Unit Analysis and Testing

SEI Curriculum Module SEI-CM-9-2.0

June 1992

Larry J. Morell
Hampton University

Lionel E. Deimel
Software Engineering Institute

Carnegie Mellon University
Software Engineering Institute

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

92052992

Unit Analysis and Testing

Acknowledgements

We would like to thank the Software Engineering Institute for its sponsorship of the writing of this curriculum module and its support of two major revisions. Norm Gibbs, Gary Ford, and Albert Johnson helped make writing version 1.0 both enjoyable and satisfying. Student assistants Suzanne Woolf and Dana Hausman, who helped construct the bibliography, made it less wearisome.

We want to thank the many people who invested their time in reviewing earlier drafts of this module: Mark Borger, Peter Feiler, Robert Goldberg, John Goodenough, Marc Graham, John Nestor, K. C. Tai, Christian Wild, and Steven Zeil. Their many comments and suggestions have been invaluable.

Linda Pesante, of the SEI, deserves special recognition for her editing of this material at various times over the past years, as well as for her cooperation and expertise, which were essential to bringing the current version to press. The clarity of our writing has been much improved through her efforts.

Contents

Unit Analysis and Testing

Module Revision History

Version 2.0 (June 1992)	Major revision; title changed to *Unit Analysis and Testing* from *Unit Testing and Analysis*
Version 1.2 (April 1989)	Outline error and other minor errors corrected
Version 1.1 (December 1988)	Minor changes and more thorough annotations in bibliography
	Approved for publication
Version 1.0 (October 1987)	Draft for public review

Unit Analysis and Testing

Capsule Description

This module examines the techniques, assessment, and management of unit analysis and testing. Analysis strategies are classified according to the view they take of the software: textual, syntactic, control flow, data flow, computation flow, or functional. Testing strategies are categorized according to whether their coverage goal is specification-oriented, implementation-oriented, error-oriented, or a combination of these. Mastery of the material in this module allows the software engineer to define, conduct, and evaluate unit analyses and tests and to assess new techniques proposed in the literature.

A Word About This Version

This version of *Unit Analysis and Testing* contains many changes that will be noticed by readers of earlier versions, including the relatively minor title change from *Unit Testing and Analysis*. Lionel Deimel functioned as an active technical editor of previous editions, and his status has been upgraded to that of coauthor.

The scope and goals of this curriculum module are largely unchanged (see *Philosophy* below) but the material has been updated and reorganized to reflect our rapidly expanding understanding of analysis and testing techniques. In particular, we have recognized that each testing technique requires determination of software characteristics, the discovery of which we have named *analysis*.[1] Since one testing technique may rely on several analysis techniques, and one analysis technique may support several testing (and other verification) techniques, analysis techniques are discussed separately from the testing techniques they support.

We believe the current organization provides greater insight into the nature of the techniques described and their relationship to one another. We have also been more deliberate about definitions and have introduced them within the context of a model of verification.

At every turn, we have had to resist turning the module into a monograph. The goal, of course, was to keep this work an approachable outline with bibliography and teaching suggestions. Achieving this goal meant deleting related material, paring the size of the bibliography, not explaining certain concepts as much as we would like, and not relating concepts to one another at greater length. The authors take full responsibility for what may seem, in places, a looseness of integration. We believe the casual reader will appreciate our brevity, however, and the careful reader will receive sufficient guidance to fill in any gaps.

The emphasis here is on *testing*, and the reader should be warned that the treatment of analysis is not comprehensive, but is meant to provide the necessary background for the discussion of the main topic. Perhaps in the future, unit analysis will receive the attention it deserves in its own curriculum module.

Philosophy

Program testing is the most practiced means of verifying that a program possesses the features required by its specification. *Testing* is a dynamic approach to *verification* in which software is executed with *test data* to assess the presence of required *features*. The inferences involved in this assessment are surprisingly complex. Testing employs *analysis* to

[1]In the earlier versions of this module “analysis” referred to verification techniques that did not require execution of the software. We feel the narrowed definition used here is more in keeping with the conventional usage of the word.

determine *software characteristics*, which are then used to evaluate whether features are present or not.

Many verification techniques have become established technologies with their own substantial literature. So that they may be given adequate treatment elsewhere, these techniques have been placed outside the scope of this module. Included among these techniques are proof of correctness, safety verification, and the more open-ended verification procedures represented by code inspections and reviews.

This module focuses on *unit*-level analysis and testing techniques; integration and systems testing are outside our scope. What constitutes a "unit" has been left imprecise—it may be as little as a single statement or as much as a set of coupled subroutines. The essential characteristic of a unit is that it can meaningfully be treated as a whole.

Because testing is a form of verification, it cannot be performed in the absence of *requirements*. Included in requirements are not only written specifications, standards, and the like, but also implicit or unwritten understandings of what the software should do.

Analysis techniques are classified here according the kinds of software characteristics they discover. Software characteristics are described as reflecting different views of the software: textual, syntactic, control flow, data flow, computation flow, or functional. By helping to discover software characteristics, analysis techniques play a part in many verification techniques, including testing.

Three major classes of testing are discussed—specification-oriented, implementation-oriented, and error-oriented—as well as some hybrid approaches. Specification-oriented testing ensures that specified major features of the software are *covered*. Implementation-oriented testing ensures that major characteristics of the code are covered. Error-oriented testing ensures that the range of typical errors is covered. The benefits of using techniques from different classes are complementary, and no single technique is comprehensive.

Assessment of unit analysis and testing techniques can be theoretical or empirical. This module presents both of these forms of assessment, and discusses criteria for selecting methods and controlling the verification process.

Management of unit analysis and testing should be systematic. It proceeds in two stages. First, techniques appropriate to the project must be selected. Then these techniques must be systematically applied.

Objectives

The following is a list of possible educational objectives based upon the material in this module. Objectives for any particular unit of instruction may be drawn from these or related objectives, as may be appropriate to audience and circumstances.

Knowledge

- Define the basic terminology of analysis and testing (particularly those terms found in the glossary on page 18 or italicized in the text).
- State the theoretical and computational limitations of analysis and testing.
- State the strengths and weaknesses of several analysis and testing techniques.
- Identify six program views.

Comprehension

- Explain the complementary nature of specification-oriented, implementation-oriented, and error-oriented testing techniques.
- Describe how the choice of analysis and testing criteria affects the selection and evaluation of test data.
- Explain the role of error collection as a feedback and control mechanism.
- Explain the program view taken by a given testing technique.

Application

- Test a software unit using specification-oriented, implementation-oriented and error-oriented techniques.
- Use configuration management to control the process of unit analysis and testing.

Analysis

- Determine the unit analysis and testing techniques applicable to a project, based upon the verification goals, the nature of the product, and the nature of the testing environment.

Synthesis

- Write a test plan tailored to accommodate project constraints.
- Design software tools to support implementation-oriented analysis and testing techniques.

Evaluation

- Evaluate the potential usefulness of new unit analysis or testing techniques proposed in the literature.

Prerequisite Knowledge

The student of unit analysis and testing should, of course, have a solid background in programming. Some mathematical sophistication is required, including a working knowledge of logic, relations, and functions. Beyond this, necessary background is dictated by the topics to be covered. Some of the specification-oriented techniques require that the student be able to read algebraic, axiomatic, and functional specifications of software modules. The implementation-oriented component requires knowledge of BNF grammars and graphs. If structural analysis tools are to be built, the student needs to have knowledge of parsing technology, parse trees, and graph algorithms at the level of an introductory compiler construction course. To understand the fundamental limitations of testing, the student should be familiar with the halting problem and reduction proofs. If underlying foundations of statistical testing are to be explored in depth, then a full year of statistics is a prerequisite. Effective use of the statistical models requires one semester of statistics.

Module Content

Outline

I. Preliminaries
  1. Concepts and terminology
  2. Adequacy of testing
  3. Limitations of testing
  4. Organization of this module

II. Program Analysis Techniques
  1. From a textual view
  2. From a syntactic view
  3. From a control flow view
  4. From a data flow view
  5. From a computation flow view
  6. From a functional view

III. Program Testing Techniques
  1. Specification-oriented testing
    a. Testing independent of the specification technique
      (i) Testing based on the interface
        (1) Input domain testing
        (2) Equivalence partitioning
        (3) Syntax checking
      (ii) Testing based on the function to be computed
        (1) Special value testing
        (2) Output domain coverage
    b. Testing dependent on the specification technique
      (i) Algebraic
      (ii) Axiomatic
      (iii) State machines
      (iv) Decision tables
  2. Implementation-oriented testing
    a. Structure-oriented testing
      (i) Statement testing
      (ii) Branch testing
      (iii) Data coverage testing
    b. Infection-oriented testing
      (i) Conditional testing
      (ii) Expression testing
      (iii) Domain testing
      (iv) Perturbation testing
      (v) Fault sensitivity testing
    c. Propagation-oriented testing
      (i) Path testing
      (ii) Compiler-based testing
      (iii) Data flow testing
      (iv) Mutation testing
  3. Error-oriented testing
    a. Error-based testing
    b. Fault-based testing
    c. Probable correctness
  4. Hybrid Testing Techniques

IV. Evaluating Unit Analysis and Testing Techniques
  1. Theoretical evaluation
  2. Empirical evaluation

V. Managerial Aspects of Unit Analysis and Testing
  1. Selecting techniques
  2. Goals
  3. Nature of the product
    a. Data processing
    b. Scientific computation
    c. Expert systems
    d. Embedded and real-time systems
  4. Nature of the testing environment
    a. Available resources
    b. Personnel
    c. Project constraints
  5. Control
    a. Configuration control
    b. Conducting tests

Annotated Outline

I. Preliminaries

The analysis and testing techniques with which this curriculum module deals are numerous and diverse. Unfortunately, there is neither a widely accepted taxonomy of them nor a standard model from which a taxonomy might be derived. Some of the classifications seen in the literature (the often-made distinction between *static analysis* and *dynamic analysis*, for example) seem to us insufficiently useful for making sense of this material. In this section, therefore, we present a framework for understanding unit analysis and testing. This framework not only provides organization for what follows, but also, we hope, provides some insight into the techniques of interest and into why their application is complex.

We begin by defining terms that will be used throughout the module. Then we examine the complementary benefits of different testing techniques, followed by the inherent limitations of testing. We conclude the section with an overview of the remaining material.

1. Concepts and terminology

As is often true in an evolving field, terminology used to describe program testing is far from settled. Although most experts would probably agree on designating certain core activities as "testing," the boundary between what is and is not testing is not so easily agreed upon.

We believe that precise definitions are important and serve to sharpen one's understanding. We further believe that testing, in the narrowest sense of executing a software unit on selected data, has analogues in other software engineering activities—in the conduct of software inspections, for example—that might reasonably be called by that name if one were willing to expand its meaning sufficiently. Such a generalization is tempting, but it demands the thorough exploration of a large gray area through which the resulting boundary of definition would have to pass. We have chosen to resist this temptation, not only because of the inherent difficult it entails, but also because the generalization is likely to *seem* more useful than it actually is. We therefore adopt a narrow definition of testing, one we think is precise and useful, and which reflects the common understanding of the term.

The standards for software test documentation [IEEE83] and for software unit testing [IEEE87], as well as the standard glossary of software engineering terms [IEEE90], define many testing-related terms. These definitions often clarify the meaning of words used inconsistently both in engineering practice and in the literature. We have drawn from these sources wherever possible, though we must point out that the standards are not 100% consistent with one another. We have introduced our own definitions where doing so seems useful. Fundamental terms (shown in italics) are discussed below and are collected in a glossary that begins on page 18.[1] Where appropriate, citations of sources are shown in the glossary.

[1]Italicized terms outside this introduction do not necessarily appear in the glossary.

Informally, testing is verification using information derived from execution of software. But by itself, a simple definition of testing fails adequately to convey either its complexity or its subtlety. Constructing a model to discuss testing can help us gain greater insight. The figure on page 20 presents such a model, in which rectangles represent objects or collections of objects, and directed arcs represent relations.

At the top left of the figure, we show "requirements," the collection of statements, diagrams, understandings, and other information that define and constrain the software to be produced.[2] As used here, requirements include not only the immediate precursor of a software unit—for example, a high-level design—but also other assertions, written or not, that serve to establish desired properties of the software.

To the right of requirements is "software," the code we wish to verify or test. (For purposes of this module, we may prefer to think of this box as being labeled "software unit.") The requirements are intended to (correctly) specify the software, to define what is and is not an acceptable implementation; *verification* seeks to determine whether the intended relationship is actually achieved. In practice, of course, either the requirements or the software may be defective.

Unfortunately, software cannot be directly measured against requirements. Instead, it is necessary to determine *software characteristics* through some process of *analysis* as one important step in verification. A characteristic is a trait, quality, or property of the software, whether intended or not. Both the number of vowels in the source code and the response of the software when executed on a given input value are software characteristics. The former characteristic is determined through static analysis (*i.e.*, without executing the software) and the latter is usually determined through dynamic analysis (*i.e.*, by some process that does involves execution). "Software characteristics" are shown in the lower right of the figure. Through analysis, the software in question is inferred to possess some set of software characteristics. Note that we do not say "possesses," as our analysis can be faulty—our argument that a loop always terminates contains a flaw—or it can lead to an insufficiently qualified result—our observation that the correct output is produced in response to a particular input does not *necessarily* mean that the software always does so, as its output may depend on an internal state affected by earlier input or by some unrecognized "input," such as time of day.

[2]We will not attempt to distinguish between the form of requirements and their abstract semantics. Because our definition of requirements includes undocumented expectations, sometimes there is no form we can point to!

In an ideal world, perhaps, verification would involve matching identified software characteristics to correct, written requirements. So simple a process is almost never appropriate, however: not everything gets written down; requirements can be incomplete, overspecified, or contradictory; stated requirements can be too complex to test directly; particularly when implied and commonsense but unstated requirements are accounted for, there may simply be too many properties of the software to verify in practice. It is therefore necessary to identify some set of *software features* against which verification takes place. The box in the lower left of the figure, therefore, represents such a collection of software characteristics specified or implied by the requirements. This collection, for purposes of verification, forms a (possibly inadequate) representation of what the software is supposed to be and to do.

Whereas verification of a piece of software attempts to determine if that software actually implements the requirements (*i.e.*, is within the class of acceptable implementations specified by the requirements), the task is actually carried out by attempting to show that the inferred software characteristics are sufficient to demonstrate that the required features are present. Of course, like the reasoning by which software features and software characteristics are derived, this process, too, is subject to error.

If the verification process relies on dynamic analysis to infer software characteristics (*i.e.*, on analysis involving executing the code), we say the verification is a form of *testing*.[3] Different methods of testing appeal to the nature of requirements, to the code itself, or to insights into how *errors* in the development process manifest themselves in requirements or software, in order to suggest *test data* and test procedures. Test procedures include those for program execution, data collection, and assessment of the results. Program execution involves the activities of selecting test data and determining the expected output, constructing an environment for executing the software with the selected data, and performing the actual execution. Information to be collected from the execution may be obtained in a very straightforward manner, such as from observation of a screen display, or it may require instrumenting the software with probes that reveal runtime behavior. Assessing the results of execution involves inferring software characteristics from the collected data and comparing those characteristics with the required features to see if the presence of the required features is indicated by the empirical evidence.

2. Adequacy of testing

As should be clear from the foregoing discussion, verification of software by testing or other means, is quite indirect. Required features and software characteristics must be derived and compared, and an argument may need to be made that establishing the presence of the features should be accepted as indicating that the software indeed satisfies the requirements. There are many opportunities for making erroneous inferences, yet resource limitations invariably dictate making inferences that are probably, though not necessarily correct.[4]

To strengthen the conclusions that can be drawn from testing, it is necessary to judiciously constrain the verification process. Conditions that are required to be satisfied during testing are called *adequacy criteria* [Weyuker86]. For example, testing may be considered inadequate if the test data do not include boundary cases specified by the requirements, do not cause execution of every line of code, or do not cause the software to deal with error-prone situations. The intent in establishing these criteria is to improve the quality of the testing. As such, adequacy criteria serve a purpose somewhat akin to software development standards, by requiring adherence to methods that have previously proved successful.

When considering the quality of testing, it is crucially important not to confuse it with the quality of the software being tested. This confusion can be seen when the goodness of testing is measured by characteristics of the program derived from testing, *e.g.*, number of *failures* found, number of faults found, mean-time-to-failure, etc. These properties presume quality testing, rather than define it. Even correct software can be poorly tested. Useful adequacy criteria seek to improve the likelihood of finding *faults* in the software, if they exist, regardless of the quality of the software being tested.

Adequacy criteria act as both specifier and judge: as specifier by indicating the constraints that must be satisfied by the testing, and as judge by indicating deficiencies in a particular test. Adequacy criteria for testing are generally expressed by stating some required *coverage* the test data should achieve. Desirable coverages include the required features, the software structure, or the potential errors that might occur in the life cycle. These coverages are discussed in depth below.

No one testing technique is so clearly superior to

[3]Compare [IEEE90]. In [IEEE83], testing is more broadly defined to include both static and dynamic verification techniques. (See *Glossary*, page 18.)

[4]If a software unit executes properly with integer inputs 1-43 and 45-100, a reasonable person testing the unit and having no cause to believe the integer 44 is in any way special, is likely to conclude the unit performs satisfactorily for inputs 1-100.

others that its exclusive use can be justified. Testing techniques are best seen as complementary rather than competing forms of verification; different techniques tend to catch different faults. This fact is reflected in the way testing techniques themselves are classified in this module: specification-oriented, implementation-oriented, and error-oriented. Techniques in each category focus on particular characteristics of the software system, leaving them susceptible to failing to uncover particular kinds of faults. Refinements introduced by the programmer for efficiency may not be executed in a specification-oriented test, for example, and a case required by the specification but omitted from the code may not be tested in an implementation-oriented test. If the errors sought in an error-oriented test are too limited, faults detectable by other methods may be missed.

Testing is a principal activity used to assess software quality. Such assessment is a subjective judgment as to the suitability of a given technique or product for a particular purpose. Evaluating software requires amassing as much information as possible about its quality. Information produced by testing is a valuable component in that evaluation, since all other forms of verification are further removed from the operational environment of the software. Inferences derived from particular test executions must be tempered by considering the environmental factors (*e.g.*, implicit inputs, compilers, operating systems, hardware, etc.) that influence the program's behavior, however. Test results are therefore susceptible to misinterpretation in a manner similar to other verification techniques.

3. Limitations of testing

Some problems cannot be solved on a computer because they are either intractable or undecidable. An *intractable* problem is one whose best known solution requires inordinate resources. An *undecidable* problem is one for which no algorithmic solution is possible. There are many such intractable and undecidable problems associated with analysis and testing. In general, programs cannot be exhaustively tested (tested for each input) because to do so is both intractable and undecidable. Huang shows that to test exhaustively a program that reads two 32-bit integers would take on the order of 50 billion years [Huang75]! Even if the input space is smaller, on the very first input it may be the case that the program does not halt within a reasonable time. It may even be the case that it is obvious the correct output will be produced if the program ever does halt. Exhaustive testing can only be completed, therefore, if all non-halting cases can be detected and eliminated. The problem of effecting such detection, however, is undecidable.

Another limitation on the power of testing is its reliance on an oracle. An *oracle* is a mechanism that judges whether or not a given output is correct for a given input. In some cases, no oracle may be available, *e.g.*, when the program is written to compute an answer that cannot, in practice, be computed by hand. Imperfect oracles may be available, but their use is risky. The absence of an oracle, or the presence of an imperfect oracle weakens significantly any conclusions drawn from testing.

4. Organization of this module

The remainder of this module is divided into four sections, discussing analysis techniques, testing techniques, methods for evaluating testing techniques, and methods for managing testing. Program analysis is treated first, as each testing technique uses one or more methods of analysis. We classify analysis methods according to the view of the software implicit in each technique. Our primary classification of testing techniques is according to the adequacy criteria they seek to satisfy. Secondary classifications refine the primary taxonomy as appropriate. Once a thorough background is laid, evaluation and managerial issues are addressed.

