

University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, &
Professional Papers

Graduate School

2001

Model-based path testing

Jie Zhang

The University of Montana

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

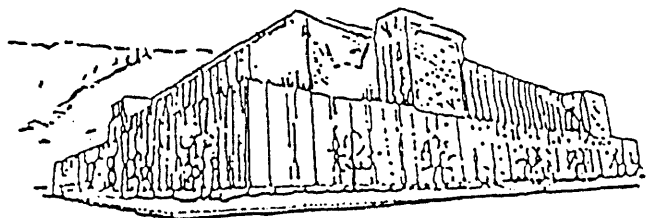
Let us know how access to this document benefits you.

Recommended Citation

Zhang, Jie, "Model-based path testing" (2001). *Graduate Student Theses, Dissertations, & Professional Papers*. 5128.

<https://scholarworks.umt.edu/etd/5128>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.



Maureen and Mike
MANSFIELD LIBRARY

The University of **MONTANA**

Permission is granted by the author to reproduce this material in its entirety,
provided that this material is used for scholarly purposes and is properly cited in
published works and reports.

*** Please check "Yes" or "No" and provide signature ***

Yes, I grant permission X
No, I do not grant permission

Author's Signature Zhang Jie

Date Jan 22, 2002

Any copying for commercial purposes or financial gain may be undertaken only with
the author's explicit consent.

MODEL-BASED PATH TESTING

by

Jie Zhang

M.S. The University of Montana, 1999

A professional paper
presented in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

The University of Montana

November 2001

Approved by:



Chairperson



Dean, Graduate School

1-29-02

Date

UMI Number: EP40592

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.

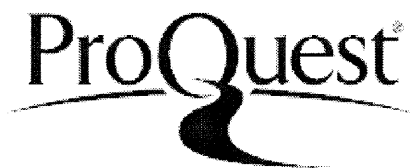


UMI EP40592

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



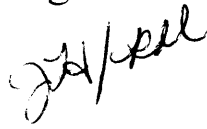
ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Jie Zhang,

Computer Science

Model-based Path Testing

Advisor: Joel Henry



This paper presents some algorithms for model-based path testing. At first, finite automata modeling technique is used to build state and transition diagram and table. Then, Maximum-path Arithmetic and Minimum-path Arithmetic are introduced for analyzing the number of path of the model. Finally, three testing approaches are discussed: a string-matching algorithm for the interested path testing; a random-walk algorithm for general path testing; and coverage testing algorithm for locating path failure.

Contents

1 Problem	1
2 Introduction	2
2.1 Traditional Software Testing.....	2
2.2 Model-Based Testing	3
2.3 Phases of Software Testing.....	4
2.3.1 Modeling Program Behavior.....	5
2.3.2 Selecting Tests	5
2.3.3 Running & Evaluating Tests.....	6
2.3.4 Measuring Test Progress.....	7
2.4 Our Work (Path Testing) and Terminology	7
3 Modeling	10
3.1 Background and Terminology	10
3.2 Model Building.....	12
3.3 Example.....	15
4 Path and Coverage Testing	19
4.1 Maximum-Path Arithmetic.....	20
4.2 Minimum-Path Arithmetic.....	29
4.3 Coverage Testing	33
4.3.1 Transition Coverage Level.....	34
4.3.2 Full Predicate Coverage Level.....	35
4.3.3 Transition-Pair Coverage Level.....	36
4.3.4 Complete Sequence Level.....	36
5 Path testing approaches and algorithms	37
5.1 Testing the Most Likely Paths	37
5.1.1 String-matching.....	39
5.1.2 Finite Automata	42
5.1.3 String-matching automata algorithm	44
5.2 Random Walk with Traversal Markers and Algorithm.....	49
5.3 Full Predicate Coverage Testing and Algorithm.....	51
5.3.1 Case Study	52
5.3.2 Full predicate coverage criterion	55
5.3.3 Test specifications.....	56
5.3.4 Get Prefix Algorithm	58
5.3.5 Generate Full-Predicate Coverage Test Cases Algorithm	60
6 Conclusion	62
REFERENCES	62

1 Problem

MATT is an application that provides enhanced test generation capability for users of MATLAB. The ability to rapidly create custom test data for running model simulations is an important time saver that frees the pains of developing a variety of test data, needed for testing and model simulation.

MATT uses information it obtains from MATLAB to create a set of data that describes the inputs for a specific model superblock. With a series of point and click selections, users may set the types of tests for data they desire for each input and adjust parameters for accuracy, constant, minimum and maximum values. Once adjustments are complete, these settings may be saved in a MATT file format known as a test Script. Test Scripts may be recalled and used again for later test generation. Once each input has been set up for a particular test type, the user may then generate the test matrix. The test matrix output may then be returned to MATLAB for simulation or it may be saved and used at a later time.

MATT does not have the capability to direct a path, or to trace paths and their coverage on running model simulations. A path is a running routine of a model that enters from a starting state, via many middle states and transitions, and ends up with final state. MATT will be highly enhanced if we can add a feature to direct a path, to trace paths and their coverage of the model, which will help the builder of the model to get testing information about model functions.

This research project is focusing on finding a testing technique and algorithm on path directing and tracing. We start the work with an introduction to software testing.

2 Introduction

2.1 Traditional Software Testing

Software testing includes executing a program on a set of test cases and comparing the actual results with the expected results. Testing and test design, as parts of quality assurance, should also focus on fault prevention. To the extent that testing and test design do not prevent faults, they should be able to discover symptoms caused by faults. Finally, tests should provide clear diagnoses so that faults can be easily corrected.

Software is tested from two different perspectives, the white box approach and the black box approach. White box strategies for testing are driven by the internal control structure of the program. There are several types of structural testing, including branch testing, control flow testing, data flow testing, slicing, and program dependency.

In the black box approach to software testing, we are interested in the inputs and outputs of the system in addition to an understanding of its behavior or functional properties that are extracted almost exclusively from the requirements. The construction of tests depends on looking at these properties while totally ignoring the structure of the implementation. Exhaustive black box testing is running the program with all possible input combinations. It can be easily seen that such a task is impossible (Whittaker, 1997; Myers, 1979). Myers

concludes that, due to the impossibility of performing exhaustive black box testing, the approach cannot be used to show the program error-free. Further, the amount of testing to be done (or selecting test data out of the infinite possibilities) becomes a major problem as it is an issue of computational and man-hour cost.

2.2 Model-Based Testing

Traditional software testing consists of the tester studying the software system and then writing and executing individual test scenarios that exercise the system. These scenarios are individually crafted and then can be executed either manually or by some form of capture/playback test tool.

This method of creating and running tests faces at least two large challenges (Robinson, 1999):

First, these traditional tests will suffer badly from the “pesticide paradox” (Beizer, 1990) in which tests become less and less useful at catching bugs, because the bugs they were intended to catch have been caught and fixed.

Second, handcrafted test scenarios are static and difficult to change, but the software under test is dynamically evolving as functions are added and changed. When new features change the appearance and behavior of the existing software, the tests must be modified to fit. If it is difficult to update the tests, it will be hard to justify the test maintenance costs.

Model-based testing alleviates these challenges by generating tests from explicit descriptions of the application. It is easier, therefore, to generate and maintain useful, flexible tests.

In recent years, there has been a growing movement in software testing to use the information contained in explicit models of software behavior to make it simpler and cheaper to do testing. (Beizer, 1995; Apfelbaum, 1997)

Model-based testing is a black-box technique that offers many advantages over traditional testing (Robinson, 1999):

- Constructing the behavioral models can begin early in the development cycle.
- Modeling exposes ambiguities in the specification and design of the software.
- The model embodies behavioral information that can be re-used in future testing, even when the specifications change.

The model is easier to update than a suite of individual tests.

2.3 Phases of Software Testing

Generally, regardless of the paradigm adopted, testing involves four phases: behavior modeling, test generation, test execution and evaluation, and measuring test progress (Whittaker, 1997, 1999).

2.3.1 Modeling Program Behavior

The task in modeling program behavior is to document all communication among the system and its users. This involves enumerating all the inputs and outputs for every user and constructing a representation of the understanding of the possible input sequences (tests): the ones the users can produce and the ones the system expects by specification. Finally, interaction among users that may have a consequential effect on the system needs to be documented. Based on this information, a model of how the software operates is constructed. The modeling products include:

- A document enumerating all the elements of software-user interaction
- A model of software behavior, based on which tests are generated. Examples of such a model include control and data flow graphs in structural testing and finite state machines in black box testing.

Modeling is the most fundamental phase of any testing process, since the rest of the phases depend on the accuracy of its artifacts.

2.3.2 Selecting Tests

This phase creates:

- A document describing each of the test adequacy criteria
- An algorithm that, based on the model constructed in the earlier phase, builds a test that meets the adequacy criteria

Selecting tests is not straightforward. Most of the work in testing has addressed test selection with various objectives in mind, such as revealing bugs, covering code, etc..

2.3.3 Running & Evaluating Tests

Running a test involves figuring out how to simulate user action so that the software “thinks” that it is in its intended environment. The task of input simulation is becoming increasingly easier. There are numerous tools that are dedicated to simulating software input. Writing code for the simulations is another feasible option, when tools are not available.

Evaluating a test involves verifying the test result against some sort of specification. Howden (Howden, 1978) states that every form of testing requires or assumes the existence of an oracle. An oracle is an independent entity that determines whether a result observed in the software after a test has been run meets expectations (i.e., whether the correct outputs have been produced; or, whether the correct control sequences have been followed). Developing an oracle is nontrivial and is often as complex as the application under test itself. Many times, in practice, the oracle is an experienced test engineer or developer upon whose expertise the decision of whether a test has been successful is based. So this phase produces:

- An input simulator that automatically executes tests
- An oracle

2.3.4 Measuring Test Progress

Generally, there are two classes of measures that testers and project managers are interested in: stopping criteria and field quality metrics. Stopping criteria describe the conditions under which it is determined that enough tests have been generated.

Field quality metrics are figures of estimation for how well the software will perform when it is released in its intended environment. For example, some of these metrics estimate how much more testing needs to be done, the time to release, the mean time between failure, the mean time to the next failure, and reliability. This phase creates:

- A document describing stopping criteria
- A document describing the field quality metrics
- The actual metrics, which are computed based on collected data (previous test runs).

2.4 Our Work (Path Testing) and Terminology

Our research is focusing on path testing. The goal is to get the information about which states and transitions in a model are covered in a path testing case. So our work focuses on the first two phases, modeling phase and selecting tests phase. And, our work will use both techniques for traditional testing and model-based testing. First we create a finite state machine, then study the criteria and algorithm to test a path within the machine. The

test will reveal which state and transition have been passed and what is covered in the machine.

Following are some very important definitions.

Path testing based on the program's control flow as a structural model is the cornerstone of testing. Methods include how to generate tests from the program's control flow, criteria for selecting paths, and how to determine path-forcing input values.

A **flowchart** is a graphical representation of a program's control structure. The programmer's original flowchart is a statement of intentions and not a program.

A **process** has one entry and one exit. It performs an operation on data. A process can consist of a single statement or instruction, a sequence of statements or instructions, a single-entry/single-exit subroutine, a macro or function call, or a combination of these. The program does not jump into or out of process. From the point of view of test cases designed from flowcharts, the details do not affect the control flow. A sequence of processing statements that is uninterrupted by **junctions** or **decisions** is usually put into one proceed block. If the processing affects the flow of control, that effect will be manifested at a subsequent decision or case statement.

