pos;
}
```

A.5 Code for "sort" ("sort()")

```
int a[20];
int size;    /* this will be n */
int a_saved[20];

void sort(a, n)
int a[];       /* 1..n indexed */
int n;
{
    int sortupto;
    int maxpos;
    int mymax;
```

```
    int index;

    sortupto = 1;
    maxpos = 1;

    while (sortupto < n)
    {
        mymax = a[sortupto];
        index = sortupto + 1;

        while (index <= n)
        {
            if (a[index] > mymax)
            {
                mymax = a[index];
                maxpos = index;
            }
            index = index + 1;
        }
        index = a[sortupto];
        a[sortupto] = mymax;
        a[maxpos] = index;
        sortupto = sortupto + 1;
    }
}
```

A.6 Code for "stat" ("stat()")

```
int a[20];
int size = 5;    /* array is [1..5] */
int xmin, xmax, sum;

void stat(a,n)
int a[];
int n;
{
    int i;
    sum = 0; xmin = 0;
```

```
    xmax = a[1];
    for ( i = 2; i <= n; i++) {
        if (a[i] < xmin)
            xmin = a[i];
        if (a[i] > xmax)
            xmax = a[i];
    }
    for (i = 1; i <= n; i++)
        sum = sum + a[i];
}
```

A.7 Code for "strmatch" ("stringmatch2()")

```
char text[20];
char pattern[20];
int textlen;
int patlen;
int answer;    /* the last return value from stringmatch2 */

int stringmatch2(pattern, text, patlen, textlen)
char pattern[];
char text[];
int patlen, textlen;
{
    int patpos, textpos;
    patpos = 1;
    textpos = 1;

    while ( (patpos <= patlen) && (textpos <= textlen))
    {
        if (pattern[patpos] == text[textpos])
        {
            textpos = textpos + 1;
            patpos = patpos + 1;
        } else {
            textpos = (textpos - patpos) + 2;
            patpos = 1;
        }
```

```
    }
    if (patpos > patlen)
        return (textpos - patlen);
    else
        return 0;
}
```

A.8 Code for “triangle” (“triangle()”)

```
/* classify triangles.  The return codes are:
 * 0 = not in order, 1 = right, 2 = obtuse, 3 = acute,
 * 4 = isoceles, 5 = equilateral
 */

int
triangle(a, b, c)
int a, b, c;
{
    int d;
L5:
    if ((a >= b) & (b >= c))
        goto L100;
    return 0;
L100:
    if (b == c)
        goto L500;
    a = a * a;
    b = b * b;
    c = c * c;
    d = b + c;
    if (a != d)
        goto L200;
    return 1;
L200:
    if (a < d)
        goto L300;
    return 2;
L300:
```

```
    return 3;
L500:
    if ( (a == b) & (a == c) )
        goto L600;
    return 4;
L600:
    return 5;
}
```

A.9 Code for common words program

```
/* common: find the k most common words in the standard input */

#define alloc calloc
#include <stdio.h>
#define MAXWORD 100+1
#include <ctype.h>
#define isletter(c) isalpha(c)

/* include hashtable */
#define HASHSIZE 07777      /* hash table size */
struct word {
    char *word;     /* the word */
    int count;      /* frequency count */
    struct word *next;  /* link to next entry */
} *hashtable[HASHSIZE+1];
/* end hashtable */
int scatter[] = {       /* map characters to random values */
    2078917053, 143302914, 1027100827, 1953210302, 755253631,
    2002600785, 1405390230, 45248011, 1099951567, 433832350,
    2018585307, 438263339, 813528929, 1703199216, 618906479,
    573714703, 766270699, 275680090, 1510320440, 1583583926,
    1723401032, 1965443329, 1098183682, 1636505764, 980071615,
    1011597961, 643279273, 1315461275, 157584038, 1069844923,
    471560540, 89017443, 1213147837, 1498661368, 2042227746,
    1968401469, 1353778505, 1300134328, 2013649480, 306246424,
    1733966678, 1884751139, 744509763, 400011959, 1440466707,
    1363416242, 973726663, 59253759, 1639096332, 336563455,
```

```
    1642837685, 1215013716, 154523136, 593537720, 704035832,
    1134594751, 1605135681, 1347315106, 302572379, 1762719719,
    269676381, 774132919, 1851737163, 1482824219, 125310639,
    1746481261, 1303742040, 1479089144, 899131941, 1169907872,
    1785335569, 485614972, 907175364, 382361684, 885626931,
    200158423, 1745777927, 1859353594, 259412182, 1237390611,
    48433401, 1902249868, 304920680, 202956538, 348303940,
    1008956512, 1337551289, 1953439621, 208787970, 1640123668,
    1568675693, 478464352, 266772940, 1272929208, 1961288571,
    392083579, 871926821, 1117546963, 1871172724, 1771058762,
    139971187, 1509024645, 109190086, 1047146551, 1891386329,
    994817018, 1247304975, 1489680608, 706686964, 1506717157,
    579587572, 755120366, 1261483377, 884508252, 958076904,
    1609787317, 1893464764, 148144545, 1415743291, 2102252735,
    1788268214, 836935336, 433233439, 2055041154, 2109864544,
    247038362, 299641085, 834307717
    };
char *progname;