II. Program Analysis Techniques

Any technique that seeks to determine software characteristics is a form of program analysis. Software characteristics are essential in development, debugging, documentation, verification, evaluation, certification, and maintenance. This section discusses analysis techniques that support verification in general, and testing in particular. In addition, analysis can help decide where to focus testing.

Analysis is employed in all stages of testing, including test data selection, program execution, data collection, and assessment of the results. Test data can be selected based upon consideration of various information sources, including the specification, the implementation, and potential errors and faults. Collecting computational information may require analysis of the program text and subsequent instrumentation of the program. Program execution itself is a form of analysis. Establishing an oracle for verification may require additional analysis.

The analysis techniques discussed here are classified according to the *view* they take of the program. Each view emphasizes different aspects of the program and enables the determination of different program characteristics. Several predominant views are described below, along with the analysis techniques they support. Some analysis techniques employ more than one view. The views below are roughly ordered by increasing information content.

1. From a textual view

From a *textual view*, a program is treated as a sequence of characters or tokens. Many primitive metrics, such as program length and frequency of occurrence of identifiers, take this view. Text editors manipulate a program as a textual object, as do scanners, line counters, etc. Coding guidelines are frequently expressed from this viewpoint. Simple prettyprinters can be based on this view.

2. From a syntactic view

A program may be viewed as a hierarchy of syntactic elements determined by the programming language's grammar. Programs decompose into subprograms that decompose into statement groups that decompose into statements, etc., until the token level is reached. This syntactic view can be obtained from a textual view (*e.g.*, by a parser) or may be constructed directly (*e.g.*, by a syntax-directed editor). Derivable program characteristics include statement counts, identifier cross references, program call graphs (what procedure calls what procedure), declared and undeclared variables, frequency of variable use, and so on. Many sophisticated program metrics are based on this view.

The syntactic view supports program *instrumentation*, in which source or object code is modified to divulge its internal workings as it executes. During such execution, a variety of program characteristics can be determined, such as what statements and branches are executed. Execution counts or even complete trace information indicating the value computed by each expression can be generated. Probert presents algorithms for such instrumentation (using control flow graphs—see next section) [Probert82], and Beizer discusses the levels of instrumentation and their resulting impact [Beizer90]. Instrumented code will necessarily be larger and slower, so that executing it may mask timing, size, and position-related faults.

3. From a control flow view

A program's *control flow relation* relates *program elements* according to their execution order. A program element is usually a condition, a single statement, or a block of statements. If element *B* can be executed immediately after element *A*, then (*A*, *B*) is in the control flow relation of the program. Successor execution is determined independent of the computation performed at *A*. Thus, if *A* refers to the condition and *B* refers to the write statement in

$$\textbf{if}\ x < x\ \textbf{then}\ \text{write}(x)$$

then (*A*, *B*) is still in the control flow relation despite the fact that *A* is always false.

The graph corresponding to the control flow relation is called a *control flow graph* [Hecht77]. Each node of the graph corresponds to a program element; a directed arc between two nodes indicates the corresponding two elements form an ordered pair in the control flow relation. A *path* through a control flow graph corresponds to a potentially executable sequence of program elements. To *execute* a path is to execute the corresponding sequence of program elements. If a program input exists that causes execution of a path, that path is called *feasible*; otherwise it is called *infeasible*. Programs with loops usually have infinitely many paths; even without loops, a program may have intractably many paths to analyze.

Control flow graphs have no labels or other annotations, distinguishing them from flowcharts, which capture additional program semantics. Control flow graphs are generally produced from a syntactic view of a program for more efficient processing of control flow information. There are many program metrics based upon the control flow view of a program.

4. From a data flow view

The *data flow relation* determined by a program relates program elements according to their data access behavior. If element *B* uses (refers to) a data object that was potentially defined at element *A*, then (*A*, *B*) is in the data flow relation of the program. A *data flow graph* [Fosdick76, Hecht77] is a directed, labeled graph corresponding to the data flow relation, in which nodes correspond to program elements and directed arcs connect *A* to *B* with label *v* if (*A*, *B*) is in the data flow relation due to a definition of *v* at *A* and a use of *v* at *B*. Data flow graphs can be produced from the syntactic view for more efficient processing of data flow information.

A program can be represented as a flow graph annotated with information about *variable definitions*, *references*, and *undefinitions*. From this representation, information about *data flow* can be deduced for use in code optimization, anomaly detection, and test data generation [Hecht77, Muchnick81].

Data flow *anomalies* are flow conditions that deserve further investigation, as they may indicate problems. Examples include: defining a variable twice with no intervening reference, referencing a variable that is undefined, and undefining a variable that has not been referenced since its last definition. Algorithms for detecting these anomalies are given in [Fosdick76] and [Osterweil76], and are refined and corrected in [Jachner84].

A *program slice* results from eliminating all statements that cannot affect the computation of an expression at a specified location [Weiser84]. Korel adapts slicing to testing and debugging in [Korel88a], [Korel88b], and [Korel90b]. His method employs a variation on a data flow graph called a *program dependence graph* [Korel87].

5. From a computation flow view

A program can be viewed as a finite representation of a (potentially infinite) set of computations. A *computation* is a trace of the data states[5] produced by a program when executing a particular input. A thorough analysis of the computation flow of a program induced by an execution may serve to estimate the number of faults remaining in the code, the strength of the test data to catch faults, and the ability of the program to hide faults.

Fault seeding is a statistical method used to assess the number and nature of the faults remaining in a program. A reprint of Harlan Mills' original proposal for this technique (where he calls it *error seeding*) appears in [Mills83]. First, faults are seeded into a program. Then the program is tested, and the number of faults discovered is used to estimate the number of faults yet undiscovered. A difficulty with this technique is that the faults seeded must be representative of the yet-undiscovered faults in the program.

Mutation analysis uses fault seeding to investigate properties of test data [Hamlet77a, Hamlet77b, DeMillo78a, DeMillo88]. Programs with seeded faults are called *mutants*. Mutants are executed to determine whether or not they behave differently from the original program. Mutants that behave differently are said to have been *killed* by the test. The product of mutation analysis is a measure of how well test data kill mutants. Mutants are produced by applying a *mutation operator*. Such an operator changes a single expression in the program to another expression, selected from a finite class of expressions. For example, a constant might be incremented by one, decremented by one, or replaced by zero, yielding one of three mutants. Applying the mutation operators at each point in a program where they are applicable forms a finite, albeit large, set of mutants.

Three conditions necessary and sufficient for a fault to cause a program failure are *execution*, *infection*, and *propagation* [Morell90, Voas91]. The fault location must be *executed*, the resulting data state must be *infected* with an erroneous value, and the succeeding computation must *propagate* the infection through erroneous data states, producing a failure. [Richardson88] and [Morell90] discuss necessary conditions for infection and propagation to occur.

Sensitivity analysis [Foster80, Voas91] investigates the three conditions required for failure, with particular focus on infection and propagation of errors. *Infection analysis* employs mutation analysis to determine the probability of a data state's being infected after a potentially faulty statement is executed. *Propagation analysis* mutates the data state to determine the probability that an infected data state will cause a program failure. [Voas91] gives means of estimating the probability that execution, infection, and propagation will occur for faults and data states from particular classes. [Morell88] and [Morell90] use symbolic execution (see below) to analyze potential error flow. Symbolic faults are introduced into the program, and the program is symbolically executed. The symbolic output captures the effect the fault would have on the program computation.

6. From a functional view

Programs may be viewed as functions by considering them as denotations for a set of ordered pairs (x, y), where y is the output produced by the program that halts on input x [Mills75]. Executing the program for input x and observing its output (if any) is an analysis technique that provides direct evidence of the program's function on the given input.

Symbolic analysis seeks to describe the function computed by a program in a more general way. A *symbolic execution system* accepts three inputs: a program to be interpreted, *symbolic input* for the program, and the path to follow. It produces two outputs: the *symbolic output* that describes the computation of the selected path, and the *path condition* for that path [Hantler76]. The specification of the path can be either interactive [Clarke76] or preselected [Howden77, Howden78b]. The symbolic output can be used to prove the program correct with respect to its specification, and the path condition can be used for generating test data to exercise the desired path. Structured data types cause difficulties, however, since it is sometimes impossible to deduce what component is being modified.

III. Program Testing Techniques

Testing is verification that relies on program execution. It includes all the activities associated with test data selection, program instrumentation and execution, and analysis of the results.

Since the conclusions of testing are drawn from execution-derived characteristics, the validity of the conclusions is directly related to the accuracy with which the execution in the test environment models an execution in the target environment. Care must be taken to ensure that all environmental factors are considered in assessing these characteristics. For example, all implicit inputs must be considered (*e.g.*, the system clock, the state of files, the load location of the unit), as well as how representative of the actual environment (*e.g.*, same compiler, loader, operating system, computer, input distribution) is the test environment.

The relationship of the test environment to the "real" execution environment is of particular concern for symbolic execution. Relying on an interpreter raises

[5]The *data state* of a program is the mapping of program variables (including temporaries and the program counter) to values.

additional concerns as to faithfulness to the programming language and hardware specifications. It is impossible to be dogmatic about what should be considered a representative execution. It is usually better to err on the side of caution when interpreting the results of a given execution, however.

Test data selection may be guided by several sources: the specification, the implementation, potential errors in the programming process, or some combination thereof. The testing techniques discussed below are organized according to these diverse sources. Specification-oriented testing seeks to show that every required software feature is addressed by some aspect of the software. Implementation-oriented testing attempts to show that the implementation contains no surprises, by showing that various aspect of the code can be exercised without violating the requirements. Error-oriented testing seeks to show that certain errors in the programming process have not occurred.

1. Specification-oriented testing

Program testing is *specification-oriented* when test data are developed from documents and understandings intended to specify a module's behavior. Sources include, but are not limited to, the actual written specification and the high- and low-level designs of the code to be tested [Howden80a]. The goal is to test for the presence of each (required) *software feature*, including the input domains, the output domains, categories of inputs that should receive equivalent processing, and the processing functions themselves.

Specification-oriented testing seeks to show that every requirement is addressed by the software. An unimplemented requirement may be reflected in a missing path or missing code in the software. Specification-oriented testing assumes a functional view of the software and sometimes is called *functional* or *black-box* testing [Howden86].

a. Testing independent of the specification technique

Specifications detail the assumptions that may be made about a given software unit. They must describe the interface through which access to the unit is given, as well as the behavior once such access is given. The *interface* of a unit includes the features of its inputs, its outputs, and their related value spaces (called *domains*). The *behavior* of a module always includes the function(s) to be computed (its *semantics*), and sometimes the *runtime characteristics*, such as its space and time complexity. Specification-oriented testing derives test data from aspects of the specification.

(i) Testing based on the interface

Testing based on the interface of a module selects test data based on the features of the input and output domains of the module and their interrelationships.

(1) Input domain testing

In *extremal testing*, test data are chosen to cover the extremes of the input domain. Similarly, *midrange testing* selects data from the interiors of domains. The motivation is inductive—it is hoped that conclusions about the entire input domain can be drawn from the behavior elicited by some representative members of it [Myers79]. For structured input domains, combinations of extremal points for each component are chosen. This procedure can generate a large quantity of data, though considerations of the inherent relationships among components can ameliorate this problem somewhat [Howden80b].

(2) Equivalence partitioning

Specifications frequently partition the set of all possible inputs into classes that receive equivalent treatment. Such partitioning is called *equivalence partitioning* [Myers79]. A result of equivalence partitioning is the identification of a finite set of functions and their associated input and output domains. For example, the specification

$$\{(x, y) \mid x \geq 0 \supset y = x \ \& \ x < 0 \supset y = -x\}$$

partitions the input into two sets, associated, respectively, with the identity and negation functions. Input constraints and error conditions can also result from this partitioning. Once these partitions have been developed, both extremal and midrange testing are applicable to the resulting input domains.

[Duran81], [Duran84], and [Hamlet90] compare equivalence partitioning to random testing, on the basis of statistical confidence in the probability of failure after testing is complete.

(3) Syntax checking

Every robust program must parse its input and handle incorrectly formatted data. Verifying this feature is called *syntax checking* [Beizer90]. One means of accomplishing this is to execute the program using a broad spectrum of test data. By describing the data with a BNF grammar, instances of the input language can be generated using algorithms from automata theory. [Duncan81] and [Bazzichi82] describe systems that provide limited control over the data to be generated.

(ii) Testing based on the function to be computed

Equivalence partitioning results in the identification of a finite set of functions and their associated input and output domains. Test data can be developed based on the known characteristics of these functions. Consider, for example, a function to be computed that has fixed points, *i.e.*, certain of its input values are mapped into themselves by the function. Testing the computation at these fixed points is possible, even in the absence of a complete specification [Weyuker82]. Knowledge of the function is essential in order to ensure adequate coverage of the output domains.

(1) Special value testing

Selecting test data on the basis of features of the function to be computed is called *special value testing* [Howden80b]. This procedure is particularly applicable to mathematical computations. Properties of the function to be computed can aid in selecting points that will indicate the accuracy of the computed solution. For example, the periodicity of the sine function suggests use of test data values which differ by multiples of 2π. Such characteristics are not unique to mathematical computations. Most prettyprinters, for example, when applied to their own output, should reproduce it unchanged. Some word processors behave this way as well.

(2) Output domain coverage

For each function determined by equivalence partitioning, there is an associated output domain. *Output domain coverage* is performed by selecting points that will cause the extremes of each of the output domains to be achieved [Howden80b]. This ensures that units have been checked for maximum and minimum output conditions and that all categories of error messages have, if possible, been produced. In general, constructing such test data requires knowledge of the function to be computed and, hence, expertise in the application area.

b. Testing dependent on the specification technique

The specification technique employed can aid in testing. An executable specification can be used as an *oracle* and, in some cases, as a test generator. Structural properties of a specification can guide the testing process. If the specification falls within certain limited classes, properties of those classes can guide the selection of test data. Much work remains to be done in this area of testing.

(i) Algebraic

In algebraic specification, properties of a data abstraction are expressed by means of axioms or rewrite rules. In one testing system, DAISTS, the consistency of an algebraic specification with an implementation is checked by testing [Gannon81]. Each axiom is compiled into a procedure, which is then associated with a set of test points. A driver program supplies each of these points to the procedure of its respective axiom. The procedure, in turn, indicates whether the axiom is satisfied. Structural coverage of both the implementation and the specification is computed. [Jalote89] discusses an approach to generating test data to verify the completeness of an algebraic specification. Algebraic methods are applicable to testing object-oriented programs.

(ii) Axiomatic

Despite the potential for widespread use of predicate calculus as a specification language, little has been published about deriving test data from such specifications. [Gourlay83] references work done on the relationship between predicate calculus specifications and path testing.

(iii) State machines

Many programs can be specified as state machines, thus providing additional means of selecting test data [Beizer90]. Since the equivalence problem of two finite automata is decidable, testing can be used to decide whether a program that simulates a finite automaton with a bounded number of nodes is equivalent to the one specified. This result can be used to test those features of programs that can be specified by finite automata, *e.g.*, the control flow of a transaction-processing system.

(iv) Decision tables

Decision tables are a concise method of representing an equivalence partitioning. The rows of a decision table specify all the conditions that the input may satisfy. The columns specify different sets of actions that may occur. Entries in the table indicate whether the actions should be performed if a condition is satisfied. Typical entries are “Yes,” “No,” or “Don’t Care.” Each row of the table suggests significant test data. *Cause-effect graphs* [Myers79] provide a systematic means of translating English specifications into decision tables, from which test data can be generated.

2. Implementation-oriented testing

In *implementation-oriented* program testing, test data selection is guided by information derived from the implementation [Howden75]. The goal is to ensure that various computational characteristics of the software are adequately covered. It is hoped that test data that satisfy these criteria have higher probability of discovering faults. Each execution of a program executes a particular path. Hence, implementation–oriented testing focuses on the following questions: What computational characteristics are desirable to achieve? What paths for this program achieve these characteristics? What test data will execute those paths? What are the computational characteristics of the set of paths executed by a given test set?

Implementation-oriented testing addresses the fact that only the program text reveals the detailed decisions of the programmer. For the sake of efficiency, a programmer might choose to implement a special case that appears nowhere in the specification. The corresponding code will be tested only by chance using specification-oriented testing, whereas use of a structural coverage measure such as statement coverage (see below) should indicate the need for test data for this case.

Implementation-oriented testing schemes may be classified according to two orthogonal axes: error orientation and program view, discussed earlier in Section II. A testing scheme's *error orientation* is the aspect of fault discovery that is emphasized: execution, infection, or propagation (see Section II.5). A testing scheme's *program view* is the program abstraction source that is used to determine desirable computational characteristics: control flow, data flow, or computation flow. Program view emphasizes how a particular strategy works; error orientation emphasizes the motivation behind the strategy and helps one to better evaluate claims made about the strategy.

The subsequent sections are organized by error orientation. Techniques that require execution of particular program elements are presented first, followed by those that attempt to force infections, and those that attempt to force propagation. It should be noted that infection and propagation techniques both require execution, and some techniques emphasize all three conditions. Within each section, the techniques are ordered by program view: control flow, then data flow, then computation flow.

a. Structure-oriented testing

A testing technique is *structure-oriented* if it seeks test data that cause various structural aspects of the program to be exercised. Assessing the coverage achieved may involve instrumenting the code to keep track of which parts of the program are actually exercised during testing. The inexpensive cost of such instrumentation has been a prime motivation for adopting structure-oriented techniques [Probert82]. Further motivation comes from consideration of the consequences of releasing a product without having executed all its parts and having the customer discover faults in untested code.

There are three essential components to be covered in structure-oriented testing: computations, branches, and data. These are discussed below.

(i) Statement testing

Statement testing requires that every statement in the program be executed. While it is obvious that achieving 100% statement coverage does not ensure a correct program, it is equally obvious that anything less means that there is code in the program that has never been executed!

(ii) Branch testing

Achieving 100% statement coverage does not ensure that each branch in the program flow graph has been executed. For example, executing an **if** ... **then** statement (no **else**) when the tested condition is true, tests only one of two branches in the flow graph. *Branch testing* seeks to ensure that *every* branch has been executed [Huang75, Tai80]. Branch coverage can be checked by probes inserted at points in the program that represent arcs from branch points in the flow graph [Probert82]. This instrumentation suffices for statement coverage as well.

(iii) Data coverage testing

In some programs, a portion of the flow control is determined by the data, rather than by the code. Knowledge-based applications, some AI applications, and table-driven code are all examples of this phenomenon. Data coverage testing seeks to ensure that various components of the data are "executed," *i.e.*, they are referenced or modified by the interpreter as it executes. Paralleling statement testing, one can ensure that each data location is accessed. Furthermore, in the area of knowledge bases, data items can be accessed in different orders, so it is important to cover each of these access orders. These access sequences are analogous to branch testing.

b. Infection-oriented testing

A testing technique is considered *infection-oriented* if it seeks to establish conditions suitable for infections to arise at locations of potential faults. This section characterizes several testing techniques that require test data to force infections if faults exist.

(i) Conditional testing

In *conditional testing,* each clause in every condition is forced to take on each of its possible values in combination with those of other clauses [Huang75]. Conditional testing thus subsumes branch testing. Instrumentation for conditional testing can be accomplished by breaking compound conditional statements into simple conditions and nesting the resulting **if** statements. This reduces the problem of conditional coverage to the simpler problem of branch coverage, enabling algorithms from the control flow view to be employed.

(ii) Expression testing

Expression testing [Hamlet77a] requires that every expression assume a variety of values during a test in such a way that no expression can be replaced by a simpler expression. If one assumes that every statement contains an expression and that conditional expressions form a proper subset of all the program expressions, then this form of testing properly subsumes all the previously mentioned techniques. Expression testing requires significant runtime support for the instrumentation [Hamlet77b].