A **decision** is a program point at which the control flow diverges. While most decisions are two-way or binary, some are a three-way branch in control flow. A case statement is a multi-way branch or decision.

A **junction** is a point in the program where the control flow merges.

A **path** through a program is a sequence of instructions or statement that starts at a junction or decision and ends at another, or possibly the same, junction or decision. A path may go through several junctions, processes, or decision, one or more time. The word “**node**” is used to mean either junction, decision or both. Paths consist of **segments**. The smallest segment is a single process that lies between two nodes, e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision. The collective term for flowchart lines that join nodes is “**link**”. A flowchart then, consists of nodes and links. A path segment is a succession of consecutive links that belongs to some path. The word “path” is also used in the more restrictive sense of a path that starts at the routine’s entrance and ends at its exit.

The term “**complete cover**”, or “**cover**” alone is used to mean that a set of tests has the potential for executing every instruction and taking all branches in all directions.

Complete coverage is a minimum mandatory testing requirement.

A **transaction** is a unit of work seen from a system user’s point of view. A transaction consists of a set of operations, some of which are performed by a system, persons, or

devices that are outside of the system. A transaction typically consists of a set of operations that begins with an input and ends with one or more outputs. At the conclusion of the transaction's processing, the transaction is no longer in the system, except perhaps in the form of historical records.

3 Modeling

Modeling is a way of representing the behavior of a system. Models are simpler than the system they describe, and they help us understand and predict the system's behavior.

A common type of model in computing is the state graph, or finite state machine. State graphs are a useful way to think about software behavior and testing (Beizer, 1995). The application begins in some state (such as "main window displayed"), the user applies an input ("invoke help dialog") and the software moves into a new state ("help dialog displayed").

3.1 Background and Terminology

Software systems are installed into environments where they are stimulated by users via inputs and where they produce outputs to be consumed by users. A software user is an element of its environment that is either responsible for generating system input or expected to consume system output. Test engineers must document communication between the software and its users occurring via inputs and outputs.

An **input** is a user-generated event recognizable by the software. An **output** is an event generated by the software directed to one or more of its users.

An input is said to be **applicable** at an identifiable point of software execution (also referred to as 'time' throughout this document) if and only if the user responsible for generating the input is capable of generating it (in such a case the input is said to be **available** to the user) and the system recognizes it as an allowable stimulus. Applicability of an input is not necessarily equivalent to its legality from the point of view of its functional specification; an applicable input is one that gets processed by the system.

An **applicable input string** is a sequence of inputs such that every input in the string is applicable after all preceding inputs in the sequence have been processed by the system.

An input is said to be **unreachable** at an identifiable point of software execution if and only if it is not applicable. Unreachable inputs are stimuli that cannot affect the system due to the unavailability of the required interface (at that particular point of execution) or that get ignored by interface components. In other words, the system under test never processes unreachable inputs, by specification.

The **functional behavior** of a software system at a particular point of execution is the manner in which it responds to inputs in it (whether it recognizes an input, ignores it, or processes it; and in the latter case, whether the response is observed as an output or goes unnoticed by the user as internal computation). This depends on the string of inputs that

has been processed by the system starting with the last invocation of the system up to the time in question.

Behavior models are discrete structures that describe every possible functional behavior and the manner in which software transitions from one behavior to another. In the context of black-box testing, finite state machines are an example representation of behavior models.

A **state** of a software system represents one and only one functional behavior of the system. The **state space** represents every possible functional behavior of the system. Therefore, a combination of values of all functionally significant data elements is a sufficient description of a state. It follows from the definition of operational modes that a state is a tuple of instantiations for all modes.

Assuming a finite-state-machine-like representation, to build a behavior model is to enumerate the states and define the state transitions of the model.

3.2 Model Building

We usually create state transition diagrams and state transition tables to describe the model.

A **state transition diagram** is a graphic representation of a state machine. State transition diagrams emphasize the logical behavior of a system. Traditionally, state

transition diagrams have been used to explain how a system with a finite set of modes, or states, can change from one mode or state to another. In Figure 1 rectangles with rounded corners represent the states in a system. The directed lines from one state to another are called transitions. These indicate the ability to change from one state to another.

Transitions are usually labeled with the conditions that must be satisfied before the transition can be taken. Several transitions can originate or terminate on the same state.

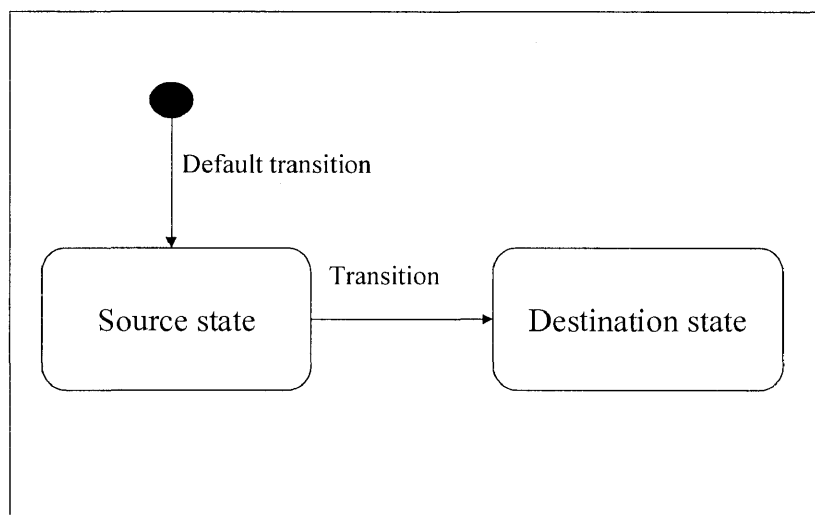


Figure 1: the basic elements of a state transition diagram

State transition diagrams are useful for visualizing logical paths through a series of states. A state transition diagram can help to clarify the exact sequence of logic that is needed to change from one state to another, particularly when each state has a small number of

transitions that originate or terminate on it. Actions associated with states and transitions enable the state diagram to interact with its external environment.

As a design tool, classic state transition diagrams are limited by scalability problems. Extended state transition diagrams, like those supported in Stateflow, overcome these limitations with constructs that handle hierarchy, parallelism, and transition re-use. Hierarchy allows states to be grouped together into a superstate so that common transitions only need to be drawn once. Parallelism allows the diagram to be partitioned into several parallel states, each with its own hierarchy of active substate(s). Parallelism prevents the state explosion that results when independent modes or attributes have numerous possible combinations.

State transition diagrams are useful for models with a relatively small number of states. Drawing and using a large state transition diagram is difficult and error-prone, even with good CAD or CASE tools and with extended state transition ideas. Usually models with 20 or more states are graphically intractable. For large models with hundreds of states, automated support is necessary. **State transition tables** provide a compact representation and ease systematic examination and use of the model.

State machines may be represented in one of several tabular formats. In the **state-to-state** format, rows represent accepting states and columns represent result states, cells represent the input/event triggering the transition. In the **state-to-event** format, rows

represent accepting states and columns represent the input/event, cells represent the result states.

3.3 Example

Consider a hypothetical design of a cruise control system (Aldrich). The inputs and outputs to the controller are shown in Figure 2. The controller uses sensor input for the brake pedal, accelerator pedal and vehicle speed. User input is generated from a Power switch, and Set, Resume, Increment, and Decrement buttons. The controller produces a throttle command used as a set point to the mechanical system that controls the throttle plate.

The target speed for the controller also serves as an output for verification even though it is not required by the other system components.

<p>Inputs:</p> <ol style="list-style-type: none">1. Vehicle speed2. Brake pedal switch3. Accelerator pedal position4. Power button5. Set button6. Resume button7. Increment button8. Decrement button <p>Outputs:</p> <ol style="list-style-type: none">1. Throttle plate command2. Target speed (for verification)

Figure 2: The cruise control input and outputs

The controller has the ability to adjust the target speed with an increment and decrement button. A list of functional requirements for the cruise control is shown in Table 1.

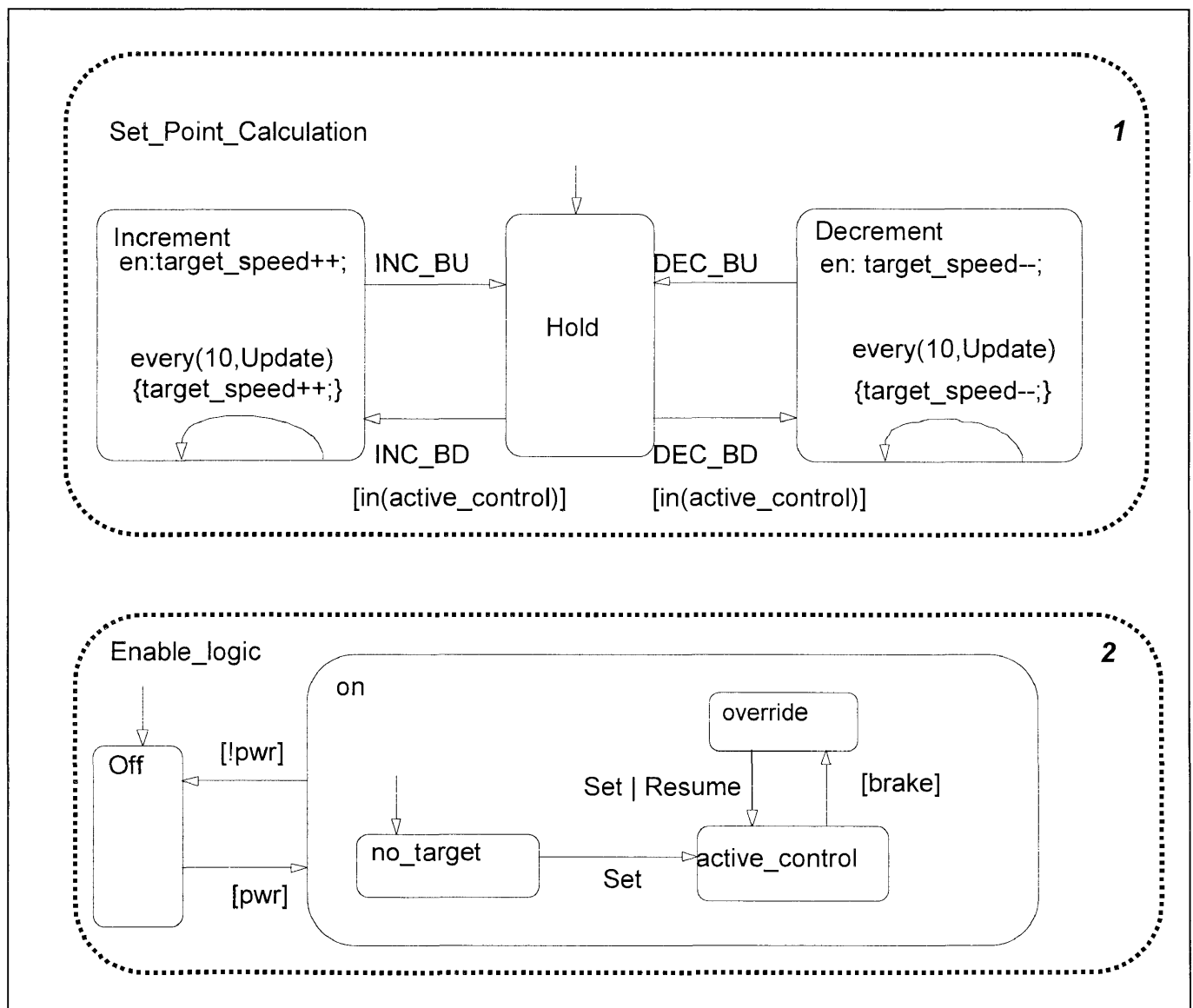
Table 1: A list of functional requirements for the cruise control

- | |
|--|
| <ol style="list-style-type: none">1. When the cruise control is powered on it shall enter an idle mode until a target speed is established that enables active control.2. When the Set button is depressed while the cruise control is on it shall set the target speed to the current vehicle speed.3. When the Resume button is depressed it shall set the target speed to the last value set by the vehicle speed since the control was powered on.4. Pressing and releasing the Inc button in less than 1 second when the control is active shall cause the target speed to increase by 1 M.P.H5. Holding the Inc button depressed when the control is active shall cause the target speed to increase by 1 M.P.H. every second.6. Pressing and releasing the Dec button in less than 1 second when the control is active shall cause the target speed to decrease by 1 M.P.H7. Holding the Dec button depressed when the control is active shall cause the target speed to decrease by 1 M.P.H. every second.8. When the cruise control is not actively controlling speed, the throttle position shall be set to the same value as the accelerator pedal.9. When the brake pedal is greater than zero and the cruise control is active the cruise control shall enter the override mode.10. When the controller is in the override mode and the Set or Resume button is depressed the controller shall return to active control. |
|--|

A portion of an extended state transition diagram for a cruise control application is shown

in Figure 3. Hierarchy allows the states that represent the powered-on modes of the controller to be grouped together in a natural manner. Diagram 2 (of **Figure 3**) is a state transition diagram showing the logic for a cruise control. When power is enabled, i.e., the condition `[pwr]` is logically true, the active state changes from `Off` to the `no_target` substate of `Active` (`Active.no_target`). When the `Set` event occurs, the mode changes to `Active.control_enabled`.

