Automated Construction of Software Behavior Models

by

Ibrahim Khalil Ibrahim El-Far

Bachelor of Science
in Computer Science
American University of Beirut
1995

A thesis
submitted to the Graduate School of
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 1999

We the undersigned committee hereby recommend that the attached document be accepted as fulfilling in part the requirements for the degree of Master of Science in Computer Science.

"Automated Construction of Software Behavior Models"
a thesis by Ibrahim Khalil Ibrahim El-Far

James A. Whittaker, Ph.D.
Associate Professor, Computer Science
Thesis Advisor

Shirley A. Becker, Ph.D.
Associate Professor, Computer Science
Committee Member

Gary W. Howell, Ph.D.
Associate Professor, Mathematics
Committee Member

William D. Shoaff, Ph.D.
Associate Professor and Program Chair
Computer Science

Abstract

Automated Construction of Software Behavior Models

by

Ibrahim Khalil Ibrahim El-Far

Thesis Advisor: James A. Whittaker, Ph.D.

In recent published work, a novel yet simple concept has been shown to be useful in understanding and constructing models of software behavior. In the context of software testing, operational modes have been used to construct Markov chains, which have been, in turn, used to select tests and compute software quality metrics.

We explore the behavior modeling problem in software testing as a state space enumeration problem. We present a framework for describing a software state space with operational modes, based on which an algorithm that automates the construction of behavior models for software systems is presented. These new findings are supported with detailed examples illustrating the construction process. Finally, we conclude with a summary and some perspectives on current and future work.

Contents

Acknowledgements

I would like to thank a number of persons without whose help and support the completion of this work would have been extremely difficult.

James Whittaker introduced me to the realm of software engineering and lent out constant academic and financial support without which my stay at Florida Tech would have been infeasible.

Anthony Berkemeyer was responsible for uncovering major flaws in earlier versions of this work through applying it to small realistic examples. I also benefited quite a bit from discussions with Mayuresh Kulkarni in the early stages of this work and recent discussions with Alvin Flanegin. The comments and observations of all three were especially helpful as to the practicality of the ideas presented here.

Lina Khatib got me interested in constraint satisfaction problems. She gave me several pointers on the subject and helped me gain focus early on in my research. Alan Jorgenson's work on operational modes has influenced the final mood of the document. Talking testing, computers and other topics over with Alan was, and still is, the make-up of the most exciting mind-teasing sessions.

Harry Robinson gave back the most beneficial feedback on this work. The phone example in this document was greatly influenced by his suggestions. Florence Mottay read this thesis for a couple of minutes very late into the writing process and uncovered several discrepancies. She also, later, gave the manuscript a thorough review.

Jasmine Jackson, Nikhil Nilakantan, and Luis Rivera accepted to review the final drafts of the work and came back with interesting insight. I also have caused Luis, who happens to be the software engineering laboratory's system administrator, some major headaches in the process of preparing this manuscript.

Gideon Wout, Nikhil Nilakantan and Roussi Rousseuv implemented a tool that put the principle ideas into practice. I also owe Nikhil and Gideon for helping me out getting resources when I was short on time.

Michael Gill and Johnny Stevenson wrote a LaTeX editor, jamLaTeX, that made the

write-up of this thesis flow just a little bit smoother.

Shirley Becker and Gary Howell graciously accepted to serve on my thesis committee and gave useful feedback in the process of reviewing the document.

Special thanks go to Billie Holiday, Otis Redding, and the handful of blues artists that kept me up the few nights I spent writing these words.

This document was generated using the LaTeX documentation preparation system written by Leslie Lamport. LaTeX is a set of macros for TeX, a typesetting program written by Donald E. Knuth. All of the work done on this document used the mikTeX distribution for the Microsoft Windows operating system maintained by Christian Schenk. The use of the workspace and printing facilities of the Center for Software Engineering Research at Florida Institute of Technology was both convenient and productive during the various production phases of this Master's thesis.

To the memory of Khalil
To Huda, Shadi, and Hadi

Chapter 1

Introduction

We proceed this work with an introduction to software testing: we suggest a definition and discuss the motivation for choosing testing over other quality assurance approaches; we briefly suggest and explain a classification of testing strategies; we explain the activities, problems, and difficulties in testing, giving bibliographical pointers whenever appropriate. Finally, the problem of modeling is presented, and the objectives of this work are stated.

1.1 Software Testing in a Flash

> *Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.*
>
> *E. Miller*
> *"Introduction to Software Testing Technology", [21]*

1.1.1 Definition

In an ideal world, the purpose of testing an implementation of a software system is to determine whether it is free of errors [11]. The IEEE standard for software testing terminology [8] defines a fault as an incorrect program component. The same standard defines an error as the incorrect behavior resulting from a fault. In theory, testing is a deductive form of verification by which errors can be detected, faults located, and the correctness of the program under test (or, otherwise, its deviation from specification) deduced.

In practice, it is very difficult to perform testing in the rigid and formal manner required to prove the correctness of programs as described in, say, [11] and [13]. It can be

observed through the software testing literature that, consequently, the focus has shifted to presenting more useful practicable techniques. These techniques have almost exclusively aimed at detecting more errors, exercising more of the implementation code in a variety of ways, or running "better quality" tests on the application under test. Myers in [25] gives a definition of testing that is widely accepted today: "Testing is the process of executing a program with the intent of finding errors."

1.1.2 Motivation for Testing

How vital and important testing is for any software development process can never be overemphasized [29] [31] [37]. The reason is primarily a matter of economics. The cost of the failure of software (which can range anywhere from loss of product image to the loss of human life) and the cost of software repair (requiring the release of patches or even new versions of the software) are too high for the software engineering management community to ignore. Some form of software quality assurance is therefore needed to help reduce the risk of incurring such costs.

The three most noteworthy approaches to attaining high software quality are formal verification, program proof, and testing. The main disadvantage of the first two is that they are inapplicable to large systems in practice, and, sometimes, they are too weakly developed technically to be effective even for small-scale software systems [21]. The very fact that formal methods are not easy to understand or to work with seems to have been reason enough for unpopularity in the testing industry [14]. Furthermore, formal verifications and proofs normally make a number of assumptions that may or may not hold in the field — the environment in which the software is actually going to be used.

Testing on the other hand is time-inexpensive and, if performed rigorously, can reveal more on how the application will perform in the "real world." Spending human, computer and tool resources on a testing strategy can reduce overall project costs.

1.1.3 For and Against Testing

The major argument against testing is that it can only show the presence of errors and never their absence [5] [6]. Testing cannot hope to settle the correctness problem unequivocally for the same reason there does not exist an algorithm to verify a program. This limitation has relegated testing in the past to a low position in the priorities of theoreticians [12]. Some have claimed that when one has given the proof of a program's correctness, one can dispense with testing altogether. In other words, since the goal of practical formal verification is to be able to prove all programs correct before they are put to use, when this goal is attained,

testing becomes obsolete [2].

Later work has shown the contrary. By constructing a theoretical framework for testing, results that affirm the necessity of testing in proving the correctness of a program have been derived. Most remarkable a work is "Toward a Theory of Test Data Selection" by Goodenough and Gerhart [11] held by Miller and Howden (among many others) as the first paper on software testing theory in [22]. Another work is "An Introduction to Proving the Correctness of Programs" by Hantler and King, a paper that shows that testing can be used to prove the first few base cases in an inductive proof of program correctness[13]. But even if it were true that testing can never find all the errors in a software system, it will always be a necessary verification technique, as has been established by today's academia and industry[33].

1.1.4 Software Testing Today

In the software engineering related literature, software testing nowadays still receives less attention than what is expected for an activity that accounts for almost half the costs of software development[2]. However, we are far from three decades ago when computer scientists simply did not bother to directly address testing issues [32]. So, what is the state of software testing today? We shall not attempt to give a full answer. Rather, we bring forth some observations.

- Software testing is still an art but is more of a science today and its automation cannot be too distant of a task as implied in early published work such as [30].
- Testing still depends on the development and domain expertise of test engineers and becomes increasingly difficult in proportion with the complexity of the program. However, we are advancing away from totally depending on the programmer's expertise in deciding when the program is sufficiently correct as argued in [4]. A whole science of systematic testing, measuring test progress, and field quality measurement has emerged and, though not mature, is far from infancy.
- Finally, on a disappointing note, although a large number of software testing techniques have been proposed during the last three decades, there has been surprisingly little concrete information about how effectively they detect errors [9], or locate faults — the very purpose of "modern-day" testing.