/* include arguments */
main(argc, argv)
int argc;
char *argv[];
{
    int i, k = 22;
    char *p, *getenv(), buf[MAXWORD];

    progname = argv[0];
    if (argc > 1)
        k = atoi(argv[1]);
    else if (p = getenv("PAGESIZE"))
        k = atoi(p);
/* end arguments */
/* include initialize */
    for (i = 0; i <= HASHSIZE; i++)
        hashtable[i] = NULL;
/* end initialize */
/* include mainprogram */
    while (getword(buf, MAXWORD) != EOF)
        addword(buf);
```

```
    printwords(k);
/* end mainprogram */
}

/* addword - add buf[0..strlen(buf)-1] to the table, bump count */
/* include fast-addword */
addword(buf)
char *buf;
{
    unsigned int h;
    int len;
    char *s, *s1, *s2, *alloc();
    struct word *wp, **q, **t;

    h = 0;      /* compute hash number of buf[0..] */
    s = buf;
    for (len = 0; *s; len++)
        h += scatter[*s++];
    t = q = &hashtable[h&HASHSIZE];
    for (wp = *q; wp; q = &wp->next, wp = wp->next)
/* include in-line-strcmp */
        for (s1 = buf, s2 = wp->word; *s1 == *s2; s2++)
            if (*s1++ == '\0') {
                wp->count++;
/* end in-line-strcmp */
                if (wp != *t) {
                    *q = wp->next;
                    wp->next = *t;
                    *t = wp;
                    }
                return;
                }
    wp = (struct word *) alloc(1, sizeof *wp);
    wp->word = alloc(len + 1, sizeof(char));
    strcpy(wp->word, buf);
    wp->count = 1;
    *q = wp;
}
/* end fast-addword */
```

```
/* getword - read next word in stdin into buf[0..size-1] */
int getword(buf, size)
char *buf;
int size;
{
    char *p;
    int c;

    p = buf;
    while ((c = getchar()) != EOF)
        if (isletter(c)) {
            do {
                if (size > 1) {
                    *p++ = c;
                    size--;
                    }
                c = getchar();
                } while (isletter(c));
            *p = '\0';
            return p - buf;
            }
    return EOF;
}

/* printwords - print the k most common words the table */
printwords(k)
int k;
{
    int i, max;
    struct word *wp, **list, *q;

    max = 0;
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = wp->next)
            if (wp->count > max)
                max = wp->count;
    list = (struct word **) alloc(max + 1, sizeof wp);
    for (i = 0; i <= HASHSIZE; i++)
        for (wp = hashtable[i]; wp; wp = q) {
            q = wp->next;
```

```
            wp->next = list[wp->count];
            list[wp->count] = wp;
            }
    for (i = max; i >= 0 && k > 0; i--)
        if ((wp = list[i]) && k-- > 0)
            for ( ; wp; wp = wp->next)
                printf("%d %s\n", wp->count, wp->word);
}
```

A.10 Instrumentation library for Chapter 3

```
#include <stdio.h>
static FILE *fpTrace;
static void initialize()
{
    if (!(fpTrace || (fpTrace = fopen("trace.out", "a")))) {
        fprintf(stderr, "Could not open trace.out\n");
        exit(1);
    }
}
void report_activation(char *module)
{
    initialize();
    fprintf(fpTrace, "0 %s\n", module);
}
int report_expression(int exp, int nodeT, int nodeF)
{
    initialize();
    fprintf(fpTrace, "%d\n", exp ? nodeT : nodeF);
    return exp;
}
void report_incidence(int node)
{
    initialize();
    fprintf(fpTrace, "%d\n", node);
}
```

A.11 Instrumentation library for Chapter 5