Figure 3: Cruise Control Top Level State Transition Diagram



We simplify figure3 into figure 4. Table 2 is the state transition table of the cruise control model.

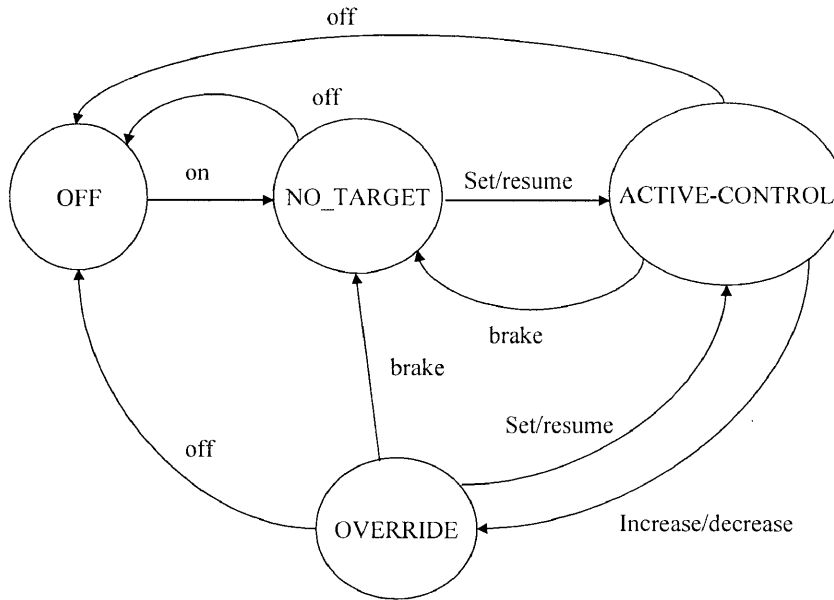


Figure 4: Cruise Control Model State Transition Diagram

Table 2 Cruise Control Model State Transition Table

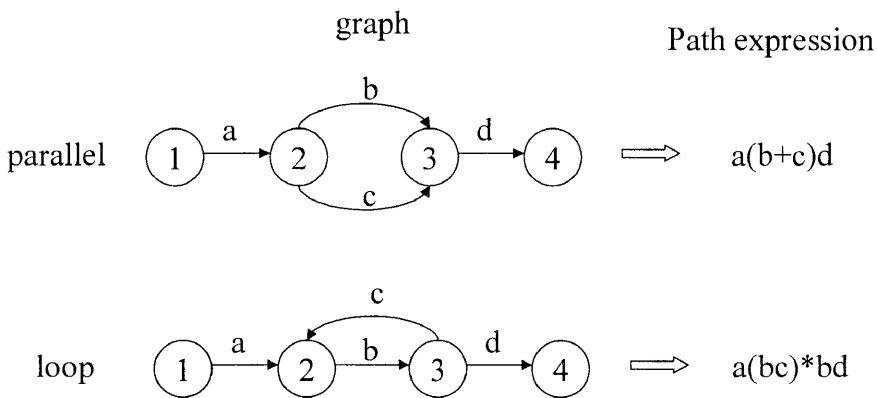
	OFF	NO_TARGET	ACTIVE_CONTROL	OVERIDE
OFF		on		
NO_TARGET	off		set/resume	
ACTIVE_CONTROL	off	brake		increase/decrease
OVERIDE	off	brake	set/resume	

4 Path and Coverage Testing

After the model is built, we trace paths through it to find a set of covering paths, a set of values that will sensitize paths, what logic function controls the flow from one state to another, or if a state is reachable or not. But before we do these, we should know what is the maximum and minimum number of paths in the model. Maximum number gives you an idea how many test cases should be generated and when you should stop. The minimum number gives you a way of efficiency to test model only once without missing a single state.

At first a review of some basic concepts. Path expressions are introduced as algebraic representations of sets of paths in a graph. With suitable arithmetic laws and weights, path expressions are converted into algebraic functions or regular expressions that can be used to examine structural properties of graphs or flowcharts.

Two basic conversions are presented as follow:



4.1 Maximum-Path Arithmetic

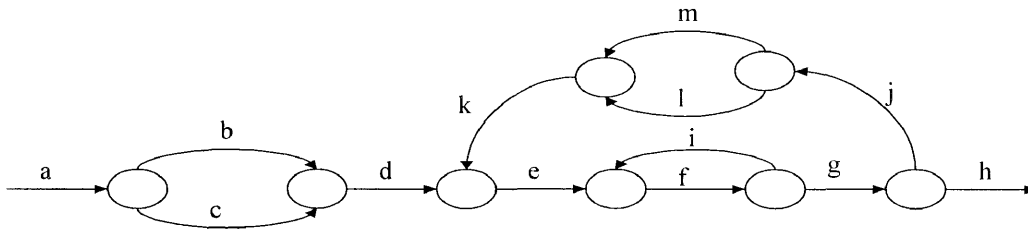
Following is the procedure for Maximum-path Arithmetic (Beizer, 1990). Start with a state transition diagram, label each link with a link weight that corresponds to the number of paths that link represents. Typically, that's one. However, if the link represented a subroutine call, say, and you wanted to consider the paths through the subroutine in the path count, then you would put that number on the link. Also mark each loop with the maximum number of times that the loop can be taken. There are three cases of interest: parallel links, serial links, and loops. In what follow, A and B are path expressions and W_A and W_B are algebraic expressions in the weights.

CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A+B$	$W_A + W_B$
SERIES	AB	$W_A W_B$
LOOP	$A^n = A^*$	$\sum_{j=0}^n W_A^j$

The arithmetic is ordinary algebra. This is not a true upper bound for the number of paths, but a larger number because the model does not include paths that might be forbidden due to correlated and dependent predicates. The rationale behind the parallel rule is simple. The path expressions denote the paths in a set of paths corresponding to that expression. The weight is the number of paths in each set. Assuming that the path expression were derived in the usual way, they would have no paths in common and consequently, the sum of the paths for the union of the sets would be the sum of the number of paths in each set. The serial rule is explained by noting that each term of the path expression (say the first one A) will be combined with each term of the second expression B , in all possible ways. If there are W_A paths in A and W_B Paths in B , then there must be $W_A W_B$ paths in the combination. The loop rule follows from the combination of the serial and parallel rules, taking into account going through zero, once, twice, and so on. If you know for a fact that the minimum number of times through the loop is not zero but some other number, say j , then you would do the summation from j to n rather than from 0 to n .

Here is a reasonably well-structured program. Its path expression, with a little work, is shown below:

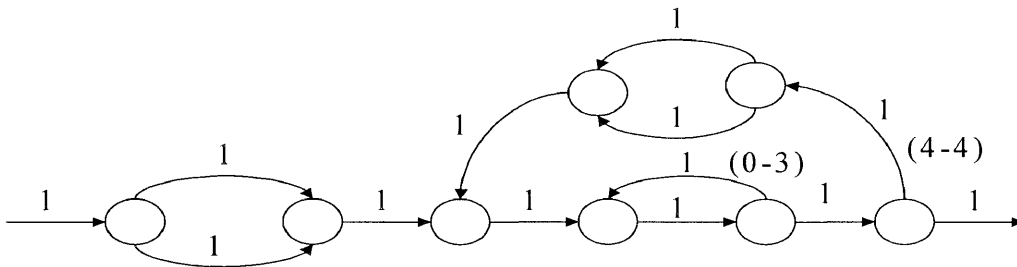
(1)



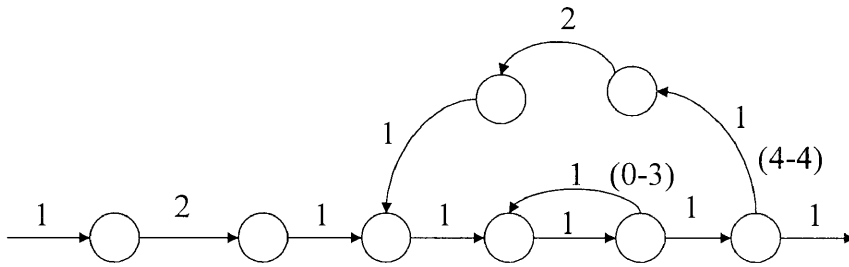
$$a(b+c)d\{e(fi)^*fgj(m+l)k\}^*e(fi)^*fgh$$

Each link represents a single link and consequently is given a weight of “1” to start. Let’s say that the outer loop will be taken exactly four times and the inner loop can be taken zero to three times. The steps in the reduction are:

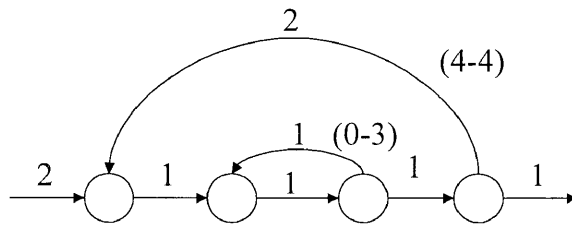
(2)



(3)

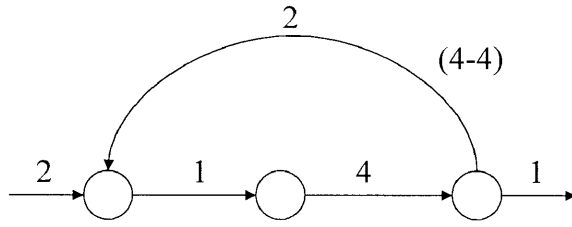


(4)