(iii) Domain testing

The input domain of a program can be partitioned according to which inputs cause each path to be executed. These partitions are called *path domains*. Faults that cause an input to be associated with the wrong path domain are called *domain faults*. Other faults are called *computation* faults. (The terms used before attempts were made to rationalize nomenclature were *domain errors* and *computation errors.*) The goal of *domain testing* is to discover domain faults by ensuring that test data limit the range of undetected faults [White80]. This is accomplished by selecting inputs close to boundaries of the path domain. If the boundary is incorrect, these points increase the chance of an infection's occurring. Domain testing assumes *coincidental correctness* does not occur, *i.e.*, it assumes a program will fail if an input follows the wrong path.[6] [Clarke82] refines the fault detection capability of this approach by requiring points to be selected that further limit the amount a boundary can shift without an infection's occurring.

[6]The definition of coincidental correctness has since been broadened to include any situation in which a fault is executed without an ensuing failure.

(iv) Perturbation testing

Perturbation testing attempts to determine a sufficient set of paths to test for various faults in the code. Faults are modeled as a vector space, and characterization theorems describe when sufficient paths have been tested to discover both computation and domain errors. Additional paths need not be tested if they cannot reduce the dimensionality of the error space [Zeil83, Zeil88].

(v) Fault sensitivity testing

[Foster80] describes a method for selecting test data that are sensitive to faults. Howden has formalized this approach in a method called *weak mutation testing* [Howden82]. Rules for recognizing fault-sensitive data are described for each primitive language construct. Satisfaction of a rule for a given construct during testing means that all alternate forms of that construct have been distinguished. This has an obvious advantage over mutation testing (discussed later)—elimination of all mutants without generating a single one! Some rules even allow for infinitely many mutants.

c. Propagation-oriented testing

A testing technique is considered *propagation-oriented* if it seeks to ensure that potential infections propagate to failures. This requires selecting paths to test based on their propagation characteristics.

(i) Path testing

In *path testing,* data are selected to ensure that all paths of the program have been executed [Howden76]. In practice, of course, such coverage is impossible to achieve, for a variety of reasons. First, any program with an indefinite loop contains infinitely many paths, one for each iteration of the loop. Thus, no finite set of data will execute all paths. The second difficulty is the infeasible path problem: it is undecidable whether an arbitrary path in an arbitrary program is executable. Attempting to generate data for such infeasible paths is futile, but it cannot be avoided. Third, it is undecidable whether an arbitrary program will halt for an arbitrary input. It is therefore impossible to decide whether a path is finite for a given input.

In response to these difficulties, several simplifying approaches have been proposed. Infinitely many paths can be partitioned into a finite set of equivalence classes based on characteristics of the loops. *Boundary and interior testing* requires executing loops zero times, one time, and, if possible, the maximum number of times [Howden75]. *Linear sequence code and jump criteria* [Woodward80] specify a hierarchy of successively more complex path coverages. [Howden78a], [Tai80], [Gourlay83], [Weyuker86], and [Ntafos88] suggest methods of studying the adequacy of path testing.

Path coverage does not imply condition coverage or expression coverage, since an expression may appear on multiple paths but some subexpressions may never assume more than one value. For example, in

$$\textbf{if } a \vee b \textbf{ then } \mathrm{S}_1 \textbf{ else } \mathrm{S}_2$$

b may be false and yet each path may still be executed.

(ii) Compiler-based testing

In [Hamlet77a] and [Hamlet77b], a compiler augmented to judge the adequacy of test data is described. Input-output pairs are encoded as a comment in a procedure, as a partial specification of the function to be computed by that procedure. The procedure is then executed for each of the input values and checked for the output values. The test is considered adequate only if each computational or logical expression in the procedure is *determined* by the test; *i.e.*, no expression can be replaced by a simpler expression and still pass the test. *Simpler* is defined in a way that allows only finitely many substitutions. Thus, as the procedure is executed, each possible substitution is evaluated on the data state presented to the expression. Those that do not evaluate the same as the original expression are rejected. Substitutions that evaluate the same, but ultimately produce failures, are likewise rejected.

(iii) Data flow testing

Data flow analysis can form the basis for testing, exploiting the relationship between points where variables are defined and points where they are used [Frankl88, Laski83, Ntafos84, Ntafos88, Podgurski90, Rapps85]. By insisting on the coverage of various definition-use pairs,[7] data flow testing establishes some of the conditions necessary for infection and partial propagation. The motivation behind data flow testing is that test data are inadequate if they do not exercise these various *def-use* combinations. It is clear that an incorrect definition that is never used during a test will not be caught by that test. Similarly, if a given location incorrectly uses a particular definition, but that combination is never tried during a test, the fault will not be detected.

Data flow connections may be determined statically [Rapps85] or dynamically [Korel88b]. Some connections may be infeasible due to the presence of infeasible subpaths. Heuristics may be developed for generating test data based on data flow information [Korel88b].

(iv) Mutation testing

Mutation testing uses mutation analysis to judge the adequacy of test data. The test data are judged adequate only if each mutant is either functionally equivalent to the original program or computes output different from the original program on the test data. Inadequacy of the test data implies that certain faults can be introduced into the code and go undetected by the test data.

Mutation testing is based on two hypotheses [DeMillo78a]. The *competent programmer hypothesis* says that a competent programmer will write code that is close to being correct; the correct program, if not the current one, can be produced by some straightforward syntactic changes to the code. The *coupling effect hypothesis* says that test data that reveal simple faults will uncover complex faults. Thus, only single mutants need be eliminated, and combinatoric effects of multiple mutants need not be considered [DeMillo78a]. [Gourlay83] formally characterizes the competent programmer hypothesis as a function of the probability of the test set's being reliable (as defined by Gourlay) and shows that under this characterization, the hypothesis does not hold. Empirical justification of the coupling effect has been attempted [Budd80, DeMillo78a, Offutt89], but theoretical analysis has shown that it may hold probabilistically, but not universally [Gourlay83, Morell88].

3. Error-oriented testing

Testing is necessitated by the potential presence of errors in the programming process. Techniques that focus on assessing the presence or absence of errors in the programming process are called *error-oriented*.

a. Error-based testing

Error-based testing seeks to demonstrate that certain *errors* have not been committed in the programming process [Weyuker80]. Error-based testing can be driven by histories of programmer errors, measures of software complexity, knowledge of error-prone syntactic constructs, or even error guessing [Myers79].

[7]If a variable x is defined at location A, referenced (or used) at location B, and there is a path from A to B with no intervening definition of x, then (A, B) is a *definition-use pair*.

Error-based testing begins with the programming process, identifies potential errors in that process and then asks how those errors are reflected as faults. It then seeks to demonstrate the absence of those faults. Howden has classified errors into two categories: abstraction and decomposition, and he has developed specific techniques for addressing these categories [Howden89, Howden90].

b. Fault-based testing

Fault-based testing aims at demonstrating that certain prescribed faults are not in the code.

Fault-based testing methods differ in both *extent* and *breadth*. One with local extent demonstrates that a fault has a local effect on computation; it is possible that this local effect will not produce a program failure. A method with global extent demonstrates that a fault *will* cause a program failure. Breadth is determined by whether the technique handles a finite or an infinite class of faults. Extent and breadth are orthogonal. Infection- and propagation-oriented techniques could be classified as fault-based if they are interpreted as seeking to demonstrate the absence of particular faults. Infection-oriented techniques are of local extent.

[Morell88] and [Morell90] define a fault-based method based on symbolic execution that permits elimination of infinitely many faults through evidence of global failures. *Symbolic faults* are inserted into the code, which is then executed on real or symbolic data. Program output is then an expression in terms of the symbolic faults. It thus reflects how a fault at a given location will impact the program's output. This expression can be used to determine actual faults that could not have been substituted for the symbolic fault and remain undetected by the test.

c. Probable correctness

Probable correctness is defined by Hamlet to be the probability that no faults exist in a tested program [Hamlet87]. Early intimations of this concept may be found in [DeMillo78b] and [Rowland81], where particular classes of functions have members that can be distinguished from other members by a finite test set. As such, each successful execution increases confidence that the implemented function is correct. [Hamlet90] explores the concept when no *a priori* bound can be placed on the number of executions needed. He bounds the number of inputs needed to obtain high confidence in a high probability of correctness.

4. Hybrid Testing Techniques

Since it is apparent that no one testing technique is sufficient, some experts have investigated ways of combining several techniques. Such integrated techniques are called *hybrid testing techniques*. These are not just the concurrent application of distinct techniques; they are characterized by a deliberate attempt to incorporate the best features of different methods into a single new technique.

In *partition analysis*, test data are chosen to ensure simultaneous coverage of both the specification and code [Richardson85]. An operational specification language has been designed that enables a structural measure of coverage of the specification. The input space is partitioned into a set of domains that is formed by the cross product of path domains of the specification and path domains of the program. Test data are selected from each non-empty partition, ensuring simultaneous coverage of both specification and code. Proof of correctness techniques can also be applied to these cross product domains.

The testing system DAISTS predates and automates this technique for algebraic specifications of abstract data types, but it does not include any notion of proof of correctness [Gannon81]. Furthermore, the emphasis in DAISTS is on test data evaluation, rather than generation. [Goodenough75] presents a less formal, integrated scheme for selecting test data based on analysis of sources of errors in the programming process. [Richardson89] applies process programming to the problem of interacting testing techniques.

IV. Evaluating Unit Analysis and Testing Techniques

The effectiveness of unit analysis and testing may be evaluated on theoretical or empirical grounds [Howden78a]. Theory seeks to understand what can be done in principle; empirical evaluation seeks to establish what techniques are useful in practice. Theory formally defines the field and investigates its fundamental limitations. For example, it is well known that testing cannot demonstrate the correctness of an arbitrary program with respect to an arbitrary specification. This does not mean, however, that testing can never verify correctness; indeed, in some cases it can [Howden78c, Tai80]. Empirical studies evaluate the utility of various practices. Though statement testing is theoretically deficient, it is immensely useful in practice, exposing many program faults.

IEEE has sponsored several workshops on testing, analysis, and verification. The National Bureau of Standards has issued a special publication that describes many of the techniques mentioned in this module and characterizes each approach according to effectiveness, applicability, learning, and cost [Powell82]. The Strategic Defense Initiative Organization commissioned a study of the state of the art in verification techniques that resulted in an extensive overview of the field [Youngblut89] and with an in-depth annotated bibliography [Brykczynski89].

1. Theoretical evaluation

Theory serves three fundamental purposes: to define terminology, to characterize existing practice, and to suggest new avenues of exploration. Unfortunately, historical terminology is inconsistent. A simple example is the word *reliable*, which is used by authors in related, but diverse ways. (Compare, for example, [Duran81], [Goodenough75], [Howden76], and [Richardson85], and do not include any literature from *reliability theory*!) [IEEE83] is an appropriate starting point for examining terminology, but it is imprecise in places and was established many years after certain (in retrospect, unfortunate) terminology had become accepted.[8] Theoretical treatments of topics in program testing are ever expanding. Goodenough and Gerhart, in [Goodenough75], made an attempt to rationalize terminology, though this work has been criticized, particularly in [Weyuker80]. Nevertheless, they anticipated the vast majority of practical and theoretical issues that have since evolved in program testing. [Goodenough75] is therefore required reading. Howden and Weyuker have both written theoretical expositions on specification-, implementation-, and error-oriented testing [Howden76, Howden78a, Howden78c, Howden82, Howden86, Rapps85, Weyuker80, Weyuker82, Weyuker84, Weyuker86]. Theoretical expositions of mutation and fault-based testing are found in [Budd80], [Budd82], [Cherniavsky87], [Davis88], [Gourlay83], [Hamlet77a], [Morell88], and [Morell90]. [Rowland81]. [Clarke89], [Podgurski90], and [Zeil88] present frameworks for understanding data flow testing.

2. Empirical evaluation

Empirical studies provide benchmarks by which to judge existing testing techniques. An excellent comparison of techniques is found in [Howden80a], which emphasizes the complementary benefits of different testing methods applied to scientific programs. Empirical studies of mutation testing are discussed in [Budd80]. [Basili87] compares the effectiveness of code-reading, specification-oriented testing, and implementation-oriented testing. [Weyuker88] discusses an empirical study of the complexity of data flow testing.

Many papers discussing experience with testing techniques can be found in conference proceedings, especially proceedings of the following:

- ACM Symposium on Principles of Programming Languages
- International Conference on Software Engineering
- Computer Software and Applications Conference
- Testing, Analysis, and Verification Conference
- International Conference on Testing Computer Software
- Software Testing and Review Conference

[8]For instance, the terminology *error-based testing* and *error seeding* became well-established long before the standard told us to use *fault*.

V. Managerial Aspects of Unit Analysis and Testing

Administration of unit analysis and testing proceeds in two stages. First, techniques appropriate to the project must be selected. These techniques must then be applied systematically. [IEEE87] provides explicit guidance for these steps.

1. Selecting techniques

Selecting the appropriate techniques from the array of possibilities is a complex task that requires assessment of many issues, including the goal of testing, the nature of the software product, and the nature of the test environment. It is important to remember the complementary benefits of the various techniques and to select as broad a range of techniques as possible, within constraints of time, cost, etc. No single analysis or testing technique is sufficient [Gerhart76]. Specification-oriented testing may suffer from inadequate code coverage, implementation-oriented testing may suffer from inadequate specification coverage, and neither technique guarantees the benefits of error-oriented coverage.

2. Goals

Different design goals impose different demands on the selection of testing techniques. Achieving correctness requires use of a great variety of techniques. A goal of reliability implies the need for statistical testing using test data representative of those of the anticipated user environment. It should be noted, however, that statistical testing still requires judicious use of “selective” tests to avoid embarrassing or disastrous situations. Testing may also be directed toward assessing the usability of software. This kind of testing requires a solid foundation in human factors [Perlman90]. Performance of the software may also be of special concern. In this case, extremal testing is essential. Timing instrumentation can prove useful.

Often, several of these goals must be achieved simultaneously. Recent attempts in process programming have sought to address this issue [Richardson89]. One approach to testing under these circumstances is to order testing by decreasing benefit. For example, if reliability, correctness, and performance are all desired features, it is reasonable to tackle performance first, reliability second, and correctness third, since these goals require increasingly difficult-to-design tests. This approach can have the beneficial effect of identifying faulty code with less effort expended.

3. Nature of the product

The nature of the software product plays an important role in the selection of appropriate techniques. Four representative types of software products are discussed below.

a. Data processing

Data processing applications appear to benefit from most of the techniques described in this module. Conventional languages such as COBOL are frequently used, increasing the likelihood of finding an instrumented compiler for doing performance and coverage analysis. Functional test cases are typically easy to identify [Redwine83]. Even domain testing, with its many restrictions, seems applicable, since most predicates in data processing programs are linear [White80].

b. Scientific computation

Howden analyzed a variety of verification techniques on the IMSL routines [Howden80a]. He concluded that functional and structure-oriented testing are complementary, that neither is sufficient, and that sometimes a hybrid approach is necessary to cover extremal values while simultaneously executing a particular path. Static verification methods found fewer errors in Howden's study, but their earlier application in the life cycle may increase their effectiveness. Extremal value testing and special value testing are vital to scientific programs. Statistical testing is perhaps less appropriate, since these programs are frequently constructed to solve problems whose characteristics are not known in advance. The IMSL package illustrates this; the designers of the package cannot make "reasonable" assumptions about the distribution of arguments to the sine routine, for instance.

c. Expert systems

Expert systems pose unique challenges to verification. Coverage of the executable code is of little use, since the behavior of the system is dominated by the knowledge base. Difficulties arise in assuring the consistency of this knowledge base. This problem is compounded by the reliance on human experts, since precise behavior is difficult to specify. A good survey of the problems related to validation of expert systems appears in [Hayes-Roth83].

Three steps can be identified as minimal requirements for verification of an expert system. First, it is necessary to clean up the knowledge base in much the same manner as is done for a BNF grammar. Inconsistencies must be detected, redundancies eliminated, loops broken, etc. Symbolic execution and data flow analysis appear to be applicable to this stage. Second, each piece of information in the knowledge base must be exercised. Mutation analysis applied to the knowledge base detects the information whose change does not affect the output and, thus, is not sufficiently exercised. Third, test case design and evaluation must be conducted by experts in the application domain.

d. Embedded and real-time systems

Embedded and real-time systems are perhaps the most complex systems to specify, design, and implement. It is no surprise that they are particularly hard to verify [Carver91, Tai91, Weiss88]. Embedded computer systems typically have inconvenient interfaces for defining and conducting tests. Ultimately, the code must execute on the embedded computer in its operational environment. *Operational testing* is performed in this environment.

Unit testing in an operational environment is rarely possible. The equipment is seldom available and may lack conventional input and output. In these cases, the embedded computer can be placed in a controlled environment that simulates the operational one. This provides the capability of conducting a *system test*. Timing constraints must be verified here. To assess time-critical software, it is essential to collect data in as unobtrusive a manner as possible. Typically, this requires hardware instrumentation, though software breakpoints sometimes suffice. Data from several points of instrumentation must be coordinated and analyzed; such a process is called *data reduction*.

If the simulated environment does not support unit testing, the embedded computer itself must be abstracted. The software can be written in assembly language, and the embedded computer can be simulated on another machine; or the software can be written in a high-level language, such as Ada, which can be cross-compiled to the target machine. At this level of abstraction, unit analysis and testing are possible. The goal during this stage is to assess correctness of individual units. Functional testing is essential, especially extremal, midrange, and special value testing, since it is impossible to ensure these tests will occur during integration or system testing. Data flow analysis of the code, especially if the system is written in assembly language, is appropriate. A simulator can be instrumented to collect necessary code coverage statistics and enable replaying of the previous executions [Carver91, Tai91].

Static analysis of concurrency based on symbolic execution is described in [Young88].

4. Nature of the testing environment

Available resources, personnel, and project constraints must be considered in selecting testing and analysis strategies.

a. Available resources

Available resources frequently determine the extent of testing. If the compiler does not instrument code, if data flow analysis tools are not at hand, if exotic tools for mutation testing or symbolic evaluation are not available, one must perform functional testing and instrument the code by hand to detect branch coverage. Hand instrumentation is not difficult, but it is an error-prone and time-consuming process. Editor scripts can aid in this process. If resources permit, successively more complex criteria involving branch testing, data flow testing, domain testing, and fault-based testing can be tried.

b. Personnel

No technique is without its personnel costs. Before introducing any new technique or tool, the impact on personnel must be considered. The advantages of any approach must be balanced against the effort required to learn the technique, the ongoing time demands of applying it, and the expertise it requires. Domain testing can be quite difficult to learn. Data flow analysis may uncover many anomalies that are not errors, thereby requiring personnel to sort through and distinguish them. Special value testing requires expertise in the application area. Analysis of the impact on personnel for many of the techniques in this module can be found in [Powell82].

c. Project constraints

The goal in selecting analysis and testing techniques is to obtain the most benefit from testing within the project constraints. Testing is indeed over when the budget or the time allotted to it is exhausted, but this is not an appropriate definition of when to stop testing [Myers79]. Estimates indicate that approximately 40% of software development time is used in the testing phase. Scheduling must reflect this fact.

5. Control

To ensure quality in unit analysis and testing, it is necessary to control both documentation and the conduct of the test.

a. Configuration control

Several types of items from unit analysis and testing should be placed under configuration management, including the test plan, test procedures, test data, and test results. A formal description of these and related items is found in [IEEE83]. The test plan specifies the goals, environment, and constraints imposed on testing. The test procedures detail the step-by-step activities to be performed during the test. *Regression testing* occurs when previously saved test data are used to test modified code. Its principal importance is that it ensures previously attained functionality has not been lost during a modification. Test results are recorded and analyzed for evidence of program failures. Software with a history of frequent failures may be a candidate for redesign or reimplementation.

b. Conducting tests

A *test bed* is an integrated system for testing software. Many such systems exist as commercial products. Minimally, they provide the ability to define a test case, construct a test driver, execute the test case, and capture the output. Additional facilities provided by such systems typically include data flow analysis, structural coverage assessment, regression testing, test specification, and report generation. [Frankl88] describes ASSET, a system for analyzing data flow coverage of a test case.

