NISTIR 6129

# Software Testing by Statistical Methods

## Preliminary Success Estimates for Approaches Based on Binomial Models, Coverage Designs, Mutation Testing, and Usage Models

**David Banks**, Div. 898
**William Dashiell**, Div. 897
**Leonard Gallagher**, Div. 897, Editor
**Charles Hagwood**, Div. 898
**Raghu Kacker**, Div. 898
**Lynne Rosenthal**, Div. 897, PI

U.S. DEPARTMENT OF COMMERCE
Technology Administration
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

NIST

NISTIR 6129

# Software Testing by Statistical Methods

## Preliminary Success Estimates for Approaches based on Binomial Models, Coverage Designs, Mutation Testing, and Usage Models

by

David Banks, Div 898
William Dashiell, Div 897
Leonard Gallagher, Div 897, Editor
Charles Hagwood, Div 898
Raghu Kacker, Div 898
Lynne Rosenthal, Div 897, PI

A Task Deliverable under a Collaboration between ITL's Statistical Engineering (898) and Software Diagnostics and Conformance Testing (897) Divisions to pursue Statistical Methods that may be applicable to Software Testing

National Institute of Standards and Technology
Information Technology Laboratory
Gaithersburg, MD 20899, USA

March 12, 1998

U.S. DEPARTMENT OF COMMERCE
William M. Daley, Secretary

TECHNOLOGY ADMINISTRATION
Gary R. Bachula, Acting Under Secretary for Technology

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Director

## - Abstract -

Software conformance testing is the process of determining the correctness of an implementation built to the requirements of a functional specification. Exhaustive conformance testing of software is not practical because variable input values and variable sequencing of inputs result in too many possible combinations to test. This paper addresses alternatives for exhaustive testing based on statistical methods, including experimental designs of tests, statistical selection of representative test cases, quantification of test results, and provision of statistical levels of confidence or probability that a program implements its functional specification correctly. The goal of this work is to ensure software quality and to develop methods for software conformance testing based on known statistical techniques, including multivariable analysis, design of experiments, coverage designs, usage models, and optimization techniques, as well as to provide quantitative measures of quality, reliability, and conformance to specifications, including statistical measures and confidence levels.

We place initial priority on statistical methods potentially applicable to software defined using conventional techniques, i.e. a functional specification with semantics defined in English, because a large majority of National, International, and Internet standards are specified via non-formal methods, and a major mission of NIST/ITL is to provide conformance testing techniques for such software standards. We focus on conformance testing of software products for which source code is not available; the only access to the product is through a published interface defined by the functional specification, i.e. black-box access. We pursue the potential applicability of several statistical methods to these goals, including binomial models, coverage designs, mutation testing, usage models, and cost-benefit optimization techniques.

**Keywords:** (binomial model; black-box; confidence level, conformance testing; coverage design; functional specification; software reliability; software quality; statistical method; usage model; validation)

## Table of Contents

## 1 Introduction

Exhaustive testing of software is not practical because variable input values and variable sequencing of inputs result in too many possible combinations to test. NIST developed techniques for applying statistical methods to derive sample test cases would address how to select the best sample of test cases and would provide a statistical level of confidence or probability that a program implements its functional specification correctly. The goal of this work is to develop methods for software testing based on statistical methods, such as Multivariable Testing, Design of Experiments, and Markov Chain usage models, and to develop methods for software testing based on statistical measures and confidence levels.

During early 1997, ITL staff from the Statistical Engineering Division (898) and the Software Diagnostics and Conformance Testing Division (897) proposed a collaborative effort to research the potential and practicality for applying statistical methods to software testing. This effort evolved into a new ITL Competency proposal, entitled Software Testing by Statistical Methods, that was presented to the NIST Director on July 7, 1997. The Competency proposal is quite broad in that it includes both white-box (source code available) and black-box (only a defined interface can be used) testing to determine if a program conforms to its functional specification, and it includes functional specifications defined using conventional techniques (see Section 2.1) or Formal Definition Techniques (formal syntax and formal mathematically rigorous semantics). The goal is to integrate statistical techniques into software testing both to ensure software quality and to provide quantitative measures of quality, reliability, and conformance to specifications.

An expanded team of ITL staff began meeting in June, 1997, to consider the best way to approach the Competency proposal and to develop a specific project plan for Fiscal Year 1998. That group, including the authors of this paper, decided to place initial priority on statistical methods potentially applicable to software defined using conventional techniques, i.e. a functional specification with semantics defined in English. The reason for focusing on conventional specifications is because a large majority of National, International, and Internet standards are specified via non-formal methods, and a major mission of ITL is to provide conformance testing techniques for such software standards. In addition, any statistical methods applicable to conventional software specifications would also be applicable to specifications written using a formal definition technique.

The first year project plan, dated July 22, 1997, focuses on statistical methods potentially applicable to the effective design, construction, and implementation of conformance testing techniques for non-formal functional specifications. This involves techniques for conformance testing of software products for which source code is not available; the only access to the product is through a published interface defined by the functional specification (i.e. black-box access). The project plan lists four statistical methods that might be applicable to conformance testing of products built to such functional specifications:

**Binomial Models** - Applicable to specifications that consist of a finite number, or a large indefinite number, of independent conformance requirements. This approach should allow quantification of uncertainty and a potential reduction in testing costs for existing falsification testing methods.

**Coverage Designs** - Coverage designs extend classical experimental methods in statistics to address the kinds of interactive failure modes that arise in software reliability and testing. New methodologies recently developed at Bellcore and AT&T Research indicate that coverage designs could lead to the construction of software testing procedures that are more efficient than designs built from more conventional statistical considerations.

**Mutation Testing** - A well-known method for white-box testing of software by seeding the program with errors and measuring the effectiveness of a test suite in identifying such errors. Potentially applicable to quantifying confidence levels for the effectiveness of any test suite.

**Usage Models** - Given a known usage pattern for a certain class of input functions, it may be possible to determine confidence levels for the reliability of software restricted to that usage pattern. Or, given experimental quantification of a known usage pattern, it may be possible to measure the reliability and to determine confidence levels for usage patterns that are measurably close to the given usage pattern.

**When to Stop Tesing** - We are interested in the efficiency of the testing process and when it makes sense to stop testing. We pursue optimal strategies, based on costs and benefits, for when to stop testing. The approaches we describe use accepted statistical methods for binomial testing with a sequential probability determination to derive stopping and decision rules in the general context of software testing.

The project plan calls for a preliminary analysis of the potential application of each of the above methodologies to black-box conformance testing, leading to a report that identifies one or more of the these approaches as a foundation for more focused efforts. A subsequent report would pursue the selected approaches in depth and choose one approach to be the basis of an experimental prototype. The first year project plan also provides for the design and preliminary development of an experimental prototype, and calls for preliminary analysis of some non-testing methods, e.g. clinical trials, that might also lead to quantitative measurements of software quality.

This report is the preliminary analysis of each of the above approaches. It begins in Section 2 with descriptions of different types of functional specifications, a definition of conformance for each type of specification, and a listing of traditional approaches to black-box testing. Each of Sections 3 through 6 takes a closer look at a known statistical method by defining it, giving some example of how it is used, and concluding with an estimate of its potential impact on black-box conformance testing. Section 7 addresses the interesting problem of "When to Stop Testing" and Section 8 reaches some conclusions about which statistical methods are most likely to be successfully applied to black-box testing. A more in-depth development of the most promising approaches will follow in a separate report.

## 2 Black-Box Conformance Testing

With black-box conformance testing, an implementation is tested only with respect to its functional specification. The internal structure of the implementation is not accessible. The tester can only infer the underlying structure by communicating with the implementation using public interfaces defined by the functional specification. Usually, an implementation is evaluated based on the state of an underlying machine or system state. That is, a conformance test would alter the state of the underlying machine and one or more descriptions of expected states of the machine. The actual states of the machine during the execution of the test would be compared with the expected states as described in the standard to provide an evaluation of the conformance of the implementation.

Black-box conformance testing consists of developing a suite of tests to determine the correctness of an implementation that is built to the requirements of a functional specification. Traditionally, the tests are designed by carefully choosing different input values, trying to design execution scenarios that will invoke every conformance rule in the functional specification. Designing such tests is labor intensive because it is not always easy to find input variables and state variables that will invoke a given conformance rule, while at the same time produce output that is known to be correct. Often it is necessary to build a reference implementation that will produce the correct output for the given input by some other means so that it can be compared with the output produced by the product being tested.

Since it is often impossible to cover every logical possibility of input variables over all possible system states, testers are usually satisfied if every rule is invoked over some system state, or over a small collection of system states. This approach, often called *falsification testing*, can only prove that a product is non-conforming, it cannot be certain that a product is conforming even if it passes all of the tests. The purpose of this research is to determine if statistical methods can be applied in test design to minimize both the number of tests needed and the number of executions needed of each test under different input variables and system states to reach a given level of confidence that a product conforms to a functional specification.

The following sections give a brief overview of the different types of functional specifications for which NIST has developed test suites in the past, defines conformance for each type of functional specification, and lists the various approaches to black-box testing that assist in determining when a test suite has "covered" all of the functional requirements of a given functional specification. Since complete coverage is often impossible, each approach has a definition of "coverage" that is claimed to be adequate for the purposes of that approach. Our statistical research should lead to quantification of "coverage" and a "level of confidence" in the correctness of a product that satisfies a certain coverage requirement.

## 2.1 Conventional Types of Functional Specifications

There is no agreed precise definition of the term functional specification, but there is general agreement that a functional specification consists of requirements for both syntax and semantics. Usually, the syntax is formally specified with the semantics for a given situation specified by conformance rules written in English. In a high-quality specification, the conformance rules will address every possible legal syntactic combination. Traditionally, for NIST, a functional specification is a National or International standard (e.g., ISO, ANSI, IEEE). The role of NIST has been to develop tests or test methods that can be used to determine whether or not a commercial implementation of a standard conforms to the requirements of that standard. NIST has traditionally been involved with conformance testing of the following types of standards:

**Language Standard:** A language standard specifies the syntax and semantics of a programming language (e.g. Cobol, Ada, C++) or a systems language (e.g. SQL). A language standard is made up of a set of Syntax Rules, usually defined via Backus Normal Form (BNF) or some close derivative, and a set of General Rules. The General Rules define the action required when the defined syntax is invoked. In most cases the General Rules are written in natural language rather than by a formal method.

**Class Library Standard:** A class library standard specifies a library of functions or procedures, the function names, number and type of input parameters, rules for invoking the procedures, and General Rules for defining the semantics of what the functions should do. Like the Syntax Rules for programming language standards, the library's syntactic requirements can be formally specified and automatically checked or generated. However, in most cases the General Rules for semantics are written in natural language, or at best some quasi-formal language.

**Protocol Standard:** A protocol standard specifies a set of rules or conventions, known as a protocol, that allows peer layers of two or more open systems to communicate. Each individual communication for each layer is specified as a functional unit, each with protocol requirements. The key elements of a protocol are syntax, semantics, and timing. Syntax is usually defined by a formal method that can be automatically checked; semantics and timing are defined by a finite state model together with General Rules to specify legal state transitions and timing requirements. In most cases, the General Rules are written in natural language.

**Interchange Standard:** An interchange standard specifies the syntax and semantics for the representation and exchange of a structured object (e.g., graphics picture). The standard defines specific objects, Syntax Rules for a file structure to represent the objects, and a set of General Rules to describe the semantics of the objects. As above, the Syntax Rules can be formally specified and automatically checked. In most cases, the General Rules for semantics are written in natural language.

## 2.2 Definitions of Conformance

Software conformance testing is the process of determining whether or not a software implementation satisfies its design specification. Conformance testing captures the technical description of a specification and measures whether a product faithfully implements the specification. The outcome of a conformance test is generally a pass or fail result, possibly including reports of problems encountered during the execution. The results of a conformance test are intended to provide a measure of confidence in the degree of conformance of the implementation tested. A common problem with conformance testing is the difficulty of associating a degree of confidence with a given test or test suite. One of the goals of this project is to address the confidence quantification problem in a statistically significant way.