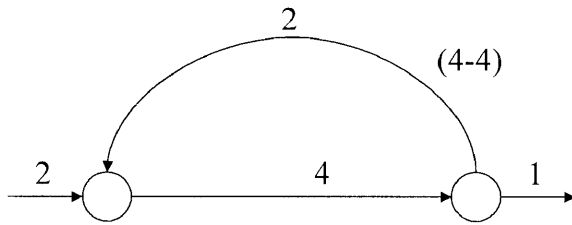


For the inner loop

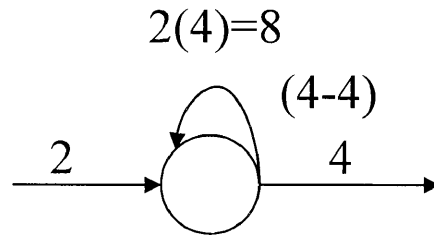
(5)



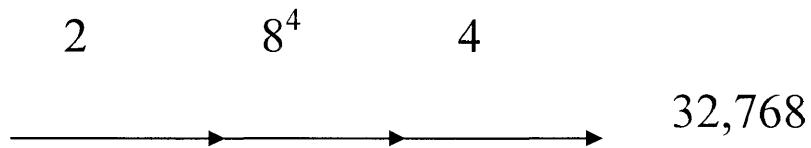
(6)



(7)



(8)



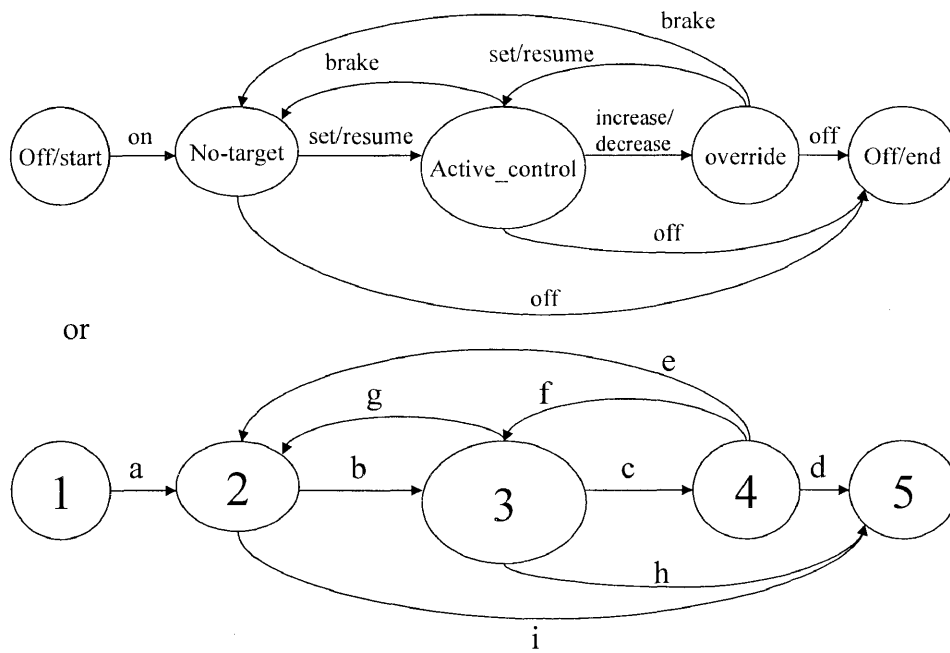
Alternatively, you could have substituted a “1” for each link in the path expression and then simplified, as follows:

$$\begin{aligned}
 & 1(1+1)1(1(1\times 1)^3 1\times 1\times 1(1+1)1)^4 1(1\times 1)^3 1\times 1\times 1 \\
 & = 2(1^3 1\times (2))^4 1^3 \\
 & \text{but } 1^3 = 1+1^1+1^2+1^3 = 4 \\
 & = 2(4\times 2)^4 \times 4 \\
 & = 2\times 8^4 \times 4 \\
 & = 32,768
 \end{aligned}$$

This is the same result we got graphically. Reviewing the steps in the reduction, we:

1. Annotated the flowchart by replacing each link name with the maximum number of paths through that link (1) and also noting the number of possibilities for looping. The inner loop was indicated by the range (0-3) as specified, and the outer loop by the range (4-4).
2. combined the first pair of parallels outside of the loop and also the pair corresponding to the IF-THEN-ELSE construct in the outer loop. Both yielded two possibilities.
3. Multiplied things out and removed notes to clear the clutter.
4. Took care of the inner loop: there were four possibilities, leading to the four values. Then we multiplied by the link weight following (originally link g) whose weight was also 1.
5. Got rid of link e.
6. Used the cross-term to create the self-loop with a weight of $8 = 2 \times 4$ and passed the outer 4 through.

For the cruise control example as figure 4, we re-draw the graph as follow in order to derive the path expressions easier:



or

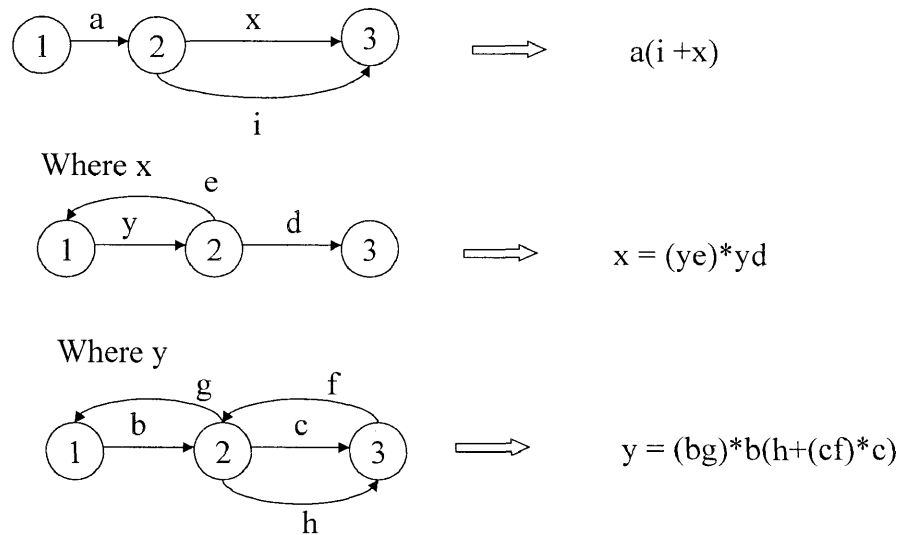
Where

- | | | |
|--------------------------|---------|----------------------|
| 1 – Off or start state | a | – on |
| 2 – No_target state | b, f | -- set/resume |
| 3 – Active_control state | c | -- increase/decrease |
| 4 – Override state | d, h, i | -- off |
| 5 – off/end state | e, g | – brake |

The path expression is:

$$a(i + ((bg)*b(h+(cf)*c)e)* (bg)*b(h+(cf)*c)d)$$

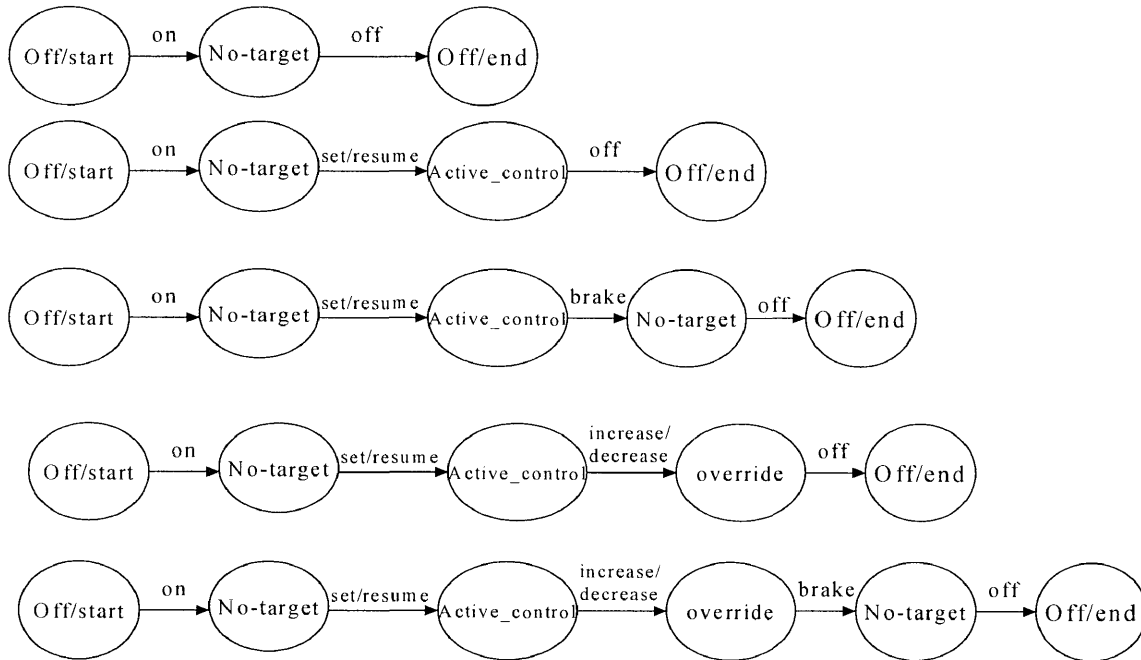
The above expression is derived after the following substitution:



Assume only taking loop once and we get the maximum-path arithmetic of the cruise control as:

$$\begin{aligned}
 & 1(1 + ((1 \times 1) * 1(1 + (1 \times 1) * 1)1) * (1 \times 1) * 1(1 + (1 \times 1) * 1)1) \\
 &= 1(1 + ((1 \times 1)^1 1(1 + (1 \times 1)^1 1)1)^1 (1 \times 1)^1 1(1 + (1 \times 1)^1 1)1) \\
 &= 1 + 2 \times 2 \\
 &= 5
 \end{aligned}$$

And they are:



4.2 Minimum-Path Arithmetic

A lower bound on the number of paths in a routine can be approximated for structured flowcharts (Beizer, 1990). It is not a true lower bound because again, forbidden paths could reduce the actual number of paths to a lower number yet. The appropriate arithmetic is:

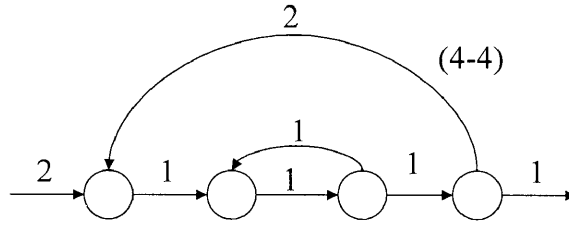
CASE	PATH EXPRESSION	WEIGHT EXPRESSION
PARALLEL	$A+B$	$W_A + W_B$
SERIES	AB	$\text{MAX}(W_A, W_B)$
LOOP	$A^n = A^*$	$1, W_1$

The parallel case is the same as before. The values of the weights are the number of members in a set of paths. There could be an error here because both sets could contain the null path, but because of the way the loop expression is defined, this cannot happen. The series case is explained by noting that each term in the first set will be combined with at least one term in the number of possibilities in the first set and the second set. The loop case requires that you use the minimum number of loops—possibly zero. Loops are always problematic. If the loop can be bypassed, then you can ignore the term in the loop. But it is better to use a value of 1, so that we are asserting that we'll count the number of paths under the assumption that the loop will be taken once. Because in creating the self-loop, we used the cross-term expression, there will be a contribution to the links following the loop, which will take things into account.

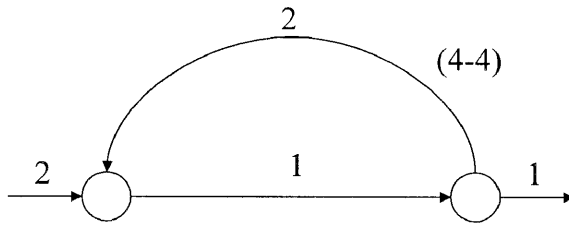
Alternatively, you could get a higher lower bound by arguing that if the loop were to be taken once, then the path count should be multiplied by the loop weight. This however, would be equivalent to saying that the loop was assumed to be taken both zero and once, because again, the cross-term that created the self-loop was multiplied by the series term. Generally, if you ask for a minimum number of paths, it is more likely that the minimum is to be taken under the assumption that the routine will loop once—because this is consistent with coverage.