Glossary

The following terminology is used throughout the module, except possibly in the abstracts in the bibliography. Additional terms are defined in the text. Note that older literature is replete with inconsistencies in the use of such terms as “error,” “failure,” and “fault.” Consistent use of these terms has been attempted here, but such consistency may itself lead to confusion in the many cases where “modern” usage conflicts with prior usage in the literature.

adequacy criteria
Conditions that must be satisfied before testing is considered complete.

analysis
The process of determining software characteristics. Analysis is *dynamic* if it is execution-based, otherwise it is *static*.

coverage
Used in conjunction with a software feature or characteristic, the degree to which that feature or characteristic is tested or analyzed. Examples

include input domain coverage, statement coverage, branch coverage, and path coverage.

error

A human action that produces an incorrect result [IEEE90].

failure

The inability of a system or component to perform its required functions within specified performance requirements [IEEE90].

fault

An incorrect step, process, or data definition in a computer program [IEEE90].

oracle

A mechanized procedure that decides whether a given input-output pair is acceptable.

software characteristic

An inherent, possibly accidental, trait, quality, or property of software [IEEE87].

software feature

A software characteristic specified or implied by requirements documentation [IEEE87].

test bed

An environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test [IEEE90].

test data

Data developed to test a system or system component [IEEE83].

testing

Verifying a system or component using software characteristics derived from dynamic analysis.

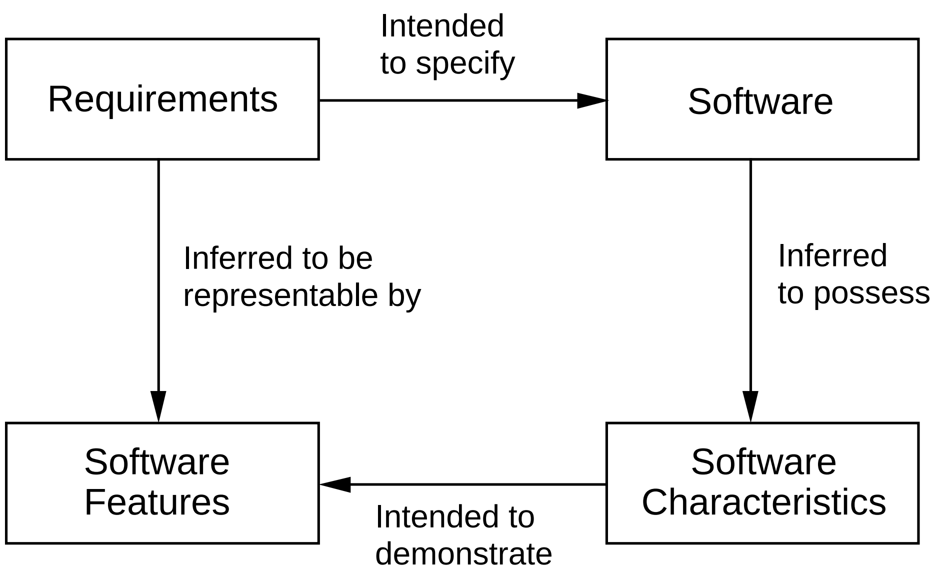
unit

A software element that can be treated meaningfully as a whole.

verification

The process of determining, for a system or component, whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (Adapted from [IEEE90].)

Figure: Software Verification Model



Teaching Considerations

Textbooks

There has been an explosion of books on the subjects of analysis and testing, only a few of which will be mentioned here. [Myers79] is dated but provides a good overview of structural coverage and some specification-based testing. It can still serve well as a supplementary text in an introductory software engineering course. [Beizer90] is eclectic, containing more testing techniques than any other reference. The text is written in a captivating style and makes frequent appeals to the experiences of actual projects. [Howden87] is the first text to approach analysis and testing within a unified framework. It contains all the necessary theoretical and practical background, and could be used as a text for a graduate seminar. [Ould86] succinctly places unit analysis and testing in its context within the overall verification effort.

To gain a full appreciation of the important issues, each of these texts must be supplemented with readings from the current literature. *Suggested Reading Lists* (page 22) contains a table categorizing entries in the annotated bibliography according to their potential use.

Suggested Schedules

The following are suggestions for using the material in this module in various classroom contexts. Numbers in parentheses represent suggested lecture hours to be allocated to each topic.

One-Term Undergraduate Introduction to Software Engineering. The large quantity of material to be covered in this course makes it difficult to deal with any topic in depth. The following minimum coverage of unit analysis and testing issues is suggested:

- Theory (0.5)
- Program Views and Related Analyses (1.5)
- Specification-Based, Implementation-Based, and Error-Based Testing (1.5)
- Managerial Aspects (1.5)

Total: 5.0 hours

Undergraduate Course on Verification Techniques. A course covering proof of correctness, review techniques, and analysis and testing provides a springboard for understanding the complicated issues of verification. Suggested coverage:

- Theory (1.5)
- Program Views and Related Analyses (4.5)
- Specification-Based Testing (3.0)
- Implementation-Based Testing (4.5)
- Error-Based Testing Testing (3.0)
- Managerial Aspects (1.5)

Total: 18 hours

Graduate Seminar on Analysis and Testing. As indicated in *Suggested Reading Lists*, there is a wealth of material to support a graduate seminar in testing. The entire outline of this module can be covered, with additional topics included as deemed appropriate. The suggestions given below focus on how this material can be taught in a seminar format.

The instructor delivers an introductory lecture in each of the major topic areas. Lectures should be based on references in the “essential” category (column 1 of the table). A subset of papers from the “recommended” list (column 2) is selected to be read by all students; one student should act as presenter for each paper. For this approach to succeed, papers and presenters must be selected well in advance, and both presenters and participants must be prepared. To ensure this advance preparation, the instructor should:

- Approve all paper selections.
- Meet with each presenter at least two weeks in advance of the presentation to answer questions, determine presentation format, and together write a set of exercises for the other students.
- Distribute the assigned reading as soon as possible and the set of exercises at least one week in advance of the presentation.
- Be prepared to assist each student at his or her presentation, if necessary.

This approach requires discipline on everyone’s part.

Broad coverage of material is aided by requiring each student to write a term paper in one of the areas related to the course. Readings listed in the "essential" and "recommended" columns provide breadth, while those categorized as "detailed" or "expert" provide depth.

Suggested coverage:

- Background (1.0)
- Program Views and Related Analyses
    - Textual and syntactic views (1.0)
    - Control flow view (2.0)
    - Data flow view (4.0)
    - Data state view (3.0)
    - Functional view (2.0)
- Specification-Based Testing
    - Testing independent of the specification technique (1.0)
    - Testing dependent on the specification technique (3.0)
- Implementation-Based Testing
    - Structure-based testing (2.0)
    - Infection-based testing (3.0)
    - Propagation-based testing (3.0)
- Error-Oriented Testing
    - Error-based testing (3.0)
    - Fault-based testing (3.0)
    - Probable correctness (1.0)
- Hybrid Testing Techniques (3.0)
- Evaluation of Unit Analysis and Testing
    - Theoretical (2.0)
    - Empirical (2.0)
- Managerial Aspects of Unit Analysis and Testing
    - Selecting techniques (1.0)
    - Configuration items (1.0)
    - Test beds (1.0)

Total: 42 hours

Exercises

It is not sufficient merely to study techniques—they must be applied to software and evaluated. Fortunately there is no lack of software to be verified! The traditional projected-oriented software engineering course clearly should have a testing component. If a testing seminar is held concurrently with such a course, the students taking the seminar can act as an independent test organization, as tool builders, as consultants, etc., for the software engineering class. Alternatively, programs can be obtained from another class or from industry for sustained testing.

In a testing seminar, the complementary benefits of specification-based and implementation-based testing can be illustrated by dividing the seminar participants into two groups. Have each group produce a specification and fault-filled program. The specification-based group receives the specification and object code from the implementation-based group, which, in turn, receives only the source code from the specification-based group. After testing is complete, the groups compare results. Roles of the two groups can then be reversed.

Testing tools are prime candidates for projects. Rudimentary test beds, data flow analyzers, and code instrumenters can be implemented in one term. Tools developed during one term can serve both as test tools and test objects for the next.

Suggested Reading Lists

The following lists categorize items in the bibliography by applicability. "Essential" reading is composed of references that provide appropriate entry points into the literature on topics treated in the module. It is not necessary for the instructor to read every one of these references, but time will be well spent reading those addressing topics that will actually be taught. Many items in this category are accessible to students as well. "Recommended" reading provides additional background, building on the groundwork laid by reading from the essential list. Readings in the "Detailed" list are narrower in scope and generally require background reading from the first two categories. These papers can serve as the basis for class projects. "Expert" reading requires background in areas of mathematics such as computability theory, statistics, or algorithm analysis. Most of the papers in this category are theoretical.

Classification of References

Essential	Recommended	Detailed	Expert
Carver91	Basili87	Bazzichi82	Budd80
DeMillo78a	Beizer90	Clarke76	Budd82
Duran84	Berztiss88	DeMillo78b	Cherniavsky87
Fosdick76	Brykczynski89	DeMillo88	Davis88
Gerhart76	Clarke82	Duncan81	Duran81
Goodenough75	Clarke89	Duran80	Gourlay83
Hamlet87	Foster80	Frankl88	Hamlet77a
Hantler76	Gannon81	Hecht77	Howden78c
Howden76	Hamlet77b	Howden77	Muchnick81
Howden80a	Hamlet88	Howden78b	Rowland81
Howden86	Hamlet90	Howden89	Weiss88
IEEE83	Hayes-Roth83	Howden90	Weyuker84
IEEE87	Howden75	Jalote89	Zeil89
IEEE90	Howden78a	Korel88b	
Laski83	Howden80b	Morell87	
Mills75	Howden82	Morell88	
Ould86	Howden87	Ntafos88	
Rapps85	Huang75	Offutt89	
Richardson85	Jachner84	Osterweil76	
Voas91	Korel87	Probert82	
Weiser84	Korel88a	Richardson89	
Weyuker80	Korel90a	Tai80	
Weyuker86	Korel90b	Weyuker88	
White80	Miller81	Woodward88	
	Mills83	Young88	
	Morell90	Zeil88	
	Myers79	Zweben89	
	Ntafos84		
	Perlman90		
	Podgurski90		
	Powell82		
	Redwine83		
	Richardson88		
	Tai91		
	Weyuker82		
	Woodward80		
	Youngblut89		
	Zeil83		

Bibliography

See *Suggested Reading Lists* (page 22) for additional information about how these references can be used in teaching.

Basili87

Basili, Victor R., and Richard W. Selby. "Comparing the Effectiveness of Software Testing Strategies." *IEEE Trans. Software Eng. SE-13*, 12 (Dec. 1987), 1278-1296.

Abstract: *This study applies an experimentation methodology to compare three state-of-the-practice software testing techniques: a) code reading by stepwise abstraction, b) functional testing using equivalence partitioning and boundary value analysis, and c) structural testing using 100 percent statement coverage criteria. The study compares the strategies in three aspects of software testing: fault detection effectiveness, fault detection cost, and classes of faults detected. Thirty-two professional programmers and 42 advanced students applied the three techniques to four unit-sized programs in a fractional factorial experimental design. The major results of this study are the following. 1) With the professional programmers, code reading detected more software faults and had a higher fault detection rate than did functional or structural testing, while functional testing detected more faults than did structural testing, but functional and structural testing were not different in fault detection rate. 2) In one advanced student subject group, code reading and functional testing were not different in faults found, but were both superior to structural testing, while in the other advanced student subject group there was no difference among the techniques. 3) With the advanced student subjects, the three techniques were not different in fault detection rate. 4) Number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested. 5) Code reading detected more interface faults than did the other methods. 6) Functional testing detected more control faults than did the other methods. 7) When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate estimates.*

"Functional testing" corresponds to specification-based testing, as used in this module. This paper should be read as much for its detailed description of experimental design as for its conclusions. The complexity of designing and conducting an experiment of this magnitude is clearly illustrated. The references provide a fairly complete list of other studies that have compared the effectiveness of software testing strategies.

This paper is important reading for anyone thinking of conducting an experiment in software testing. Basic statistical competence is assumed.

Bazzichi82

Bazzichi, Franco, and Ippolito Spadafora. "An Automatic Generator for Compiler Testing." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 343-353.

Abstract: *A new method for testing compilers is presented. The compiler is exercised by compatible programs, automatically generated by a test generator. The generator is driven by a tabular description of the source language. This description is in a formalism which nicely extends context-free grammars in a context-dependent direction, but still retains the structure and readability of BNF. The generator produces a set of programs which cover all grammatical constructions of the source language, unless user supplied directives instruct otherwise. The programs generated can also be used to evaluate the performance of different compilers of the same source language.*

A significant example from Pascal is presented, and experience with the generator is reported.

The approach taken here is one similar to that of a two-level grammar for specifying context sensitivity. The problems inherent in specifying semantic constraints on a programming language are clearly presented. However, the presentation is difficult to understand without consulting the cited references.

This paper or [Duncan81] should be read by the instructor. It is a difficult paper for students, though its goal should be apparent.

Beizer90

Beizer, Boris. *Software Testing Techniques, 2nd Ed.* New York: Van Nostrand Reinhold, 1990.

This book offers enough breadth and depth to warrant its use for a variety of academic and training courses. Using a flamboyant style that captures the reader's attention, Beizer explains a multitude of testing techniques within a management framework of his own devising. The book contains an extensive taxonomy of bugs (faults) and related bug counts. The book stresses implementation-based testing, especially path testing.

Berztiss88
Berztiss, Alfs, and Mark A. Ardis. *Formal Verification of Programs.* Curriculum Module SEI-CM-20-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Capsule Description: *This module introduces formal verification of programs. It deals primarily with proofs of sequential programs, but also with consistency proofs for data types and deduction of particular behaviors of programs from their specifications. Two approaches are considered: verification after implementation that a program is consistent with its specification, and parallel development of a program and its specification. An assessment of formal verification is provided.*

Brykczynski89
Brykczynski, Bill R., and Christine Youngblut. *Bibliography of Testing and Evaluation Reference Material.* IDA Memorandum Report M-496, Institute for Defense Analyses, Alexandria, Va., Aug. 1989.

Over 1900 entries with full abstracts are classified by author and topic. This bibliography is a companion to [Youngblut89].

Budd80
Budd, Timothy A., Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. "Theoretical and Empirical Studies on Using Program Mutations to Test the Functional Correctness of Programs." *Conf. Record 7th Ann. ACM Symp. on Principles of Prog. Lang.* New York: ACM, Jan. 1980, 220-233.

This paper presents little-known results on mutation testing, both theoretical and empirical. The theoretical section can be safely ignored, except for the analysis of decision tables and straight-line Lisp programs. The empirical results are more interesting, since they provide insight into the mutant operators used and their success on buggy programs.

The theoretical section is useful only for those who wish to pursue mutation testing at an expert level. The empirical section is of some use in demonstrating when mutation testing does and does not work.

Budd82
Budd, Timothy A., and Dana Angluin. "Two Notions of Correctness and Their Relation to Testing." *Acta Informatica 18*, 1 (1982), 31-45.

Abstract: *We consider two interpretations for what it means for test data to demonstrate correctness. For each interpretation, we examine under what conditions data sufficient to demonstrate correctness exists, and whether it can be automatically detected and/or generated. We establish the relation between these questions and the problem of deciding equivalence of two programs.*

This paper requires a good background in computability theory. A theoretical analysis of mutation testing is presented in excellent style.

This paper is for experts. Students without a course in computability will be lost.

Carver91
Carver, Richard H., and Kuo-Chung Tai. "Replay and Testing for Concurrent Programs." *IEEE Software 8*, 2 (March 1991), 66-74.

This is a good starting point for understanding problems involved in testing concurrent programs. The mechanism described by the authors enables sufficient state information of the program to be captured to allow prior behavior to be repeated for the purpose of debugging or testing.

Cherniavsky87
Cherniavsky, John C., and Carl H. Smith. "A Recursion Theoretic Approach to Program Testing." *IEEE Trans. Software Eng. SE-13*, 7 (July 1987), 777-784.

Abstract: *Inductive inference, the automatic synthesis of programs, bears certain ostensible relationships with program testing. For inductive inference, one must take a finite sample of the desired input/output behavior of some program and produce (synthesize) an equivalent program. In the testing paradigm, one seeks a finite sample for a function such that any program (in a given set) which computes something other than the object function differs from the object function on the finite sample. In both cases, the finite sample embodies sufficient knowledge to isolate the desired program from all other possibilities. These relationships are investigated and general recursion theoretic properties of testable sets of functions are exposed.*

[Budd82] presents similar theory from a programming language theoretic view. [Rowland81] presents related theory for a narrower problem domain.

This paper requires a strong background in the notations and conventions of computability theory. For experts only.

Clarke76
Clarke, Lori A. "A System to Generate Test Data and Symbolically Execute Programs." *IEEE Trans. Software Eng. SE-2,* 3 (Sept. 1976), 215-222.

Abstract: *This paper describes a system that attempts to generate test data for programs written in ANSI Fortran. Given a path, the system symboli-*

cally executes the path and creates a set of constraints on the program's input variables. If the set of constraints is linear, linear programming techniques are employed to obtain a solution. A solution to the set of constraints is test data that will drive execution down the given path. If it can be determined that the set of constraints is inconsistent, then the given path is shown to be non-executable. To increase the chance of detecting some of the more common programming errors, artificial constraints are temporarily created that simulate error conditions and then an attempt is made to solve each augmented set of constraints. A symbolic representation of the program's output variables in terms of the program's input variables is also created. The symbolic representation is in a human readable form that facilitates error detection as well as being a possible aid in assertion generation and automatic program documentation.

This paper reports on an early symbolic execution system that allows a user to specify interactively the path to be analyzed. The use of artificial constraints foreshadows the application of symbolic execution in fault-based testing.

A symbolic execution system for a simple language makes an excellent class project. This paper (along with [Howden78b] and [Howden77]) provides sufficient detail for implementing such a project.

Clarke82

Clarke, Lori A., Johnette Hassell, and Debra J. Richardson. "A Close Look at Domain Testing." *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 380-390.

Abstract: *White and Cohen have proposed the domain testing method, which attempts to uncover errors in a path domain by selecting test data on and near the boundary of the path domain. The goal of domain testing is to demonstrate that the boundary is correct within an acceptable error bound. Domain testing is intuitively appealing in that it provides a method for satisfying the often suggested guideline that boundary conditions should be tested.*

In addition to proposing the domain testing method, White and Cohen have developed a test data selection strategy, which attempts to satisfy this method. Further, they have described two error measures for evaluating domain testing strategies. This paper takes a close look at their strategy and their proposed error measures. It is shown that inordinately large domain errors may remain undetected by the White and Cohen strategy. Two alternative domain testing strategies, which improve on the error bound, are then proposed and the complexity of each of the three strategies is analyzed. Finally, several other issues that must be addressed by domain testing are presented and the general applicability of this method is discussed.

This paper recommends the selection of additional test points to narrow the range of domain shifts that remain undetected by the domain testing strategy suggested in [White80], which is prerequisite reading. The paper makes several important suggestions for relaxing the restrictions of [White80].

This is essential reading for the instructor if domain testing is to be discussed. Also, it serves as a good source of thought questions for examinations. It is advanced reading for students.

Clarke89

Clarke, Lori A., Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. "A Formal Evaluation of Data Flow Path Selection Criteria." *IEEE Trans. Software Eng. 15*, 11 (Nov. 1989), 1318-1332.

Abstract: *A number of path selection criteria have been proposed throughout the years. Unfortunately, little work has been done on comparing these criteria. To determine what would be an effective path selection criterion for revealing faults in programs, we have undertaken an evaluation of these criteria. This paper reports on the results of our evaluation of path selection criteria based on data flow relationships. We show how these criteria relate to each other, thereby demonstrating some of their strengths and weaknesses. In addition, we suggest minor changes to some criteria that improve their performance. We conclude with a discussion of the major limitations of these criteria and directions for future research.*

The authors begin with a thorough overview of the three principal data flow approaches to path selection (see [Korel83], [Ntafos84], [Rapps85], and [Ntafos88]), demonstrating the interrelationships among them using a subsumption hierarchy. They then suggest modifications to the path selection criteria to remedy noted deficiencies. The authors discuss issues other than subsumption that must be considered in evaluating the criteria: the effect of infeasible paths, the relative cost of the criteria, and the fault detection capabilities of the criteria.