Some progress has been made on the above points [Ling93, Sank94, Amma95, Heit96, Blac97] when the semantics of a standard are specified according to some rigorous, mathematically based Formal Method. However, most popular standards (e.g. Internet standards) do not have the benefit of a formal specification. We hope to make some progress on the above points even when the General Rules of the standard's specification are written (clearly we assume) in natural language.

The exact definition of conformance varies with the type of functional specification. We use the following definitions for conformance related to the above types of functional specifications:

**Language Standard:** A program written in the programming language is said to be a *conforming program* if it satisfies the Syntax Rules of the standard specification. An implementation of the standard is said to be a *conforming implementation* if it accepts any conforming program and correctly processes that program according to the General Rules of the standard specification. Typically, a test developer will produce a number of test programs that it knows abide by the syntax of the standard specification and applies those programs to the product to be tested to see if it produces the correct output. If the standard specifies error conditions on data, then test programs will attempt to simulate those situations to see if the implementation properly diagnoses the error.

**Class Library Standard:** Conformance to a class library standard is defined analogously to conformance for programming language standards. Instead of Syntax Rules, a *conforming function* or *conforming procedure* must satisfy the syntactic restrictions on the number and position of its parameters, as well as data type and range constraints on all input parameters. A *conforming implementation* of a class library standard must accept any conforming function or procedure as input and produce correct output, and correct changes to the underlying machine state, according to the General Rules of the functional specification.

**Protocol Standard:** Conformance to a protocol standard is specified in terms of the *client role* and the *server role.* An implementation may claim conformance to one or both of these roles. An implementation conforming to the client role must abide by the protocols for initiating a connection to another open system; in addition, it must correctly create protocol service requests for the other functional units it claims to support. An implementation conforming to a server role must abide by the protocols for opening a connection requested by another open system; in addition, it must correctly respond to the protocol service requests for the functional units it claims to support.

**Interchange Standard:** Conformance to an interchange standard is usually defined in terms of interchange Files, Readers, and Writers. An interchange File is a *conforming file* if it satisfies the Syntax Rules of the standard specification. A Writer is a *conforming writer* if it accepts an object of a specific type (with a proprietary definition) and produces an output stream that abides by the standard File specification and correctly represents the structures of the input object. A Reader is a *conforming reader* if it can accept and parse a conforming file and produce correct results. Typically a test developer has a trusted Writer that is used to test Readers and a trusted Reader that is used to test Writers. The test developer will produce a collection of objects that invoke all of the rules of the specification in order to produce a correct interface File, and vice versa.

In general, a test developer will produce as many tests as are needed to cover syntactic alternatives to a satisfactory degree, and whose execution requires the implementation to exercise all of the general rules correctly at least once. It is not uncommon for a test developer to try to produce at least one simple program to test every General Rule in the standard's specification. If an implementation passes all of the tests, then it is said to be a conforming implementation (even though all possible syntactic alternatives, and combinations of alternatives, have not been tested). If an implementation fails even a single test, it is said to be deficient. We can say with certainty, that an implementation does not conform; however, we cannot say with certainty that an implementation does conform. One goal of this research is to use valid statistical methods to quantify the level of confidence one has that an implementation conforms to a given functional specification.

## 2.3 Traditional Categories of Black-Box Testing

Traditionally, black-box conformance testing has been as much an art as it is a science. Often tests are based on hunches of what an implementation is most likely to do wrong; the tester thinks of mistakes that are likely to be made, and then designs tests to ensure that an implementation did not make those particular mistakes. Experienced testers stress that all testing should be based on a strategy, with a set of rules by which one is able to determine if a particular test does or does not satisfy the strategy. A good testing strategy will focus on a particular aspect of the testing process, e.g. finding bugs derived from programming carelessness, finding bugs derived from common programming mistakes, finding logical errors introduced by common misunderstandings of the stated requirements, finding errors based on branching mistakes, finding errors based on looping mistakes, finding errors resulting from failure to cover all possible scenarios, etc. A good testing strategy applied to a formal representation of the functional requirements will result in a collection of tests that is complete as far as that testing strategy is concerned. Boris Beizer [Beiz95] recommends representing functional specifications as directed graphs, which can then be structured to implement a given strategy in a way that completeness can be measured as a structural property of the graph itself. His 1995 book lists the following categories of black-box test development:

**Control-Flow Testing**

Black-box control-flow testing is analogous to white-box testing of a program's flow of control structures, i.e. order of execution, branching, and looping. Since the source program is not available, flow of control must be deduced from rules in the functional specification. A directed graph, with processing nodes and directed flow-of-control arcs, can represent the control-flow as follows:

Processing Node - Any enclosed sequence of processing steps from the functional specification, provided that if any part of the sequence is executed then every part of the sequence will be executed, can be represented as a single node in the directed graph.

Input Node - A processing node in which an input value is accepted through an interface defined by the functional specification. An input node has no incoming arcs.

Output Node - A processing node in which an output value is made available through an interface defined by the functional specification. An output node has no outgoing arcs.

FlowOfControl Arc - An arc from one node to another in which the processing of the first node precedes the processing of the second node.

Predicate Node - A processing node with two or more outgoing arcs each of which is determined by a predicate result in the processing node. If there are exactly two outgoing arcs, then the predicate node is called a *logical predicate node*, otherwise it is called a *selector predicate node*. A compound predicate with N conditions connected by AND's or OR's, can be modeled by a tree of $2^N$ logical predicate nodes and $2^N$+N-1 complementary arcs labeled as True or False.

Junction Node - A processing node with two or more incoming arcs.

Path - A sequence of nodes with exactly one identified arc from the i-th to the i+1-st node. An *I/O-Path* is a path from an input node to an output node. An *achievable path* is an I/O-path for which it is possible to find input values that will cause the program to follow that path. An *unachievable* path is an I/O-path for which there is no set of inputs that will cause that path to be traversed.

The objective of control-flow testing is to model the functional specification as a directed graph with all possible processing nodes and all possible achievable paths identified, and then to find input values for a minimal number of test cases to ensure that every achievable path is traversed by at least one of the test cases. The first step is to construct a directed graph with all possible I/O-paths identified; the second step is to select the achievable paths by eliminating all possible unachievable paths; the third step is to *sensitize* the achievable paths by choosing input values that will cause the program to traverse that path; each such choice of input values will represent a test case. Selecting the achievable paths and sensitizing them can each be labor intensive steps. One objective of this research is to determine if statistical methods can be used to reduce these labor intensive steps or to automate some or all of this process.

**Data-Flow Testing**

Data-flow testing is a generalization of control-flow testing that depends more on data values than on flow-of-control. It can be modeled with a data-flow graph, which is a generalization of a control-flow graph that includes more data information, including nodes to represent where every data value comes from. In a data-flow graph each processing node is further subdivided into nodes that represent every possible definition of a data object. A separate node is defined whenever a data object is assigned a new value. Thus a data object, e.g. a variable, may be represented by a processing node when it is created, and then represented by different processing nodes whenever the object is initialized, instantiated, or assigned a value as the result of some calculation. Under this generalization, a data-flow node can be hought of as a calculated function that is dependent upon the data values of the incoming nodes it is connected to. The incoming arcs are a generalization of control flow arcs in that they represent the relationship that the data value of the node at the tail of the directed arc is directly used to calculate the value of the node at the tip of the arc. The only difference between a control-flow graph and a data-flow graph is that unimportant processing sequences are not represented in a data-flow graph. The order of processing is represented by an arc only when the calculated value is dependent upon the order of processing.

Data-flow can be modeled at several different levels. At one level, an entire database can be considered as a data object, with operations to Select, Insert, Update, or Delete collections of records. At another level, data operations can be modeled for each record, or for each field in a record. Since data-flow and control-flow models don't mix well, one can treat a control flow model as a separate level of a data-flow model. In this way a complicated system can be represented by several simpler data-flow models and each model will result in a set of test cases. Since the different levels represent different data requirements, the sets of test cases should have very little overlap. However, there is no guarantee that the union of test cases from simpler data-flow graphs are as powerful as the set of test cases derived from a single, more comprehensive, data-flow graph.

The construction of test cases from a data-flow graph is somewhat more complex than that of a pure control-flow graph. Instead of just working just with paths, we need to consider data-flow slices:

> <u>DataFlow Slice</u> - A data flow slice is a subgraph of a data-flow graph that depends upon a single given node (usually an output node) for its definition. In general it consists of all data flows (i.e. nodes and arcs) that can reach the given node and all data flows that can be reached from that node. If the given node is an output node, then the data-flow slice includes all of the paths from input nodes that can influence the value of that output node.

There are several different strategies that might be used in data-flow testing. Each strategy in the following list is more powerful than the previous ones, but each requires more effort in designing test cases.

> <u>Input/Output Cover</u> - A very weak testing strategy that considers each output node of the data-flow graph separately and then chooses values for a set of input nodes that leads to some output value for the given output node. This strategy covers all output nodes and in most cases covers all input nodes. This Input/Output covering strategy is too weak to be useful; it only assures us that the software works for a finite collection of input values. It does not even guarantee that a given percentage of the data-flow slices have been covered.

> <u>All Predicates</u> - A slightly stronger strategy that, in addition to input/output cover, chooses additional input values to ensure that all predicates (including control-flow predicates for branching and looping) get covered and that each outgoing arc from a predicate node gets covered. In particular, this means choosing at least two sets of input values for every logical predicate node to ensure that both output arcs from that node get traversed. This strategy is about as powerful as control-flow branch testing, but it is still too weak to draw strong conclusions about conformance of the implementation to the given functional specification.

> <u>All Definitions</u> - The all definitions strategy is to ensure that every processing node that is not a predicate or control-flow node is covered. Since every such processing node is a computational node that results in the definition of a calculated value, it is called the all definitions strategy. However, there is no guarantee that any arcs dealing with predicates and selectors have been traversed. Sometimes the all definitions strategy is used in combination with control-flow testing or with all predicates testing as defined above. If control-flow can be separated from data-flow as much as possible, then each strategy becomes relatively simple, and all definitions becomes quite powerful. One advantage of the all definitions strategy is that it is based on a formal, verified theory.

> <u>All Nodes</u> - The all nodes strategy ensures that all nodes of the data-flow graph are covered, including all data selector nodes and all control-flow nodes, as well as all definition nodes. It subsumes the all definitions strategy, so is clearly more powerful, but it doesn't guarantee that every outgoing arc from a selector node gets traversed. It can be used in combination with all predicates to ensure two separate sets of test cases that cover

most logical possibilities, but it isn't strong enough to guarantee that all interactions between predicate nodes and definition nodes have been covered.

All Uses - The all uses strategy ensures that every arc in the data-flow graph is traversed at least once. If the data-flow graph is connected (as it usually is), then this strategy subsumes the all nodes strategy and is clearly more powerful. However, choosing input values for the input nodes to guarantee that every arc gets covered by some test case can be a very labor intensive process, and it still doesn't guarantee that every potential slice to an output node gets traversed. It only guarantees that some slice to every output node gets traversed.

All Define Use Paths - The all define use paths strategy extends the all uses strategy to ensure that every potential slice leading to an output node gets traversed. The strategy is to choose, for every selector node, a separate slice for each incoming arc to the selector node, rather than just picking enough slices to cover the arcs elsewhere in the graph. This strategy becomes combinatorially more complex than the all uses strategy in that the number of slices leading to test cases increases geometrically with the number of selector nodes in a path from an input node to an output node. Luckily, there is some empirical evidence [Weyu90] to suggest that the all uses strategy comprises most of the benefits of the all define use paths strategy.

All Uses plus Loops - Often a data-flow graph can be designed without having to consider loops, but if loops are present, then the all uses plus loops strategy is to unfold each loop by replacing its node and return arc with separate logical selector nodes and return arcs to ensure that a separate slice is defined for not looping at all, looping once, looping twice, etc. This can dramatically increase the number of defined slices leading to separate test cases, especially if the all define use paths strategy is being used.