1.2 Black-Box versus White-Box Testing

No introductory discussion of software testing is complete without presenting a classification scheme of testing methods. This is mainly because this brings one closer to an understanding of what testing is all about. The most popular classification of software techniques is that which characterizes them as either black box or white box. Most testing strategies usually fall exclusively in one class or the other, but it is important to be aware that such is not always the case.

1.2.1 The Black Box Approach

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 [36] [25]. 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.

Howden in [16] mentions another weakness of the approach. He states that the disadvantage of black box approach is that it ignores important functional properties of the program, which are part of its design or implementation and which are not described in the requirements (on which black box tests are based)[1].

1.2.2 The White Box Approach

White box (also known as structural) strategies for testing are driven by the internal control structure of the program. White box testing, by taking the internal functional properties of the program into consideration when generating tests, constitutes an attempt to overcome this particular limitation of black-box testing [17].

There are several types of structural testing, including branch testing, control flow testing, data flow testing, slicing, and program dependency, just to name a few. Structural

[1] Howden has proposed a refined black box approach to testing that also makes use of structural information: functional program testing [17] [18] [19]. Some refer to black box testing as functional testing (as opposed to structural testing which is equivocal to white box testing). To avoid ambiguity, the term black box is used throughout this document.

testing is probably the most widely used class of program testing strategies [26]. It is interesting to note that very little has been written on black box testing in comparison to its white box counterpart, an approach that was supposed to be an improvement over black box testing [17].

White box testing is not without problems or limitations. Stocks and Carrington, in an elegant argument for specification-based testing, state that not only is it impossible to test without some sort of specification, but that any testing based only on program implementation is fundamentally flawed, as also argued by Goodenough and Gerhart in their landmark work cited earlier[31].

1.2.3 Current Trends

The popularity of structural testing has been consistent throughout the past years, but, particularly during the past decade, black box testing techniques have made a major comeback through statistical testing [40] [1], state-machine-based testing [3] [20] [10], and specification-based testing [7] [27] [28].

What the testing literature suggests is that there is no magic software testing technique — black box or white box. When it comes to uncovering errors and locating faults, techniques have been shown to be extremely efficient in certain situations and poor-performing in others.

Future testing will most probably feature the best of both the black box and white box paradigms. By combining the understanding of the requirements and specification and the actual behavior of the program, future techniques may very well reach the level of sophistication needed for them to be instrumental in proving the correctness of software.

1.3 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 [36] [37]. We describe each phase by its objectives and the artifacts it produces. We then briefly comment on how well the activities in each phase have been treated in the literature.

By lightly touching on what is involved in every phase, we hope to convey some of the difficulties — both practical and theoretical — encountered in the field.

1.3.1 Modeling Program Behavior

Objectives The first 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.

Artifacts

1. A document enumerating all the elements of software-user interaction.
2. 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 Markov chains and finite state machines in black box testing.

Comments Modeling is the most fundamental phase of any testing process, since the rest of the phases (selecting tests, running and evaluating tests, and measuring test progress) depend on the accuracy of its artifacts. A lot of work has been done on developing structural models of software, but in the black box testing arena, building models is one of the least addressed of all issues. Most works in testing assume the existence of a model based on which tests are to be generated, and it seems that it is the task of constructing the model that is almost always left for the test engineers in their individual projects.

1.3.2 Selecting Tests

Objectives We have mentioned how there are infinitely more tests than we can run on the program in black box testing. This also applies for white box testing of applications of sufficient complexity (that contain, for example, loops in their branches, paths, and control and data flow graphs). Therefore, in order to distinguish "interesting" tests from all others, certain test adequacy criteria have to be set.

Artifacts

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

Comments Selecting tests is not straightforward, and it is what "makes or breaks" a testing strategy. Most of the work in testing has addressed test selection with various objectives in mind (such as revealing bugs, covering code, etc. . .).

1.3.3 Running & Evaluating Tests

Objectives

Running a test involves figuring out how to simulate user action so that the software "thinks" 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. In addition, where such tools are not available, their own as a part of their individual testing endeavors. And writing code for the simulations is another feasible option, when tools are not available.

Evaluating a test involves verifying the test result (typically either output or variable values) against some sort of specification. Howden in [15] 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 been produced; or, whether the correct control sequences 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.

Artifacts

1. An input simulator that automatically executes tests.
2. An oracle.

Comments The oracle problem is another fundamental problem that has been addressed by theoreticians before [28]. Unfortunately very few of the approaches have turned out to be practical or inexpensive enough for the bulk of the software industry to adopt.

1.3.4 Measuring Test Progress

Objectives 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 (what percentage of tests will not cause the software to fail in the field).

Artifacts

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

Comments There is an immense body of literature on software metrics, a small percentage of which focus on testing metrics. A lot of the published work falls under software reliability engineering [23] [24] and related work.

1.4 Software Modeling for Black Box Testing

This work focuses on the modeling phase of black box testing. We have mentioned earlier that this area is addressed neither with the frequency nor the depth worthy of the difficulties encountered. This can be explained by the fact that not enough work has been written on black box testing, and what has been written focused on test selection and measuring field quality. From this perspective, the black box paradigm is a few years behind the white box one. It took quite a few years until serious work on constructing white box models (control, data and program dependency graphs) came into existence.

Another explanation to why the testing research community has avoided tackling this problem is that there is no unified or universal method of expressing requirements and specifications. Since these are the artifacts that black box testers study when constructing the model, establishing a generalized model-building strategy is understandably complicated. In comparison, programming languages have matured from a structural point of view. Hence, in a white box model, the same representation can be derived rather simply for equivalent control or data program components.

The first step towards resolving this problem would be to have a proper understanding of its details. In black box software-modeling, there is one central idea that needs to be understood and emphasized: state space. A state, from a black-box perspective, is a representation of how the software reacts to input and produces output. An error is detected if the software produces the wrong output for a certain sequence of inputs. This is verified

against specifications. This means that the software, due to either corrupt input, corrupt internal data or both, is in a state that it should not be in. Either it is in a legal state that it has been mistakenly driven into, or it is in an illegal state not described by the specification.

Understanding states is therefore crucial to any black box testing technique. Moreover, there is a need to distinguish the legal states of software form the illegal ones. Throughout this document, we shall call the set of legal states the state space.

In practice, as well as in scholarly journals, we have seen several examples of state space representations. Finite state machines have been used to generate tests in a variety of ways; a couple of interesting pointers can be found in [10] and [3]. Markov chains (particularly finite-state, discrete-parameter chains) have achieved recent popularity in the field for two reasons: the random variables of the chain are equivalent to states, and the statistical properties of Markov chains can be exploited in computing statistical software quality metrics. The fact that researchers have chosen state machines and their equivalents to work with is at least an unconscious knowledge of the centrality of states in software.

1.5 About This Work

We hope to have served the following points in our introduction:

- Testing is a difficult problem that has received attention in both academia and industry.
- One popular characterization of testing methodologies is to state whether one is black box or white box.
- Testing depends on the existence of some sort of model that encodes behavior of the system and guides the performance of all testing activities.
- It is essential that test engineers understand the state space of the software under test. This task is usually hindered by the fact that most documentation (particularly specifications) is not written in a form that can be easily mapped back to states and transitions.
- In order for testers of any software project to reap the benefits of a model-based testing strategy, a systematic way of constructing that model needs to be established.

This work expands on earlier efforts in stochastic software testing [36] with new observations leading to results that will take the modeling of software for black box testing one step closer to automation.

The objectives this thesis is hoped to meet can be summarized as follows:

Introduce Operational Modes A brief history of earlier work on operational modes is presented. An improved set of definitions is then suggested and supported with examples.

Describe a Model Construction Algorithm We present an algorithm that automates the construction of models of software behavior using the definitions we set forth concerning operational modes.

Discuss the Significance of This Work We discuss how this work contributes to software testing in general and software behavior modeling in particular. We then discuss areas of current and future research that spin off this work.

Chapter 2

An Introduction to Operational Modes

Although the importance of understanding states has not escaped the attention of software testing researchers, the problem of enumerating states has not been directly isolated and addressed before Whittaker's work in [36]. This chapter introduces the concept of an operational mode that constitutes the core of a novel, simple approach to abstract and then construct the state space of a software system. Operational modes are variables that collectively describe all the states of a system and can be used to reduce the tediousness of behavior modeling.