By collecting the various data flow definitions in one place, the authors have done everyone a service. If any one paper on data flow analysis is to be studied by the instructor, this is probably the appropriate one. The paper, of course, contains a heavy dose of graph theory.

Davis88

Davis, Martin, and Elaine J. Weyuker. "Metric Space-Based Test-Data Adequacy Criteria." *Computer J. 31*, 1 (Feb. 1988), 17-24.

Abstract: *Since software testing cannot ordinarily be expected to provide conclusive evidence that a program is correct, software engineers have had to be satisfied with the vague notion of a set of test data being* adequate *for a given program. In this paper a theoretical model is provided for the notion of adequacy. Adequacy criteria are seen as serving to distinguish a given program from a certain class of programs. In particular, notions of* distance *between programs are studied, and adequacy of a test set is taken to mean that the set successfully distinguishes the program being tested from all programs that are sufficiently near to it, and differ in input-output behaviour from the given program. Certain points, called* critical, *are identified which must occur in every adequate test set. Finally, lower bounds are obtained on the size of test sets which are minimally adequate, in the sense that they have no adequate proper subsets.*

The concept of the *distance* between two programs is defined in terms of transformations needed to convert one program into the other. The adequacy discussed is with respect to a *finite* neighborhood (in the sense of [Budd82]), limiting the usefulness of the results.

The authors presume a strong background in programming language theory. This paper is appropriate only for those doing research into the theory of fault-based testing.

DeMillo78a

DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer 11*, 4 (April 1978), 34-41. Reprinted in [Miller81].

This paper should win a prize for introducing more catchy new terms than any other—*mutation testing, competent programmer hypothesis, coupling effect*. Beware! It is easy to fall under the spell of the latter two terms and assume they are well-defined and justified. Beware also of the typographical error that occurs in several places on page 37, where '1' is substituted for 'I'. This substitution leads to the (wrong) impression that mutation testing is performed conventionally on double mutants. The description on page 39 is confusing and seems to imply that 14 mutants are equivalent to the original, yet four of them are not. [Duran81] draws exactly the opposite conclusion based on the random generation example!

Despite flaws, this paper is an excellent introduction to mutation testing and is, therefore, essential reading for both instructor and student.

DeMillo78b

DeMillo, Richard A., and Richard J. Lipton. "A Probabilistic Remark on Algebraic Program Testing." *Information Processing Letters 7*, 4 (June 1978), 193-195.

This is the first paper to introduce the concept of determining statistical confidence in the absence of faults in programs that compute functions in a specified class (*e.g.*, polynomials).

The paper is in-depth reading for the instructor.

DeMillo88

DeMillo, Richard A., *et al*. "An Extended Overview of the Mothra Software Testing Environment." *Proc. Second Workshop on Software Testing, Verification, and Analysis*. Washington, D.C.: IEEE Computer Society Press, 1988, 142-151.

Abstract: *Mothra is a software testing environment that supports mutation-based testing of software systems. Mothra is interactive; it provides a high-bandwidth user interface to make software testing faster and less painful. Mothra currently runs on a variety of systems under 4.3BSD UNIX, UNIX System V, and ULTRIX-32 1.2. This paper begins with a brief introduction to mutation analysis. We then take the reader on a guided tour of Mothra, emphasizing how it interacts with the tester. Then we present [*sic*] with a short discussion of Mothra's internal design. Next, we discuss some major problems with using mutation analysis and discuss possible solutions. We conclude by presenting a solution to one of these problem [*sic*]—a new method of automatically generating mutation-adequate test data.*

The authors present an overview of mutation analysis and its problems, and they describe the Mothra environment, which supports mutation testing. A principal problem is the generation of mutation-adequate test data; the authors discuss heuristics for the generation of requisite test data.

This article is appropriate for the student interested in mutation operators and the operation of a mutation system.

Duncan81

Duncan, A. G., and J. S. Hutchinson. "Using Attributed Grammars to Test Designs and Implementations." *5th Intl. Conf. on Software Eng.* New York: IEEE, March 1981, 170-178.

Abstract: *We present a method for generating test cases that can be used throughout the entire life cycle of a program. This method uses attributed translation grammars to generate both inputs and outputs, which can then be used either as is, in order to test the specifications, or in conjunction with* *automatic test drivers to test an implementation against the specifications.*

The grammar can generate test cases either randomly or systematically. The attributes are used to guide the generation process, thereby avoiding the generation of many superfluous test cases. The grammar itself not only drives the generation of test cases but also serves as a concise documentation of the test plan.

In the paper, we describe the test case generator, show how it works in typical examples, compare it with related techniques, and discuss how it can be used in conjunction with various testing heuristics.

This is a practical paper on the means of generating test data based on a BNF grammar. The use of "attributes" here is unconventional and is not directly related to attribute grammars.

This paper or [Bazzichi82] should be read by the instructor. It can be useful for in-depth study by the student.

Duran80

Duran, Joe W., and John J. Wiorkowski. "Quantifying Software Validity by Sampling." *IEEE Trans. on Reliability R-29,* 2 (June 1980), 141-144.

Abstract: *The point of all validation techniques is to raise assurance about the program under study, but no current methods can be realistically thought to give 100% assurance that a validated program will perform correctly. There are currently no useful ways for quantifying how 'well-validated' a program is. One measure of program correctness is the proportion of elements in the program's input domain for which it fails to execute correctly, since the proportion is zero i.f.f. the program is correct. This proportion can be estimated statistically from the results of program tests and from prior subjective assessments of the program's correctness. Three examples are presented of methods for determining* s-*confidence bounds on the failure proportion. It is shown that there are reasonable conditions (for programs with a finite number of paths) for which ensuring the testing of all paths does* not *give better assurance of program correctness.*

The authors are interested in program testing, particularly in quantifying how well a program has been tested. Both random testing and path testing are considered. A strong statistical background is presumed.

This is expert reading for the instructor.

Duran81

Duran, Joe W., and John J. Wiorkowski. "Capture-Recapture Sampling for Estimating Software Error Content." *IEEE Trans. Software Eng. SE-7,* 1 (Jan. 1981), 147-148.

Abstract: *Mills' capture-recapture sampling method allows the estimation of the number of errors in a program by randomly inserting known errors and then testing the program for both inserted and indigenous errors. This correspondence shows how correct confidence limits and maximum likelihood estimates can be obtained from the test results. Both fixed sample size testing and sequential testing are considered.*

It is essential that Mills's original article be read first (see Chapter 9 of [Mills83]). A strong statistics background is needed.

This reading is for experts only.

Duran84

Duran, Joe W., and Simeon C. Ntafos. "An Evaluation of Random Testing." *IEEE Trans. Software Eng. SE-10,* 4 (July 1984), 438-444.

Abstract: *Random testing of programs has usually (but not always) been viewed as a worst case of program testing. Testing strategies that take into account the program structure are generally preferred. Path testing is an often proposed ideal for structural testing. Path testing is treated here as an instance of partition testing, where by partition testing is meant any testing scheme which forces execution of at least one test case from each subset of a partition of the input domain. Simulation results are presented which suggest that random testing may often be more cost effective than partition testing schemes. Also, results of actual random testing experiments are presented which confirm the viability of random testing as a useful validation tool.*

This paper challenges many ideas about program testing, especially the notion that random testing is of no value. Experiments were conducted to validate an error model, and the structural coverage accomplished by such testing is reported. Knowledge of statistics helps.

This is essential reading for the instructor. It is challenging for the student, but it should be read.

Fosdick76

Fosdick, Lloyd D., and Leon J. Osterweil. "Data Flow Analysis in Software Reliability." *ACM Computing Surveys 8,* 3 (Sept. 1976), 305-330.

Abstract: *The ways that the methods of data flow analysis can be applied to improve software reliability are described. There is also a review of the basic terminology from graph theory and from data flow analysis in global program optimization. The notation of regular expressions is used to describe actions on data for sets of paths. These expressions provide the basis of a classification scheme for data flow which represents patterns of* *data flow along paths within subprograms and along paths which cross subprogram boundaries. Fast algorithms, originally introduced for global optimization, are described and it is shown how they can be used to implement the classification scheme. It is then shown how these same algorithms can also be used to detect the presence of data flow anomalies which are symptomatic of programming errors. Finally, some characteristics of and experience with DAVE, a data flow analysis system embodying some of these ideas, are described.*

This article is a most readable and thorough introduction to data flow analysis. Read this first and compare with [Jachner84].

This is essential reading for both instructor and student.

Foster80

Foster, Kenneth A. "Error Sensitive Test Cases Analysis (ESTCA)." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 258-264.

Abstract: *A hardware failure analysis technique adapted to software yielded three rules for generating test cases sensitive to code errors. These rules, and a procedure for generating these cases, are given with examples. Areas for further study are recommended.*

A set of error-sensitive test case analysis rules are given for producing inputs that are "error-sensitive." The rules are *ad hoc*, and no theoretical justification is given for them. Results of this paper are clarified in *Software Engineering Notes 10*, 1 (Jan. 1985), 62-67.

This paper contains many classical examples and is useful for that reason. It is not essential reading, but it raises many questions about why the proposed ideas seems to work.

Frankl88

Frankl, Phyllis G., and Elaine J. Weyuker. "An Applicable Family of Data Flow Testing Criteria." *IEEE Trans. Software Eng. 14*, 10 (Oct. 1988), 1483-1498.

Abstract: *A test data adequacy criterion is a predicate which is used to determine whether a program has been tested "enough." An adequacy criterion is applicable if for every program there exists a set of test data for the program which satisfies the criterion. Most test data adequacy criteria based on path selection fail to satisfy the applicability property because, for some programs with unexecutable paths, no adequate set of test data exists.*

In this paper, we extend the definitions of the previously introduced family of data flow testing criteria to apply to programs written in a large subset of Pascal. We then define a family of adequacy criteria called feasible data flow testing criteria, which are derived from the data flow testing criteria. The feasible data flow testing criteria circumvent the problem of nonapplicability of the data flow testing criteria by requiring the test data to exercise only those definition-use associations which are executable. We show that there are significant differences between the relationships among the data flow testing criteria and the relationships among the feasible data flow testing criteria.

We also discuss a generalized notion of the executability of a path through a program unit. A script of a testing session using our data flow testing tool, ASSET, is included in the Appendix.

The emphasis in this paper is on the term "feasible." [Clarke89] points out that this shift in concern does not entirely resolve undecidable issues. It is crucial to read [Rapps85] before reading this paper, and perhaps [Clarke89] as well.

This paper requires significant background in data flow testing.

Gannon81

Gannon, John, Paul R. McMullin, and Richard G. Hamlet. "Data-Abstraction, Implementation, Specification, and Testing." *ACM Trans. Prog. Lang. and Syst. 3*, 3 (July 1981), 211-223.

Abstract: *A compiler-based system DAISTS that combines a data-abstraction language (derived from the SIMULA* **class***) with specification by algebraic axioms is described. The compiler, presented with two independent syntactic objects in the axioms and implementing code, compiles a "program" that consists of the former as test driver for the latter. Data points, in the form of expressions using the abstract functions and constant values, are fed to this program to determine if the implementation and axioms agree. Along the way, structural testing measures can be applied to both code and axioms to evaluate the test data. Although a successful test does not conclusively demonstrate the consistency of axioms and code, in practice the tests are seldom successful, revealing errors. The advantage over conventional programming systems is threefold:*

(1) The presence of the axioms eliminates the need for a test oracle; only inputs need be supplied.

(2) Testing is automated: a user writes axioms, implementation, and test points; the system writes the test drivers.

(3) The results of tests are often surprising and helpful because it is difficult to get away with "trivial" tests: what is not significant for the code is

liable to be a severe test of the axioms, and vice versa.

The system described here covers diverse aspects of program testing. It is a specification-dependent hybrid approach that takes advantage of the orthogonality between implementations and algebraic axioms.

This paper is recommended reading for the instructor. With some background in algebraic specification, students can readily comprehend the system.

Gerhart76

Gerhart, Susan L., and Lawrence Yelowitz. "Observations of Fallibility in Applications of Modern Programming Methodologies." *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 1976), 195-207.

Abstract: *Errors, inconsistencies, or confusing points are noted in a variety of published algorithms, many of which are being used as examples in formulating or teaching principles of such modern programming methodologies as formal specification, systematic construction, and correctness proving. Common properties of these points of contention are abstracted. These properties are then used to pinpoint possible causes of the errors and to formulate general guidelines which might help to avoid further errors. The common characteristic of mathematical rigor and reasoning in these examples is noted, leading to some discussion about fallibility in mathematics, and its relationship to fallibility in these programming methodologies. The overriding goal is to cast a more realistic perspective on the methodologies, particularly constructive recommendations for their improvement.*

This paper is a masterpiece of analysis of how errors occur in the life cycle. Though the authors "nit-pick" in places, they succeed in convincing the most adamant skeptic of the need for dynamic testing of computer programs. The paper is best understood after some formal specifications and proofs of correctness are attempted.

This paper is essential reading for both instructor and student.

Goodenough75

Goodenough, John B., and Susan L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 156-173. Reprinted in [Miller81].

Abstract: *This paper examines the theoretical and practical role of testing in software development. We prove a fundamental theorem showing that properly structured tests are capable of demonstrating the absence of errors in a program. The theorem's proof hinges on our definition of test reliability and validity, but its practical utility hinges on being able to show when a test is actually reliable. We explain what makes tests unreliable (for example, we show by example why testing all program statements, predicates, or paths is not usually sufficient to insure test reliability), and we outline a possible approach to developing reliable tests. We also show how the analysis required to define reliable tests can help in checking a program's design and specifications as well as in preventing and detecting implementation errors.*

Despite the flaws indicated in [Weyuker80], this paper remains a classic. It is essential reading for the instructor. Students find it very difficult; do not use it as an introduction to testing!

Gourlay83

Gourlay, John S. "A Mathematical Framework for the Investigation of Testing." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 686-709.

Abstract: *Testing has long been in need of mathematical underpinnings to explain its value as well as its limitations. This paper develops and applies a mathematical framework that 1) unifies previous work on the subject, 2) provides a mechanism for comparing the power of methods of testing programs based on the degree to which the methods approximate program verification, and 3) provides a reasonable and useful interpretation of the notion that successful tests increase one's confidence in the program's correctness.*

Applications of the framework include confirmation of a number of common assumptions about practical testing methods. Among the assumptions confirmed is the need for generating tests from specifications as well as programs. On the other hand, a careful formal analysis shows that the "competent programmer hypothesis" does not suffice to ensure the claimed high reliability of mutation testing. Hardware testing is shown to fit into the framework as well, and a brief consideration of it shows how the practical differences between it and software testing arise.

This paper is expert reading.

Hamlet77a

Hamlet, Richard G. "Testing Programs with Finite Sets of Data." *Computer J. 20*, 3 (Aug. 1977), 232-237.

Abstract: *The techniques of compiler optimization can be applied to aid a programmer in writing a program which cannot be improved by these techniques. A finite, representative set of test data can be useful in this process. This paper presents the theoretical basis for the (nonconstructive) existence of test sets which serves as maximally effective* *stand-ins for an unlimited number of input possibilities. It is argued that although the time required by a compiler to fully exercise a program on a set of data may be large, the corresponding improvement in the reliability of the program may also be large if the set meets the given theoretical requirements.*

As a theoretical companion to [Hamlet77b], this paper explores the notion of assessing test data adequacy via program mutations. The article requires some background in computability, especially in reduction proofs involving the halting problem.

The paper could be used as an introduction to computability for students with limited background; all its theorems are relevant to issues involved in program testing.

Hamlet77b

Hamlet, Richard G. "Testing Programs with the Aid of a Compiler." *IEEE Trans. Software Eng. SE-3*, 4 (July 1977), 279-290.

Abstract: *If finite input-output specifications are added to the syntax of programs, these specifications can be verified at compile time. Programs which carry adequate tests with them in this way should be resistant to maintenance errors. If the specifications are independent of program details they are easy to give, and unlikely to contain errors in common with the program. Furthermore, certain finite specifications are maximal in that they exercise the control and expression structure of a program as well as any tests can.*

A testing system based on a compiler is described, in which compiled code is utilized under interactive control, but "semantic" errors are reported in the style of conventional syntax errors. The implementation is entirely in the high-level language on which the system is based, using some novel ideas for improving documentation without sacrificing efficiency.

This paper provides an excellent description of a system that anticipated many of the fault-based methods of program testing practice and theory. It represents the first fault-based system using program mutation in a context that determines test data adequacy by demonstrating that no simpler programs can be substituted for the original and still pass the test.

The paper is easy to understand and motivates discussion of mutation testing and test data adequacy. It is recommended reading for the instructor; for the student, it provides an interesting comparison of tradeoffs among mutation methods.

Hamlet87

Hamlet, Richard G. "Probable Correctness Theory." *Information Processing Letters 25*, 1 (April 1987), 17-25.

Abstract: *A theory of 'probable correctness' is proposed to assess the reliability of software through testing. Current research in testing is not adequate for this assessment. Most testing methods are intended for debugging, to find failures and connect them to program faults for repair. When these methods no longer expose errors, no analysis has been done to find the confidence that may be placed in the software. (Preliminary results here are that this confidence should be low.) Other work applies conventional decision theory to inputs as samples of a program's use. The application is suspect because the necessary independence and distribution assumptions may be violated; in any case, the results are intuitively incorrect. The proposed theory relies on a uniform distribution of test samples, but relates these to textually occurring faults. Preliminary results include an analysis of partition testing, and suggestions for textual sampling. It is crucial that any such confidence theory be plausible, so the foundations of program sampling are examined in detail.*

This paper lays the foundation for a new area of investigation in program testing. Probable correctness theory estimates the probability that a program has no faults. (Reliability theory, on the other hand, tries to bound the probability that a program will fail.) This theory provides a means of computing bounds on the trustworthiness of software. The theory is improved in [Hamlet90].

Understanding the probability model developed here requires a significant investment on the part of the reader. It explores different sample spaces in which faults may lie.

This paper is essential reading for the instructor who wants to discuss the statistical confidence that can be associated with a successful test.

Hamlet88

Hamlet, Richard G. "Special Section on Software Testing." *Comm. ACM 31*, 6 (June 1988), 662-667.

Hamlet provides an overview of three papers on testing. He discusses difficulties encountered in trying to infer statistical confidence based upon test results.

Hamlet90

Hamlet, Richard G., and Ross Taylor. "Partition Testing Does Not Inspire Confidence." *IEEE Trans. Software Eng. 16*, 12 (Dec. 1990), 1402-1411.

Abstract: *Partition testing, in which a program's* *input domain is divided according to some rule and tests conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes subdomain boundaries and random sampling that ignores the partition gives the counterintuitive result that partitioning is of little value. In this paper we improve the negative results published about partition testing, and try to reconcile them with its intuitive value. Theoretical models allow us to study partition testing in the abstract, and to describe the circumstances under which it should perform well at failure detection. Partition testing is shown to be more valuable when the partitions are narrowly based on expected failures and there is a good chance that failures occur. For gaining confidence from successful tests, partition testing as usually practiced has little value.*

By "confidence," the author means statistical confidence. The paper challenges many long-held beliefs about the value of partition testing, in particular, its value in ensuring overall confidence in the correct operation of the program. Hamlet reviews and extends the work of Duran and Ntafos [Duran84] on random testing on a failure-rate model of program confidence and presents his own defect-rate model of probable correctness for assessing partition testing.

This paper requires some background in probability and statistics. It may be necessary to read some of the cited references before fully comprehending the material.

Hantler76

Hantler, Sidney L., and James C. King. "An Introduction to Proving the Correctness of Programs." *ACM Computing Surveys 8*, 3 (Sept. 1976), 331-353. Reprinted in [Miller81].