Each of the above strategies leads to an algorithm for creating slices that traverse paths from a set of input nodes to an output node in a data-flow graph. More formal definitions are given in [Fran88]. As with control-flow testing, the objective of data-flow testing is to model the functional specification as one or more data-flow graphs with all relevant processing nodes and all achievable paths identified, then to define a minimal set of slices according to the strategy being employed, and then to find input values for each slice to ensure that every achievable path of the slice is traversed by the test case resulting from those input values. As with control-flow testing, eliminating all unachievable paths (and thus unachievable slices) and sensitizing the achievable slices by choosing input values that will cause the program to traverse the paths of that slice, can each be labor intensive steps. One objective of this research is to determine if statistical methods can be used to reduce these labor intensive steps or to automate some or all of this process.

**Domain Testing**

Domain testing is used to test software, or smaller units of software, that contain substantial numerical processing of real numbers, where values for testing are taken from various domains of the input space. It is possible that path sensitization from the control-flow and data-flow testing methods described above will lead to domains defined over the input space. In that situation there is a choice as to what are the best values to be selected from the domain for the test cases. Domain testing may also arise naturally from the category-partition method described below. One major advantage of domain testing is that it is a formal, mathematical technique that can be automated both for test design and test execution.

For simplification we assume that there are N numeric input variables for each test case and that the N values can be considered as the components of a vector in N-dimensional Euclidean space. A domain is a subset of the input space defined by a smooth boundary. We know from mathematical theory that a smooth boundary can be approximated arbitrarily closely by a set of linear inequalities, so without loss of generality we assume that all domains are defined by linear inequalities. With suitable transformations non-linear boundaries resulting from polynomials, conic sections, exponential functions, power functions, and logarithmic functions can be represented by linear inequalities. This simplification is further justified by [Jeng94], which shows that if you are only concerned with the existence of a bug in the software, and not necessarily in the kind of domain error the bug represents, then except for a small class of bugs, the issues of non-linear domain boundaries do not matter. The domains of a well-designed domain test will form a partition of the input space; some domains will lead to valid computations and others will lead to errors. We will want to test values from every domain in the partition to ensure that the software under test properly handles all cases, whether it leads to a valid result or to an error condition.

There are many domain testing strategies, e.g. [Afif92, Whit87, Zeil89, Jeng94], but [Jeng94] gives a theoretical argument that by choosing test points in a suitable way, one can be almost certain to catch all but the most unusual boundary implementation errors. Jeng presents two basic *weak* strategies, a strategy called 1x1 and a family of strategies called Nx1. The 1x1 strategy is sufficient for detecting boundary errors and the Nx1 strategy is effective for determining exactly what type of boundary error is present, where N is the dimension of the input space. The 1x1 strategy tests two points for each boundary inequality, one on the boundary and one off the boundary. The two points are chosen as close as possible to one another to ensure that domain shift errors are properly detected. If two domains share a boundary, then the off point is always in the domain that is open with respect to that boundary. The Nx1 strategy tests N+1 points for each boundary inequality, N points on the boundary and 1 point off the boundary. The off point is chosen at or near the centroid of the on points. One must always be careful to choose the off point so that it is in a valid domain, i.e. one that leads to valid computations or to a specific error condition, otherwise the point may be rejected for coincidental reasons, e.g. by an initial analysis that rejects all points not in a domain that leads to subsequent processing.

The above weak testing strategies require one set of test cases for each boundary inequality. Weak strategies assume that the boundary extends to infinity in all directions, that there are no gaps in the boundary, and that closure is consistent along the boundary's entire length. If these conditions are not satisfied, then one can apply a stronger testing strategy with similar results. A *strong* strategy requires one set of test points for each boundary *segment* instead of just one set for each boundary. Boundary segments are determined by gaps in the boundary, by adjacent domains that share a boundary, and by places where the inequality changes direction so that the boundary closure changes.

The main objective of domain testing is to ensure that the boundary conditions are properly represented in the software being tested and that the functional specification is free of contradictions and ambiguities in the domains derived from it. We will want to ensure that adjacent domains have the proper boundary inequality between them and that there are not any extra boundaries that serve no purpose in the software.

## Finite State Testing

The finite state machine model dates to the mid-fifties [Meal55, Moor56] where it was used to support automata and switching theory for testing hardware logic. Later it was used for protocol modeling and testing [Bosi91] and more recently it is being used for testing menu-driven applications [Beiz95] and to represent abstract data types in object modeling [Booc91, Rumb91] and testing [Hong96]. Modeling finite states as a Markov chain, with transition probabilities among the states, also leads to interesting stochastic observations [Whit94, Poor97]. Finite state machines are often represented by directed graphs with the states as nodes and the transitions between states as directed arcs; however, the models differ in that some add nodes to represent the output of a transition, or the event that triggers a transition, whereas others overload the arcs to represent both the transition and the event that caused the transition. Each approach has advantages, depending on the objective one is trying to achieve. Here we follow the finite state model of [Hong96] because it follows the popular definitions in [Rumb91] and is specialized to support the testing of class definitions in object-oriented programming, a major goal of this research.

The behavior of a class in object-oriented programming can be specified in terms of states and events. When an event is received, e.g. a message is received or a function is invoked, the next state of the object depends on the current state as well as the content or action of the event. A change of state caused by an event is called a transition. The properties of states and transitions are as follows:

> State - A state is an abstraction of the data attributes of a class. A set of attribute values is grouped together into a state according to the properties that affect the gross behavior of the class. Thus the state of a class can be defined by predicate conditions on the values of the data attributes of the class.
>
> Transition - A transition is composed of three items, an event, a guard, and an action. An event can be represented as a call to a member function of the class. A guard is a condition that must be satisfied in order for the transition to occur, i.e. for the function call to be successful. It can be a predicate condition on any of the data attributes of the class or on any of the input parameters of the function call. If the guard condition evaluates to TRUE, then the action is invoked. An action is a set of operations that are performed when the

transition occurs. The action can reference or manipulate any of the data attributes of the class or any of the output parameters of the calling function.

Special States - The states of a class include an initial state, $s_0$, and a final state, $s_f$. The predicate conditions on the class attributes for $s_0$ and $s_f$ are undefined. The initial state is the source state for any transition whose event is a member function of the class that creates the first object instance. The final state is the target state for any transition whose event is a member function of the class that destroys the last object instance. In simple class modeling and testing, at most one object instance can exist at any given time. The model generalizes if the states are extended to include the number of object instances in existence at any given time. If the class model includes specification of error conditions, then the states of the class include an error state, $s_e$. The predicate conditions on the class attributes for $s_e$ are undefined. The error state is the target state for any transition whose event is a member function of the class that raises an error condition. The model generalizes if each error condition is considered to be a distinct error state.

A simple class state model, i.e. where at most one object instance of a class C can exist at any one time, can be represented by the algebraic tuple M = (V, F, S, T) where V is a finite set of the data attributes of C, F is a finite set of member functions of C, S is a finite set of states of C, i.e. S = {s : s = (predV)} where predV is a predicate on the data attributes in V, and T is a finite set of transitions, i.e. T = {t : t = (source, target, event, guard, action)}, where source in S, target in S, event in F, guard is a predicate on attributes in V or parameters of functions in F, and action is a computation on attributes in V or parameters of functions in F. A simple class state model can be represented as a directed graph with S as the set of nodes and members of T as labeled arcs from the source node of T to the target node of T.

Hong and co-authors [Hong96] provide a straight-forward algorithm to transform a simple class state model M to a pure data flow graph G. Each state of M becomes a state-processing node of G, each non-trivial guard (i.e. guard not identically TRUE) becomes a guard-predicate node of G, and each transition of M becomes a transition-processing node of G. The arcs of G are determined directly from the transitions such that every t in T produces: 1) an arc from the transition-processing node determined by t to its target state-processing node, 2) if the guard of t is identically TRUE then an arc from the source state-processing node of t to the transition-processing node determined by t, 3) if the guard of t is not identically TRUE, then an arc from the source state-processing node of t to the guard-predicate node determined by the guard of t and an arc from the guard-predicate node determined by the guard of t to the transition-processing node determined by t.

The final step in transforming a class state model M to a data flow graph G is to decide which of the nodes of G are predicate nodes, which are definition nodes, and which are just processing nodes, and similarly, which of the arcs are predicate-used. This is achieved as follows: all guard-predicate nodes are considered to be data flow predicate nodes; all state-processing nodes are considered to be non-definition data flow processing nodes; for every attribute v in V, v is said to be *defined* at a transition-processing node t if the action of t assigns a value to v, and v is said to be *computation-used* at a transition-processing node t if the action of t references v; similarly, v is said to be *predicate-used at a state-transition* arc (s,t) or a *state-guard* arc (s,g) if the event of s references v, and v is said to be *predicate-used at a guard-transition* arc (g,t) if g references v. At this point G can be considered as a pure data flow graph and all of the conventional data flow testing techniques [c.f. Fran88] can be applied to G.

The above paragraphs describe how the specification of an object class can be modeled as a finite state machine, and then how the finite state machine can be transformed into a pure data flow graph. The various coverage techniques for testing data flow graphs can then be applied, thereby giving a conformance testing method for functional specifications of object classes. In particular, the All-Uses data flow method can be applied to determine conformance of class implementations to their functional specification.

## Category-Partition Testing

Category-partition testing is a generalization and formalization of a classical functional testing approach that first partitions the input domain of the functional unit to be tested, and then selects test data from each class of the partition. The basic assumption is that all elements within an equivalence class are essentially the same for the purposes of testing and that any element of a class will expose an error in the software as well as any other one. As described in [Ostr88], the category-partition method is a systematic, specification-based approach, that first uses partitioning to generate a formal test specification, and then uses a generator tool to eliminate undesirable or impossible combinations of parameters and environments, and then automatically produces functional tests for each functional unit of a complex software system. The main advantage of this approach is the creation of a formal test specification, written in a test specification language, e.g. TSL, to represent an informal or natural language functional specification. The test specification lends itself to automation, both in the analysis of constraints to eliminate impossible test cases and to generate input parameters for the test cases themselves. The main disadvantage of this approach is that it only produces input values for test cases, it does not automatically validate the results of the test parameters to determine correctness with respect to the functional specification. The formal test specification gives the tester a logical way to control test generation and is easily modified to accommodate changes or mistakes in the functional specification. In a later paper, [Amma95] extends this approach to replace TSL by the formal specification language Z and to specify a procedure for satisfying a minimal coverage criterion for category-partition testing. The use of Z allows more flexibility in the specification of constraints and more formality in the representation of categories and partitions. It also allows the specification of post conditions that will help in test case validation. An analysis of different partition testing strategies is given by Weyuker [Weyu91].

The main steps in the category-partition testing method are as follows:

Analyze the Specification. Identify individual functional units from the functional specification that can be separately tested. For each unit, identify: the parameters of each unit, the characteristics of each parameter, objects in the environment whose state might affect the functional unit's operation, and the characteristics of each environment object. Parameters are the explicit inputs to a functional unit and environmental conditions are characteristics of the system's state at the time of execution of a functional unit.

Identify Categories. A category is a classification of the major properties or characteristics of a parameter or environmental object. Properties of a character string input parameter might be its length, its length relative to other objects, restrictions on characters that make up the string, treatment of special characters in the string, etc. Environmental conditions might be the status of various switches, the content of system parameters, the content of peripheral data objects, size restrictions on peripheral data objects, etc. Categories can be derived directly from the functional specification, from implicit design information, or from the experience and intuition of the test developer. Often categories will be derived from preconditions and type information about the input parameters and system state components.

Partition the Categories into Choices. Partition each category into distinct choices that include all the different kinds of values that are possible for the category. [Weyu91] describes strategies for designing partitions. Each choice will be an equivalence class of values that will be assumed to have identical properties as far as testing and error detection are concerned. The choices must be disjoint and must cover the entire category. Some categories partition nicely into normal cases and unusual or illegal cases based directly on the functional specifications; partitions for other categories may depend on implicit design constraints or the test developer's intuition. The selection of a single choice from each category will determine a test frame. In the absence of constraints (see next paragraph) the number of potential test frames will be the product of the number of choices in each category, likely a very large number.