Applying this arithmetic to the earlier example gives us the identical steps until Step 3, where we pick up:

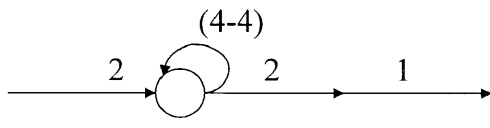
(4)



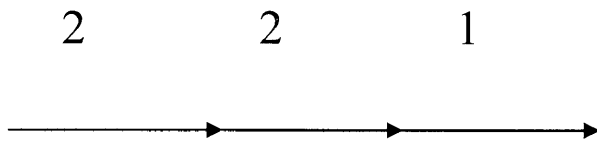
(5)



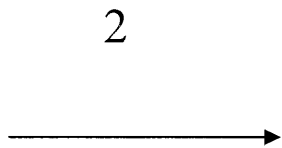
(6)



(7)

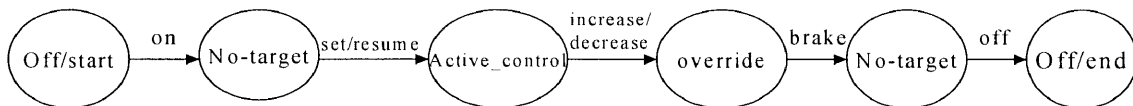


(8)



If you go back to the original graph, you will see that it takes a minimum of two paths to cover, and it can be done in two paths. The reason for restricting the algorithm to structured graphs is that for nonstructured graph the result can depend on the order in which nodes are removed. Structured or not, it's worth calculating this value to see if you have at least as many paths as the minimum number of paths calculated this way. If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

It is obvious that for cruise control the minimum-path arithmetic is 1, which means that there is a single path that can cover all states and inputs. It is:



4.3 Coverage Testing

There are four levels of path and coverage testing (Offutt, Abdurazik, 1999). (1) the transition coverage level, (2) the full predicate coverage level, (3) the transition-pair coverage level, and (4) the complete sequence level.

It is possible to apply all levels, or to choose a level based on a cost/benefit tradeoff. The first two are related; the transition coverage level requires many fewer test cases than the full predicate coverage level, but if the full predicate coverage level is used, the tests will also satisfy the transition coverage level (full predicate coverage subsumes transition coverage). Thus only one of these two should be used. The latter two levels are meant to be independent; transition-pair coverage is intended to check the interfaces among states, and complete sequence testing is intended to check the software by executing the software through complete execution paths. As it happens, transition-pair coverage subsumes transition coverage, but they are designed to test the software in very different ways.

4.3.1 Transition Coverage Level

It seems reasonable to expect that to test the software adequately, the tester should at minimum use tests that cause every transition in every statechart to be taken. This level requires just that, by requiring test cases that satisfy each precondition in the specification at least once. In the criteria definitions, T is a set of test cases, and SG is a specification graph, a graph that represents the transitions in a statechart. Although the tests are intended to be executed on an implementation of the specification, we say that a test

traverses a transition to indicate that, from a modeling perspective, the test causes the transition's predicate to be true, and the implementation will change from the transition's pre-state to its post-state.

4.3.2 Full Predicate Coverage Level

Small inaccuracies in the specification predicates can lead to major problems in the software. The full predicate coverage level takes the philosophy that to test the software we should at least provide inputs to test each clause in each predicate. This level requires that each clause in each predicate on each transition be tested independently, thus attempting to address the question of whether each clause is necessary and is formulated correctly. The Boolean operators are AND, OR, and NOT. A clause is a Boolean expression that contains no Boolean operators. For example, relational expressions and Boolean variables are clauses. A predicate is a Boolean expression that is composed of clauses and zero or more Boolean operators. A predicate without a Boolean operator is also a clause. If a clause appears more than once in a predicate, each occurrence is a distinct clause.

Full predicate coverage is based on the philosophy that each clause should be tested independently, that is, while not being influenced by the other clauses. In other words, each clause in each predicate on every transition must independently affect the value of the predicate. That is, for each predicate P on each transition, T must include tests that

cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c . Here, "directly correlated" means that c controls the value of P , that is, one of two situations occurs. Either c and P have the same value (c is true implies P is true and c is false implies P is false), or c and P have opposite values (c is true implies P is false and c is false implies P is true). This explicitly disallows cases such as c is true implies P is true and c is false implies P is true.

Note that if full predicate coverage is achieved, transition coverage will also be achieved. To satisfy the requirement that the test clause controls the value of the predicate, other clauses in the predicate must be either True or False. For example, if the predicate is $(X \wedge Y)$, and the test clause is X , then Y must be True. Likewise, if the predicate is $(X \vee Y)$, Y must be False.

4.3.3 Transition-Pair Coverage Level

The previous testing levels test transitions independently, but do not test sequences of state transitions. This level requires that pairs of transitions be taken. That is, for each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$ in SG , T contains a test that traverses the pair of transitions in sequence.

4.3.4 Complete Sequence Level

It seems very unlikely that any successful test method could be based on purely mechanical methods; at some point the experience and knowledge of the test engineer must be used. Particularly at the system level, effective testing probably requires detailed domain knowledge. A complete sequence is a sequence of state transitions that form a complete practical use of the system. In most realistic applications, the number of possible sequences is too large to choose all complete sequences. In many cases, the number of complete sequences is infinite. So for complete sequence level testing, the test engineer must define meaningful sequences of transitions on the statechart diagram by choosing sequences of states that should be entered.

5 Path testing approaches and algorithms

5.1 Testing the Most Likely Paths

It would be helpful if there were a way to guide the path testing into areas that are of more interest to the tester. For instance, you might want to see if a path includes all the activities that a user is more likely to perform. Or, you might only want to test the minimum number of paths that cover all of the states in the model.

For example (Beizer, 1990), a program that detects the character sequence “zczc” can be in the following states:

1. neither zczc nor any part of it has been detected

2. z has been detected
3. zc has been detected
4. zcz has been detected
5. zczc has been detected

The inputs are:

1. Z
2. C
3. Any character other than Z or C, which we'll denote by A

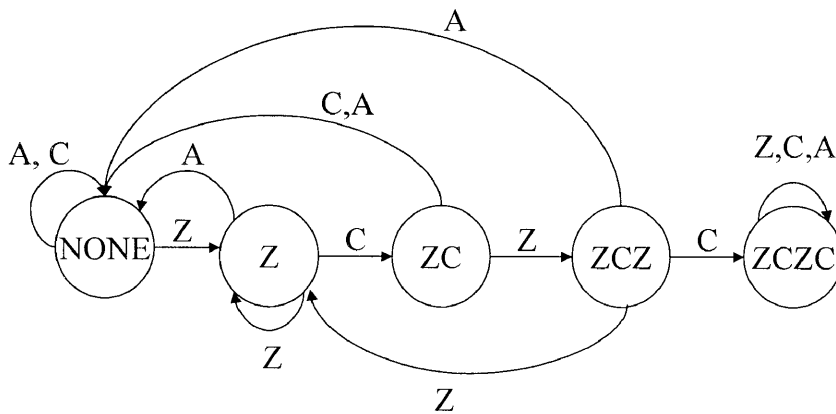


Figure 5: ZCZC sequence detector state graph

1. If the system is in the “NONE” state, any input other than a Z will keep it in that state.
2. If a Z is received, the system transitions to the “Z”.
3. If the system is in the “Z” state, and a Z is received, it will remain in the “Z” state. If a C is received it will go to the “ZC” state, and if any other character is received, it will go back to the “NONE” state because the sequence has been broken.
4. A Z received in the “ZC” state progresses to the “ZCZ” state, but any other character breaks the sequence and causes a return to the “NONE” state.
5. A C received in the “ZCZ” state completes the sequence and the system enters the “ZCZC” state. A Z breaks the sequence and causes a transition back to the “Z” state; any other character cause a return to “NONE” state.
6. No matter what is received in the “ZCZC” state, the system stays there.

String matching with a finite automata algorithm (Cormen, 1990) can be used in testing the paths interested in the model. You need just change concepts for the character string into transition input strings and the text into a path pool of your model.

5.1.1 String-matching

We formalize the string-matching problem as follows. We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length m . We further

assume that the elements of P and T are characters drawn from a finite alphabet Σ . For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \dots, z\}$. The character arrays P and T are often called strings of characters.

We say that pattern P occurs with shift s in text T (or, equivalently, that pattern P occurs beginning at position $s+1$ in text T) if $0 \leq s \leq n-m$ and $T[s+1..s+m] = P[1..m]$ (that is, if $T[s+j] = p[j]$, for $1 \leq j \leq m$). If P occurs with shift s in T , then we call s a valid shift; otherwise, we call s an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T . Figure 6 illustrates these definitions.

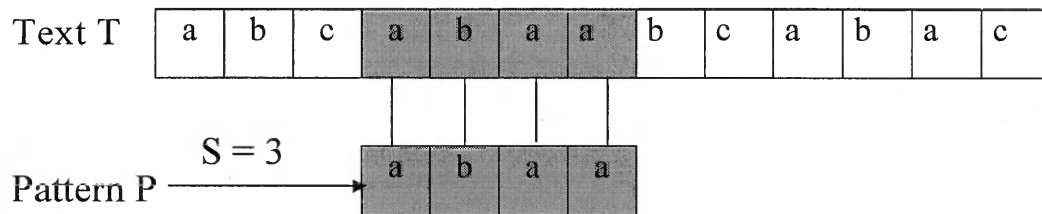


Figure 6: The string-matching problem

The goal is to find all occurrences of the pattern $P = abaa$ in the text $T = abcabaabcbac$.

The pattern occurs only once in the text, at shift $s = 3$. The shift $s = 3$ is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

We shall let Σ^* denote the set of all finite-length strings formed using characters from the alphabet Σ . The zero-length empty string, denoted ϵ , also belongs to Σ^* . The length of a string x is denoted $|x|$. The concatenation of two strings x and y , denoted xy , has length $|x| + |y|$ and consists of the characters from x followed by the characters from y .

We say that a string w is a prefix of a string x , denoted $w \subset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \subset x$, then $|w| \leq |x|$. Similarly, we say that a string w is a suffix of a string x , denoted $w \supset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \supset x$ that $|w| \leq |x|$. The empty string ε is both a suffix and a prefix of every string. For example, we have $ab \subset abcca$ and $cca \supset abcca$.

5.1.2 Finite Automata

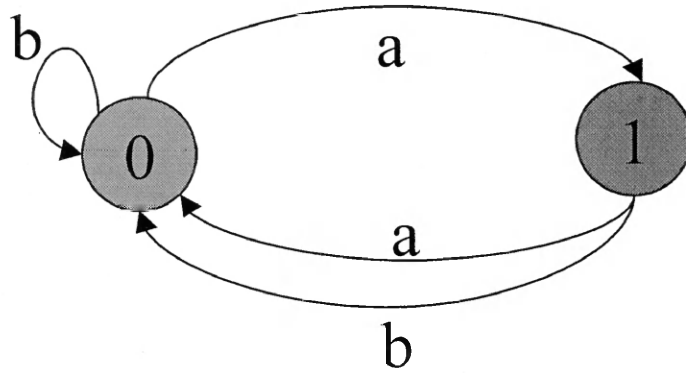
A finite automata M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of states,
- $q_0 \in Q$ is the start state,
- $A \subseteq Q$ is a distinguished set of accepting states,
- Σ is a finite input alphabet,
- δ is a function from $Q \times \Sigma$ into Q , called the transition function of M .