Abstract: *This paper explains, in an introductory fashion, the method of specifying the correct behavior of a program by the use of input/output assertions and describes one method for showing that the program is correct with respect to those assertions. An initial assertion characterizes conditions expected to be true upon entry to the program and a final assertion characterizes conditions expected to be true upon exit from the program. When a program contains no branches, a technique known as symbolic execution can be used to show that the truth of the initial assertion upon entry guarantees the truth of the final assertion upon exit. More generally, for a program with branches one can define a symbolic execution tree. If there is an upper bound on the number of times each loop in such a program may be executed, a proof of correctness can be given by a simple traversal of the (finite) symbolic execution tree.*

However, for most programs, no fixed bound on the number of times each loop is executed exists and the corresponding symbolic execution trees are infinite. In order to prove the correctness of such programs, a more general assertion structure must be provided. The symbolic execution tree of such programs must be traversed inductively rather than explicitly. This leads naturally to the use of additional assertions which are called "inductive assertions."

This highly readable article provides a gentle introduction to three important areas: program correctness, formal verification, and symbolic execution.

The instructor who needs to learn about the relationship of symbolic execution to verification should begin here. Additional references can be found in [Berztiss88]. This is ideal reading for students.

Hayes-Roth83

Hayes-Roth, Frederick, and Donald Arthur Waterman, eds. *Building Expert Systems*. Reading, Mass.: Addison-Wesley, 1983.

This book outlines verification activities applicable to expert systems.

Hecht77

Hecht, Matthew S. *Flow Analysis of Computer Programs*. New York: Elsevier North-Holland, 1977.

This is the standard text covering the theory of data flow analysis as applied to program optimization. Application of data flow analysis to verification is not covered.

Howden75

Howden, William E. "Methodology for the Generation of Program Test Data." *IEEE Trans. Computers C-24*, 5 (May 1975), 554-560.

Abstract: *A methodology for generating program test data is described. The methodology is a model of the test data generation process and can be used to characterize the basic problems of test data generation. It is well defined and can be used to build an automatic test data generation system.*

The methodology decomposes a program into a finite set of classes of paths in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. The test data generation problem is theoretically unsolvable: there is no algorithm which, given any class of paths, will either generate a test case that causes some path in that class to be followed or determine that no such data exist. The methodology attempts to generate test data for as many of the classes of paths as possible. It operates by constructing *descriptions of the input data subsets which cause the classes of paths to be followed. It transforms these descriptions into systems of predicates which it attempts to solve.*

This paper contains a nuts-and-bolts presentation of symbolic execution techniques.

The instructor may find this paper useful, but dated. Students who are not enthusiastic about using structural coverage to generate test data should avoid this one.

Howden76

Howden, William E. "Reliability of the Path Analysis Testing Strategy." *IEEE Trans. Software Eng. SE-2*, 3 (Sept. 1976), 208-215. Reprinted in [Miller81].

Abstract: *A set of test data T for a program P is* reliable *if it reveals that P contains an error whenever P is incorrect. If a set of tests T is reliable and P produces the correct output for each element of T then P is a correct program. Test data generation strategies are procedures for generating sets of test data. A testing strategy is reliable for a program P if it produces a reliable set of test data for P. It is proved that an effective testing strategy which is reliable for all programs cannot be constructed. A description of the path analysis testing strategy is presented. In the path analysis strategy data are generated which cause different paths in a program to be executed. A method for analyzing the reliability of path testing is introduced. The method is used to characterize certain classes of programs and program errors for which the path analysis strategy is reliable. Examples of published incorrect programs are included.*

This is an excellent paper, which established much of the terminology and influenced much of the work in path testing.

This is essential reading for both the instructor and student.

Howden77

Howden, William E. "Symbolic Testing and the DISSECT Symbolic Evaluation System." *IEEE Trans. Software Eng. SE-3*, 4 (July 1977), 266-278. Reprinted in [Miller81].

Abstract: *Symbolic testing and a symbolic evaluation system called DISSECT are described. The principle features of DISSECT are outlined. The results of two classes of experiments in the use of symbolic evaluation are summarized. Several classes of program errors are defined and the reliability of symbolic testing in finding bugs is related to the classes of errors. The relationship of symbolic evaluation systems like DISSECT to classes of program errors and to other kinds of program testing and program analysis tools is also discussed. Desirable improvements in DISSECT, whose importance was revealed by the experiments, are mentioned.*

This paper provides a detailed look into the strengths and weaknesses of a symbolic execution system. Several interesting notions are introduced, such as using two-dimensional output to improve readability of symbolic output and using a path description language. For more detailed information on the DISSECT system, see [Howden78b].

The paper is necessary only for in-depth understanding of symbolic execution. It is easily understood by students.

Howden78a

Howden, William E. "Theoretical and Empirical Studies of Program Testing." *IEEE Trans. Software Eng. SE-4*, 4 (July 1978), 293-298.

Abstract: *Two approaches to the study of program testing are described. One approach is theoretical and the other empirical. In the theoretical approach situations are characterized in which it is possible to use testing to formally prove the correctness of programs or the correctness of properties of programs. In the empirical approach testing strategies reveal the errors in a collection of programs. A summary of the results of two research projects which investigated these approaches are presented. The differences between the two approaches are discussed and their relative advantages and disadvantages are compared.*

This paper is recommended reading for the instructor who wishes to compare the theoretical approach with the empirical approach. It is readily understood by students.

Howden78b

Howden, William E. "DISSECT—A Symbolic Evaluation and Program Testing System." *IEEE Trans. Software Eng. SE-4*, 1 (Jan. 1978), 70-73.

Abstract: *The basic features of the DISSECT symbolic testing tool are described. Usage procedures are outlined and the special advantages of the tool are summarized. Cost estimates for using the tool are provided and the results of experiments to determine its effectiveness are included. The background and history of the development of the tool are outlined. The availability of the tool is described and a listing of reference materials is included.*

This paper provides detailed information in the use of a batch-oriented symbolic execution system. For a broader perspective, see [Howden77] and [Muchnick81].

The paper is necessary only for in-depth understanding of symbolic execution. It should be read with [Clarke76].

Howden78c

Howden, William E. “Algebraic Program Testing.” *Acta Informatica 10*, 1 (1978), 53-66.

Abstract: *An approach to the study of program testing is introduced in which program testing is treated as a special kind of equivalence problem. In this approach, classes of programs* P* *and associated classes of test sets* T* *are defined which have the property that if two programs* P *and* Q *in* P* *agree on a set of tests from* T*, *then* P *and* Q *are computationally equivalent. The properties of a class* P* *and the associated class* T* *can be thought of as defining a set of assumptions about a hypothetical correct version* Q *of a program* P *in* P*. *If the assumptions are valid then it is possible to prove the correctness of* P *by testing. The main result of the paper is an equivalence theorem for classes of programs which carry out sequences of computations involving the elements of arrays.*

This reading is for expert knowledge.

Howden80a

Howden, William E. “Applicability of Software Validation Techniques to Scientific Programs.” *ACM Trans. Prog. Lang. and Syst. 2*, 3 (July 1980), 307-320. Reprinted in [Miller81].

Abstract: *Error analysis involves the examination of a collection of programs whose errors are known. Each error is analyzed and validation techniques which would discover the error are identified. The errors that were present in version five of a package of Fortran scientific subroutines and then later corrected in version six were analyzed. An integrated collection of static and dynamic analysis methods would have discovered the error in version five before its release. An integrated approach to validation and the effectiveness of individual methods are discussed.*

The author gives an excellent description of what errors are discovered by what techniques.

This paper is essential reading for the instructor and student alike.

Howden80b

Howden, William E. “Functional Program Testing.” *IEEE Trans. Software Eng. SE-6*, 2 (March 1980), 162-169.

Abstract: *An approach to functional testing is described in which the design of a program is viewed as an integrated collection of functions. The selection of test data depends on the functions used in the design and on the value spaces over which the functions are defined. The basic ideas on the method were developed during the study of a collection of scientific programs containing errors. The method was the most reliable testing technique for discovering the errors. It was found to be significantly more reliable than structural testing. The two techniques are compared and their relative advantages and limitations are discussed.*

By “functional program testing,” Howden means testing those aspects of a program that have any form of external specification, including design documents or even comments within the code.

This paper is a precursor of [Howden86].

Howden82

Howden, William E. “Weak Mutation Testing and Completeness of Test Sets.” *IEEE Trans. Software Eng. SE-8*, 4 (July 1982), 371-379.

Abstract: *Different approaches to the generation of test data are described.* Error-based *approaches depend on the definition of classes of commonly occurring program errors. They generate tests which are specifically designed to determine if particular classes of errors occur in a program. An error-based method called* weak mutation testing *is described. In this method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests. The method is systematic, and a tool can be built to help the user apply the method. It is extensible in the sense that it can be extended to cover additional classes of errors. Its relationship to other software testing methods is discussed. Examples are included.*

Different approaches to testing involve different concepts of the adequacy or completeness *of a set of tests. A formalism for characterizing the completeness of test sets that are generated by error-based methods such as weak mutation testing as well as the test sets generated by other testing methods is introduced. Error-based, functional, and structural testing emphasize different approaches to the test data generation problem. The formalism which is introduced in the paper can be used to describe their common basis and their differences.*

Weak mutation testing provides a viable alternative to its more expensive cousin, mutation testing, and bears close resemblance to the system described in [Hamlet77a]. This paper formalizes the notion of completeness of a test set based on its ability to detect local changes to the code. A good comparison of testing methods is made, using notation introduced in the paper. The paper is more easily understood if [DeMillo78], [White80], and [Foster80] are read first.

This paper is recommended for the instructor, especially if error-based or fault-based testing is to be covered in depth. Given sufficient background, students should find the paper accessible. It could form the basis of a class project to develop a weak mutation system.

Howden86

Howden, William E. “A Functional Approach to Program Testing and Analysis.” *IEEE Trans. Software Eng. SE-12,* 10 (Oct. 1986), 997-1005.

Abstract: *An integrated approach to testing is described which includes both static and dynamic analysis methods and which is based on theoretical results that prove both its effectiveness and efficiency. Programs are viewed as consisting of collections of functions that are joined together using elementary functional forms or complex functional structures.*

Functional testing is identified as the input-output analysis of functional forms. Classes of faults are defined for these forms and results presented which prove the fault revealing effectiveness of well defined sets of tests.

Functional analysis is identified as the analysis of the sequences of operators, functions, and data type transformations which occur in functional structures. Functional trace analysis involves the examination of the sequences of function calls which occur in a program path; operator sequence analysis the examination of the sequences of operators on variables, data structures, and devices; and data type transformation analysis the examination of the sequences of transformations on data types. Theoretical results are presented which prove that it is only necessary to look at interfaces between pairs of operators and data type transformations in order to detect the presence of operator or data type sequencing errors. The results depend on the definition of normal forms for operator and data type sequencing diagrams.

This paper represents the culmination of the development of Howden’s ideas on program testing as a full-blown theory. It summarizes his book [Howden87] and should be consulted before selecting the book in a course. By an interesting twist of terminology, Howden has managed to incorporate all of structural testing into functional testing. He presumes the availability of external functions that specify the behavior of components of the program, even those as small as an expression. Thus, conventional structural issues such as branch testing are converted into questions like, “Does this condition compute this (externally defined) function?” Of course, the existence of these external functions for every line of code can be questioned, but Howden has a quick reply—you can use the code to generate the function! While such sleight-of-hand may be disturbing at first, it is clear that in some cases this procedure is appropriate, as when a section of code fits a standard paradigm and is headed by a comment such as “sort list.” To understand Howden’s development fully, it is necessary to see his progress through several papers, especially [Howden76], [Howden80b], and [Howden82].

This paper is essential reading for the instructor. The presentation is at such a high level that it will be difficult for an uninitiated student to understand, even though it is very well written.

Howden87

Howden, William E. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

This book contains an excellent chapter on theoretical foundations of program testing, including material found nowhere else. The model of functional testing and analysis presented in the book requires detailed internal specifications of behavior, however, which are seldom available. Extensions to the model are seen in [Howden89].

Howden89

Howden, William E. “Validating Programs Without Specifications.” *ACM Software Eng. Notes 14*, 8 (Dec. 1989), 2-9.

This article does not contain an abstract, but its contents are summarized in the following excerpt:

> In the error based approach to program testing and analysis, the focus is on errors that a programmer or designer may make during the software development process, and on techniques that can be used to detect their occurrence. ... It is often the case that a program is constructed without any formal, detailed specification. In this case the code itself is the only complete specification. This means that the only way to verify such a program is to ensure that no errors were made by the programmer during programming. The term “errors” here means errors that occur due to human fallibility. This requires that we study the ways in which humans make mistakes in the construction of artifacts, and then build methods to detect when they have occurred. ... We have used a simple model in which human errors are classified as being either errors of *decomposition* or errors of *abstraction*. ... Flavor analysis is a kind of dynamic type checking. It allows the programmer to document properties of objects that change during the operation of a program, and to check if assumptions about an object’s current set of properties are correct.

This article provides excellent motivation for the use of flavor analysis in large systems for the detection of decomposition errors. It complements the work found in [Howden90].

Howden90

Howden, William E. "Comments Analysis and Programming Errors." *IEEE Trans. Software Eng. 16*, 1 (Jan. 1990), 72-81.

Abstract: *Software validation is treated as the problem of detecting errors that programmers make during the software development process. This includes fault detection, in which the focus is on techniques for detecting the occurrence of local errors which result in well defined classes of program statement faults. It also includes detecting other kinds of errors, such as* decomposition errors. *These occur when there is an inconsistency between two parts of a program and are the result of a false assumption made in one part of the program about the properties of some other part. The main focus of the paper is on a decomposition error analysis technique called* comments analysis. *In this technique, errors are detected by analyzing special classes of program comments. Comments analysis has been applied to a variety of different kinds of systems, including both a data processing program and an avionics real-time program. The use of comments analysis for sequential and concurrent systems is discussed and the basic features of comments analysis tools are summarized. The relationship of comments analysis to other techniques, such as event sequence analysis, are discussed, and the differences between it and earlier work are explained.*

This paper is not primarily directed to unit testing. It is included here because it illustrates one practical method of employing error-based knowledge in the testing process, especially in the context of concurrent programming. Related work is found in [Howden89].

Huang75

Huang, J. C. "An Approach to Program Testing." *ACM Computing Surveys 8*, 3 (Sept. 1975), 113-128. Reprinted in [Miller81].

Abstract: *One of the practical methods commonly used to detect the presence of errors in a computer program is to test it for a set of test cases. The probability of discovering errors through testing can be increased by selecting test cases in such a way that each and every branch in the flowchart will be traversed at least once during the test. This tutorial describes the problems involved and the methods that can be used to satisfy the test requirement.*

This paper discusses a method for determining path conditions to enable achievement of branch coverage.

The paper is very easy to understand and should cause no problems for students. It will introduce them to predicate calculus notation for expressing path conditions. It is recommended reading for both instructor and students.

IEEE83

IEEE. *IEEE Standard for Software Test Documentation*, ANSI/IEEE Std 829-1983. New York: IEEE, 1983.

Test documentation analogous to the documentation of the traditional waterfall life-cycle development model is defined and illustrated in this standard.

IEEE87

IEEE. *IEEE Standard for Software Unit Testing*, ANSI/IEEE Std 1008-1987. New York: IEEE, 1987.

The processes and products of unit testing are defined and illustrated in this standard.

IEEE90

IEEE. *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std 610.12-1990. New York: IEEE, 1990.

This is a revision and re-designation of an earlier glossary, ANSI/IEEE Std 729-1983). The "Corrected Edition" is dated Feb. 1991, but it retains "1990" in its number.

Students should not only learn and employ acceptable terminology, but they should also learn why standardized terminology is important. Comparing definitions here with those that appear in books and papers will help them learn both lessons.

Jachner84

Jachner, Jacek, and Vinod K. Agarwal. "Data Flow Anomaly Detection." *IEEE Trans. Software Eng. SE-10*, 4 (July 1984), 432-437.

Abstract: *The occurrence of a data flow anomaly is often an indication of the existence of a programming error. The detection of such anomalies can be used for detecting errors and to upgrade software quality. This paper introduces a new, efficient algorithm capable of detecting anomalous data flow patterns in a program represented by a graph. The algorithm based on static analysis scans the paths entering and leaving each node of the graph to reveal anomalous data action combinations. An algorithm implementing this type of approach was proposed by Fosdick and Osterweil [2]. Our approach presents a general framework which not* *only fills a gap in the previous algorithm, but also provides time and space improvements.*

This paper corrects a problem in [Fosdick76] and cannot be understood without having read that article.

Instructors who use [Fosdick76] must also read this paper. The paper opens up the possibility of a meta-discussion about the need to analyze papers critically. The shock effect on students of the reliability of published papers is not to be underestimated. Other illustrations of the need for critical analysis can be found in [Gerhart76], [Weyuker80], and [Zweben89].

Jalote89

Jalote, Pankaj. "Testing the Completeness of Specifications." *IEEE Trans. Software Eng. 15*, 5 (May 1989), 526-31.

Abstract: *Specifications are means to define formally the behavior of a system or a system component. Completeness is a desirable property for specifications. In this paper, we describe a system that* tests *for the completeness of axiomatic specifications of abstract data types. For testing, the system generates a set of test cases and an implementation of the data type from the specifications. The generated implementation is such that if the specifications are not complete, the implementation is not complete, and the behavior of all of the sequences of valid operations on the data type is not defined. This implementation is tested with the generated test cases to detect the incompleteness of specifications. The system is implemented on a VAX system running Unix.*

The paper illustrates fault-based testing of a *specification*. The fault under consideration is one of missing axioms. A brief overview of algebraic specifications is given. The paper defines an ADT to be sufficiently complete if and only if, for every possible instance of the abstract type, the result of all behavior operations is defined by the specifications. The paper considers only incompleteness caused by missing axioms. It presents heuristics based on test data generated from the syntactic portion of the specification for discovering that omission. The paper cites further references to the implementation of the system described.

The paper raises some interesting problems, such as whether heuristics exist for discovering other classes of faults in algebraic specifications, and whether testing categorically proves the absence of those faults.

This paper is appropriate for students only after they have been exposed to the principles of algebraic specifications. It can be used to illustrate how fault-based testing techniques can be applied to testing specifications.

Korel87

Korel, Bogdan. "The Program Dependence Graph in Static Program Testing." *Information Processing Letters 24*, 2 (Jan. 1987), 103-108.

Abstract: *In this paper, new techniques for static program testing are presented. The techniques are based on the program dependence graph, which models the structure of the program in terms of data and control dependences between program instructions. First, a new approach for redundant code detection is proposed. The main idea is based on the observation that each program instruction should have influence on the output of the program, otherwise it is considered redundant. Second, an input output relationship analysis, which reflects the influence of specific input data on specific output data of the program, is proposed. It is shown that the presented techniques can increase the number of detectable errors as compared with error detection through data flow analysis alone.*

This paper is crucial to understanding the work of Korel and others who seek to use data flow information in novel ways. The paper is self-contained and is essential reading for anyone studying methods of representing data flow information.

Korel88a

Korel, Bogdan, and Janusz Laski. "Dynamic Program Slicing." *Information Processing Letters 29*, 3 (Oct. 1988), 155-163.

Abstract: *A dynamic program slice is an executable subset of the original program that produces the same computations on a subset of selected variables and inputs. It differs from the static slice (Weiser, 1982, 1984) in that it is entirely defined on the basis of a computation. The two main advantages are the following: Arrays and dynamic data structures can be handled more precisely and the slice can be significantly reduced, leading to a finer localization of the fault. The approach is being investigated as a possible extension of the debugging capabilities of STAD, a recently developed System for Testing and Debugging (Korel and Laski, 1987; Laski, 1987).*

This paper should be read before [Korel90b].

Korel88b

Korel, Bogdan, and Janusz Laski. "STAD—A System For Testing and Debugging: User Perspective." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 13-20.