Determine Constraints among Choices. Constraints are restrictions on the way that choices within different categories can interact with one another. A typical constraint will specify that a choice from one category cannot occur together in a test frame with certain choices from other categories. Choices and constraints will be derived from the natural language functional specifications, but can often be specified by formal methods, thus making their analysis easier to automate. With a careful specification of constraints, the number of potential test frames can usually be reduced to a manageable number. In any case, [Amma95] defines a minimal covering criterion for choosing test frames whose number is linear in the number of choices, rather than the product of the number of choices in each category.

Formalize and Evaluate the Test Specification. Specify the categories, choices, and constraints using a formal specification technique that is compatible with the test generation tool, i.e. TSL for [Ostr88], Z for [Amma95], or SCR notation for [Heit96, Blac97]. Most test generation tools also provide automated techniques for evaluating the internal consistency of the formal test specification. This evaluation often discovers errors or inconsistencies in the specification of constraints and sometimes leads to discovery of errors in the source functional specification.

Generate and Validate the Test Cases. Use the test generation tool compatible with the Test Specification method to instantiate test cases by specifying actual data values for each choice. The tool can then generate test cases that satisfy all of the constraints. If the Test Specification includes post conditions that must be satisfied, then the tool can check that the post conditions will be satisfied for each execution of a test case. If a reference implementation is available, then the tests can be validated by checking their results against the reference implementation. In the absence of a reference implementation or comprehensive post conditions, the validation of test cases can be a labor intensive process.

## 3 Binomial Trials

The binomial model describes the number of successes in *n* independent trials, where each trial has probability of success *p*. It is widely used in statistical analyses, and offers a flexibly robust approach to the qualitative description of many real-world problems. The paradigm application is the determination of the probability of obtaining exactly *k* heads in *n* tosses of a (possibly unfair) coin. Formally, the binomial model states that

$$P[X=k] = \binom{n}{k} p^k (1-p)^{n-k}$$

where *k* may be any integer between 0 and *n*, inclusive, $0 \leq p \leq 1$, and *n* is some positive integer. By convention, *X* is the random variable and *k* is the value it takes; for example, *X* is the number of heads in 10 tosses, and *k* might be 3.

To use this model in conformance testing, we imagine that *n* is the number of tests built from the conformance requirements. We assume that a new piece of software will pass each test independently with unknown probability *p*. The interest of the assessor is in estimating *p*, and we describe below a number of standard statistical methods that are available to achieve that end.

Regarding the validity of the assumptions that underlie this application, there are two questions:

* Is *p* constant across tests?

* Are the tests independent?

The constancy of *p* is a slightly philosophical issue, and can be satisfied in several ways. One way is to imagine that there is a large set of *N* possible tests, from which *n* are chosen in some random way (either as simple random sampling, or, more realistically, as stratified sampling from strata determined by code functionality). The independence of the tests is an issue if several tests pertain to the same kind of functionality, so that failure on one test would suggest or ensure failure on others. But for this application, the effect of dependence will be to make the inference more conservative; correlated failures will deflate the value of *p*, making the assessor less likely to approve the code. The problem of correlated successes should not arise in any stringent test (since even a small number of failures is damning) provided that the tests provide good coverage of the range of functionality.

Our goal in employing this model is to develop fast, easy, methods that enable conformance testers to:

* quantify their uncertainty about the performance of the software under test,

* make statistical forecasts of that software's future performance, and

* reduce testing costs.

The techniques described in the following sections support all three of these goals.

## 3.1 Binomial Inference

In the standard binomial model described above, one estimates $p$ by $\hat{p} = X/n$, where $X$ is the number of successes. Also, to quantify one's uncertainty about $\hat{p}$, one uses a 100(1-$\alpha$)% one-sided lower confidence interval of the form

$$L = \hat{p} - z_{\alpha} \sqrt{\hat{p}\,(1 - \frac{\hat{p}}{n})} \qquad (1)$$

where $z_{\alpha}$ is a percentile point from the standard normal distribution. To interpret this interval, one says that out of 100 independent replications of this experiment, 100(1-alpha)% of the time the true but unknown value of $p$ will be greater than the value of $L$ generated by the experiment. (In other applications one often uses the more familiar two-sided confidence interval

$$U, L = \hat{p} \pm z_{\alpha/2} \sqrt{\hat{p}\,(1 - \frac{\hat{p}}{n})}$$

but traditional thinking in conformance tests puts no value on finding upper bounds U for the probability of code success.)

It is desirable that these confidence intervals have small width, since this indicates great certitude about the value of $p$. The confidence interval given in (1) is a very accurate approximation based upon the Central Limit Theorem; however, if needed, an exact interval can be obtained by numerical solution of an integral or by employing the Clopper-Pearson tables.

## 3.2 Sequential Binomial Tests

To reduce testing costs under the binomial model, one can use sequential analysis. Here $n$ is allowed to grow until some stopping condition is satisfied. In this application, the natural stopping rule to consider is:

> Stop at the first failure, or whenever a 95% lower bound $L$ on $p$ exceeds .98.

This rule assumes that conformance testing stops when the first failure is found. In some cases, it may be better to adopt a more latitudinarian rule that stops after a small fixed number of noncritical failures are found. Of course, the values of 95% and .98 should be chosen to reflect a cost-benefit comparison that takes account of the specific risks of approving faulty code.

## 3.3 Binomial Model for a Finite Population

Another approach to reducing test costs arises if we can assume that there are a known finite number $N$ of tests, from which a subset of $n$ have been selected at random (without duplication) and performed. In this case, one can modify the confidence interval (1) to take account of the fact that the population of tests is finite; thus

$$L = \hat{p} - z_{\alpha} \sqrt{\hat{p}\,(1-\frac{\hat{p}}{n})} \sqrt{\frac{N-n}{N-1}}$$

The last term is called the *finite population correction*; it occurs analogously in the two-sided confidence interval as well.

Notice that the effect of the finite population correction is to reduce the uncertainty in the estimate of $p$, since $(N-n)/(N-1)$ is always less than 1. This enables one to achieve the same level of certitude with fewer tests than would be possible using the analysis described in subsection 3.1. This correction can be used in conjunction with the sequential analysis in the obvious way, by adjusting the value of the lower bound in the second part of the stopping rule.

The value of this approach depends upon the validity of the assumption that there are only a finite number $N$ of possible tests. This assumption is not unreasonable in the context of conformance testing, where there are a fixed number of listed requirements, each of which will spawn only some fixed number of tests. However, the inference on $p$ will refer only to the proportion of requirement tests that would be passed, and is not especially relevant to forecasting the behavior of the code after release unless the requirements fully cover the entire range of function the code must express.

## 3.4 Bayesian Analysis with a Beta-Binomial Model

A third strategy for achieving cost reduction and a formal statement of uncertainty is to employ Bayesian methods. This approach used to be controversial, in that it allows the tester to use prior information, accumulated from previous experience in similar situations, when designing the experiment and making inferences.

Heuristically, the experimenter starts with a prior distribution on the value of $p$, and then observes the outcomes from $n$ independent tests. After seeing that data, the experimenter then updates the prior distribution to find the posterior distribution, which now describes the experimenter's beliefs about the value of $p$ based on both the prior belief and the observed data. The tool for that updating is Bayes' Theorem, which says

$$f^*(p|x) = \frac{g(x|p)\, f(p)}{\int g(x|y) f(y)\, dy} \qquad (2)$$

where $f^*$ denotes the posterior distribution based upon the data $x$, $f$ denotes the prior distribution on the unknown probability of success $p$, and $g(x|y)$ denotes the distribution of the data, x, when the chance of success on each independent trial is fixed at y.

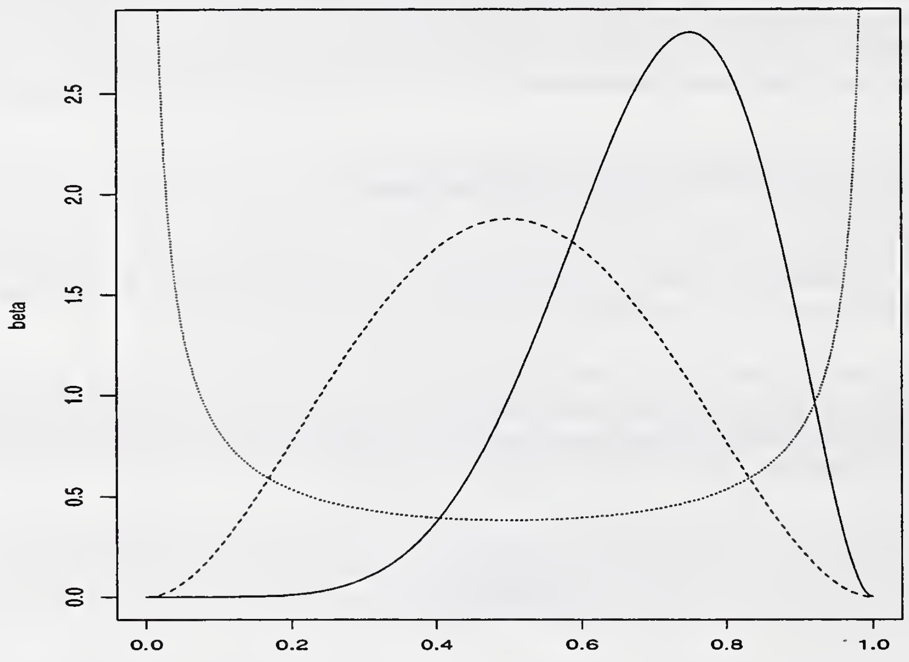
The formula applies for any distribution $f(p)$, so that an unscrupulous experimenter could choose a distribution which so overwhelmingly weighted the belief in (say) a small $p$ that no amount of data could reverse that prejudice in any fixed number of trials. But that requires arrant dishonesty, against which no conformance testing protocol is secure. In general, with reasonable oversight, the Bayesian approach can accomplish substantial savings by permitting testers to take formal advantage of their experience.

Although (2) applies with any distribution $f(p)$, some choices are mathematically convenient. For this reason, many similar analyses take a beta distribution as the prior on $p$. Here

$$f(p) = \frac{\Gamma(r+s)}{\Gamma(r)\,\Gamma(s)}\, p^{r-1}(1-p)^{s-1}$$

where $r,s > 0$ and $0 \le p \le 1$. The values of $r$ and $s$ are the parameters of the beta, and they allow a conveniently flexible range of beliefs to be expressed about $p$. For example, taking $r = s = 1$ gives the uniform prior, saying that $p$ is equally likely to take any value between 0 and 1. Taking $r > s$ says one's prior belief is that $p$ tends to be large, and $s > r$ implies the opposite. If $r + s$ is small, then the experimenter is injecting little prior information into the problem, but if it is large, the prior has great weight; this is why $r + s$ is sometimes called the *prior sample size.*

The following figure shows the shapes of three representative beta densities. The continuous line is the B(7,3) density; the dashed line is the B(3,3) density; the dotted line is the B(.25,.25) density. Note the great range of shapes available to model different kinds of prior belief.



Densities of three beta distributions, to indicate the range of possible shapes.

The mathematical consequence of choosing a beta prior is that the posterior is again a beta distribution. The posterior beta no longer has parameters *r* and *s*, but rather the parameters *r+x* and *s+n-x*, respectively. Thus large values of *x*, corresponding to many successful tests, will increase the posterior belief that *p* is large. The mean of this posterior is

$$E[p|x] = \frac{r+x}{r+s+n} = \frac{x}{n}\,\frac{n}{r+s+n} + \frac{r+s}{r+s+n}\,\frac{r}{r+s}$$

which is a weighted average of the prior mean and the sample mean, with weights determined by *r*, *s*, and *n*. Thus the inference blends prior belief with information in the data. Also, the posterior variance of *p* is

$$Var[p|x] = \frac{(r+x)(s+n-x)}{(r+s+n)^2\,(r+s+n+1)}$$

Thus as *n* increases, the posterior variance, a measure of statistical uncertainty, goes to 0.