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a , it moves (“makes a transition”) from state q to state $\delta(q, a)$. Whenever its current state q is a member of A , the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected.

state	input	
	a	b
0	1	0
1	0	0

(a)



(b)

Figure 7

Figure 7 illustrates these definitions with a simple two-state automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{a, b\}$. Figure 7 (a) is a tabular representation of the transition function δ . Figure 7 (b) is an equivalent state-transition diagram. State 1 is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates $\delta(1, b) = 0$. This automaton accepts those strings that end in an odd number of a 's. More precisely, a string x is accepted if and only if $x = yz$, where $y = \varepsilon$ or y ends with a b , and $z = a^k$, where k is odd. For example, the sequence of states this automaton enters for input $abaaa$ (including the start state) is $\langle 0, 1, 0, 1, 0, 1 \rangle$, and so it accepts this input. For input $abbaa$, the sequence of states is $\langle 0, 1, 0, 0, 1, 0 \rangle$, and so it rejects this input.

A finite automaton M induces a function ϕ , called the final-state function, from Σ^* to Q such that $\phi(\omega)$ is the state M ends up in after scanning the string ω . Thus, M accepts a string ω if and only if $\phi(\omega) \in A$. The function ϕ is defined by the recursive relation

$$\phi(\varepsilon) = q_0,$$

$$\phi(\omega a) = \delta(\phi(\omega), a) \quad \text{for } \omega \in \Sigma^*, a \in \Sigma.$$

5.1.3 String-matching automata algorithm

There is a string-matching automaton for every pattern P ; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string. Figure 2 illustrates this construction for the pattern $P = \text{ababaca}$. From now on, we shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation.

In order to specify the string-matching automaton corresponding to a given pattern $P[1..m]$, we first define an auxiliary function σ , called the suffix function corresponding to P . The function σ is a mapping from Σ^* to $\{0, 1, \dots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x :

$$\sigma(x) = \max \{ k: P_k \supset x \}.$$

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, for the pattern $P = ab$, we have $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$, and $\sigma(ccab) = 2$. For a pattern P of length m , we have $\sigma(x) = m$ if and only if $P \supset x$. It follows from the definition of the suffix function that if $x \supset y$, then $\sigma(x) \leq \sigma(y)$.

We define the string-matching automaton corresponding to a given pattern $P[1..m]$ as follows.

- The state set Q is $\{0, 1, \dots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a :

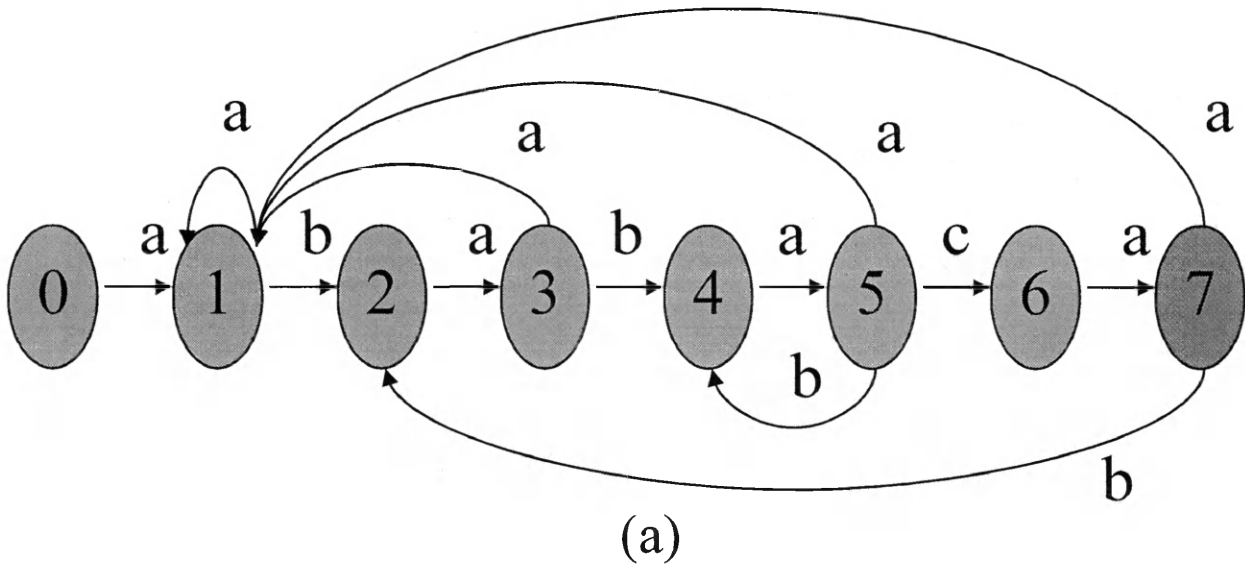
$$\delta(q, a) = \sigma(P_q a).$$

Here is an intuitive rationale for defining $\delta(q, a) = \sigma(P_q a)$. The machine maintains as an invariant of its operation that $\phi(T_i) = \sigma(T_i)$. In word, this means that after scanning the first i characters of the text string T , the machine is in state $\phi(T_i) = q$, where $q = \sigma(T_i)$ is the length of the longest suffix of T_i that is also a prefix of the pattern P . If the next character scanned is $T[i+1] = a$, then the machine should make a transition to state $\sigma(T_{i+1}) = \sigma(T_i a)$. That is, to compute the length of longest suffix of $T_i a$ that is a prefix of P , we can compute the longest suffix of $P_q a$ that is a prefix of P . At each state, the machine only needs to know the length of the longest prefix of P that is a suffix of what

has been read so far. Therefore, setting $\delta(q, a) = \sigma(P_q a)$ maintains the desired invariant

$$\phi(T_i) = \sigma(T_i).$$

In the string-matching automaton of figure 8, for example, we have $\delta(5, b) = 4$. This follows from the fact that if the automaton reads a b in state $q = 5$, then $P_q b = ababab$, and the longest prefix of P that is also a suffix of ababab is $P_4 = abab$.



state	Input			Pattern
	a	b	c	
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	-	1	2	3	4	5	6	7	8	9	10	11
T[i]	-	a	b	a	b	a	b	a	c	a	b	a
State $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figure 8

Figure 8 (a) A state –transition daigram for the string-matching automaton that accepts all strings ending in the string ababaca. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents $\delta(i, a) = j$. The right-going edges forming the “spine” of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; if a state i no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i, a) = 0$. Figure 8 (b) The corresponding transition function δ , and the pattern string $P = ababaca$. The entries corresponding to successful matches between pattern and input characters are shown shaded. Figure 8 (c) The operation of the automaton on the text $T = abababacaba$. Under each text character $T[i]$ is given the state $\phi(T_i)$ the automaton is in after processing the prefix T_i . One occurrence of the pattern is found, ending in position 9.

Following is the algorithm for simulating the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern P of length m in an input text $T[1..n]$. As for any string-matching automaton for a pattern of length m , the state set Q is $\{0,1,\dots,m\}$, the start state is 0 , and the only accepting state is state m .

FINITE-AUTOMATON-MATCHER(T, δ, m)

1. $N \leftarrow \text{length}[T]$
2. $q \leftarrow 0$
3. for $i \leftarrow 1$ to n
4. do $q \leftarrow \delta(q, T[i])$
5. if $q = m$
6. then $s \leftarrow i - m$
7. print "Pattern occurs with shift" s

The following procedure computes the transition function δ from a given pattern $P[1..m]$.

COMPUTE-TRANSITION-FUNCTION(P, Σ)

1. $m \leftarrow \text{length}[P]$
2. for $q \leftarrow 0$ to m
3. do for each character $a \in \Sigma$
4. do $k \leftarrow \min(m+1, q+2)$
5. repeat $k \leftarrow k-1$
6. until $P_k \supset P_q a$
7. $\delta(q, a) \leftarrow k$
8. return δ

This procedure computes $\delta(q, a)$ in a straight forward manner according to its definition.

The nested loops beginning on lines 2 and 3 consider all states q and characters a , and lines 4-7 set $\delta(q,a)$ to be the largest k such that $P_k \supset P_q a$. The code starts with the largest conceivable value of k , which is $\min(m, q+1)$, and decreases k until $P_k \supset P_q a$.

5.2 Random Walk with Traversal Markers and Algorithm

A random walk (sometimes called a “drunkard’s walk”) is simple to describe: from the current node, choose an outgoing link at random, follow that link to the next node and repeat the process. Traversal marker is to record all the paths executed and to see if the succession of link names correspond exactly to the expected path name.

Random walks are very simple to implement because they have no real guiding, overall plan. Interestingly, they can be very useful in software testing because their very lack of a plan makes them fairly resistant to the pesticide paradox. Random walks have been used with great success in some of Microsoft’s testing efforts. (Nyman, 1998)

There are however, several difficulties and weakness in random test data (Beizer, 1990), especially if that is the only kind of test that’s done.

1. Random data produces a statistically insignificant sample of the possible paths through most routines (Huang, 1975; Moranda, 1978). Because it may be difficult to determine how many feasible paths there are, even copious tests based on random data may not allow you to produce a statistically valid prediction of the routine’s reliability.
2. There is no assurance of coverage. Running the generator to the point of 100% coverage could take centuries. Especially, random walks tend to be very inefficient

about covering a large graph quickly. Since they have no notion of where they have already been in the graph, they tend to re-traverse links they have already visited. For instance, a random walk on a typical application might invoke the Help screen many times before moving on to testing the parts of the application that you want it to test.

3. If the data are generated in accordance with statistics that reflect expected data characteristics, the test cases will be biased to the normal paths—the very paths that are least likely to have bugs.
4. It may be difficult or impossible to predict the desired outputs and therefore to verify that the routine is working properly; all you might learn is that it did not blowup but not whether what it did made sense or not. In many cases, the only way to produce the output against which to make a comparison is to run the equivalent of the routine; which equivalence is as likely to have bugs as the routine being tested.

If random path generation is to be used, instead of generating test cases in accordance with the probability of traversals at decisions, the test cases should be generated in accordance with the complementary probability. This would, at least, bias the paths away from the normal cases and toward the weird case that are more likely to have bugs.

Use the notation for finite automata in the last section. The algorithm for a random walk:


```

RANDOM-WALK(Q,  $\delta$ ,  $\Sigma$ )
1  q  $\leftarrow$  0
2  for i  $\leftarrow$  1 to n (n might be the maximum number of pass of model)
3      do
4          j  $\leftarrow$  RandomNumber
5          q  $\leftarrow$   $\delta$  (q,  $\Sigma$  [j])
6          print "q"
7  until q = A (accepting state)

```

5.3 Full Predicate Coverage Testing and Algorithm

Treating testing as sampling requires determining the scope of the test by understanding the input population. Specifically, testers must analyze the environment in which the system operates and identify each input source. Each input source is essentially a subpopulation that we further decompose by determining relevant subclasses that might be (or must be) tested separately. In addition to sources of input, we also identify output devices that receive data from the system under test. Sometimes, the internal state of such devices can affect how the system under test behaves.

The *operational environment* is the set of all systems, components and people that interact with the system under test or affect the system under test in any manner. Informally the operational environment is the “environment in which the software operates.” The process of understanding the operational environment and dividing it into subpopulations is called *domain decomposition*. This is the first activity testers pursue when treating testing as sampling.