Abstract: *A recently developed, experimental, integrated System for Testing and Debugging is presented. Its testing part supports three data flow*

coverage criteria. The debugging part guides the programmer in the localization of faults by generating and interactively verifying hypotheses about their location.

The three data flow coverage criteria referred to are U- and L-context testing and chain testing. The tool reports coverage in terms of these three criteria.

This paper extends the concepts found in [Laski83]. It would serve well as an example of program instrumentation.

Korel90a

Korel, Bogdan. "Automated Software Test Data Generation." *IEEE Trans. Software Eng. 16*, 8 (Aug. 1990), 870-879.

Abstract: *Test data generation in program testing is the process of identifying a set of test data which satisfies given testing criterion. Most of the existing test data generators ... use symbolic evaluation to derive test data. However, in practical programs this technique frequently requires complex algebraic manipulations, especially in the presence of arrays. In this paper we present an alternative approach of test data generation which is based on actual execution of the program under test, function minimization methods, and dynamic data flow analysis. Test data are developed for the program using actual values of input variables. When the program is executed, the program execution flow is monitored. If during program execution an undesirable execution flow is observed (e.g., the "actual" path does not correspond to the selected control path) then function minimization search algorithms are used to automatically locate the values of input variables for which the selected path is traversed. In addition, dynamic data flow analysis is used to determine those input variables responsible for the undesirable program behavior, leading to significant speedup of the search process. The approach to generating test data is then extended to programs with dynamic data structures, and a search method based on dynamic data flow analysis and backtracking is presented. In the approach described in this paper, values of array indexes and pointers are known at each step of program execution, and this approach exploits this information to overcome difficulties of array and pointer handling; as a result, the effectiveness of test data generation can be significantly improved.*

This is an excellent paper, which presents an innovative and complex approach to the problem of test data generation. The paper addresses some of the most complex issues involving arrays and dynamic data structures. Appropriate references to the implemented systems are cited.

A sophisticated background in data structures and data flow theory is required to read this paper.

Korel90b

Korel, Bogdan, and Janusz Laski. "Dynamic Slicing of Computer Programs." *J. Syst. and Software 13*, 3 (Nov. 1990), 187-195.

Abstract: *Program slicing is a useful tool in program debugging ... . Dynamic slicing introduced in this paper differs from the original static slicing in that it is defined on the basis of a computation. A dynamic program slice is an executable part of the original program that preserves part of the program's behavior for a specific input with respect to a subset of selected variables, rather than for all possible computations. As a result, the size of a slice can be significantly reduced. Moreover, the approach allows us to treat array elements and fields in dynamic records as individual variables. This leads to a further reduction of the slice size.*

The application of dynamic analysis to program slicing is clearly advantageous for debugging and for data flow testing.

The paper is accessible to students only after reading [Korel88a] and [Weiser84].

Laski83

Laski, Janusz W., and Bogdan Korel. "A Data Flow Oriented Program Testing Strategy." *IEEE Trans. Software Eng. SE-9*, 3 (May 1983), 347-354.

Abstract: *Some properties of a program data flow can be used to guide program testing. The presented approach aims to exercise use-definition chains that appear in a program. Two such data oriented testing strategies are proposed; the first involves checking liveness of every definition of a variable at the point(s) of its possible use; the second deals with liveness of vectors of variables treated as arguments to an instruction or program block. Reliability of these strategies is discussed with respect to a program containing an error.*

This paper provides a transition from the use of data flow to detect anomalies in programs to its use as a method for selecting and evaluating test data.

The paper should be read by the instructor and is accessible to students. The instructor should emphasize the difference between using a criterion for evaluation and using it for generation.

Miller81

Miller, Edward, and William E. Howden, eds. *Tutorial: Software Testing & Validation Techniques, 2nd Ed.* New York: IEEE Computer Society Press, 1981.

This collection of articles is dated and out-of-print, but it contains copies of many of the older articles discussed in this module. A new edition is in preparation.

Mills75

Mills, Harlan D. “The New Math of Computer Programming.” *Comm. ACM 18*, 1 (Jan. 1975), 43-48.

Abstract: *Structured programming has proved to be an important methodology for systematic program design and development. Structured programs are identified as compound function expressions in the algebra of functions. The algebraic properties of these function expressions permit the reformulation (expansion as well as reduction) of a nested subexpression independently of its environment, thus modeling what is known as stepwise program refinement as well as program execution. Finally, structured programming is characterized in terms of the selection and solution of certain elementary equations defined in the algebra of functions. These solutions can be given in general formulas, each involving a single parameter, which display the entire freedom available in creating correct structured programs.*

The functional view of programs is introduced in this classic paper. The paper is essential reading for the instructor and student alike if the functional view is to be given serious consideration.

Mills83

Mills, Harlan D. *Software Productivity*. Boston: Little, Brown, 1983.

This collection of articles on the subject of software processes was written by Harlan Mills over a period of years. Mills’s seminal article on error seeding is reprinted here.

Morell87

Morell, Larry J. “A Model for Assessing Code-Based Testing Techniques.” *Proc. Fifth Ann. Pacific Northwest Software Quality Conf.* Portland, Ore.: Lawrence & Craig, 1987, 309-326.

Abstract: *A theory of fault-based program testing is defined and explained. Testing is* fault-based *when it seeks to demonstrate that prescribed faults are not in a program. It is assumed here that a program can only be incorrect in a limited fashion specified by associating* alternate expressions *with program expressions. Classes of alternate expressions can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates.*

A particular form of fault-based testing based on symbolic execution is presented. In symbolic testing *program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a* propagation equation *whose solutions determine those alternatives which are not differentiated by this test.*

This paper contains a gentle introduction to (symbolic) fault-based testing. The coupling effect is discussed and formally characterized. It is then shown that for particular classes of faults and program constructs, the probability is small that a double fault remains undetected if each of the single faults is eliminated.

This paper is useful to those interested in theoretical analysis of the coupling effect.

Morell88

Morell, Larry J. “Theoretical Insights into Fault-Based Testing.” *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 45-62.

Abstract: *Testing is* fault-based *when its goal is to demonstrate the absence of prespecified faults. This paper presents a framework that characterizes fault-based testing schemes based on how many prespecified faults are considered and on the contextual information used to deduce the absence of those faults. Established methods of fault-based testing are placed within this framework. Most methods either are limited to finite fault classes, or focus on local effects of faults rather than global effects. A new method of fault-based testing called* symbolic testing *is presented by which infinitely many prespecified faults can be proven to be absent from a program based upon the global effect the faults would have if they were present. Circumstances are discussed as to when testing with a finite test set is sufficient to prove that infinitely many prespecified faults are not present in a program.*

Fault-based testing seeks to demonstrate that a given program is unique among a neighborhood of programs defined by classes of faults. Fault-based testing schemes may be classified according to breadth of the neighborhood (finite or infinite) and the extent of the propagation (local or global) used to distinguish the programs from other members of the neighborhood.

This paper contains proofs of the theorems cited in [Morell90] and a useful history of fault-based testing. After some background reading on symbolic execution, most of the paper should be understandable to the student. The theorems require familiarity with the halting problem of computability theory.

Morell90
Morell, Larry J. “A Theory of Fault-Based Testing.” *IEEE Trans. Software Eng. 16*, 9 (Aug. 1990), 844-857.

Abstract: *A theory of fault-based testing is defined and explained. Testing is* fault-based *when it seeks to demonstrate that prescribed faults are not in a program. Is is assumed here that a program can only be incorrect in a limited fashion specified by associating* alternate expressions *with program expressions. Classes of alternate expression can be infinite. Substitution of an alternate expression for a program expression yields an alternate program that is potentially correct. The goal of fault-based testing is to produce a test set that differentiates the program from each of its alternates.*

A particular form of fault-based testing based on symbolic execution is presented. In symbolic testing *program expressions are replaced by symbolic alternatives that represent classes of alternate expressions. The output from the system is an expression in terms of the input and the symbolic alternative. Equating this with the output from the original program yields a* propagation equation *whose solutions determine those alternatives which are not differentiated by this test. Since an alternative set can be infinite, it is possible that no finite test differentiates the program from all its alternates. Circumstances are described as to when this is decidable.*

This paper extends the work of [Morell88] by including analysis of both symbolic faults and symbolic errors. Prerequisite reading in the area of symbolic execution may be necessary.

Muchnick81
Muchnick, Steven S., and Neil D. Jones, eds. *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

This book delves deeply into the subject of data flow analysis and many areas of its application to testing, including static analysis tools and symbolic execution.

This book is for experts.

Myers79
Myers, Glenford J. *The Art of Software Testing*. New York: John Wiley, 1979.

This book is an often-cited reference on software testing. Although it is somewhat dated, students find it helpful and easy to read.

Ntafos84
Ntafos, Simeon C. “On Required Element Testing.” *IEEE Trans. Software Eng. SE-10*, 6 (Nov. 1984), 795-803.

Abstract: *In this paper we introduce two classes of program testing strategies that consist of specifying a set of required elements for the program and then covering those elements with appropriate test inputs. In general, a required element has a structural and a functional component and is covered by a test case if the test case causes the features specified in the structural component to be executed under the conditions specified in the functional component. Data flow analysis is used to specify the structural component and data flow interactions are used as a basis for developing the functional component. The strategies are illustrated with examples and some experimental evaluations of their effectiveness are presented.*

The author establishes a general framework for integrating structural testing with data flow information.

The paper could be useful to the instructor, but it is less accessible to the student. It may be helpful to first read [Rapps85], which is more comprehensive in its treatment of approaches.

Ntafos88
Ntafos, Simeon C. “A Comparison of Some Structural Testing Strategies.” *IEEE Trans. Software Eng. 14,* 6 (June 1988), 868-874.

Abstract: *In this paper we compare a number of structural testing strategies in terms of their relative coverage of the program’s structure and also in terms of the number of test cases needed to satisfy each strategy. We also discuss some of the deficiencies of such comparisons.*

This paper contains an extended overview of data flow testing methods, surveying the main papers in this area. It also corrects a mistake in an earlier version of Ntafos’s *k*-dr testing strategy. The paper extends the subsumption hierarchy introduced in [Rapps85] by including $TER_n = 1$ (see [Woodward80]), boundary-interior testing methods, and *k*-dr testing (see [Ntafos84]).

Because it provides a historical perspective on data flow testing, this paper could be used as the first reading in the area of data flow testing, followed by some of the earlier papers.

Offutt89
Offutt, A. Jefferson. “The Coupling Effect: Fact or Fiction?” *ACM Software Eng. Notes 14*, 8 (Dec. 1989), 131-140.

Abstract: *Fault-based testing strategies test software by focusing on specific, common types of errors. The* coupling effect *states that test data sets that detect simple types of faults are sensitive enough to detect more complex types of faults. This paper describes empirical investigations into the coupling effect over a specific domain of software faults. All the results from the investigation support the validity of the coupling effect. The major conclusion from this investigation is that by explicitly testing for simple faults, we are also implicitly testing for more complicated faults. This gives confidence that fault-based testing is an effective means of testing software.*

The difficulty with the notion of coupling effect is the imprecision of the terms "simple" and "complex." Offutt uses the interpretation of these terms given by Morell: simple faults are denoted by single mutants; double faults are denoted by double mutants. In no case examined in this study did any non-equivalent double-order mutant survive two sets of mutation-adequate test data. Theoretical treatment of the coupling effect may be found in [Morell87].

This paper is easy to read, but it requires a thorough background in mutation testing. Fruitful class discussion can be generated concerning the experimental design and the validity of the conclusions drawn.

Osterweil76

Osterweil, Leon J., and Lloyd D. Fosdick. "DAVE—A Validation Error Detection and Documentation System for Fortran Programs." *Software—Practice and Experience 6*, 4 (Oct.-Dec. 1976), 473-486. Reprinted in [Miller81].

Abstract: *This paper describes DAVE, a system for analyzing Fortran programs. DAVE is capable of detecting the symptoms of a wide variety of errors in programs, as well as assuring the absence of these errors. In addition, DAVE exposes and documents subtle data relations and flows within programs. The central analytic procedure used is a depth first search. DAVE itself is written in Fortran. Its implementation at the University of Colorado and some early experience is described.*

After an abrupt introduction to data flow anomalies, the paper gives two algorithms for computing the input/output classification of a variable. The relationship between these algorithms and the detection of data flow anomalies is not immediately obvious. [Fosdick76] should be read first and compared with this article. The algorithms here are expressed in an Algol-like language, making them more palatable than those in [Fosdick76].

The paper could serve as detailed reading for the instructor. The density of notation makes it difficult for the student.

Ould86

Ould, Martyn A., and Charles Unwin, eds. *Testing in Software Development*. Cambridge, England: Cambridge University Press, 1986.

This excellent monograph focuses on four views of testing: the manager's, the user's, the designer's, and the programmer's. All levels of testing (acceptance, system, integration, and unit) are discussed.

This book is a good supplement for a project-oriented software engineering course. When supplemented with readings from the literature, it provides a sufficient framework for a course in software testing.

Perlman90

Perlman, Gary. *User Interface Development.* Curriculum Module SEI-CM-17-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Jan. 1990.

Capsule Description: *This module covers the issues, information sources, and methods used in the design, implementation, and evaluation of* user interfaces, *the parts of software systems designed to interact with people. User interface* design *draws on the experiences of designers, current trends in input/output technology, cognitive psychology, human factors (ergonomics) research, guidelines and standards, and on the feedback from evaluating working systems. User interface* implementation *applies modern software development techniques to building user interfaces. User interface* evaluation *can be based on empirical evaluation of working systems or on the predictive evaluation of system design specifications.*

Podgurski90

Podgurski, Andy, and Lori A. Clarke. "A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance." *IEEE Trans. Software Eng. 16*, 9 (Sept. 1990), 965-979.

Abstract: *A formal, general model of program dependences is presented and used to evaluate several dependence-based software testing, debugging, and maintenance techniques. Two generalizations of control and data flow dependence, called weak and strong syntactic dependence, are introduced and related to a concept called semantic dependence. Semantic dependence models the ability of a program statement to affect the execution behavior of other statements. It is shown, among other things, that weak syntactic dependence is a necessary but not sufficient condition for semantic dependence and that strong syntactic dependence is a necessary but not sufficient condition for a re-*

stricted form of semantic dependence that is finitely demonstrated. These results are then used to support some proposed uses of program dependences, to controvert others, and to suggest new uses.

This paper is highly recommended for its clear and concise definitions in the area of data flow phenomena (dependences). The substantial effort necessary to understand the definitions will prove a useful investment when reading other data flow papers.

This paper could be used to lay the mathematical foundation necessary for understanding data flow in programs.

Powell82

Powell, Patricia B., ed. *Software Validation, Verification, and Testing Technique and Tool Reference Guide*, NBS Special Publication 500-93. Washington, D.C.: National Bureau of Standards, 1982.

This book covers most of the testing and analysis techniques covered in this module. The techniques are compared as to their effectiveness, applicability, ease of learning, and costs. The assessments are accurate and succinct.

This is recommended reading for the instructor; it contains many examples useful in the classroom.

Probert82

Probert, Robert L. "Optimal Insertion of Software Probes in Well-Delimited Programs." *IEEE Trans. Software Eng. SE-8*, 1 (Jan. 1982), 34-42.

Abstract: *A standard technique for monitoring software testing activities is to instrument the module under test with counters or probes before testing begins; then, during testing, data generated by these probes can be used to identify portions of as yet unexercised code. In this paper the effect of the disciplined use of language features for explicitly delimiting control flow constructs is investigated with respect to the corresponding ease of software instrumentation. In particular, assuming all control constructs are explicitly delimited, for example, by END IF or equivalent statements, an easily programmed method is given for inserting a minimum number of probes for monitoring statement and branch execution counts without disrupting source code structure or paragraphing. The use of these probes, called statement probes, is contrasted with the use of standard (branch) probes for execution monitoring. It is observed that the results apply to well-delimited modules written in a wide variety of programming languages, in particular, Ada.*

The author surveys program instrumentation techniques and describes a specific method. The paper is self-contained, and the method described is applicable to most modern languages.

The paper should be read by the instructor if instrumentation is discussed. A background in graph theory and formal grammars is necessary. The paper is explicit enough to form the basis of a class project.

Rapps85

Rapps, Sandra, and Elaine J. Weyuker. "Selecting Software Test Data Using Data Flow Information." *IEEE Trans. Software Eng. SE-11*, 4 (April 1985), 367-375.

Abstract: *This paper defines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria, which examine only the control flow of a program, are inadequate. Our procedure associates with each point in a program at which a variable is defined, those points at which the value is used. Several test data selection criteria, differing in the type and number of these associations, are defined and compared.*

This paper explores the hierarchical relationships among several data flow testing techniques. The emphasis is on specifying criteria that should be satisfied by test data, not on generating the data.

The paper should be read by the instructor if data flow is to be treated in depth. The paper is likely to overwhelm students.

Redwine83

Redwine, Samuel T., Jr. "An Engineering Approach to Software Test Data Design." *IEEE Trans. Software Eng. SE-9*, 2 (March 1983), 191-200.

Abstract: *A systematic approach to test data design is presented based on both practical translation of theory and organization of professional lore. The approach is organized around five domains and achieving coverage (exercise) of them by the test data. The domains are processing functions, input, output, interaction among functions, and the code itself. Checklists are used to generate data for processing functions. Separate checklists have been constructed for eight common business data processing functions such as editing, updating, sorting, and reporting. Checklists or specific concrete directions also exist for input, output, interaction, and code coverage. Two global heuristics concerning all test data are also used. A limited discussion on documenting test input data, expected results, and actual results is included.*

Use, applicability, and possible expansions are covered briefly. Introduction of the method has similar difficulties to those experienced when introducing any disciplined technique into an area where discipline was previously lacking. The ap-

proach is felt to be easily modifiable and usable for types of systems other than the traditional business data processing ones for which it was originally developed.

This is one of the best papers on a systematic means of testing data processing software. The value of this paper lies in its pragmatic approach to test data selection; there is little theory presented here.

As an example of applied testing in business applications, this paper is a winner. It could serve as a self-assessment test for students who must develop an integrated method.

Richardson85

Richardson, Debra J., and Lori A. Clarke. "Partition Analysis: A Method Combining Testing and Verification." *IEEE Trans. Software Eng. SE-11*, 12 (Dec. 1985), 1477-1490.

Abstract: *The partition analysis method compares a procedure's implementation to its specification, both to verify consistency between the two and to derive test data. Unlike most verification methods, partition analysis is applicable to a number of different types of specification languages, including both procedural and nonprocedural languages. It is thus applicable to high-level descriptions as well as to low-level designs. Partition analysis also improves upon existing testing criteria. These criteria usually consider only the implementation, but partition analysis selects test data that exercise both a procedure's intended behavior (as described in the specifications) and the structure of its implementation. To accomplish these goals, partition analysis divides or* partitions *a procedure's domain into subdomains in which all elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. This partition divides the procedure domain into more manageable units. Information related to each subdomain is used to guide in the selection of test data and to verify consistency between the specification and the implementation. Moreover, the testing and verification processes are designed to enhance each other. Initial experimentation has shown that through the integration of testing and verification, as well as through the use of information derived from both the implementation and the specification, the partition analysis method is effective for evaluating program reliability. This paper describes the partition analysis method and reports the results obtained from an evaluation of its effectiveness.*

This paper contains an excellent presentation of a hybrid approach, in which simultaneous coverage of both code and specification is attempted. Prerequisite reading includes domain testing [White80] and symbolic execution and formal verification [Hantler76].

This paper is essential reading for both instructor and student.

Richardson88

Richardson, Debra J., and Margaret C. Thompson. "The RELAY Model of Error Detection and Its Application." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 223-230.

This paper discusses the uses of conditions that must be satisfied in order for an error (infection) to be introduced into the state of the program and to transfer (propagate) to the output.

The paper illustrates the complexity encountered when considering how infections propagate. Propagation is broken into two stages: to the initial infection of a portion of the program's data state, and through successive execution to output. The paper proposes a system for studying infection and propagation analysis.