2.1 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. Let I and O be the set of all inputs and that of all outputs, respectively.

An input $i \in I$ 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 i is capable of generating it (in such a case i 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 $i \in I$ 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, unreachable inputs, by specification, never get processed by the system under test.

The **functional behavior** of a software system at a particular point of execution is the manner in which it responds to inputs in I (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.

2.2 A Brief History

Whittaker and Thomason in [40] define **usage variables** as data elements that are of significance when it comes to applying inputs. States of the Markov chains that they build in their example are constructed as combinations of values of these usage variables. Although they do not explain how these variables and their respective domains are designed, it is implicit that they are to be extracted from a functional specification of the system or any viable substitute. This is the first time operational modes are used in the manner described in the context of this paper.

Whittaker carries this work further in [36], where he proposes a systematic way of constructing a behavior model from operational modes. For a real application, the number of states that needs to be generated makes the construction cumbersome and labor-intensive. The major contribution of the paper was, in addition to directly addressing the concept of operational modes, the introduction of a hierarchical construction technique that builds the model level by level, each level corresponding to one operational mode. Whittaker does not discuss the statistics and is only interested in the structure of his Markov chain.

2.3 A Basic Definition of Operational Modes

We approach the definition of operational modes in a substantially more formal manner than in earlier work. Our treatment of the subject features a number of modifications to the theoretical ground on which operational modes stand. This will set the stage for the use of constraint satisfaction problems as an elegant formulation of the problem of automating behavior model construction.

An **operational mode** m is a variable that abstracts a partition of the collection of system variables that are responsible for all identifiable functional behavior. A variable of a software governs functional behavior at a particular point of execution if its value at the time dictates the manner in which the system reacts to user input by either not recognizing or ignoring the input, performing internal computation, or producing output in response. Such a variable is called a **functionally significant data element**. Denote by $\mathcal{M}$ the set of all operational modes.

The domain of each operational mode m, $domain(m)$, is a finite set of discrete values (also referred to as **modal values**), each of which is an abstraction of a distinct partition of possible combinations of values of the variables abstracted by that mode. The partitions abstracted by any two modal values should be such that they do not have identical effects on the system's functional behavior.

Example 1 Strictly Positive Integers

Consider the simple program written in BASIC-like pseudocode in figure 2.1. Let $Number$ with domain $\{StrictlyPositive, Other\}$ be a variable that abstracts $\{n\}$ such that $StrictlyPositive$ stands for all the possible values of n strictly greater than 0 and less than or equal to MAXINT and $Other$ stands for all other values. $Number$ satisfies the definition of an operational mode.

Consider another variable $Number$ of domain $\{-MaxInt, \ldots, 0, \ldots, MaxInt\}$ that abstracts $\{n\}$ at the lowest possible level of abstraction. $Number$ is not an operational mode since the values $-MaxInt, \ldots, 0$ have an identical effect on the program (and the same can be said about the values of $1, \ldots, MaxInt$). ■

2.4 Behavior Models

It follows from the definition of operational modes that they collectively describe the conditions under which software receives and internally responds to inputs. Operational modes

```
Sub main ()
    Dim n As Integer
    Input n
    If (n > 0) Then
      Print n; " is strictly positive."
    End If
End Sub
```

Figure 2.1: Strictly Positive Integers Program

constitute a good ground to build a description of applicable input strings and the state space of software.

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. In other words, we have to identify the combinations of variable-instantiations that, by specification, can be made simultaneously. We then have to determine the effect of every input of the system on every value in every combination (whether a mode changes to another value).

2.5 A Detailed Description of Modal Values

With this perspective on the problem, and with automation in mind, we have carried out an investigation of descriptors of modal values, a layer of detail that allows us to determine the relationship between the availability of an input and modal values. In addition, that same layer must allow us to determine the effect of every input (if any) on every modal value.

Upon investigation of a number of descriptors of modal values, some have turned out to be useful in verifying the extracted operational modes, providing a formal characterization of the values, and contributing to our definitions of operational states and behavior models.

A value v of an operational mode m can be described by:

- $I_n \subseteq I$, the set of all inputs that are necessarily applicable by the user when the mode has legally acquired v.
- $I_u \subseteq I$, the set of all inputs that are unreachable by the user when the mode has legally acquired v.
- $I_c \subseteq I$, the set of all **value-critical inputs** that, when applicable, and possibly under the condition that particular assignments to other modes have been made, will cause m to change to some other distinct value in the domain.
- $\varphi : I_c \times \prod_{m_i \neq m} domain(m_i) \longrightarrow domain(m) - \{v\}, i = 1, \ldots, |\mathcal{M}|$, is the **value-transition function**. φ deterministically describes the effect of a value-critical input on the value v of m, taking into consideration the values of all other modes m_i.

Notation 1 *Let v be a modal value; we write $v : I_n$, $v : I_u$, $v : I_c$, and $v : \varphi$ to denote the I_n, I_u, I_c and φ attributes of v.*

Notation 2 *Let v be a modal value of m; for clarity, we sometimes write this as $m = v$.*

Notation 3 *Consider the modal value $m = v$; if $v : \varphi(i_c, v', \ldots, v^{(|\mathcal{M}|-1)}) = w$ (where w is another value of m, and $v', \ldots, v^{(|\mathcal{M}|-1)}$ are the values of the other modes when m is v), we say that i_c changes v to w under the condition $(v', \ldots, v^{(|\mathcal{M}|-1)})$.*

Notation 4 *Consider the modal value $m_q = v$; if i_c changes v regardless of the values of the other modes $m_1, \ldots, m_{q-1}, m_{q+1}, \ldots, m_{|\mathcal{M}|}$ to w, we write $v : \varphi(i_c, \epsilon, \ldots, \epsilon, \epsilon, \ldots, \epsilon) = w$ or $v : \varphi(i_c) = w$ for simplicity. We also say that i_c changes v to w unconditionally.*

Example 2 Light Switch with a Lock

The inputs of the light switch system are $Invoke$, $Terminate$, $Switch$, and $Lock$. $Invoke$ starts the program, and $Terminate$ ends it. $Switch$ increases the intensity of light from one level to the next in the following sequence: off, dim, normal, bright. $Lock$ toggles the lock of the switch; when the switch is unlocked, $Switch$ can be applied; otherwise, $Switch$ is unreachable.

Consider the following functional specification of a light switch system in table 2.1. (An easy-to-understand tabular form is used to present the specification for the sake of illustration.) Assume the inputs are unreachable under conditions not described in the specification.

Input	**Condition**	**Output**	**Explanation**
Invoke	The program has not been started	Off;Unlocked	When the system starts the lights are off.
Terminate	The program has been started	Off;Unlocked	When the system is shutdown the lights are turned off and the lock is unlocked.
Switch	Switch is unlocked	Off/Dim/Normal/Bright	Switch to the next light intensity (from Off to Dim, Dim to Normal, and so on . . .) provided the light switch is unlocked
Lock	Switch is unlocked and the system has been started	Locked	Lock the light switch.
Lock	Switch is locked and the system has been started	Unlocked	Unlock the light switch.

Table 2.1: Light Switch Specification

By looking at the conditions in the specification table, the following three operational modes can be extracted: $SwitchSystem$ of domain $\{inactive, active\}$ (to distinguish between situations in which the system is not started and those in which it is up and running), $LightIntensity$ of domain $\{off, dim, normal, bright\}$ (to distinguish among situations in which the light intensity is off, dim, normal or bright respectively) ,and $LockStatus$ of domain $\{locked, unlocked\}$ (to differentiate situations in which the switch is locked and no change in light intensity can be made by the user, and all other situations).