5.3.1 Case Study

As an example (Offutt, Liu, 1999), the cruise control system (note that it does not model the throttle) has four states: OFF (the initial state), NO_TARGET, ACTIVE_CONTROL, and OVERRIDE. The system's environmental conditions indicate whether the automobile's ignition is on (Ignited), the engine is running (Running), the automobile is going too fast to be controlled (Toofast), the brake pedal is being pressed (Brake), and whether the cruise control level is set to Activate, Deactivate, or Resume. Table 3 is the state transition table of cruise control system with environmental conditions.

Previous State	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	New State
OFF	@T	-	-	-	-	-	-	NO_TARGET
NO_TARGET	@F	-	-	-	-	-	-	OFF
	t	t	-	f	@T	-	-	ACTIVE_CONTROL
ACTIVE_CONTROL	@F	-	-	-		-	-	OFF
	t	@F	-	-		-	-	NO_TARGET
	t	-	@T	-		-	-	
	t	t	f	@T		-	-	OVERRIDE
	t	t	f	-		@T	-	
OVERRIDE	@F	-	-	-		-	-	OFF
	t	@F	-	-		-	-	NO_TARGET
	t	t	-	f	@T	-	-	ACTIVE_CONTROL
	t	t	-	f		-	@T	

Table 3: State transition table for the Cruise Control System

Each row in the table specifies a conditioned event that activates a transition from the state on the left to the state on the right. A table entry of @T or @F under a column header C represents a triggering event @T(C) or @F(C). This means that the value of C must change for the transition to be taken, that is, "@T(C)" means C must change from false to true, and "@F(C)" means C must change from true to false. A table entry of t or f represents a WHEN condition. WHEN[C] means the transition can only be taken if C is true, and WHEN[¬C] means it can only be taken if C is false. If the value of a condition C does not affect a conditioned event, the table entry is marked with a hyphen "-" (don't care condition).

Table 4 shows the transitions of the specification with the trigger events expanded in predicate form, numbered P₁ through P₁₂. A triggering event is a change in a value for a variable, expression, or expressions that causes the software to transition from one state to another. A triggering event actually specifies two values, a before-value and an after-value. To fully test predicates with triggering events, test engineers must distinguish between them by controlling values for both before-values and after-values. This paper suggests implementing this by assuming two versions of the triggering event variable, X and X', where X represents the before-value of X and X' represents its after-value. Figure 9 shows the state transition diagram with the edges labeled with the predicate numbers.

Predicate No.	Previous State	Predicates	New State
P ₁	OFF	$\neg \text{Ignited} \wedge \text{Ignited}'$	NO_TARGET
P ₂	NO_TARGET	$\text{Ignited} \wedge \neg \text{Ignited}'$	OFF
P ₃	NO_TARGET	$\neg \text{Activate} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Brake} \wedge \text{Activate}'$	ACTIVE_CONTROL
P ₄	ACTIVE_CONTROL	$\text{Ignited} \wedge \neg \text{Ignited}'$	OFF
P ₅	ACTIVE_CONTROL	$\text{Running} \wedge \text{Ignited} \wedge \neg \text{Running}'$	NO_TARGET
P ₆	ACTIVE_CONTROL	$\neg \text{Toofast} \wedge \text{Ignited} \wedge \text{Toofast}'$	NO_TARGET
P ₇	ACTIVE_CONTROL	$\neg \text{Brake} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Toofast} \wedge \text{Brake}'$	OVERRIDE
P ₈	ACTIVE_CONTROL	$\neg \text{Deactivate} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Toofast} \wedge \text{Deactivate}'$	OVERRIDE
P ₉	OVERRIDE	$\text{Ignited} \wedge \neg \text{Ignited}'$	OFF
P ₁₀	OVERRIDE	$\text{Running} \wedge \text{Ignited} \wedge \neg \text{Running}'$	NO_TARGET
P ₁₁	OVERRIDE	$\neg \text{Activate} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Brake} \wedge \text{Activate}'$	ACTIVE_CONTROL
P ₁₂	OVERRIDE	$\neg \text{Resume} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Brake} \wedge \text{Resume}'$	ACTIVE_CONTROL

Table 4: state transition table with the trigger events expanded in predicate form

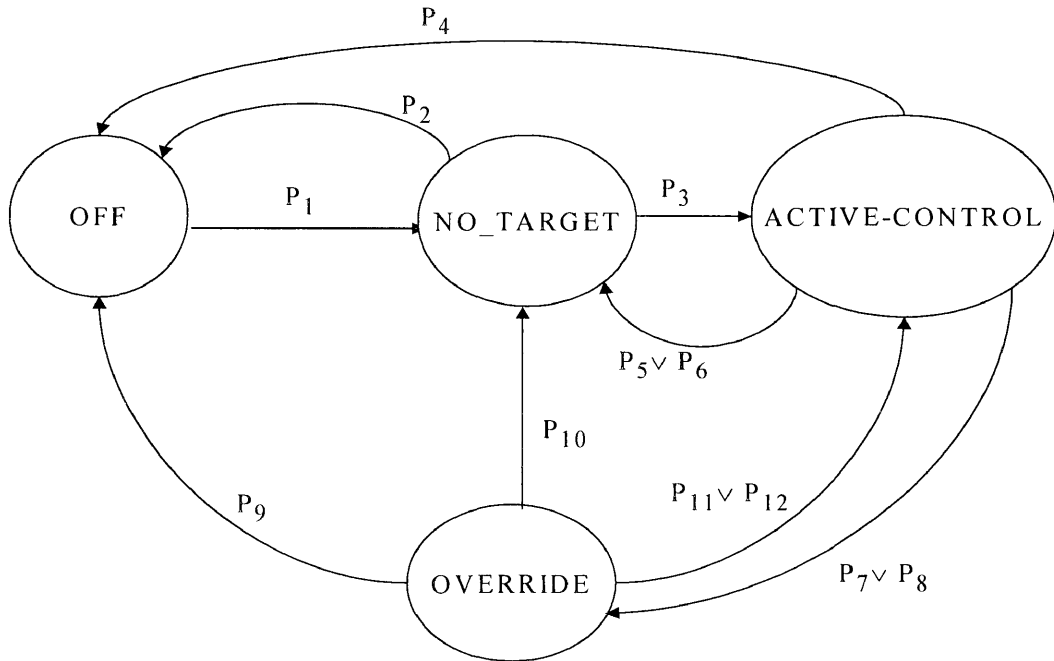


Figure 9: shows the state transition diagram with the edges labeled with the predicate numbers.

5.3.2 Full predicate coverage criterion

There are nine transitions in the cruise control specifications, and twelve disjunctive predicates. For convenience, the technique is applied by considering each predicate specification separately. Both the before-values and after-values of the triggering event should be separately tested. This is handled by treating @ as an operator and expanding it algebraically. If X represents a before-value and X' an after-value, the relevant expansions are:

- $@T(X) \equiv \neg X \wedge X'$
- $@T(X \wedge Y) \equiv \neg (X \wedge Y) \wedge (X' \wedge Y') \equiv (\neg X \vee \neg Y) \wedge X' \wedge Y'$
- $@T(X \vee Y) \equiv \neg (X \vee Y) \wedge (X' \vee Y') \equiv \neg X \wedge \neg Y \wedge (X' \vee Y')$

There are 54 separate test case requirements for the full predicate coverage. The third transition, P_3 , is used to illustrate the test case requirement derivation. The variable values are taken from the predicates, and are shown as T, F, t, f, and -. A T or F means the clause is triggering, and the table contains a before-value and after-value. The values for the test case are the new value for the triggering clause (T or F), and the t and f values from the WHEN conditions. The expected output for the test specification is derived from the triggering event, the post-state, and any terms or variables that are defined as a result of the transition. P_3 has four clauses:

$@T \text{ Activate} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Brake}$

and its expanded version is:

$$\neg \text{Activate} \wedge \text{Ignited} \wedge \text{Running} \wedge \neg \text{Brake} \wedge \text{Activate}'$$

Its six test case requirements are:

Pre State	Activate	Ignited	Running	Brake	Activate'	Post State
1. NO-TARGET	F	t	t	f	T	ACTIVE_CONTROL
2. NO-TARGET	F	f	t	f	T	NO-TARGET
3. NO-TARGET	F	t	f	f	T	NO-TARGET
4. NO-TARGET	F	t	t	t	T	NO-TARGET
5. NO-TARGET	T	t	t	f	T	NO-TARGET
6. NO-TARGET	F	t	t	f	F	NO-TARGET

The first row is the predicate as it appears in the specification; every clause is True. This corresponds to a valid test input (and is also the transition coverage test case for this transition). The subsequent rows make each clause False in turn, corresponding to invalid inputs. Because there are no OR operators, the full predicate coverage criterion is satisfied by holding all other clauses True. The post-states are the expected values. Five of them represent invalid transitions, and it is assumed that the software will remain in the same state.

5.3.3 Test specifications

The actual test specifications and test scripts are mechanically derived from the test requirements. The predicate P_3 is chosen as an illustrative example. P_3 has six full

predicate level tests. For the first test case for P_3 , the test case must reach the NO-TARGET state; this forms the Prefix. The Test case values set the before-value for the triggering event, and the WHEN condition variables of Inactive, Running, and Brake, and then sets Activate to be True as the triggering event. The Verify and Exit parts of the specifications are not shown, as they depend on the software. The software can safely be assumed to automatically print the current state, and to not require an exit.

1. Test specification P_3 -1:

Prefix: Ignited = True -- Reach NO-TARGET state
 Test case value: Activate = False -- Trigger before-value
 Running = True -- Condition variable
 Brake = False -- Condition variable
 Activate = True -- Triggering event
 Expected outputs: ACTIVE_CONTROL

2. Test specification P_3 -2:

Prefix: Ignited = True -- Reach NO-TARGET state
 Test case value: Activate = True -- Trigger before-value
 Running = True -- Condition variable
 Brake = False -- Condition variable
 Activate = True -- Triggering event
 Expected outputs: NO-TARGET

3. Test specification P_3 -3:

Prefix: Ignited = True -- Reach NO-TARGET state
 Test case value: Activate = False -- Trigger before-value
 Ignited = False -- Condition variable
 Running = True -- Condition variable
 Brake = False -- Condition variable
 Activate = True -- Triggering event
 Expected outputs: NO-TARGET

4. Test specification P_3 -4:

Prefix: Ignited = True -- Reach NO-TARGET state
 Test case value: Activate = False -- Trigger before-value
 Running = False -- Condition variable
 Brake = False -- Condition variable
 Activate = True -- Triggering event
 Expected outputs: NO-TARGET

5. Test specification P₃-5:

Prefix: Ignited = True -- Reach NO-TARGET state
Test case value: Activate = False -- Trigger before-value
Running = True -- Condition variable
Brake = True -- Condition variable
Activate = True -- Triggering event
Expected outputs: NO-TARGET

6. Test specification P₃-6:

Prefix: Ignited = True -- Reach NO-TARGET state
Test case value: Activate = False -- Trigger before-value
Running = True -- Condition variable
Brake = False -- Condition variable
Activate = False -- Triggering event
Expected outputs: NO-TARGET

There is an interesting point to note about these test specifications. It should be clear that there is some redundancy; some of the condition variables do not need to be explicitly set, as they will already have the appropriate values.

Following are two algorithms (Offutt, Abdurazik, 1999), Get Prefix Algorithm and Generate Full-Predicate Coverage Test Cases Algorithm. A prefix generation algorithm was used in test data generation algorithms to create the values necessary to reach a particular state.