This paper should be read by the instructor after reading [Voas91].

Richardson89

Richardson, Debra J., Stephanie Leif Aha, and Leon J. Osterweil. "Integrating Testing Techniques Through Process Programming." *ACM Software Eng. Notes 14*, 8 (Dec. 1989), 219-228.

Abstract: *Integration of multiple testing techniques is required to demonstrate high quality of software. Technique integration has four basic goals: reduced development costs, incremental testing capabilities, extensive error detection, and cost-effective application. We are experimenting with the use of process programming as a mechanism for integrating testing techniques. Having set out to develop a process that provides adequate coverage and comprehensive fault detection, we proposed synergistic use of DATA FLOW testing and RELAY to achieve all four goals. We developed a testing process program much as we would develop a software product from requirements through design to implementation and evaluation. We found process programming to be effective for explicitly integrating the techniques and achieving the desired synergism. Used in this way, process programming also mitigates many of the other problems that plague testing in the software development process.*

The paper requires a grounding in the concept of process programming, "programming" the process of software development.

This paper is most appropriate for those interested in research into the testing process, rather than testing, *per se*.

Rowland81

Rowland, John H., and Philip J. Davis. "On the Use of Transcendentals for Program Testing." *J. ACM 28*, 1 (Jan. 1981), 181-190.

Abstract: *The element* z *is called a transcendental for the class* F *if functions in* F *can be uniquely identified by their values at* z*. Conditions for the existence of transcendentals are discussed for certain classes of polynomials, multinomials, and rational functions. Of particular interest are those transcendentals having an exact representation in computer arithmetic. Algorithms are presented for reconstruction of the coefficients of a polynomial from its value at a transcendental. The theory is illustrated by application to polynomials, quadratic forms, and quadrature formulas.*

This paper presents many techniques for demonstrating that a particular function has been implemented in a computer program. The paper requires a good background in functional analysis to grasp all the details. It is very well written, though it has limited application.

The paper can prove useful to the instructor, especially in gaining understanding of issues involved in selecting test data for particular program paths. It is not recommended for students.

Tai80

Tai, Kuo-Chung. "Program Testing Complexity and Test Criteria." *IEEE Trans. Software Eng. SE-6*, 6 (Nov. 1980), 531-538.

Abstract: *This paper explores the* testing complexity *of several classes of programs, where the testing complexity is measured in terms of the number of test data required for demonstrating program correctness by testing. It is shown that even for very restrictive classes of programs, none of the commonly used test criteria, namely, having every statement, branch, and path executed at least once, is nearly sufficient to guarantee absence of errors.*

Based on the study of testing complexity, this paper proposes two new test criteria, one for testing a path and the other for testing a program. These new criteria suggest how to select test data to obtain confidence in program correctness beyond the requirement of having each statement, branch, or path tested at least once.

This paper analyzes the complexity of achieving several structural coverage measures. The inadequacy of these measures is again shown, along with new criteria for demonstrating correctness for a limited class of programs.

The paper should be read by the instructor to gain an appreciation of when testing is equivalent to proving correctness. It is in-depth reading for a student interested in structural testing.

Tai91

Tai, Kuo-Chung, Richard H. Carver, and Evelyn E. Obaid. "Debugging Concurrent Ada Programs by Deterministic Execution." *IEEE Trans. Software Eng. 17*, 1 (Jan. 1991), 45-63.

Abstract: *An execution of a concurrent program* P *with input* X *nondeterministically exercises a sequence of synchronization events, called a* synchronization sequence *(or* SYN-sequence*). Thus, multiple executions of* P *with the same input* X *may exercise different SYN-sequences and produce different results. When debugging an erroneous execution of* P *with input* X*, it is often necessary to repeat this execution in order to collect more debugging information. However, there is no guarantee that this execution will be repeated by executing* P *with input* X*. To solve this problem requires* deterministic execution debugging*, which is to force a* deterministic execution *of a concurrent program according to the SYN-sequence of a previous execution of this program.*

In this paper, we present a language-based approach to deterministic execution debugging of concurrent Ada programs. Our approach is to define SYN-sequences of a concurrent Ada program in terms of Ada language constructs and to replay such SYN-sequences without the need of system-dependent debugging tools. We first show how to define a SYN-sequence of a concurrent Ada program in order to provide sufficient information for deterministic execution. Then we show how to transform a concurrent Ada program P *so that the SYN-sequences of previous executions of P can be replayed. This transformation adds an Ada task to* P *that controls program execution by synchronizing with the original tasks in* P*. We also briefly describe the implementation of tools supporting deterministic execution of concurrent Ada programs.*

This paper provides a technical introduction to testing concurrent Ada programs using a model that captures the sequence of concurrent interaction and enables it to be replayed. See [Carver91] for a less technical introduction and [Weiss88] for a theoretical discussion.

The paper should be read by those conducting research into testing concurrent programs.

Voas91

Voas, Jeffrey, Larry J. Morell, and Keith Miller. "Predicting Where Faults Can Hide from Testing." *IEEE Software 8*, 2 (March 1991), 41-48.

This paper introduces the concept of *sensitivity analysis*, which estimates the probability that a program location can hide a fault. The paper is built on the fault/failure model that is used to structure the implementation-based testing section of this mod-

ule. For a program to fail on a given input, three necessary and sufficient conditions must be satisfied: a fault location must be executed, the succeeding data state must be infected, and the data-state error must propagate to the output. The paper gives an overview of execution, infection, and propagation analysis and discusses how the results of these analyses can be used to identify program locations where faults can easily hide.

This paper is the best introduction to the fault/failure model discussed in this module. It should be read before some of the more theoretical presentations in [Morell88], [Richardson88], [Zeil89], and [Morell90].

Weiser84

Weiser, Mark. "Program Slicing." *IEEE Trans. Software Eng. SE-10*, 4 (July 1984), 352-357.

Abstract: *Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a "slice," is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.*

Some properties of slices are presented. In particular, finding statement-minimal *slices is in general unsolvable, but using data flow analysis is sufficient to find approximate slices. Potential applications include automatic slicing tools for debugging and parallel processing of slices.*

This article underscores the point that the same analysis technique—data flow in this case—can be used effectively in many areas of software engineering. Although slicing has not been applied to testing, the linkage between data flow testing and program slicing is inescapable. [Korel90] extends the static concept of slicing to a dynamic one.

This paper provides a detailed discussion of the theory underlying program slicing. It requires careful reading and significant background in data flow analysis. This is essential reading for the instructor.

Weiss88

Weiss, Stewart N. "A Formal Framework for the Study of Concurrent Program Testing." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 106-113.

Abstract: *Representing a concurrent program as a set of simulating, sequential programs provides a solution to the reproducible testing problem as well as a formal foundation for a theory of concurrent program testing. It is shown how this model of concurrent programs is used to extend the methods and theory of testing sequential programs to concurrent programs.*

This paper gives an overview of Weiss's Ph.D. dissertation on the testing of concurrent programs.

Only those with a significant background in concurrency and program testing should read this.

Weyuker80

Weyuker, Elaine J., and Thomas J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 236-246. Reprinted in [Miller81].

Abstract: *The theory of test data selection proposed by Goodenough and Gerhart is examined. In order to extend and refine this theory, the concepts of a revealing test criterion and a revealing subdomain are proposed. These notions are then used to provide a basis for constructing program tests.*

A subset of a program's input domain is revealing if the existence of one incorrectly processed input implies that all of the subset's elements are processed incorrectly. The intent of this notion is to partition the program's domain in such a way that all elements of an equivalence class are either processed correctly or incorrectly. A test set is then formed by choosing one element from each class. This process represents perfect program testing. For a practical testing strategy, the domain is partitioned into subdomains which are revealing for errors considered likely to occur.

Three programs which have previously appeared in the literature are discussed and tested using the notions developed in the paper.

This is the foundational paper for error-based testing. The criticism of [Goodenough75] is crisp, and the paper's theoretical approach has established it as a classic.

This is essential reading for the instructor. The student who wishes to pursue error-based testing must read it also.

Weyuker82

Weyuker, Elaine J. "On Testing Non-testable Programs." *Computer J. 25*, 4 (Nov. 1982), 465-470.

Abstract: *A frequently invoked assumption in program testing is that there is an oracle (i.e. the tester or an external mechanism can accurately decide whether or not the output produced by a program is correct). A program is non-testable if either an oracle does not exist or the tester must expend some extraordinary amount of time to deter-*

mine whether or not the output is correct. The reasonableness of the oracle assumption is examined and the conclusion is reached that in many cases this is not a realistic assumption. The consequences of assuming the availability of an oracle are examined and alternatives investigated.

Oracles may be unavailable for a number of reasons, *e.g.*, the correct output may not be known or may be extremely difficult to compute.

This paper is essential reading for the instructor, and it provides students with a useful description of pragmatic difficulties of testing theory and practice.

Weyuker84

Weyuker, Elaine J. "The Complexity of Data Flow Criteria for Test Data Selection." *Information Processing Letters 19,* 2 (Aug. 1984), 103-109.

This paper analyzes the theoretical upper bound on the number of test cases necessary to cover all the definition-use pairs of a program.

This is expert reading in data flow testing.

Weyuker86

Weyuker, Elaine J. "Axiomatizing Software Test Data Adequacy." *IEEE Trans. Software Eng. SE-12,* 12 (Dec. 1986), 1128-1138.

Abstract: *A test data adequacy criterion is a set of rules used to determine whether or not sufficient testing has been performed. A general axiomatic theory of test data adequacy is developed, and five previously proposed adequacy criteria are examined to see which of the axioms are satisfied. It is shown that the axioms are consistent, but that only two of the criteria satisfy all of the axioms.*

A set of meta-criteria (called axioms) are established for evaluating test data adequacy criteria. Criticism of this article appears, along with a reply by Weyuker, in [Zweben89].

This article is for researchers in program testing theory.

Weyuker88

Weyuker, Elaine J. "An Empirical Study of the Complexity of Data Flow Testing." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 188-195.

Abstract: *A family of test data adequacy criteria employing data flow information has been previously proposed, and theoretical complexity analysis performed. This paper describes an empirical study to help determine the actual cost of using these criteria. This should help establish the practical usefulness of these criteria in testing software, and serve as a means of predicting the amount of testing needed for a given program.*

The programs studied were taken from the book *Software Tools in Pascal* by Brian W. Kernighan and P. J. Plauger. The study was motivated by the theoretical work in [Weyuker84], which indicated that data flow testing can require a number of tests exponentially related to the number of statements in the software. The study found that for most of the software tested, only linearly many tests were needed. The impact of infeasible definition-use pairs was important, however.

The paper contains some interesting discussion of experimental design. The software tested is readily available, making this paper a good starting point for comparison experiments.

White80

White, Lee J., and Edward I. Cohen. "A Domain Strategy for Computer Program Testing." *IEEE Trans. Software Eng. SE-6,* 3 (May 1980), 247-257. Reprinted in [Miller81].

Abstract: *This paper presents a testing strategy designed to detect errors in the control flow of a computer program, and the conditions under which this strategy is reliable are given and characterized. The control flow statements in a computer program partition the input space into a set of mutually exclusive* domains, *each of which corresponds to a particular program path and consists of input data points which cause that path to be executed. The testing strategy generates test points to examine the boundaries of a domain to detect whether a domain error has occurred, as either one or more of these boundaries will have shifted or else the corresponding predicate relational operator has changed. If test points can be chosen within* ε *of each boundary, under the appropriate assumptions, the strategy is shown to be reliable in detecting domain errors of magnitude greater than* ε*. Moreover, the number of test points required to test each domain grows only linearly with both the dimensionality of the input space and the number of predicates along the path being tested.*

This is the fundamental paper on domain testing, an error-based testing strategy. The paper focuses on testing errors in the control flow of programs whose predicates have linear interpretation in the input variables. Note that the restrictions specified in the paper, especially linearity and the absence of arrays, limit the applicability of this strategy mostly to data processing programs. The strategy is examined closely in [Clarke82] and complemented by the approach in [Zeil83].

This paper is very well written and requires little background, though [Howden76] should probably be read first. It is essential reading for the instructor, and students will find it very readable.

Woodward80

Woodward, Martin R., David Hedley, and Michael A. Hennell. "Experience with Path Analysis and Testing of Programs." *IEEE Trans. Software Eng. SE-6*, 3 (May 1980), 278-286. Reprinted in [Miller81].

Abstract: *There are a number of practical difficulties in performing a path testing strategy for computer programs. One problem is in deciding which paths, out of a possible infinity, to use as test cases. A hierarchy of structural test metrics is suggested to direct the choice and to monitor the coverage of test paths. Another problem is that many of the chosen paths may be infeasible in the sense that no test data can ever execute them. Experience with the use of "allegations" to circumvent this problem and prevent the static generation of many infeasible paths is reported.*

This paper introduces the concept of LCSAJ, a linear code sequence and jump, which has since been used as a structural measure in several diverse experiments.

The paper should be read by the instructor interested in practical methods of structural testing. Students will find the paper difficult but rewarding.

Woodward88

Woodward, M. R., and K. Halewood. "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 152-158.

Abstract: *Despite the intrinsic appeal of the mutation approach to testing, its disadvantage in being computationally expensive has hampered its widespread acceptance. When weak mutation was introduced as a less expensive and less stringent form of mutation testing, the original technique was renamed strong mutation. This paper argues that strong mutation testing and weak mutation testing are in fact extreme ends of a spectrum of mutation approaches. The term* ***firm mutation*** *is introduced to represent the middle ground in this spectrum. The paper also argues, by means of a number of small examples, that there is a potential problem concerning the criterion for deciding whether a mutant is 'dead' or 'live.' A variety of solutions are suggested. Finally, practical considerations for a firm mutation testing system, with greater user control over the nature of result comparison, are discussed. Such a system is currently under development as part of an interpretive development environment.*

Firm mutation represents the practical implementation of the extent property of fault-based techniques discussed in [Morell88]. This paper is indicative of a growing understanding of the importance of analysis of propagation in program testing.

The paper assumes a good grounding in mutation testing and knowledge of practical problems associated with it.

Young88

Young, Michal, and Richard N. Taylor. "Combining Static Concurrency Analysis with Symbolic Execution." *IEEE Trans. Software Eng. 14*, 10 (Oct. 1988), 1499-1511.

Abstract: *Static concurrency analysis detects anomalous synchronization patterns in concurrent programs, but may also report spurious errors involving infeasible execution paths. Integrated application of static concurrency analysis and symbolic execution sharpens the results of the former without incurring the full costs of the latter applied in isolation. Concurrency analysis acts as a path selection mechanism for symbolic execution, while symbolic execution acts as a pruning mechanism for concurrency analysis. Methods for combining the techniques follow naturally from explicit characterization and comparison of the state spaces explored by each, suggesting a general approach for integrating state-based program analysis techniques in a software development environment.*

Many have proposed augmenting flow analysis with symbolic execution to minimize the impact of infeasible paths. This paper clearly presents the advantages and the difficulties of such an integration.

Since this paper treats the intersection of three subjects—concurrency, flow analysis, and symbolic execution)—significant background is necessary before reading.

Youngblut89

Youngblut, Christine, *et al. SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks.* IDA Paper P-2132, Institute for Defense Analyses, Alexandria, Va., Feb. 1989.

This report discusses almost all areas of program analysis and testing at all levels (unit, integration, system, and acceptance) and evaluates them in the context of SDI applications. (The report was prepared for the Strategic Defense Initiative Organization.) An extensive glossary is included. This is a companion to [Brykczynski89].

Zeil83

Zeil, Steven J. "Testing for Perturbations of Program Statements." *IEEE Trans. Software Eng. SE-9,* 3 (May 1983), 335-346.

Abstract: *Many testing methods require the selection of a set of paths on which tests are to be conducted. Errors in arithmetic expressions within program statements can be represented as perturbing functions added to the correct expression. It is then possible to derive the set of errors in a chosen functional class which cannot possibly be detected using a given test path. For example, test paths which pass through an assignment statement* "X := f (Y)" *are incapable of revealing if the expression* "X - f (Y)" *has been added to later statements. In general, there are an infinite number of such undetectable error perturbations for any test path. However, when the chosen functional class of error expressions is a vector space, a finite characterization of all undetectable expressions can be found for one test path, or for combined testing along several paths. An analysis of the undetectable perturbations for sequential programs operating on integers and real numbers is presented which permits the detection of multinomial error terms. The reduction of the space of (potential) undetected errors is proposed as a criterion for test path selection.*

This paper describes a method for deducing sufficient path coverage to ensure the absence of prescribed errors in a program. It models the program computation and potential errors as a vector space. This enables the conditions for non-detection of an error to be calculated. The strategy assumes the existence of a reliable testing strategy for paths, which, of course, does not exist.

Exposure to [White80] should provide sufficient background for appreciating the context in which the techniques are to be used. To understand the mathematics requires some background in linear algebra, especially if some of the references are to be read. The paper explores an interesting area and deserves to be read by the instructor. This is advanced reading for students, however.

Zeil88

Zeil, Steven J. "Selectivity of Data-Flow and Control-Flow Path Criteria." *Proc. Second Workshop on Software Testing, Verification, and Analysis.* Washington, D.C.: IEEE Computer Society Press, 1988, 216-222.

Abstract: *A given path selection criterion is more selective than another such criterion with respect to some testing goal if it never requires more, and sometimes requires fewer, test paths to achieve that goal. This paper presents canonical forms of control-flow and data-flow path selection criteria and demonstrates that, for some simple testing goals, the data-flow criteria as a general class are more selective than the control-flow criteria. It is shown, however, that this result does not hold for general testing goals, a limitation that appears to stem directly from the practice of defining data-flow criteria on the computation history contributing to a single result.*

This paper challenges the reader to consider methods of comparing path-selection criteria other than subsumption.

The paper assumes the reader is familiar with data flow testing at the level of [Clarke89].

Zeil89

Zeil, Steven J. "Perturbation Techniques for Detecting Domain Errors." *IEEE Trans. Software Eng. 15,* 6 (June 1989), 737-746.

Abstract: *Perturbation testing is an approach to software testing which focuses on faults within arithmetic expressions appearing throughout a program. In this paper perturbation testing is expanded to permit analysis of individual test points rather than entire paths, and to concentrate on domain errors. Faults are modeled as perturbing functions drawn from a vector space of potential faults and added to the correct form of an arithmetic expression. Sensitivity measures are derived which limit the possible size of those faults that would go undetected after the execution of a given test set. These measures open up an interesting new view of testing, in which attempts are made to reduce the volume of possible faults which, were they present in the program being tested, would have escaped detection on all tests performed so far. The combination of these measures with standard optimization techniques yields a new test data generation method, called* arithmetic fault detection.

This paper extends Zeil's earlier paper [Zeil83] by treating aspects of propagation not considered before. Program computations are modeled as vector spaces, and program faults as perturbations of those vector spaces. The perturbation model is somewhat analogous to the symbolic execution model discussed in [Morell90].

The paper requires significant mathematical sophistication to understand the model proposed. It is expert reading in error-based testing.

Zweben89

Zweben, Stuart H., and John S. Gourlay. "On the Adequacy of Weyuker's Test Data Adequacy Axioms." *IEEE Trans. Software Eng. 15,* 4 (April 1989), 496-501.

Abstract: *Weyuker has recently proposed a set of*

properties which should be satisfied by any reasonable criterion used to claim that a computer program has been adequately tested. She called these properties "axioms." She also evaluated several well-known testing strategies with respect to these properties, and concluded that some of the commonly used strategies failed to satisfy several of the properties.

We question both the fundamental nature of the properties and the precision with which they are presented, and illustrate how a number of ideas in Weyuker's paper can be simplified and clarified through greater precision and a more consistent set of definitions. We also reanalyze the testing strategies after accounting for these inconsistencies. The strategies tend to fare much better as a result of this reanalysis.

The authors raise the issue of what makes an axiomatic system, as well as what constitutes a proper axiom. This criticism must be read with along with [Weyuker86]. Weyuker responds to the criticism at the end of the article.

If students have never seen such a professional interchange, this is worth reading for that aspect alone.