```
#include <stdio.h>
typedef struct II {  /* Instrumentation Information */
    struct II *pIINext;
    int cCounts;
    char *szMod;
    int rgCounts[1];  /* variable-length */
} II;
static FILE *fpTrace; /* file to write count data */
II IINil; /* nonzero sentinel for list of data */
II *pIIFirst = &IINil; /* head of list of data */
static void initialize()
{/* Initialize SIT system if not already initialized */
    if (!(fpTrace || (fpTrace = fopen("trace.out","a")))) {
        fprintf(stderr,"Could not open trace.out\n");
        exit(1);
    }
}
inst_dump_counts()
{/* Write count data for all modules to file */
    II *pII; int i; int *rgCounts;
    initialize();
    for (pII = pIIFirst; pII != &IINil; pII = pII->pIINext) {
        fprintf(fpTrace,"%s\n",pII->szMod);
        fprintf(fpTrace,"%d\n",pII->cCounts);
        rgCounts = &pII->rgCounts[0];
        for (i = 0; i < pII->cCounts; i++)
            fprintf(fpTrace,"%d\n",rgCounts[i]);
    }
}
inst_link_module(void *pv)
{/* Link a module's count information into system */
    II *pII = pv;
    if (pIIFirst == &IINil) /* once, set up to dump on exit */
        atexit(inst_dump_counts);
    pII->pIINext = pIIFirst;
    pIIFirst = pII;
}
```

```
inst_reset_counts()
{/* Reset all counts for all modules in program to zero */
    II *pII; int i; int *rgCounts;
    for (pII = pIIFirst; pII != &IINil; pII = pII->pIINext) {
        rgCounts = &pII->rgCounts[0];
        for (i = 0; i < pII->cCounts; i++)
        rgCounts[i] = 0;
    }
}
```

Appendix B

Basic Definitions

This appendix presents the basic definitions of structured testing and a discussion of the related mathematical model of software control flow as used in this dissertation. These definitions are taken from McCabe [37, 38], and the presentation is adapted from Watson and McCabe [58].

B.1 Modules and control flow graphs

A *module* is a unit of software with a single entry point that can be called and a single exit point that returns to the caller. In C, functions are modules, as are procedures and subroutines in other common languages.

A *control flow graph* is an abstract directed graph that describes the control structure of a module. The nodes of the graph represent computational statements or conditional expressions, and the edges represent transfer of control between nodes. The graph only represents code that is reachable from the module entry (ignoring statically "dead" code), and from which the exit is reachable. This model accommodates `return` statements by considering them as transferring control to the single exit node rather than as individual exit nodes. Modules can not, however, contain "infinite loops," which would prevent the exit node from being reachable. Every execution of a module corresponds to a path from the entry node to the exit node of the control flow graph.

B.2 Paths and vectors

Structured testing uses a correspondence between paths and edge frequency vectors in control flow graphs. Each path through a module's graph has a corresponding vector with entries equal to the number of times each edge of the graph was traversed along that path. Note that two different paths can correspond to the same vector, if they traverse the same edges the same number of times but in different orders.

The *edge space* of a graph is the vector space generated over the field of rational numbers by the vectors corresponding to all possible paths through the graph. It consists of all vectors that obey conservation of flow equations at each vertex as well as between the entry and the exit. Note that not every vector in the edge space has a corresponding path (for example, the zero vector).

A *basis set of paths* is a set of paths whose corresponding vectors form a basis for the edge space of the graph. Recall from linear algebra that a basis of a vector space is a set of vectors that generate all vectors in the space by linear combination.

B.3 Cyclomatic complexity

The *cyclomatic complexity* of a module is $2 + e - n$, where e and n are the numbers of edges and nodes in the module's control flow graph, respectively. It is also the dimension of the edge space of the graph, and therefore the number of paths in any basis set of paths.

Cyclomatic complexity is also known as $v(G)$, where v refers to the cyclomatic number in graph theory and G indicates that complexity is a function of the module's graph.

The *cyclomatic number* of a graph is $1 + e - n$, and is typically characterized as the number of fundamental (or basic) cycles in connected, undirected graphs [3]. It also gives the dimension of the edge space of a strongly connected directed graph. McCabe [37] derived cyclomatic complexity from the cyclomatic number by adding a "virtual" exit-to-entry edge to module control flow graphs to make them strongly connected. Since the flow through the virtual edge is a linear combination of the flows through the other edges, adding it does not change the dimension of the edge space. To account for the virtual edge, the cyclomatic complexity of a control flow graph is one greater than its cyclomatic number.

B.4 Structured testing

The *structured testing*, or *basis path testing* criterion states that a basis set of paths for a module must be executed during testing. Given the correspondence between paths and vectors, this means that the tested paths generate all possible paths by linear combination, and are therefore in some sense a representative set of paths. The minimum number of tests to satisfy this criterion is the cyclomatic complexity.

The *weak structured testing* criterion is to execute each edge of the control flow graph using a number of tests at least equal to the cyclomatic complexity. This criterion is subsumed by the structured testing criterion and is sometimes confused with it, since McCabe proposed them both under the name "structured testing."

The *baseline method* is a technique for generating a basis set of paths. It is sometimes confused with basis path testing, and prior to this dissertation it was the only published technique for satisfying the structured testing criterion.