5.3.4 Get Prefix Algorithm

Figure 10 gives an algorithm for generating test prefix values from a specification graph. The input is a state (the test state) in the graph, and it finds a path from an initial state in the graph to the test state.

algorithm: **GetPrefix (State)**
 input: Test state of a transition.
 output: Inputs to get to the given state.
 output criteria: No redundant inputs.
 declare: prefix (s) -- Inputs to reach state s.
 incomingTrans (s) -- The set of incoming transitions.
 event (otr) -- Trigger event for transition otr.
 whenCondition (otr) -- Precondition for otr.
 nextState (otr) -- Next state for transition otr.
 expectedOutput -- Post-state after transition.
 TCValue (otr) -- Value assignments for the trigger
 event and when condition variables for otr.

GetPrefix (State)

```

BEGIN -- Algorithm GetPrefix
  s = State
  prefixStates = prefixStates ∪ s
  WHILE (s IS NOT initial state) LOOP
    get incomingTrans (s)
    prefix (s) = EMPTY
    IF (∃ transition itr ∈ incomingTrans (s) such that
      prevState (itr) = initialState) THEN
      s = prevState (itr)
      prefixStates = prefixStates ∪ s
      EXIT
    ELSE
      s = prevState (itr) such that itr ∈ incomingTrans (s) ∧
        prevState (itr) ∉ prefixStates
      prefixStates = prefixStates ∪ s
    END IF
  END LOOP
END Algorithm GetPrefix
  
```

Figure 10: The GetPrefix Algorithm

5.3.5 Generate Full-Predicate Coverage Test Cases Algorithm

Figure 11 gives an algorithm for generating test cases for the full predicate coverage criterion. Algorithm `GenerateFullPredicateCoverageTCs` takes a state transition table as input, and generates test cases for the full predicate coverage criterion. It processes each outgoing transition of each source state, generates a test case that makes the transition valid, and then generates test cases that make the transition invalid. When generating a test case, `GetPrefix()` is used to obtain prefixes to reach the source state of a transition. Then each variable in the transition predicate is assigned a test case value. To avoid redundant test case value assignments, those variables that already have assigned values in the prefixes are not considered in the test case value assignment process. After all test case values are generated, an additional algorithm is run on the test cases to identify and remove redundant test cases.

algorithm:	GenerateFullPredicateCoverageTCs (STTable)
input:	State transition table.
output:	Test cases for full predicate coverage.
output criteria:	Test cases contain prefix, test case values, and expected output.
assumption:	Clauses are disjunctive. No redundant assignments in prefix and test cases.
declare:	prefix (s) -- Inputs to get to the state s. outgoingTrans (s) -- Set of outgoing transitions. event (otr) -- Trigger event for transition otr. whenCondition (otr) -- Precondition for otr. nextState (otr) -- Next state for transition otr. expectedOutput -- Post-state after transition. TCValue (otr) -- Value assignments for the trigger event and when condition variables for otr.

`GenerateFullPredicateCoverageTCs (STTable)`

```

BEGIN -- Algorithm GenerateFullPredicateCoverageTCs
  TestCaseSet = EMPTY
  FOR EACH source state s in STTable
    prefix (s) = GetPrefix (s)
    get outgoingTrans (s)
    -- Generate one test case for each transition
    FOR EACH outgoing transition otr  $\in$  outgoingTrans (s)
      expectedOutput = nextState (otr)
      TCValue (otr) = EMPTY
      get event (otr) and whenConditions (otr)
      -- Check for redundancy
      IF ( $\neg \exists$  a condition variable var  $\in$  prefix (s) s.t.
        var.name = event (otr).name  $\wedge$  var.value = event (otr).value)
        TCValue (otr) = TCValue (otr)  $\cup$ 
          {(event (otr).name, event (otr).beforeValue)}
      END IF
      -- Assign value for clauses in when condition
      FOR EACH clausei in whenConditions (otr)
        IF ( $\neg \exists$  a condition variable var  $\in$  prefix (s) s.t.
          var.name = clausei.name  $\wedge$  var.value = clausei.value)
          TCValue (otr) = TCValue (otr)  $\cup$ 
            {( clausei.name, clausei.value)}
        END IF
      END FOR
      TCValue (otr) = TCValue (otr)  $\cup$  {(event (otr).name,
        event (otr).afterValue)}
      TestCaseSet = TestCaseSet  $\cup$  {(prefix (s), TCValue (otr),
        ExpectedOutput)}
      -- get test cases for invalid transitions
      expectedOutput = current state s
      FOR EACH variable var in TCValue (otr)
        TCValue (otr) = TCValue (otr) - {(var.name, var.value)}
        var.value =  $\neg$ var.value
        TCValue (otr) = TCValue (otr)  $\cup$  {(var.name, var.value)}
        TestCaseSet = TestCaseSet  $\cup$  {(prefix (s), TCValue (otr),
          expectedOutput)}
      END FOR
    END FOR
  END FOR
END Algorithm GenerateFullPredicateCoverageTCs

```

Figure 11: The GenerateFullPredicateCoverageTCs Algorithm

6 Conclusion

When conducting model testing, you may want to know how the state and transition are passed in your model. Using state transition diagram designed in the software design phase, you can conduct the path testing. If you would like to test the specific path in your mind, you can use string-matching algorithm to do the test. If you just want to know how many different paths are working as expected or get the general idea about model, you may conduct the random walk testing. Coverage testing is a good way to find out where and why some paths are not working.

REFERENCES

- Aldrich, William. **Coverage Analysis for Model Based Design Tools**, The Math Works, Inc.
- Apfelbaum, L. (1997) **Model-Based Testing**, Proceedings of Software Quality Week 1997
- Avritzer, A. and Weyuker, E. (1995), **Automatic Generation of Load Test Suites and the Assessment of the Resulting Software**, *IEEE Transactions on Software Engineering*, 21, 9, 705-716.
- Beizer, B. (1990) **Software Testing Techniques**, 2nd Edition
- Beizer, Boris. **Black Box Testing: Techniques for Functional Testing of Software and Systems**, New York, John Wiley & Sons, 1995
- Beltrami, E. (1977) **Models for Public Systems Analysis**
- Binder Robert V. **Testing Object-Oriented Systems, Models, Patterns, and Tools**, Addison-Wesley 1999
- Bodin, L. and Tucker, A. (1983) **A Model for Municipal Street Sweeping Operations** in Modules in Applied Mathematics Vol. 3: Discrete and System Models

Chow, T.S. (1978) **Testing Software Design Modeled by Finite-State Machines**, IEEE Transactions on Software Engineering 4

Cormen, Thomas. **Introduction to Algorithms**, Cambridge, Massachusetts, The MIT Press, 1990

Dahbura, A. and Uyar, M. (1986) **Optimal Test Sequence Generation for Protocols: The Chinese Postman Algorithm Applied to Q.931**, IEEE GLOBECOM, Dec. 1986

Dahle, O. (1995), **Statistical Usage Testing Applied to Mobile Telecommunication Systems**, Master's Thesis, Department of Computer Science, University of Trondheim, Norway.

Dill, D., Ho, R., Horowitz, M. and Yang, C. (1995) **Architecture Validation for Processors**, Proceedings of the 22nd annual International Symposium on Computer Architecture

Duran, J. and Ntafos, S. (1984), **An Evaluation of Random Testing**, *IEEE Transactions on Software Engineering*, 10, 4, 438-444.

Duran, J. and Wiorowski, J. (1984), **Quantifying Software Validity by Sampling**, *IEEE Transactions on Reliability*, 29, 2, 141-144.

Feller, W. (1950), **An Introduction to Probability Theory and its Application**, Vol. 1, Wiley, New York.

Gross, J. and Yellen, J. (1998) **Graph Theory and its Applications**

Hamlet, D. and Taylor, R. (1990), **Partition Testing Does Not Inspire Confidence**, *IEEE Transactions on Software Engineering*, 16, 12, 1402-1411.

Houghtaling, M. (1996), **Automation Frameworks for Markov Chain Statistical Testing**, In *Proceedings of the Automated Software Test and Evaluation Conference*, Washington, DC.

Howden, Willaim E. **Theoretical and empirical studies of program testing**. *IEEE Transactions on Software Engineering*, 4(4):293-298, July 1978.

Huang, J.C. **An approach to program testing**. ACM Computing Surveys 7: 113-128(1975).

Kwan, M-K. (1962) **Graphic Programming Using Odd and Even Points**, Chinese Journal of Mathematics, Vol. 1

Moranda, P.B. **Limits to program testing with random number inputs**. Proceedings COMSAC'78, New York: IEEE, 1978.

- Myers, Glenford J. **The Art of Software Testing**. Wiley, 1979.
- Nyman, N. (1998) **GUI Application Testing with Dumb Monkeys**, Proceedings of STAR West 1998
- Ntafos, Simeon C. **A comparison of some structural testing strategy**. IEEE Transactions on Software Engineering, 14(6):868{874, June 1988.
- Offutt, J. and Abdurazik, A. **Generateing tests from UML Specifications**, Second International Conference on the Unified Modeling Language (UML99), 1999
- Offutt, Jefferson, A, Liu, Shaoying, and Abdurazik, Aynur, **Generating Test Data Feom State-based Specifications**, 1999.
- Ostrand, T. and Balcer, M. (1988), **The Category-Partition Method for Specifying and Generating Functional Tests**, Communications of the ACM, 31, 6, 676-686.
- Poore, J. H., Mills, H. D. and Mutchler, D. M. (1993), **Planning and Certifying Software System Reliability**, *IEEE Software*, 88-99.
- Pressman, Roger **Software Engineering: A Pracitioner's Approach**, 5th Edition, McGraw-Hill companies Inc. 2001
- Rautakorpi, M. (1995), "**Application of Markov Chain Techniques in Certification of Software**", Master's Thesis, Department of Mathematics and Systems Analysis, Helsinki University of Technology, Helsinki, Finland.
- Robinson, Harry. "**Graph Theory Techniques in Model-Based Testing**", **Proceedings of the International Conference on Testing Computer Software 1999**
- Robinson, Harry, "**Finite State Model-Based Testing on a Shoestring**", Presented at STAR West 1999
- Shen, Y. and Lombardi, F. (1992) "**Graph Algorithms for Conformance Testing Using the Rural Chinese Postman Tour**", *SIAM Journal on Discrete Mathematics*, Vol. 9
- Skiena, S. (1998) **The Algorithm Design Manual**
- Thevenod-Fosse, P. and Waeselynck, H. (1993), "**STATEMATE Applied to Statistical Software Testing**," In *Proceedings of the International Symposium on Software Testing and Analysis*, ACM Press, Cambridge, MA, pp.99-109.
- Whittaker, James A. "**Stochastic software testing**". *Annals of Software Engineering*, 4:115{131, August 1997.

Whittaker, James A.. “**Software testing: What it is, and why it is so difficult**”. To appear in IEEE Software, 1999.

Whittaker, J. A. (1992), “**Markov Chain Techniques for Software Testing and Reliability Analysis**”, Ph.D. Dissertation, Dept. of Comp. Sci., Univ. of Tennessee, Knoxville, TN.

Whittaker, James A. and El-Far, Ibrahim K. “**Automated Construction of Behavior Models for Software Testing**”, IEEE Transactions on Software Engineering, (submitted)

Whittaker, J. and Thomason, M. (1994), “**A Markov Chain Model for Statistical Software Testing**”, *IEEE Transactions on Software Engineering*, 20, 10, 812-824..