When $SwitchSystem$ equals $inactive$, none of the inputs can be applied except for $Invoke$, which changes it to $active$ unconditionally. When $SwitchSystem$ is $active$, the inputs $Lock$ and $Terminate$ are always applicable, whereas $Invoke$ is not. $Terminate$ will unconditionally change $active$ to $inactive$. This can be written as follows:

$$SwitchSystem = inactive : \begin{pmatrix} I_n = \{Invoke\}, \\ I_u = \{Lock, Switch, Terminate\}, \\ I_c = \{Invoke\}, \\ \varphi(Invoke) = active \end{pmatrix}$$

$$SwitchSystem = active : \begin{pmatrix} I_n = \{Terminate, Lock\}, \\ I_u = \{Invoke\}, \\ I_c = \{Terminate\}, \\ \varphi(Terminate) = inactive \end{pmatrix}$$

The values of $LightIntensity$ do not constrain the applicability of any of the inputs except for $Invoke$ which can never be applied whenever $LightIntensity$ is dim, $normal$ or $bright$ since it is implicit in these cases that $SwitchSystem$ is $active$. The values govern how the system responds to the input $Switch$ by changing light intensity. $Switch$, whenever applicable, changes the value of $LightIntensity$ from off to dim, from dim to $normal$, from $normal$ to $bright$, and from $bright$ to off provided that the switch is not locked ($LockStatus = unlocked$). Finally, $Terminate$ changes dim, $normal$, and $bright$ to off unconditionally.

$$LightIntensity = off : \begin{pmatrix} I_n = \emptyset, \\ I_u = \emptyset, \\ I_c = \{Switch\}, \\ \varphi(Switch, LockStatus = unlocked) = dim \end{pmatrix}$$

$$LightIntensity = dim : \begin{pmatrix} I_n = \{Terminate, Lock\}, \\ I_u = \{Invoke\}, \\ I_c = \{Terminate, Switch\}, \\ \varphi(Terminate) = off, \\ \varphi(Switch, LockStatus = unlocked) = normal \end{pmatrix}$$

$$LightIntensity = normal : \begin{pmatrix} I_n = \{Terminate, Lock\}, \\ I_u = \{Invoke\}, \\ I_c = \{Terminate, Switch\}, \\ \varphi(Terminate) = off, \\ \varphi(Switch, LockStatus = unlocked) = bright \end{pmatrix}$$

$$LightIntensity = bright : \begin{pmatrix} I_n = \{Terminate, Lock\}, \\ I_u = \{Invoke\}, \\ I_c = \{Terminate, Switch\}, \\ \varphi(Terminate) = off, \\ \varphi(Switch, LockStatus = unlocked) = off \end{pmatrix}$$

Finally, when $LockStatus$ is $locked$, $Switch$ should not be applicable by specification. When its value is $unlocked$, $Switch$ is only applicable when the system is up and running (and that has nothing to do with $LockStatus$). $Lock$ toggles the value of $LockStatus$ which can be written as follows:

$$LockStatus : unlocked \begin{pmatrix} I_n = \emptyset, \\ I_u = \emptyset, \\ I_c = \{Lock\}, \\ \varphi(Lock) = locked, \end{pmatrix}$$

$$LockStatus : locked \begin{pmatrix} I_n = \emptyset, \\ I_u = \{Switch, Invoke\}, \\ I_c = \{Lock\}, \\ \varphi(Lock) = unlocked, \end{pmatrix}$$

■

2.6 The Phone Example

It is useful to look at another example of slightly greater complexity. Consider a software that simulates a phone device that communicates with other phone devices through a phone switch center (a user of that system). For simplicity, let us make the following assumptions:

- The users of the system are humans (who pick up and hang up the receiver and dial a number) and a phone switch (that informs the system: when there is an incoming call, when the party being dialed is busy, when the incoming call is canceled by the calling party, when the connected calling party disconnects, and when the device of the party being called is ringing)
- The phone switch only processes seven-digit phone numbers.
- Only two parties can be connected at the same time.
- Real-time issues, such as when the party being called hangs up at roughly the same time the phone places the call, are not considered.
- The human user dials a number of an existing party.
- Dialing digits is disabled after the device is informed that the party dialed is not available.

Refer to table 2.2 for a detailed listing and description of the inputs of the Phone System.

Example 3 The Phone System — Operational Modes

In what follows, we list and briefly discuss the operational modes of the phone system. A complete description of all the values follows.

1. $PhoneReceiver$ of domain $\{onhook, offhook\}$.

   $onhook$ describes the situation when the receiver is on the device's hook, and, therefore, certain inputs such as $DialDigit$ are unreachable. The only input the human user is necessarily able to apply whenever $PhoneReceiver$ is $onhook$ is $PickUp$. $offhook$ describes the other possible status of the receiver. When $PhoneReceiver$ is $offhook$, we have no indication of whether any of the system inputs is applicable (with the obvious exception of $PickUp$ and $HangUp$).

2. $DeviceStatus$ of domain $\{idle, ringing, dialtone, partyringing, connected, busysignal\}$.

   The values of $DeviceStatus$ distinguish among the following situations:

Input	**Description**
HangUp	The human user hangs up the receiver of the device.
PickUp	The human user picks up the receiver of the device.
DialDigit	The human user dials a digit that is assumed to get processed eventually by the phone switch user.
PartyNotAvailable	The switch informs the device that the party whose number has been supplied is not available (either busy or disconnected service situations).
PartyCalling	The device is informed that a party is calling.
CallingPartyCanceled	The device is informed that the party attempting to establish connection has cancelled its request (either by hanging up or being disconnected in some form).
PartyPickUp	The party being called picks up and establishes connection.
PartyHangUp	The calling party, having established connection, hangs up or terminates connection in some manner.
PartyDeviceRinging	The device of the party being called is ringing; connection can be established when the other party picks up.

Table 2.2: The Phone Inputs

(a) $DeviceStatus = idle$. *The device is idle (with its receiver on hook), and there is no party requesting a connection.* This situation necessitates that only $PickUp$ and $PartyCalling$ be applicable. When applied, $PickUp$ and $PartyCalling$ change this value to $ringing$ and $dialtone$, respectively.

(b) $DeviceStatus = ringing$. *The device's receiver is on hook, and there is a party that has already requested connection, so the device is ringing.* This situation necessitates that only $PickUp$ and $PartyCalling$ be applicable. $PickUp$ and $PartyCalling$ change $ringing$ to $connected$ and $idle$, respectively.

(c) $DeviceStatus = dialtone$. *The human user is about to start dialing digits, is in the process of doing so, or has just completed dialing a seven digit number but no indication on whether the party dialed is available has been received yet.* It is necessary, in this situation, that $HangUp$ and $DialDigit$ be applicable. No action by any party device trying to call is possible (so, $PartyCalling$ and $CallingPartyCanceled$ are unreachable). Also, since no connection has been established at this point, no action by the party being dialed is possible (thus, $PartyPickUp$ and $PartyHangUp$ are unreachable). The situation is changed if the human user hangs up, or (once all seven digits have been dialed) the switch applies either $PartyNotAvailable$ or $PartyDeviceRinging$.

(d) $DeviceStatus = partyringing$. *The party dialed is available for connection, and its device is in ringing status.* In this case, the inputs $HangUp$, $DialDigit$, and $PartyPickUp$ are necessarily applicable, and the rest of system's inputs are unreachable. $HangUp$ and $PartyPickUp$ change $partyringing$ to $idle$ and $connected$, respectively.

(e) $DeviceStatus = connected$. *The party dialed has picked up and connection has been established.* In this case, it is necessary that $HangUp$, $DialDigit$, and $PartyHangUp$ be applicable and the rest of the inputs unreachable. $HangUp$ and $PartyHangUp$ change $connected$ to $idle$ and $dialtone$, respectively.

(f) $DeviceStatus = busysignal$. *The party dialed has been reported by the switch not to be available; the device is busy until the human user hangs up.* $HangUp$ changes the value back to $idle$.