Depending upon how the tests are generated, the Bayesian approach can enable predictive forecasts of software reliability. If the tests may be described as an independent sample of typical applications, rather than a random sample from a requirements list, then the estimated probability that the next use of the system will be successful is the posterior mean, $(r+x)/(r+s+n)$. This result is called Laplace's Law of Succession, and was originally derived in an attempt to calculate the probability that the sun would rise tomorrow.

There are other ways in which a Bayesian perspective can improve conformance testing. For example, in the sequential testing paradigm, a skilled tester may have some prior guesses as to which tests are most stringent, and by administering them first, can more quickly disqualify failing software and thereby reduce the time-on-test. A similar approach, used with the finite-population correction test, can also yield early termination (although the estimate of $p$ will be biased downwards).

## 3.5 Impact on Black-Box Conformance Testing

In standard black-box conformance testing, one has a list of requirements which must be exhaustively and thoroughly tested. This is labor-intensive, especially when new tests must be developed. The binomial models described in this section offer tools that will bring about modest decreases in the number of tests needed to achieve the same level of certitude as present practice. Moreover, these methods allow testers to make explicit probability statements about the degree of uncertainty in their findings, as opposed to current practice, in which the uncertainty in the certification is swept under the carpet.

## 4 Coverage Designs

Coverage designs extend classical experimental methods in statistics to address the kinds of interactive failure modes that arise in software reliability and testing. Many of the methods are relatively new, and relatively mathematical. Alfred Renyi [Reny71] posed the mathematical problem of listing the smallest set of binary sequences of length $k$ such that for each pair of positions $i, j$, all four possible pairs of entries [i.e, (0,0), (0,1), (1,0), and (1,1)] were represented. Sloane [Sloa93] extended this to the consideration of triplets of positions, but formal mathematical statements quickly become arduous. The Renyi/Sloane covering designs might have remained a mathematical curiosity, but Dalal and Mallows [Dala97] argued that it could lead to the construction of software testing procedures that were more efficient than designs built from more conventional statistical considerations. At present it is unclear whether the linkage they proposed is sound, but some early success has been claimed. This research will attempt to assess the potential of these covering methods for software reliability and testing, and also to find ways to make their regular use more practical.

We plan to examine the extent to which these designs can be usefully applied to black-box conformance testing. Subsection 4.1 describes the reasons why coverage designs may apply in conformance testing, and subsection 4.2 details these designs more carefully, with an example layout. Subsection 4.3 explains the directions we shall consider in further evaluating this methodology.

## 4.1 Experimental Designs for Software Testing

In conventional statistical designs, one is chiefly concerned to estimate main effects (such as the effect of fertilizer and variety on the yield of corn), and secondary interest pertains to interactions between factors (such as the residual joint interactive effect of fertilizer and variety on yield, after explaining all that is possible through main effects). Additionally, it typically happens that the greatest source of uncertainty in experimental designs is the natural random variation (residual error) that arises in replicated measurements at the same levels of the factors (e.g., the variation in yield between different corn plants of the same variety that have received the same fertilizer).

This paradigm is not closely applicable in software testing. First, one can argue that there is no random variation; either the software passes or fails the test, and if one were to repeat the same test over and over, the outcome would always be the same. Second, it is widely believed that most failures in software are interactive, meaning that failures are observed only when two or more modules happen to be in specific states (as opposed to "main-effect" failures, that would arise whenever a given module is employed).

Dalal and Mallows [Mall96, Dala97] argue that the failure of the usual statistical paradigm opens the door to coverage or Renyi designs. These designs attempt to exercise every pair or triplet or r-tuplet of modules in a program together, in all their states, to determine whether specific combinations of module states produce software errors. This assumes that modules (factors) can be identified, which is often problematic in black-box testing, although this can be implicit in functional specifications. Also, it assumes that there are only two (or, in more advanced realizations of the coverage design strategy, only a few) states that each module can take. When these assumptions are satisfied, there is a stunning reduction in the number of experiments that need be done, as contrasted with the combinatorial explosion that arises in conventional design theory. But more work is needed to assess whether these designs can reliably supercede the more exhaustive testing that is usual. In fact, it is not clear that the new paradigm advanced by Dalal and Mallows is actually a useful description of the software testing problem.

## 4.2 Optimal Coverage Designs

Suppose that a software system has $k$ modules, each of which may be either *on* or *off* in a given application. We assume that in order for the system to fail, at least $r$ of the modules have to be in a particular set of states. For example, if the modules are labelled A, B, C, ..., and each can be in either the 0 or 1 (*on* or *off*) state, then a system might fail if (A, B) is (0,0), (0,1), (1,0), or (1,1), and all four such possibilities must be checked during conformance testing.

The idea behind coverage designs is that a single test of a computer program must necessarily set all modules to one or the other state, and thus if one is clever, one can simultaneously assess many pairs. For example, if (A,B,C,D) is (0,0,0,0), then one is testing six possible pairs of module states, all at the *off-off* level, in a single execution of the software. The heart of the coverage design strategy is to exploit these economies as efficiently as possible.

To be somewhat more concrete, assume that one has $k = 10$ modules, and that one is satisfied to search for failures involving $r = 2$ or fewer modules. A naive tester might assume that $4\binom{k}{r} = 180$ tests are required, but by loading many tests onto each execution, that entire effort can be reduced to six tests. The table below shows how this is done.

| Module | A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|---|
| Run 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Run 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| Run 3 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| Run 4 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| Run 5 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 |
| Run 6 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Note that every pair (0,0), (0,1), (1,0), and (1,1) occurs at least once for each possible combination of modules.

Similar designs can be constructed for testing for failures that involve triplets of module states. In the case with $k = 10$ and $r = 3$, Sloane [Sloa93] gives a construction that requires 12 runs to test all possible conditions. But in general, it is a difficult problem to construct the binary array that implements an optimal coverage design. An even more complex problem arises when the modules can take more than two states.

## 4.3 Impact on Black-Box Conformance Testing

The important contribution of coverage designs is their potential to drastically reduce the number of runs needed to make a conformance test of a program. The difficulty in applying them is that it is not obvious how one identifies a module, especially in the context of black-box testing, and it is unclear whether the modules can assume only a small number of states. A further, and more practical issue, is that it can be hard to develop conformance tests that exercise together exactly the modules specified in the design.

Nonetheless, Cohen, *et al*, [Cohe96] report success with this method, and the software testing community feels the technique deserves attention. Our hope is that one can generally determine modular structure in a program from functional requirements, and that the model of a small number of states is sufficient to describe some (or most) of the operational failure modes. To pursue this further will require interaction with technicians who have had experience in implementing these designs to assess software systems.

## 5 Mutation Testing

Mutation testing is a two-step process for determining the correctness of a given software program with respect to a given functional specification. The first step is to determine the adequacy of an existing test suite for its ability to distinguish the given program from a similar but incorrect program, i.e. a mutant of the given program. A mutation of a program is a modification of the program created by the introduction of a single, small, legal syntactic change in the software. Some mutations are equivalent to the given program, i.e given the same input they will produce the same output, and some are different. Test suite adequacy is determined by mutation analysis, a method for measuring how well a test suite is able to discover that a mutated program is either equivalent to, or different from, the original program. A test suite is 100% adequate if it is able to discover every possible non-equivalent mutation. Since 100% adequacy is nearly impossible to achieve, the second step of the mutation testing process is to make an inadequate test suite more robust by adding new test cases in a controlled manner. The test suite enhancement process stops when mutation analysis shows that the test suite has reached a desired level of adequacy. The modified test suite, together with its adequacy measure, is then used to determine a level of confidence for the correctness of the original source program with respect to its functional specification.

Adequacy of a test suite with respect to a given collection of mutant programs is often defined to be the ratio of the number of discovered non-equivalent mutants from that collection divided by the total number of non-equivalent mutants in the collection. Since the number of possible mutations can be very large, often proportional to the square of the number of lines of code, testing to estimate this ratio can be a costly and time consuming process. Mutation testing and mutation analysis have matured since 1977 and researchers have found a number of ways to get equivalent results with smaller and more carefully designed classes of mutant programs. Recent researchers have shown that by choosing an appropriate class of mutations, mutation testing can be equivalent to data-flow testing or domain testing in its ability to determine program correctness. Since many of the steps in mutation analysis can be automated, the mutation testing process is now suitable for industrial use as a white-box methodology for unit testing.

Early researchers in mutation testing include A.T. Acree, T. Budd, R.A. DeMillo, F.G. Sayward, A.P. Mathur, A.J. Offutt, and E.H. Spafford. Active research centers include Yale University, the Georgia Institute of Technology, Purdue University, and George Mason University, as well as industrial software houses. Automated mutation systems include Pilot [Budd77] at Yale University and Mothra [DeMi88]at Purdue University.

## 5.1 Advantages and Disadvantages of Mutation Testing

**Advantages**

- Research has shown that there is a strong correlation between simple syntax errors and complex syntax errors in a program, i.e. usually a complex error will contain some simple error. By choosing a set of mutant programs that do a good job of distinguishing simple syntax errors, one has some degree of confidence (not statistically measurable) that the set of mutant programs will also discover most complex syntax errors.

- By proper choice of mutant operations, comprehensive testing can be performed. Research has shown that with an appropriate choice of mutant programs mutation testing is as powerful as Path testing or Domain analysis [Fran93]. Other research has tried to compare mutation testing with Data Flow testing [Math94, Offu95] with evidence (not proof) that mutation testing is at least as powerful.

- Mutation analysis is more easily automated that some other forms of testing. Robust automated testing tools have been developed at multiple universities and industrial testing groups, e.g. Yale (Pilot) [Budd77], Purdue (Mothra) [DeMi88], Georgia Institute of Technology, and George Mason Univ.

- Mutation testing lends itself nicely to stochastic analysis (see Section 5.4).

**Disadvantages**

- Each mutation is equal in size and complexity to the original program, and a relatively large number of mutant programs need to be tested against the candidate test suite.

- Each mutant may need to be tested against many of the test cases before it can be distinguished from the original program.

- Some research has shown that a "good" set of mutant programs may require a number of mutants proportional to the square of the number of lines of code in the original program [Sahi90], potentially a very large number.

## 5.2 Why consider Mutations for Black-Box Testing?

Since mutation testing is clearly dependent upon the source code of the program being tested, why is it being considered for use in black-box conformance testing? Listed below are some of the reasons why we identified mutation testing as a potentially useful technique for black-box conformance testing.:

- Once a test suite derived from mutation analysis is available, the program it tests need not be further modified; the tests can be applied in a black-box fashion to the public interface. It might be possible to use mutations of a reference implementation of a functional specification to produce a test suite adequate for testing any other implementation of the same functional specifications. We were unable to find any literature supporting this point of view, but we were unable to reject it as a possible technique. It is worthy of further consideration.

- Many articles in the mutation testing literature compare data flow testing with mutation testing [e.g. Offu95, Math94]. Since data flow testing has variants for both black-box and white-box testing, we thought that comparisons between mutation testing and data flow testing in white-box environments might lead to a mutation testing variant for black-box conformance testing. However, the comparisons we analyzed used experimental rather than theoretical arguments for evaluating the effectiveness of mutation testing versus data flow testing. Thus we were unable to find any theoretical or algorithmic way to derive a black-box mutation testing variant. This line of research looks like it leads to a dead end.

- Some articles in the mutation testing literature have titles like "using mutation analysis to test the functional correctness of programs" [Budd80]. We were hoping that the reference to functional correctness was a reference to black-box testing of functional specifications. It wasn't! Like most of the other mutation testing literature, the emphasis was on syntactic mutations of the given program. It seems like none of the current mutation literature considers mutation testing in anything other than white-box terms, i.e. syntactic modifications of a given program.