3. $DigitsDialed$ of domain $\{0, 1, 2, 3, 4, 5, 6, 7\}$. 0 collectively describes all possible situations in which either dialing a digit is not possible or it does not represent the dialing of any of the seven digits needed to establish a connection with a party. That includes both the *busy* and *connected* situations. $1, 2, 3, 4, 5, 6$ describe situations in which $1, 2, 3, 4, 5, 6$

digits have been dialed, respectively. 7 describes the case in which a seven-digit number has been dialed, and in which the device is awaiting one of the switch's possible inputs at this time: $PartyNotAvailable$ and $PartyDeviceRinging$.

$$PhoneReceiver = onhook : \left(\begin{array}{rcll} I_n & = & \{ & PickUp\}, \\ I_u & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyNotAvailable, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & PickUp\}, \\ \varphi(PickUp) = offhook & & & \end{array} \right)$$

$$PhoneReceiver = offhook : \left(\begin{array}{rcll} I_n & = & \{ & HangUp\}, \\ I_u & = & \{ & PickUp\}, \\ I_c & = & \{ & HangUp\}, \\ \varphi(HangUp) = onhook & & & \end{array} \right)$$

$$DeviceStatus = idle : \left(\begin{array}{rcll} I_n & = & \{ & PickUp, \\ & & & PartyCalling\}, \\ I_u & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyNotAvailable, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & PartyCalling \\ & & & PickUp\}, \\ \varphi(PartyCalling) = ringing & & & \\ \varphi(PickUp) = dialtone & & & \end{array} \right)$$

$$DeviceStatus = ringing : \left(\begin{array}{lcll} I_n & = & \{ & PickUp, \\ & & & CallingPartyCanceled\}, \\ I_u & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyNotAvailable, \\ & & & PartyCalling, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & PickUp, \\ & & & CallingPartyCanceled\}, \\ \varphi(PickUp) = connected & & & \\ \varphi(CallingPartyCanceled) = idle & & & \end{array} \right)$$

$$DeviceStatus = dialtone : \left(\begin{array}{lcll} I_n & = & \{ & HangUp, \\ & & & DialDigit\}, \\ I_u & = & \{ & PickUp, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyHangUp\}, \\ I_c & = & \{ & HangUp, \\ & & & PartyNotAvailable, \\ & & & PartyDeviceRinging\}, \\ \varphi(HangUp) = idle & & & \\ \varphi(PartyNotAvailable, DigitsDialed = 7) = busysignal & & & \\ \varphi(PartyDeviceRinging, DigitsDialed = 7) = partyringing & & & \end{array} \right)$$

$$DeviceStatus = partyringing : \left(\begin{array}{lcll} I_n & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyPickUp\}, \\ I_u & = & \{ & PickUp, \\ & & & PartyNotAvailable, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & HangUp, \\ & & & PartyPickUp\}, \\ \varphi(HangUp) = idle & & & \\ \varphi(PartyPickUp) = connected & & & \end{array} \right)$$

$$DeviceStatus = connected : \left(\begin{array}{lcll} I_n & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyHangUp\}, \\ I_u & = & \{ & PickUp, \\ & & & PartyNotAvailable, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & HangUp, \\ & & & PartyHangUp\}, \\ \varphi(HangUp) = idle & & & \\ \varphi(PartyHangUp) = dialtone & & & \end{array} \right)$$

$$DeviceStatus = busysignal : \left(\begin{array}{lcll} I_n & = & \{ & HangUp\}, \\ I_u & = & \{ & PickUp, \\ & & & DialDigit, \\ & & & PartyNotAvailable, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & HangUp\}, \\ \varphi(HangUp) = idle & & & \end{array} \right)$$

$$DigitsDialed = 0 : \left(\begin{array}{lcll} I_n & = & \emptyset, & \\ I_u & = & \{ & PartyNotAvailable, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & DialDigit\}, \\ \varphi(DialDigit, DeviceStatus = dialtone) = 1 & & & \end{array} \right)$$

$$DigitsDialed = k : \left(\begin{array}{lcll} I_n & = & \{ & HangUp, \\ & & & DialDigit\}, \\ I_u & = & \{ & PickUp, \\ & & & PartyNotAvailable, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ & & & PartyDeviceRinging\}, \\ I_c & = & \{ & DialDigit, \\ & & & HangUp\}, \\ \varphi(DialDigit) = k + 1 & & & \\ \varphi(HangUp) = 0 & & & \end{array} \right) ; k = 1, 2, 3, 4, 5, 6$$

$$DigitsDialed = 7 : \left(\begin{array}{lcll} I_n & = & \{ & HangUp, \\ & & & DialDigit, \\ & & & PartyNotAvailable, \\ & & & PartyDeviceRinging\}, \\ I_u & = & \{ & PickUp, \\ & & & PartyCalling, \\ & & & CallingPartyCanceled, \\ & & & PartyPickUp, \\ & & & PartyHangUp, \\ I_c & = & \{ & HangUp, \\ & & & PartyNotAvailable, \\ & & & PartyDeviceRinging\}, \\ \varphi(HangUp) = 0 & & & \\ \varphi(PartyNotAvailable) = 0 & & & \\ \varphi(PartyDeviceRinging) = 0 & & & \end{array} \right)$$

■

2.7 Notes on Operational Modes

An operational mode only partially describes the state of a software system. Therefore, an arbitrary input $i \in I$ is not necessarily in either of I_n or I_u. In the phone example, the inputs *PartyCalling* and *CallingPartyCanceled* are neither in the I_n nor the I_u attributes of $PhoneReceiver = onhook$. This is because it is the values of *DeviceStatus* that dictate whether these inputs are applicable. Also one could easily think of examples in which a value-critical input can be affected by its membership in the I_u attribute of some value of a different mode. For example, in the light switch system, *Switch* is in $LightIntensity = off : I_c$. However, if the mode *SwitchSystem* is *inactive*, *Switch* will not affect $LightSwitch = off$ simply because it is also in $SwitchSystem = inactive : I_u$. On the other hand, if an input is in $I_n \cap I_c$, then it will always be applicable and critical to the value whenever it is legally acquired. In the phone example, *PartyHangUp* is in both the I_c and the I_n attributes of $DeviceStatus = connected$. In other words, regardless of the values of *PhoneReceiver* and *DigitsDialed*, *PartyHangUp* will always be applicable whenever $DeviceStatus = connected$ and it will always change that value to *dialtone* if it gets applied.

Finally the values of an operational mode m are always such that either

1. the modal values $v_1, \ldots, v_{|domain(m)|}$ have mutually different I_n sets,
2. the modal values $v_1, \ldots, v_{|domain(m)|}$ have mutually different I_u sets,
3. it must be that either
   (a) $v_1 : I_c, \ldots, v_{|domain(m)|} : I_c$ are pairwise unequal or,
   (b) for a pair of values of m, v_i and v_j, if $v_i : I_c = v_j : I_c$, then for at least one i_c in that set, $v_i : \varphi$ maps i_c to a different value from (or to the same value under different conditions than) $v_j : \varphi$, or
4. any combination of 1, 2, and 3.

Chapter 3

Building Models of Software Behavior

The main objective of this thesis is to present a technique to enumerate the state space of a software system. Building on the material of the previous chapter, we derive some results including that the state enumeration problem is a constraint satisfaction problem that can be solved automatically. We briefly introduce the concept of a constraint satisfaction problem, present a state generation algorithm, and then extend it to generate the state transitions.

3.1 Constraint Satisfaction Problems

A **constraint satisfaction problem** [34] [35] or **CSP** is a triplet (X, C, O), where X is a set of variables with possibly different domains, and C is a set of constraints or rules governing the assignment of values to these variables. Typically, the objective, O, is to find the first, best, or all solutions (i.e. combinations of variable-assignments) that satisfy the constraints in C. Let $SOLUTIONS(P)$ denote the set of all tuple solutions to a CSP P; then

$$SOLUTIONS(P) = \{sol \in \prod_{x \in X} domain(x) | \forall c \in C, sol \text{ satisfies } c\}$$

Example 4 Couples of Distinct Digits

Let P be the problem of generating all the couples of distinct ternary (base 3) digits (i.e. 0, 1, and 2). P can be formulated as a constraint satisfaction problem as follows:

1. The set of variables is $X = \{firstdigit, seconddigit\}$, where both variables have the domain $\{0, 1, 2\}$.

2. The assignment of $firstdigit$ and $seconddigit$ is constrained by that $firstdigit \neq seconddigit$. One way to write C is for it to be the set of all impossible solutions to P.
3. The objective is to find all solutions.

We write

$$P = (\{firstdigit, seconddigit), \{(d, d) | d \in \{0, 1, 2\}\}, \text{'all solutions'})$$

$$SOLUTIONS(P) = \{(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)\}$$

■

There is a wealth of algorithms and implementations to solve CSP's that are widely used by the artificial intelligence community. (Tsang's "Foundations of Constraint Satisfaction" [34] is one example book that can orient the reader into the basics of CSP's and CSP-solver algorithms.) We can then be assured that an automated solution can be found to any problem that can be formulated as a constraint satisfaction problem.

3.2 State Generation Problem

The problem of enumerating all the legal states of a software system from its operational modes is a constraint satisfaction problem. The set of variables is the set of operational modes. The variable assignment is constrained by the fact that some combinations of values cannot exist in a legal state. The objective, of course, is to generate all possible solutions — i.e., all the states — to the problem. We call this the **state generation constraint satisfaction problem** or **SGCSP**.

We now can define a **software operational state** in terms of what we just discussed as an element of $SOLUTIONS(SGCSP)$. The **operational state space** is exactly $SOLUTIONS(SGCSP)$.

Our early discussion of behavior models (section 2.4) implies that any definition of a state should be such that not only every state represents exactly one functional behavior, but that there is one and only one state for each functional behavior. Proving the following claim establishes this fact for our definition of a software operational state.

Notation 5 *If a value v is one of the values constituting a state s in $SOLUTIONS(SGCSP)$, we say that v is in s.*

Claim 1 *For every possible functional behavior of a software system, there exists exactly one descriptive operational state as defined above.*

Proof

We proceed to prove this claim by showing the existence of an operational state for every functional behavior and then the uniqueness of such a state.

1. *Existence (There exists at least one state)*

   Assume there exists no state for the behavior. This implies that there is no combination of modal values that reflects that behavior. Consequently, there is no possible internal data element(s) that would cause such a behavior (since, by definition, operational modes collectively abstract all such internal data elements). We could only infer one of two things: either the functional behavior is impossible (in which case the assumption is contradicted), or there exists some operational mode or modal-value detail that accounts for this behavior and that is missing from the test design (in which case the claim is out of context).

   By contradiction, there exists at least one operational state for the functional behavior.

2. *Uniqueness (There exists at most one state)*

   Assume there are two or more states, and hence value-combinations in $SOLUTIONS(SGCSP)$ for the same behavior. This implies that some modes have more than one value for which the system will behave in the same exact manner — given a combination of fixed values for the other modes. By definition, no two values can contribute to the same behavior under the same conditions. Therefore, the two different states are identical, which contradicts the assumption.

   By contradiction, there exists at most one operational state for the functional behavior.

It follows from 1 and 2 that for every functional behavior of a software system there exists exactly one operational state. ■

3.3 An Example State Generation Algorithm

Let $S = SOLUTIONS(SGCSP)$, the set of states. We present algorithm 1 to show that the automatic generation of the set of states S from $\mathcal{M}$ is feasible (a result used in

the behavior model constructor algorithm presented later in this chapter). The algorithm assumes a $SGCSP$ formulated as $(\mathcal{M}, C, \text{'all solutions'})$, where C is the set of all tuples of modal values that cannot coexist in a state in S.

Notation 6 *Let A and B be two sets. Denote by $A - B$ the difference of the set B from set A, the set of all elements in A but not in B.*

Algorithm 1 A Simple State-Generator

- Input: $SGCSP = (\mathcal{M}, C, \text{'all solutions'})$
- Output: State Set S.

1. Let $S = \prod_{m \in \mathcal{M}} domain(m)$
2. For every s in S, if there are two or more values in s that constitute a tuple in C then $S = S - \{s\}$.

■

Example 5 Generating the States of Light Switch System Using Algorithm 1

$$SGCSP \text{ is } \begin{pmatrix} \{SwitchSystem, LightIntensity, LockStatus\} \\ \{(SwitchSystem = inactive, LightIntensity = dim), \\ (SwitchSystem = inactive, LightIntensity = normal), \\ (SwitchSystem = inactive, LightIntensity = bright), \\ (SwitchSystem = inactive, LockStatus = locked)\} \\ \text{'all solutions'} \end{pmatrix}$$

1. $S = domain(SwitchSystem) \times domain(LightIntensity) \times domain(LockStatus)$
2. S starts with $2 \times 4 \times 2 = 16$ elements. After removing every element $s \in S$ such that there is a couple $(v, v') \in C$ and both v and v' are in S (such as $(inactive, dim, unlocked)$), we end up with the following states in S:
   (a) (inactive, off, unlocked)
   (b) (active, off, unlocked)
   (c) (active, dim, locked)
   (d) (active, normal, unlocked)
   (e) (active, bright, locked)
   (f) (active, off, locked)

(g) (active, dim, unlocked)

(h) (active, normal, locked)

(i) (active, bright, unlocked)

■

3.4 Behavior Models

A behavior model is the couple (S, δ), where $S = SOLUTIONS(SGCSP)$ is the state space and $\delta : S \times I \longrightarrow S$ is the deterministic transition function of the model.

By looking at the examples presented so far, we notice that extracting operational modes is not overly complicated. Also, the automation of state generation is now only dependent on determining the constraints among modal values. This is a major step up from manually determining the states. Nevertheless, the task of manually enumerating the constraints is still marginally tedious.

We have introduced the attributes of values with the purpose of improving on our ability to determine the constraints of SGCSP. We will next present a way to automate the generation of these constraints.

The **binary modal-value constraint**, $\otimes$, is a subset of $\cup_{i<j} domain(m_i) \times domain(m_j)$, where $i = 1, \ldots, |\mathcal{M}| - 1$ and $j = 2, \ldots, |\mathcal{M}|$. $(m = v, m' = v') \in \otimes$ if and only if there is no operational state s such that both v and v' are in s. From a black box perspective, such a situation arises only when $v : I_n \cap v' : I_u \neq \emptyset$ or $v : I_u \cap v' : I_n \neq \emptyset$.

Conflicting Couples of Values		The Inputs Causing the Conflicts in	
v	v'	$v : I_n$	$v' : I_n$
inactive	*dim*	*Invoke*	*Lock*, *Terminate*
inactive	*normal*	*Invoke*	*Lock*, *Terminate*
inactive	*bright*	*Invoke*	*Lock*, *Terminate*
inactive	*locked*	*Invoke*	None

Table 3.1: Constraints of the Light Switch

Conflicting Couples of Values		The Inputs Causing the Conflicts in	
v	v'	$v : I_n$	$v' : I_n$
onhook	*dialtone*	$PickUp$	$HangUp$, $DialDigit$
onhook	*partyringing*	$PickUp$	$HangUp$, $DialDigit$, $PartyPickUp$
onhook	*connected*	$PickUp$	$HangUp$, $DialDigit$, $PartyHangUp$
onhook	*busysignal*	$PickUp$	$HangUp$
onhook	1,2,3,4,5,6	$PickUp$	$HangUp$, $DialDigit$
onhook	7	$PickUp$	$HangUp$, $DialDigit$, $PartyNotAvailable$, $PartyDeviceRinging$
$offhook$	*ringing*	$HangUp$	$PickUp$
$offhook$	*idle*	$HangUp$	$PickUp$
idle	1,2,3,4,5,6	$PickUp$, $PartyCalling$	$HangUp$, $DialDigit$
idle	7	$PickUp$, $PartyCalling$	$HangUp$, $DialDigit$, $PartyNotAvailable$, $PartyDeviceRinging$

continued on next page

continued from previous page

Conflicting Couples of Values		The Inputs Causing the Conflicts in	
v	v'	$v : I_n$	$v' : I_n$
ringing	1,2,3,4,5,6	$PickUp$, $CallingPartyCanceled$	$HangUp$, $DialDigit$
ringing	7	$PickUp$, $CallingPartyCanceled$	$HangUp$, $DialDigit$, $PartNotAvailable$, $PartyDeviceRinging$
partyringing	1,2,3,4,5,6	$PartyPickUp$	None
partyringing	7	$PartyPickUp$	$PartyNotAvailable$, $PartyDeviceRinging$
connected	1,2,3,4,5,6	$PartyHangUp$	None
connected	7	$PartyHangUp$	$PartyNotAvailable$, $PartyDeviceRinging$
busysignal	1,2,3,4,5,6	None	$DialDigit$
busysignal	7	None	$DialDigit$, $PartyNotAvailable$, $PartyDeviceRinging$

Table 3.2: Constraints of the Phone Device

3.5 The Construction Algorithm

We proceed by formulating the state generation constraint satisfaction problem using the $\otimes$ as the set of constraints, thus obtaining the state set S. Next the algorithm iterates on every state s, determining the transition function δ that maps s to other states based on the attributes of all the values in that state.

Notation 7 *Let $s : I_u$, the set of all inputs unreachable from state s, be the union of all the unreachable-set-attributes of all the values in s, $\cup_{v \ in \ s} v : I_u$.*

Notation 8 *Let $s : I_c$ denote the set of all the state-changing inputs. $s : I_c$ is computed as $\cup_{v \ in \ s} v : I_c - s : I_u$.*

Notation 9 *Let $s : I_l$ denote the set of all inputs that do not change state s. $s : I_l$ is computed as $I - s : I_c - s : I_u$.*

Algorithm 2 Behavior Model Constructor

- Input: Operational Mode Set $\mathcal{M}$.
- Output: Behavior Model (S, δ).