- Several good articles on application of statistical methods to mutation testing [e.g. Sahi90, Sahi92] might have analogues useful for black-box testing. However, the methods discussed in mutation testing are similar to those already under consideration in other parts of this report, i.e. Bayesian binomial methods and sequential probability ratios to decide when to stop testing and whether to accept or reject a candidate test suite. There don't appear to be any new statistical methods discussed in the mutation testing literature that might be applicable to black-box conformance testing.

Of the above potential reasons for pursuing mutation testing as potentially applicable to black-box conformance testing, only the first reason in the list shows any promise for further research and analysis. That reason is further discussed in Section 5.5 below.

## 5.3 Generic Mutation Testing Methodology

The following algorithmic method describes the generic methodology for mutation analysis and the building of a robust test suite for mutation testing:

1) Begin with a software program P and a set of test cases T known to be correct tests with respect to a given functional specification S.

- Usually T will be designed to already test most of the individual requirements of a functional specification. We use mutation analysis to make T more robust.

2) Apply the tests in T to P. If one or more of the tests fail, then P is in error. If there are no failures, then continue with step 3.

3) Create a set of mutant programs $\{P_i\}$, each differing from P by a simple, syntactically correct modification of P.

- There is a lot of research on methods for creating efficient mutant sets for individual programming languages and common programmer errors. In addition, the process of creating and testing mutants can be automated.

- Sets of mutant programs may be categorized by Type, e.g. some Types are good for checking common programmer errors, some might check all possible syntactic alternatives of operators in the language, some check for data flow or domain analysis. The set of all possible mutants may be too large for reasonable testing, but these special collections may be quite manageable, especially if we are looking only for a certain type of error in the program.

4) If a mutant program, $P_i$, produces any different results for the tests in T, then $P_i$ is clearly distinguishable from P and is said to be *killed* by the test suite T. If $P_i$ produces exactly the same results as P for the tests in T, then one of the following is true:

a) either T is *not adequate* to distinguish $P_i$ from P, or

b) P and $P_i$ are *equivalent* and their behavior cannot be distinguished by any set of tests.

- Adequacy is the ratio of the number of killed programs over the number of non-equivalent mutants. There is a lot of research on how to distinguish and eliminate mutant programs that are equivalent to P.

- Since 100% adequacy is not practical, there is a lot of research to estimate the adequacy of a test suite with respect to a given type of mutant programs.

5) If the estimated adequacy of T is not sufficiently high, then design a new test that satisfies the functional specification and distinguishes $P_i$ from P; add that new test to T, and go to step 2. If the estimated adequacy of T is beyond an appropriate threshold, then accept T as a good measure of the correctness of P with respect to the set of mutant programs and STOP designing new tests.

6) Declare P to be a correct implementation of the functional specification with respect to the set of mutant programs. Use the definition of adequacy in step 4 to derive a numeric confidence level for the statement that P is a correct implementation of the functional specification.

## 5.4 Stochastic Considerations

If Step 5 of the preceding mutation testing methodology is based on a statistical analysis, then it will consist of a stopping rule and a decision rule. The stopping rule will specify, for the current contents of the test suite T, whether to stop or continue improving the quality of T by constructing additional tests. If the stopping rule is invoked, then a decision rule based on the current contents of T will determine the likelihood within a stated confidence level that the given program P conforms to the functional specification S.

Sahinoglu and Spafford [Sahi90, Sahi92] use accepted statistical methods for Binomial Testing with a Sequential Probability Ratio Test to derive stopping and decision rules for the mutation testing methodology. Based on the notion of counting defective parts from manufacturing, they define test suite quality to be the ratio of non-killed mutants divided by the number of non-equivalent mutants, i.e. 1 - test suite adequacy. A test suite that has fewer non-killed mutants will be of higher quality than one that misses its opportunity to distinguish a mutant of P from the original program P. For Binomial Testing, with p being the probability that an erroneous mutant program will not be killed by T and 1-p the probability that it will, [Sahi90] uses the binomial sequential probability ratio test based on sequential random sampling from the set of mutations of P to determine Type I and Type II errors based on the null hypothesis that $p = p_1$ versus the alternative hypothesis that $p = p_2$, where $p_1 < p_2$. Thus a Type I error is the probability of mistakenly rejecting a test suite of good quality, i.e. $p \leq p_1$, and a Type II error is the probability of mistakenly accepting a bad test suite, i.e. $p \geq p_2$. [Sahi90] goes on to show that if $C_{21}$ is the decision cost incurred if a software product with a fraction of live mutants less than or equal to $p_1$ is rejected, analogous to the Type I probability of rejecting a good program P mistakenly, and if $C_{12}$ is the decision cost incurred if a software product with a fraction of live mutants greater than or equal to $p_2$ is accepted, analogous to the Type II probability of accepting a bad product mistakenly, then so long as the cost of inspection depends merely on the total number of mutants inspected, and no overhead cost is involved, a sequential sampling inspection plan for mutant programs will be the most economical plan.

[Sahi92] derives stopping and decision criteria for a most economical mutation testing plan based exclusively on the following four input variables, under the assumptions that one mutant program is inspected at a time and the inspection is stopped as soon as sufficient evidence is observed in favor of either of the hypotheses:

| | |
|---|---|
| $p_1$ | the *acceptable* quality limit for the quality of a software test suite, expressed as a fraction of live mutants allowed to pass undetected. For conformance testing we want $p_1$ to be very small. |
| $p_2$ | the *unacceptable* quality limit for the quality of a software test suite, expressed as a fraction of live mutants allowed to pass undetected. For conformance testing we want $p_2$ to be no more than twice the value of $p_1$. |
| $\alpha$ | the maximum probability of mistakenly rejecting test suites of good quality, i.e. $p \leq p_1$. For conformance testing, this value becomes important only when the relative cost of producing new test cases becomes prohibitively expensive. |
| $\beta$ | the maximum probability of mistakenly accepting test suites of poor quality, i.e. $p \geq p_2$. For conformance testing we want $\beta$ to be very small. |

The criterion for accepting or rejecting a mutation test suite constructed by sequential inspection of mutant programs is given by two parallel straight lines in the xy-plane with equations:

$$f_1(x) = m*x - h_1 \quad \text{and}$$
$$f_2(x) = m*x + h_2$$

where x is the number of mutant programs tested, f(x) is the number of live mutants remaining after testing, $-h_1$ is the y-intercept of the lower line, and $h_2$ is the y-intercept of the upper line. The slope and intercepts of the two lines are uniquely determined by the above four test suite quality requirements as follows:

$$g_1 = \ln(p_2/p_1)$$
$$g_2 = \ln((1-p_1)/(1-p_2))$$
$$a = \ln((1-\beta)/\alpha)$$
$$b = \ln((1-\alpha)/\beta)$$
$$h_1 = b/(g_1+g_2)$$
$$h_2 = a/(g_1+g_2)$$
$$m = g_2/(g_1+g_2)$$

The region in the first quadrant (i.e. $x \geq 0$, $y \geq 0$) of the xy-plane below the line determined by $f_1$ is the region for accepting the hypothesis that the test suite is good, with confidence 100%(1-$\beta$), and the region in the first quadrant above the line determined by $f_2$ is the region for rejecting the hypothesis that the test suite is good, with confidence 100%(1-$\alpha$) that the rejection is correct. The region between the two lines calls for continued sampling.

In step 5 of the mutation testing method, after mutation analysis of the first n randomly selected mutant programs, if the number of live mutants is less than $f_1(n)$, then stop constructing new test programs and conclude with confidence 100%(1-$\beta$) that the test suite T is adequate to catch all but 100%*$p_1$ of the errors in program P; otherwise, continue constructing new test cases to increase the quality of T with respect to the type of errors that T is designed to catch.

We observe that the line $f_1(x)$ can also be used to determine the minimum number of mutant programs that will be needed in order to have any chance of concluding that a given test suite is adequate or that a given program is correct with respect to its functional specification. But $f_1(x) > 0$ only if $mx > h_1$, or $g_2x > b$. Thus to distinguish $p_1 = .001$ from $p_2 = .002$ at 99% confidence against mistakenly accepting a bad test suite and 95% confidence against mistakenly rejecting a good test suite, we need a minimum of 4,547 mutant programs to work with. If $p_1$ and $p_2$ are increased to .01 and .02 respectively, the minimum required number of mutant programs reduces to 299; thus a collection of less than 1,000 independently generated mutant programs may be sufficient to pursue reasonable mutation analysis of a test suite T.

## 5.5 Impact on Black-Box Conformance Testing

In Section 5.2 above, we discuss some of the reasons why we considered mutation testing as a potential method for black-box conformance testing. Most of those reasons lead to dead-ends, but one reason could potentially lead to an approach for black-box conformance testing. Suppose we have the following given information:

> A functional specification, F, with a finite number of rules that define in detail the functional requirements of a software product claiming conformance to F.
>
> A test suite, T, that tests the main effects of the rules of F.
>
> A reference implementation, R, that satisfies all of the tests in T.
>
> A new test suite, T*, derived from T by mutation analysis using an appropriate collection of mutations of the reference implementation R. Assume that T* is derived using the methods of Sahinoglu and Spafford [Sahi92] so that T* satisfies pre-specified probabilities for Type I and Type II errors related to accepting R as a correct implementation of the functional specification F.

We should pursue the following black-box conformance question:

> If C is a commercial product, different from R, that claims conformance to the functional specification F, can the test suite T* be used to make any quantitative judgements, with appropriate confidence levels, about the probability of C's conformance or non-conformance to F.

# 6 Usage Models

The usage model provides a test design strategy for black-box conformance testing by focusing on test suites that find software bugs common to the environment of users. Historically, the input domain has been taken to be identical to the usage domain and under general assumptions, the reliability estimate is an unbiased estimate of the operational reliability. Thus, accurate confidence intervals can be determined for software reliability. Furthermore, these estimates can be used to determine the size of the input domain needed to provided a specified amount of confidence in the operational reliability of the software.

One shortcoming of usage models is that they do not take into account the fact that, in many cases, the input domain for the conformance test is chosen a priori. Thus, the input domain and the usage domain may differ. To rectify this, we propose a biased sampling approach to the problem.

## 6.1 Reliability and Usage Models

The reliability of a software system is the probability that it will function properly. By functioning properly we mean, the production of answers that are not deviant from that required. In discussing the statistical measurement of software reliability, it is convenient to think of two types of situations:

(i) those where the reliability measurement involves the occurrence of failure over time, i.e. the time until the next failure, and

(ii) the static model, where the reliability depends only on the successful performance of software given an intensive input interrogation. Weiss and Weyuker [Weis88] call these time-dependent and time-independent statistical software reliability analyses.

In the time dependent reliability model the raw data available to the tester is a sequence $t_1, t_2, \ldots, t_n$ of execution times between successive failures. As failures occur, attempts are made to correct the underlying faults in the software. Because the errors are corrected when detected and the corrections do not lead to further errors, it is reasonable to assume that the number of failures occurring in the time interval $(0,t]$, $N_t$ is a nonhomogeneous Poisson process. Thus the observed times can be regarded as realizations of random variables $T_1, T_2, \ldots, T_n$, the interarrival times of $N_t$. The problem is to compute the conditional distribution of $T_k$ given that, $S_{k-1} = \sum_{i=1}^{k-1} T_i = t$ , and is given by

$$R(x|t) = P(T_k > x \mid S_{k-1} = t)$$

$R(x \mid t)$ is just the probability that a failure does not occur in $(t, t+x)$ given the last failure occurred at time $t$. Of course, $R(x \mid t)$ is calculated from the distributional properties of $N_t$ and the observed failure times. Note that here the error detection times $t_1, t_2, \ldots, t_n$ will, more often than not, depend on the class of users.