1. Formulate $SGCSP$ as $(\mathcal{M}, \otimes,$ 'all solutions').
2. *Generate the states.* Using a CSP-solver, such as algorithm 1 or simple backtracking, say, generate $S = SOLUTIONS(SGCSP)$.
3. *Generate the transitions.* For every state $s = (v_1, \ldots, v_{|\mathcal{M}|}) \in S$, perform steps 4, 5, and 6, thus generating δ.
4. *Define the impossible transitions.* An input that is unreachable from the state should not affect the system in that state in any way; in other words, it is impossible to compute a corresponding transition.
   $\forall i \in s : I_u$,
   $$\delta(s, i) = \epsilon.$$
5. *Define the loop transitions.* If an input is neither in the unreachable set of the state, nor critical to any value of the state, then a loop transition must be computed for that input.
   $\forall i \in s : I_l$,
   $$\delta(s, i) = s.$$
6. *Define the state-changing transitions.* Consider every input that is not unreachable from the state currently under investigation and that is critical to one of the values in that same state. Take into consideration all the values' value-transition functions to determine the effect of that input on

the values and therefore the states:

$\forall i \in s : I_c$,

$$\delta(s, i) = (w_1, \ldots, w_{|\mathcal{M}|})$$

where

$$w_k = \begin{cases} v_k : \varphi(i, v_1, \ldots, v_{k-1}, v_{k+1}, \ldots, v_{|\mathcal{M}|}) & \text{if } i \in v_k : I_c \\ v_k & \text{otherwise} \end{cases}$$

$$k = 1, \ldots, |\mathcal{M}|$$

■

3.6 How the Algorithm Works in the Phone Example

The first three steps are simple enough:

1. $SGCSP$ is $\left(\begin{array}{l} \{PhoneReceiver, DeviceStatus, DigitsDialed\} \\ \{(onhook, dialtone), (onhook, partyringing), (onhook, connected), \\ (onhook, busysignal), (onhook, k), (offhook, idle), \\ (offhook, ringing), (idle, k), (ringing, k), \\ (partyringing, k), (connected, k), (busysignal, k)\} \\ \text{'all solutions'} \end{array} \right)$

   where $k = 1, \ldots, 7$.

2. The thirteen states in S are:

   (a) (onhook,idle, 0)

   (b) (onhook,ringing, 0)

   (c) (offhook,dialtone, 0)

   (d) (offhook,dialtone, 1)

   (e) (offhook,dialtone, 2)

   (f) (offhook,dialtone, 3)

   (g) (offhook,dialtone, 4)

   (h) (offhook,dialtone, 5)

   (i) (offhook,dialtone, 6)

   (j) (offhook,dialtone, 7)

   (k) (offhook,partyringing, 0)

(l) (offhook,connected, 0)

(m) (offhook,busysignal, 0)

We give three examples of generating transitions: from $(onhook, idle, 0)$, from $(onhook, ringing, 0)$, and from $(offhook, dialtone, 7)$. Steps 4 and 5 of the algorithm are straightforward. We elaborate on step 6:

$s = (onhook, idle, 0)$

- $s : I_u = \{HangUp\ DialDigit, PartyNotAvailable, PartyPickUp, PartyHangUp, PartyDeviceRinging, CallingPartyCanceled\}$
- $s : I_l = \emptyset$
- $s : I_c = \{PickUp, PartyCalling\}$

1. $PickUp \in onhook : I_c$ and $PickUp \in idle : I_c$.

$$\delta((onhook, idle, 0), PickUp) = (offhook, dialtone, 0)$$

2. $PartyCalling \in idle : I_c$.

$$\delta((onhook, idle, 0), PartyCalling) = (onhook, ringing, 0)$$

$s = (onhook, ringing, 0)$

- $s : I_u = \{HangUp\ DialDigit, PartyNotAvailable, PartyPickUp, PartyHangUp, PartyDeviceRinging, PartyCalling\}$
- $s : I_l = \emptyset$
- $s : I_c = \{PickUp\ CallingPartyCanceled\}$

1. $PickUp \in onhook : I_c$ and $PickUp \in ringing : I_c$.

$$\delta((onhook, ringing, 0), PickUp) = (offhook, connected, 0)$$

2. $CallingPartyCanceled \in ringing : I_c$.

$$\delta((onhook, ringing, 0), CallingPartyCanceled) = (onhook, idle, 0)$$

$s = (offhook, dialtone, 7)$

- $s : I_u = \{PickUp, PartyCalling, CallingPartyCanceled, PartyHangUp, PartyPickUp\}$
- $s : I_l = \{DialDigit\}$

- $s : I_c = \{HangUp,\ PartyNotAvailable,\ PartyDeviceRinging\}$

1. $HangUp \in offhook : I_c$, $HangUp \in dialtone : I_c$, and $HangUp \in 7 : I_c$.

$$\delta((offhook, dialtone, 7), HangUp) = (onhook, idle, 0)$$

2. $PartyNotAvailable \in dialtone : I_c$ and $PartyNotAvailable \in 7 : I_c$.

$$\delta((offhook, dialtone, 7), PartyNotAvailable) = (offhook, busysignal, 0)$$

3. $PartyDeviceRinging \in dialtone : I_c$ and $PartyDeviceRinging \in 7 : I_c$.

$$\delta((offhook, dialtone, 7), PartyDeviceRinging) = (offhook, partyringing, 0)$$

The completely generated transition function δ is displayed in tables 3.3 and 3.4.

	HangUp	*PickUp*	*DialDigit*	*PartyNotAvailable*	*PartyCalling*
onhook *idle* 0	ϵ	*offhook* *dialtone* 0	ϵ	ϵ	*onhook* *ringing* 0
onhook *ringing* 0	ϵ	*offhook* *connected* 0	ϵ	ϵ	ϵ
offhook *dialtone* 0	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 1	ϵ	ϵ
offhook *dialtone* 1	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 2	ϵ	ϵ
offhook *dialtone* 2	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 3	ϵ	ϵ
offhook *dialtone* 3	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 4	ϵ	ϵ
offhook *dialtone* 4	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 5	ϵ	ϵ
offhook *dialtone* 5	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 6	ϵ	ϵ
offhook *dialtone* 6	*onhook* *idle* 0	ϵ	*offhook* *dialtone* 7	ϵ	ϵ
offhook *dialtone* 7	*onhook* *idle* 0	ϵ	loop	*offhook* *busysignal* 0	ϵ
offhook *partyringing* 0	*onhook* *idle* 0	ϵ	loop	ϵ	ϵ
offhook *connected* 0	*onhook* *idle* 0	ϵ	loop	ϵ	ϵ
offhook *busysignal* 0	*onhook* *idle* 0	ϵ	ϵ	ϵ	ϵ

Table 3.3: The Phone Device Transition Function

	$CallingPartyCanceled$	$PartyPickUp$	$PartyHangUp$	$PartyDeviceRinging$
$onhook$ $idle$ 0	ϵ	ϵ	ϵ	ϵ
$onhook$ $ringing$ 0	$onhook$ $idle$ 0	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 0	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 1	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 2	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 3	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 4	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 5	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 6	ϵ	ϵ	ϵ	ϵ
$offhook$ $dialtone$ 7	ϵ	ϵ	ϵ	$offhook$ $partyringing$ 0
$offhook$ $partyringing$ 0	ϵ	$offhook$ $connected$ 0	ϵ	ϵ
$offhook$ $connected$ 0	ϵ	ϵ	$offhook$ $dialtone$ 0	ϵ
$offhook$ $busysignal$ 0	ϵ	ϵ	ϵ	ϵ

Table 3.4: The Phone Device Transition Function

Chapter 4

Conclusions

4.1 Summary

Testers face difficulties as they perform the various tasks during software testing. The behavior modeling phase for black box testing is particularly labor-intensive, often hindered by lack of proper and complete specification. The problem most central to the behavior modeling phase in black box testing is that of enumerating the state space. Operational modes can help testers address this issue.