In conformance testing the focus is on the time-independent case. The software system is subjected to an intensive test suite in order to discover errors. In many cases the tester follows some international standard, ISO, ANSI, IEEE, set up to specify minimal requirements the software must satisfy. Traditionally, the most important issue in conformance testing is determining whether an implementation performs as required, essentially a pass/fail test. But, in practice, we cannot test software for correctness as Parnas , *et al* [Parn90] point out: ``Because of the large number of states (and the lack of regularity in its structure), the number of states that would have to be tested to assure that software is correct is preposterously large. Testing can show the presence of bugs, but, except for toy problems, it is not practical to use testing to show that software is free of design errors." Therefore for the conformance tester it is incumbent that the problem be restricted to a solvable form. To accomplish this, the testing domain is restricted to an input domain that reflects the user environment. This will surely reduce the magnitude of the testing problem. Then, an uncertainty statement can be computed for software reliability. This uncertainty statement, of course, will be user specific, but does represent a reasonable approach to a difficult problem.

Let the input domain, $E$, of the software be the set of inputs intended for excution of the program and also reflecting the user environment. Partition $E$ into subdomains, $E_1, E_2, \ldots, E_n$. The $E_i$'s represent different parts of the software system, in general not all equally likely to be called. If one lets $p_i$ denote the probability of choosing input from $E_i$ and let

$$q_i = \begin{cases} 1 & \textit{if input } E_i \textit{ results in no failure} \\ 0 & \textit{otherwise} \end{cases}$$

then the reliability is equal to

$$R_1 = \sum_{i=1}^{n} p_i q_i .$$

This is the Nelson input-domain based model [Nels73].

In the Nelson model, the $q_i$ are obtained by testing all the elements of $E_i$, thus a large amount of testing is necessary in order to compute the reliability. Brown and Lipow [Brow75] proposed that the estimation of the reliability be carried out using the basic statistical theory of stratified sampling.

Choose $n_i$ input values from each $E_i$, $i = 1, 2, ..., n$ and denote by $f_i$ the number of failures that occur out of the $n_i$ tests. The reliability is then estimated by

$$R_2 = 1 - \sum_{i=1}^{n} \frac{f_i}{n_i} p_i$$

where the $p_i$ as above represent the probability of choosing input from $E_i$.

It can be shown that under general conditions, the estimate of reliability in Nelson's model is an unbiased estimate of the operational (user) reliability provided the input domain is identical to the user domain. Under the same conditions the model of Brown and Lipow [Brow75] will be asymptotically unbiased.

In conformance testing many times test suites are chosen *a priori* and the input domain may not be identical to the user domain. Then a biased form of stratified sampling can be used to estimate the system's reliability.

## 6.2 Impact on Black-Box Conformance Testing

The application of usage models to black box conformance testing can yield confidence statements about the operational reliability of the software based on its test performance. Furthermore, the statistical analysis used will provide a means of estimating the size of the test suites needed to provide a desired level of operational reliability.

## 7 When To Stop Testing

We are interested in the efficiency of the testing process and when it makes sense to stop testing. The approach we describe was developed in the context of software testing by Dalal and Mallows [Dala88]. It presents a strategy for sequential determination, based on the severity of bugs and the length of time required to discover them, of the optimal moment to stop the testing effort. This research will attempt to assess the potential of this strategy, and also to find ways to make its regular use more practical.

The original version of this problem was developed by Starr [Star74], who gave an explicit solution for the problem of deciding when to stop proofreading a document, assuming that each undiscovered typographical error entails a random cost and that the proofreader is paid by the hour. For this situation, it is obvious that after a certain point, it becomes unprofitable to continue proofreading; however, finding that point entails the solution of a nontrivial dynamic programming problem.

Dalal and Mallows [Dala88] recast this problem in the language of software, generalizing Starr's result in ways that were more faithful to the conformance testing paradigm. In particular, they allowed the costs of the bugs to vary randomly, they dropped the distributional assumptions on the length of time needed to find the bugs, and they obtained asymptotic results suggesting that the method applies when code is of inhomogeneous quality. However, it remains unclear whether their asymptotic results enable testers to achieve significant cost reductions in the finite-horizon reality of standard software testing.

Section 7.1 details the essential mathematics behind Starr's results. Section 7.2 describes the generalizations made by Dalal and Mallows. Section 7.3 suggests ways in which the efficacy of this technique might be assessed, and how the strategy can be simplified without sacrificing much of the optimality.

## 7.1 Starr's Result

Consider a set with a large number of objects, but with $N$ distinguished and hidden objects. The problem is to find, observe, or catch some or all of the group of $N$ objects, where there is a cost for searching and a reward for finding. The object could be prey, software bugs, etc.

We use the following notation:

| | | |
|---|---|---|
| $N$ | = | the number of objects, labeled 1, 2, ..., $N$ |
| $z_j$ | = | time it takes to capture the *j-th* object if one can search indefinitely |
| $y_j$ | = | value of the *j-th* object |
| $b$ | = | search cost per unit time |
| $k_t$ | = | number of objects caught by time $t$ |
| $w_t$ | = | total value of all objects caught at time $t = y_1 + ... + y_{kt}$ |
| *payoff* | = | $w_t - bt$ |

and make the following assumptions:

$z_1, z_2, ..., z_N$ are independent random variables with common exponential distribution function

$$Prob[z_j \le t] = 1 - e^{-\mu t} \quad \text{where} \quad t \ge 0$$

$y_1, y_2, ..., y_N$ are independent with common mean $a$.

$(z_1, z_2, ..., z_N)$ are independent of $(y_1, y_2, ..., y_N)$

Let **S** be the set of all possible stopping schemes that tell when to stop searching and collect a payoff (none should involve looking into the future). The optimal scheme, $\sigma \in \mathbf{S}$, satisfies

$$\begin{aligned} E[payoff] &= max! \\ &= max_{s \in S} E[w_s - bs] \end{aligned}$$

Since $wt = y_1 + ... + y_{kt}$ one gets

$$\begin{aligned} E[w_\sigma - b\sigma] &= E[y_1 + ... + y_{k_\sigma} - b\sigma] \\ &= E[ak_\sigma - b\sigma] \\ &= aE[k_\sigma - c\sigma] \text{ where } c = a/b \end{aligned}$$

Let $x_t = k_t - ct$, then the best search scheme is $\sigma \in \mathbf{S}$ satisfying

(*i*) $E[x_\sigma] = max_{s \in S}[x_s]$

(*ii*) with optimal payoff $\sigma$ where $E[x_\sigma] = k_\sigma - c\sigma$

**Theorem 1:** The optimal scheme satisfies:

$$\sigma = \textit{first time } t \ge 0 : k_t \ge N - c/\mu$$

This optimal scheme says that one should continue searching until the number of prey remaining in the area is no more than $c/\mu$, the expected cost of a capture in an area inhabited by a single prey.

**Proof:** Let **S*** be the subclass of stopping schemes which permit stopping at either zero or any subsequent time. For such rules:

First we decide whether to search at all: If not

$$E[payoff] = 0$$

If we do start seaching:

$$E[payoff] = V(N) = \max_{s>0: s\in S^*} E[x_s]$$

where we assume from dynamic programming that the expected payoff is some function $V$ dependent upon the number of objects $N$.

Therefore we should try our luck if $V(N) > 0$.

Suppose we decide to search, then we have to decide whether to continue after we make our first capture.

$u_1$ = *time of the first capture.*

If we stop after our first capture, then: $E[payoff] = 1 - cu_1$

If we continue, then: $E[payoff] = V(N-1) + (1 - cu_1)$

This is due to the lack of memory property of the exponential distribution. If we continue, our prospects for the future are the same as we started , with two important differences:

(i) there are now $(N$-1) objects left

(ii) we have a profit of $1 - cu_1$

where the max is over the class $s \in S^* : s > u_1$.

Therefore we should continue beyond the first capture if:

$$V(N-1) + (1 - cu_1) > xu_1$$

or

$$V(N-1) > 0$$

Preceeding inductively, it is easily shown that having caught $k$ prey we should continue if $V(N-k) > 0$ or we should stop if $V(N_k) \leq 0$. Since the supremum sets decrease

$$V(N) \geq V(N-1) \geq \ldots \geq V(1) \geq V(0) = -\infty$$

the optimal stopping rule is defined by

$\sigma$ = *first time* $t \geq 0 : V(N-k_t) \leq 0$.

Since $V(N-k_t)$ is increasing and depends only on the number $k_t$ of prey caught up to time $t$ then we need only consider stopping rules of the form

$\sigma_k$ = *first time* $t \geq 0$ *such that* $k_t = k$.

We want to determine the best $\sigma_k$, that is, for what $k$ is the payoff maximum?

For each $k$ the expected payoff is $V_k(N) = k - cE[u_k]$, where $u_k$ is the random variable denoting the waiting time until the *k-th* event of a Poisson process. Thus it has a gamma distribution and we have

$$E[u_k] = \frac{1}{\mu}\sum_{j=0}^{k-1}\frac{1}{N-j} \quad and$$

$$V_k(N) = k - \frac{c}{\mu}\sum_{j=0}^{k-1}\frac{1}{N-j} \quad where \quad V_0(N)=0.$$

The maximizing $k$ is given by: "$k$ is the smallest integer not less than $N - c/\mu$". This completes the proof.

Suppose $N$ is unknown, then a logical approach to finding an optimal stopping time is to replace $N$ in Theorem 1 by an estimate, say $\tilde{N}$, and use the stopping scheme

$$\sigma = first\ time\ \tilde{N} - k_t \leq c/\mu$$

The following unbiased estimate of $N$ is used,

$$\tilde{N} = k_t / (1 - e^{-\mu t}).$$

Why is $\tilde{N}$ an unbiased estimate of $N$? Because, $k_t$ is a Binomial random variable with parameters $N$ and $p = 1 - e^{-\mu t}$. That is $k_t$ is the number of successes in $N$ pass/fail tests with $p$ the probability of pass. Indeed, $k_t$ by definition is the number of objects caught by time $t$, that is it is the number of $z_j \leq t$. Let

$$I_j = \begin{cases} 1 & if\ \ z_j \leq t \\ 0 & if\ \ z_j < t. \end{cases}$$

The $I_j$ are indicators of the pass/fail events. Then,

$$k_t = \sum_{j=1}^{N} I_j$$

Therefore taking the expectation,

$$E[k_t] = E[\sum_{j=1}^{N} I_j] = \sum_{j=1}^{N} E[I_j] = N\ Pr[z_j \leq t] = 1 - e^{-\mu t}$$

Thus, $E[k_t/(1 - e^{-\mu t})] = N$. Similarly, the other parameters can be estimated from the data and substituted into the optimal time.

## 7.2 Dalal and Mallows' Generalization

The work by Starr assumes that the length of time needed to find the error labeled $i$ is independent of all other discovery times and that it follows an exponential distribution. Also, it requires that the number of typographic errors is fixed and known, that new errors are not introduced during correction, that errors are equally likely to be caught on each examination of the text, that errors do not occur in bunches, and that the costs of testing, correction, and discovery are all known precisely. These conditions enable an elegant mathematical solution, but are unrealistically idealized. To remedy these idealizations, Dalal and Mallows sacrificed considerable mathematical tractability, but (asymptotically) found results similar to Starr's optimal stopping rule. In particular, their generalizations include:

- the density of bugs is allowed to vary across modules, rather than having a fixed intensity.
- the number of bugs in module $i$ is random (Poisson, with parameter $\lambda_i$), where the unknown $\lambda_i$ has a gamma distribution (this is further weakened later, when an asymptotic argument requires only two finite moments on the distributions for the number of bugs).
- the time needed to discover a bug is random with arbitrary known distribution $G$ (later, an asymptotic argument allows $G$ to belong to a family of distributions, indexed by a finite-dimensional parameter that is learnt adaptively).
- there can be different categories of bugs, having different costs and different distributions for the time-to-discovery.
- bugs can occur in clusters.

The technical details involved in this extension are described in Dalal and Mallows [Dala88]. The three basic mathematical tools that are used are Bayesian inference, dynamic programming, and local asymptotic minimaxity. Overall, Dalal and Mallows' mathematics is unusual in the degree of concern it takes for allowing realistic generality. To some extent, every practical concern that arises in a critique of Starr's method has been addressed. But so far, it is unclear whether the extensions, and their asymptotic formulation, can lead to useful economies in practice.

## 7.3 Efficacy and Enhancement

As it stands, the full generality of the optimal stopping approach is dauntingly complex to implement. Also, it is uncertain whether the asymptotic optimality is usefully successful in finite-horizon applications. Thus one needs to find simplifications of the stopping rules, and to undertake a simulation study to evaluate the comparative performance of different rules.

The kinds of simplification we propose would be to assign order-of-magnitude costs to delays in software release and the failures to discover bugs of specific categories. Also, we would assume a convenient form for the time-to-discovery distribution $G$, and use this to build an approximately optimal stopping rule according to the blueprint developed by Dalal and Mallows. Obviously, this rule will be suboptimal, because of the simplifying assumptions; however, it would be more simply prescriptive, and might achieve nearly the same level of cost reduction.

Further, we propose to undertake a simulation study that examines the performance of the simplified rules, the optimal rules, and current conformance testing practice. These would be compared across a range of realistic scenarios, enabling better quantification of the advantages and disadvantages among the different testing protocols.

## 8 Conclusions

This paper addresses alternatives for exhaustive software testing based on statistical methods, including multivariable analysis, design of experiments, coverage designs, usage models, and optimization techniques. It provides quantitative measures of quality and reliability, as well as statistical measures and confidence levels that a program implements its functional specification correctly. The overall goal is to ensure software quality and to develop methods for software conformance testing based on relevant statistical techniques.

The above sections provide a preliminary analysis of the potential application of each of several statistical methods to black-box conformance testing. In traditional black-box conformance testing, one has a list of requirements which must be exhaustively and thoroughly tested by *falsification methods*, i.e. looking for test failures. The statistical methods described herein offer measurement tools that will bring a new quality and quantitative measure of confidence to these traditional methods. In addition, some of these statistical methods are quite recent and may lead to significant improvements in the way one addresses software testing, possibly leading to new approaches and techniques. We will choose one or more of these statistical methods as a basis for formalization of quality and confidence levels in traditional methods of black-box testing, as well as a foundation for experimenting with new or alternative approaches involving experimental prototypes.

The binomial models described in Section 3 above are the basis of the classical application of statistical methods to software testing. These models offer tools that will bring about modest decreases in the number of tests needed to reach dependable quality and confidence measurements in traditional software testing. They will be especially helpful in allowing testers to make explicit probability statements about the degree of uncertainty in their evaluations of commercial implementations of formal software standards and other widely-adopted functional specifications.

The coverage designs discussed in Section 4 above have the potential to drastically reduce the number of runs needed to make a conformance test of a program. They offer the possibility of developing significant new contributions to category-partition testing methods and to the formal testing of class definitions in object-oriented software. They are particularly appropriate for the testing of combinations in component testing of software modules. The difficulty in applying them is that it is not obvious how one identifies a module, especially in the context of black-box testing, and it is unclear whether the modules can assume only a small number of states. A further, and more practical issue, is that it can be hard to develop conformance tests that exercise together exactly the modules specified in the design. We intend to address all of these problems in more focused follow-on efforts with coverage designs. Nonetheless, Cohen, *et al*, report success with this method, and the software testing community feels the technique deserves attention. Our hope is that one can generally determine modular structure in a program from functional requirements, and that the model of a small number of states is sufficient to describe some (or most) of the operational failure modes.

The reasons for initially pursuing mutation testing for its potential application to black-box conformance testing are given in Section 5 above. All but one of these reasons lead to dead-ends as far as black-box testing is concerned. The one idea worth pursuing is the potential for extending mutation analysis from a given implementation of a functional specification to a whole class of implementations of that specification. Given a functional specification and a reference implementation, we may be able to use the test suite generated by mutation analysis for the reference implementation as a general test suite for all commercial applications claiming conformance to the same functional specification. The approach would then allow testers to establish pre-specified probabilities for Type I and Type II errors related to accepting the commercial product as a correct implementation of the functional specification.

The application of usage models to black box conformance testing is discussed in Section 6 above. Such models can yield confidence statements about the operational reliability of the software based on its test performance. Furthermore, the statistical analysis used will provide a means of estimating the size of the test suites needed to provide a desired level of operational reliability.

As a final topic, we address the question of "When to stop testing?" in Section 7 above. As it stands, the full generality of the optimal stopping approach is dauntingly complex to implement, thus one needs to find simplifications of the stopping rules, and to undertake a simulation study to evaluate the comparative performance of different rules. We propose to address such simplifications and the specification of appropriate stopping rules for various software testing methods in our more focused follow-on efforts.

In the next phase of our research on software testing by statistical methods, we will formalize the applicability of statistical confidence intervals to all methods of software testing. In addition, we will pursue coverage designs and their evolutionary spin-offs as a basis for developing significant new techniques to testing of object classes and component modules. We also hope to make significant progress on specification and design of an experimental prototype for testing some of our preliminary theoretical conclusions.

## - Bibliography -

| | |
|---|---|
| [Afif92] | F.H. Afifi, L.J. White, and S.J. Zeil. Testing for linear errors in non-linear computer programs, *Proceedings of 14th International Conference on Software Engineering*, pp 81-91, 1992. |
| [Amma95] | Paul Ammann and Jeff Offutt, *Using Formal Methods to Derive Test Frames in Category-Partition Testing*, George Mason Univ ISEE technical report, 1995. |
| [Beiz95] | Boris Beizer. *Black-Box Testing Techniques for Functional Testing of Software and Systems*, John Wiley, New York, 1995. |
| [Blac97] | M.R. Blackburn, R.D. Busser, and J.S. Fontaine. Automatic Generation of Test Vectors for SCR-Style Specifications, Software Productivity Consortium, *IEEE Publication*, July 1997. |
| [Booc91] | G. Booch, *Object Oriented Design with Applications*, Benjamin Cummings, 1990. |
| [Bosi91] | B. Bosik and M. Uyar. Finite State Machine Based Formal Methods in Protocol Conformance Testing: from Theory to Implementation, *Computer Networks and ISDN Systems*, 22:7-33, 1991. |
| [Brow75] | J.R. Brown and M. Lipow. Testing for software reliability, *Proceedings of the International Conference on Reliable Software*, p518-527, Los Angeles, CA, April 1975. |
| [Budd77] | T. Budd and F. Sayward. *User's guide to the Pilot mutation system*, TR-114, Dept of Computer Science, Yale Univ, 1977. |
| [Budd80] | T. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs, *Proceedings of the ACM Principles of Programming Languages Symposium*, pp 220-233, ACM, 1980. |
| [Cohe96] | D. Cohen, S. Dalal, J. Parelius, and G. Patton. The Combinatorial Approach to Automatic Test Generation, *IEEE Software*, 13(5):83-87, September 1996. |
| [Dala88] | Dalal, S.R. and Mallows, C.L. When Should One Stop Testing Software? *Journal of the American Statistical Association*, 83: 872-879. |
| [Dala97] | S.R. Dalal and C.L. Mallows. *Covering Designs for Testing Software*, Submitted for Pub. |
| [DeMi88] | R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. " An extended overview of the Mothra software testing environment, *Proceedings of 2nd Workshop on Software Testing, Verification, and Analysis*, pp 142-151, July 1988. |
| [Fran88] | P.G. Frankl and E.J. Weyuker. An Applicable Family of Data Flow Testing Criteria, *IEEE Transactions on Software Engineering*, 14(10):1483-1498, October 1988. |
| [Fran93] | P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods, *IEEE Transactions on Software Engineering*, 19(3):202-213, March 1993. |
| [Gill88] | R.D. Gill, Y. Vardi, and J.A. Wellner. Large Sample Theory of Empirical Distributions in Biased Sampling Models, *The Annals of Statistics*, 16:1069-1112, 1988. |
| [Heit96] | C.L. Heitmeyer, R.J. Jeffords, and B.G. Labaw. Automated Consistency Checking of Requirements Specifications, *ACM Transactions on Software Engineering and Methodology*, 5(3):231-261, July 1996. |

[Hong96] H.S. Hong, Y.R. Kwon, and S.D. Cha. *Testing of Object-Oriented Programs Based on Finite State Machines*, Korea Advanced Institute of Science and Technology, Dept of Computer Science Technical Report, 1996.

[Jeng94] B.C. Jeng and E.J. Weyuker. A Simplified domain-Testing Strategy, *ACM Transactions on Software Engineering and Methodology* 3, pp 254-270, July 1994.

[Ling93] Richard C. Linger. Cleanroom Process Model, *Proceedings of the 15th International Conference on Software Engineering*, pp 2-13, 1993.

[Litt87] B. Littlewood, Editor. How Good are Software Reliability Predictions, *Software Reliability: Achievement and Assessment*, 1987.

[Mall96] Colin L. Mallows. *Covering Designs in Random Environments*, Submitted for Pub.

[Math94] A.P. Mathur and W.E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria, *The Journal of Software Testing, Analysis, and Verification*, 4(1):9-31, March 1994.

[Meal55] G.H. Mealy. A Method for Synthesizing Sequential Circuits, *Bell System Technical Journal*, 34:1045-1079, 1955.

[Moor56] E.F. Moore. Gedanken Experiments on Sequential Machines, In *Automata Studies - Annals of Mathematical Studies #34*, Princeton Univ Press, 1956.

[Nels73] E.C. Nelson, *A Statistical Basis for Software Reliability Assessment*, TRW Software Series, SS-73-03, March 1973.

[Offu95] A.J. Offut, J. Pan, K. Tewary, and T. Zhang, *An Experimental Evaluation of Data Flow and Mutation Testing*, George Mason Univ ISSE technical report, 1995.

[Ostr88] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests, *Communications of the ACM*, 31(6):676-686, June 1988.

[Parn90] A. Parnas, J. Schouwen, and S.P. Kwan. Evaluation of Safety-Critical Software, *Communications of the. ACM*, 33:636-648, 1990.

[Poor97] J.H. Poore, G.H. Walton, and J.A. Whittaker, *A Mathematical Programming Approach to the Representation and Optimization of Software Usage Models*, Univ of Tennessee Dept of Computer Science technical report, In Review 1997.

[Reny71] A. Renyi. *Foundations of Probability*, Wiley, NY, 1971.

[Rumb91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design*, Prentice Hall, 1991.

[Sahi90] Mehmet Sahinoglu and Eugene Spafford. *Sequential Statistical Procedures for Approving Test Sets Using Mutation-Based Software Testing*, Purdue Univ, SERC TR-79-P, May 1990.

[Sahi92] Mehmet Sahinoglu, Eugene Spafford, and William Hsu. *An Experimental Approach to Statistical Mutation-Based Testing*, Purdue Univ, SERC TR-63-P, August 1992.

[Sank94] Sriram Sankar and Roger Hayes, *Specifying and Testing Software Components using ADL*, Sun Microsystems technical report, SMLI TR-94-23, April 1994.

[Sloa93] N.J.A. Sloane. Covering Arrays and Intersecting Codes, *Journal of Combinatorial Designs*, 1:51-63, January 1993.

[Star74] N. Starr. Optimal and Adaptive Stopping Based on Capture Times. *Journal of Applied Probability*, 11: 294-301, November 1974.

[Weis88] Stewart Weiss and Elaine Weyuker. An extended Domain-Based Model of Software Reliability, *IEEE Transactions on Software Engineering*, 14(10):1512-1524, October 1988.

[Weyu90] Elaine J. Weyuker. The Cost of Data Flow Testing - An Empirical Study, *IEEE Transactions on Software Engineering* 16, February 1990.

[Weyu91] Elaine Weyuker and Bingchiang Jeng. Analyzing Partition Testing Strategies, *IEEE Transactions on Software Engineering*, 17(7):703-711, July 1991.

[Whit87] Lee J. White, Software Testing and Verification, *Advances in Computers*, volume 26, pp 335-391, Academic Press, New York, 1987.

[Whit94] J.A. Whitaker and M.G. Thomason. A Markov Chain Model for Statistical Software Testing, *IEEE Software*, October 1994.

[Zeil89] S.J. Zeil, Perturbation Techniques for Detecting Domain Errors, *IEEE Transactions on Software Engineering*, 15(8):737-746, June 1989.