Operational modes are variables that abstract the internal data elements responsible for the behavior of a software system. Operational modes are used to build a model of that behavior from which we can designate the applicable input strings (and the illegal ones as well). Using the formulation of constraint satisfaction problems in addition to binary constraint relationships, the problem of generating states is rendered automatable, given the set of the system's operational modes. We presented an algorithm that automates the generation of the states and state-transitions of the behavior model based on descriptive attributes of the modal values.

The following steps summarize the procedure to model a software system using operational modes:

1. Observe all available documentation including the specification, if available.
2. Enumerate the system's inputs.
3. Extract the system's operational modes.
4. Apply the algorithm of section 3.5 to the set of operational modes, which results in the system's behavior model.

4.2 Significance and Expected Impact

The contribution of this work can be summarized in the following points:

Operational Modes More Formally Defined The definition of operational modes presented here was more precise in linking the modes to functional behavior, applicability and effect of applicable inputs, in addition to behavior models. In particular the verification rules of section 2.7 on page 26 carry the artful work of testers one step closer to scientific method.

Relationship Between States and Functional Behavior Better Understood Defining functional behavior in the way presented in this paper makes such concepts as states, state space, and behavior models much more intuitive, since there is a one-to-one correspondence between functional behavior and states of the system.

Behavior Modeling Less Labor-Intensive The behavior model is of practical importance to testers since it specifies the input sequences that testers must consider. Without automation such models can be too time consuming to develop. Practitioners often work with abstracted models and fail to fully define expected behavior. Algorithm 2 allows a fully detailed model to be generated, decreasing the tester's reliance on abstractions.

There are also a number of research projects that remain to be conducted.

Identifying Operational Modes Still an Art There is no formal way of going about extracting operational modes. Clearly, the definition of operational modes and their values can yield a number of guidelines by which a variable can be verified to be an operational mode. The process of going about the review of specification and other relevant documentation and figuring out operational modes of a system in the absence of straight-to-the-point data specifications is still an adhoc one. However, the art of operational modes is not unattainable by the average tester and is indeed a practical approach to constructing a model even in the most inadequate of circumstances.

Soundness and Completeness of the Algorithm It can be argued that the behavior model constructor will not result in a situation in which s, an element of $\prod_{m \in \mathcal{M}} domain(m) - SOLUTIONS(SGCSP)$ is a legal operational state of a software described by $\mathcal{M}$ (the binary operational constraint is sound). However, there is no ground for assuming the completeness of the algorithm mainly because the definition of operational modes does not guarantee the extraction of all operational modes. The

constraints, however, can be argued to be complete from a functional point of view. In other words, if the set of modes $\mathcal{M}$ is not complete, then so is S and δ — the result of the algorithm.

Working With Modal-Value Attributes Rather Tedious Some may argue that specifying the attributes may be too much work for testers — mainly because it is almost equivalent to writing the specification of the subset of behavior being tested. The tediousness can be significantly relieved by creating a simple tool that makes the process of attribute-data entry easier. Further, this still involves several orders of magnitude less work than constructing states based on individual testing projects and then figuring out the transitions to build the model. Using the algorithm and the attributes achieves time gains over earlier endeavors with usage variables [40] and operational modes [36].

4.3 Prospects for Future Work

Future work that is immediately related to this thesis includes the following:

Generating Markov Chains The next step in our research agenda is to automate the generation of probabilities based on an operational profile. Exploiting the simplification offered by operational modes is speculated to be greatly advantageous. In our experiments that accompanied this research, we have used only uniform probabilities for our Markov chains. We are investigating the use of CSP's to generate non-uniform probability assignments. This will be particularly useful in applications for which significant prior usage data exists.

More Precise Rules for Extracting Operational Modes The completeness issues discussed earlier can be resolved upon the discovery of a more precise definition of operational modes that allows even the inexperienced tester to work with modes. We are currently investigating further descriptors of operational modes in addition to informal guidelines to extract modes from software documentation.

Verification of Generated Behavior Model The correctness of the model generated is greatly dependent on the rather large amount of data that has to be artfully extracted and manually fed as input to the algorithm. In order for testers to proceed with trust, verification techniques need to be developed to assess whether the resultant behavior model is the one desired. Discrepancies include unreachable states and sink

non-terminal states among others, and all of them should be traced to the operational modes documented in the test design.

Static Comparative Analysis of Behavior and Implementation Models One interesting research direction that we are investigating is the extraction of operational modes of both the software specification and implementation, thus producing models of the supposed and actual behavior of the software. A static comparative analysis of the two models may lead us to estimate the distance between the models. In other words, this may allow us to produce a rather-confident answer to the question: how far is the software from its specifications?

In addition to this, the work on operational modes and the construction algorithm has spawned work in the following areas:

Test Case Adequacy Criteria Random testing is inexpensive. It is also a waste of effort and time that can be spent on building better tests instead of repetitive ones. Whittaker and Al-Ghafees have achieved significant progress on methodical test case selection with criteria depending on the content of states — modal values. Early results are found in [38].

Guidelines for Better Software Design Working with operational modes forces us to think like developers. If a system under test were designed and implemented with operational modes as the back-bone of specification, most of the errors would be traced back to either missing operational modes or missing or ill-described values. In fact, it turns out that designing software with operational modes helps prevent certain classes of errors. More of this work can be found in [39].

Bibliography

[1] Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions on Software Engineering*, 21(9):705–715, September 1995.

[2] Doug Bell, Ian Morrey, and John Pugh. *Software Engineering: A Programming Approach*. Prentice Hall, second edition, 1992.

[3] Tsun S. Chow. Testing design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[4] Lori Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[5] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Number 2 in APIC Studies in Data Processing. Academic Press, 1972.

[6] E.W. Dijkstra. Structural programming. In *Software Engineering Techniques*, pages 84–88. Buxton and Randell, 1969.

[7] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):483–444, July 1984.

[8] Institute for Electrical and Electronics Engineers. *IEEE Standards Collection: Software Engineering*. The Institute of Electrical and Electronics Engineers, Inc., New York, New York, 1993.

[9] Phyllis G. Frankl and Elaine J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[10] Susumu Fujiwara, Gregor v. Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[11] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[12] John S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, September 1983.

[13] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys*, pages 331–353, September 1976.

[14] Constance Heitmeyer, James Kirby, and Bruce Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of the Annual Conference on Computer Assurance (COMPASS '97)*, 1997.

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

[16] William E. Howden. Functional program testing. *IEEE Transactions on Software Engineering*, 6(2):162–169, March 1980.

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

[18] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[19] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, 1987.

[20] J.C. Huang. State constraints and pathwise decomposition of programs. *IEEE Transactions on Software Engineering*, 16(8):880–896, August 1990.

[21] Edward Miller. Introduction to software testing technology. In *Tutorial: Software Testing & Validation Techniques*, pages 4–16. IEEE Computer Society Press, 1981.

[22] Edward Miller and William E. Howden. *Tutorial: Software Testing & Validation Techniques*. IEEE Computer Society Press, second edition, 1981.

[23] John D. Musa. A theory of software reliability and its applications. *IEEE Transactions on Software Engineering*, 1(3):312–327, September 1975.

[24] John D. Musa. Software-reliability-engineered testing. *IEEE Computer*, 29(11):61–68, November 1996.

[25] Glenford J. Myers. *The Art of Software Testing*. Wiley, 1979.

[26] Simeon C. Ntafos. A comparison of some structural testing strategy. *IEEE Transactions on Software Engineering*, 14(6):868–874, June 1988.

[27] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[28] Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[29] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, fourth edition, 1997.

[30] C.V. Rammamoorthy, K.H. Kim, and W.T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, 1(4):403–425, December 1975.

[31] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[32] Leon G. Stucki. A case for software testing. *IEEE Transactions on Software Engineering*, 2(3):194, September 1976.

[33] A. S. Tanenbaum. In defense of program testing or correctness proofs considered harmful. *ACM SIGPLAN Notices*, 11(5):64–68, May 1976.

[34] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[35] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[36] James A. Whittaker. Stochastic software testing. *Annals of Software Engineering*, 4:115–131, August 1997.

[37] James A. Whittaker. Software testing: What it is, and why it is so difficult. To appear in IEEE Software, 1999.

[38] James A. Whittaker and Mohammad Al-Ghafees. Test case adequacy criteria. Paper final writeup in progress, 1999.

[39] James A. Whittaker and Alan Jorgensen. Why software fails. To Appear in ACM Software Engineering Notes, 1999.

[40] James A. Whittaker and Michael G. Thomason. A markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, October 1994.