Markov Chain-based Test Data Adequacy Criteria: a Complete Family

Mohammed Al-Ghafees and James A. Whittaker
Florida Institute of Technology, FL, USA

malghafees@hotmail.com jw@cs.fit.edu

Abstract

The idea of using white box data flow information to select test cases is well established and has proven an effective testing strategy. This paper extends the concept of data flow testing to the case in which the source code is unavailable and only black box information can be used to make test selection decisions. In such cases, data flow testing is performed by constructing a behavior model of the software under test to act as a surrogate for the program flow graph upon which white box data flow testing is based. The behavior model is a graph representation of externally-visible software state and input-induced state transitions. We first summarize the modeling technique and then define the new data flow selection rules and describe how they are used to generate test cases. Theoretical proof of concept is provided based on a characteristic we call *transition variation*. Finally, we present results from a laboratory experiments in which we compare the fault detection capability of black box data flow tests to other common techniques of test generation from graphs, including simple random sampling, operational profile sampling and state transition coverage.

Keywords: Behavior model, operational profile, random testing, software testing, test data adequacy criteria, transition variation.

Introduction

From a white-box, code-based perspective, data and data-flow are easily defined. *Data* is represented by program variables and data structures; *data flow* is defined by uses of data (e.g., predicates) and changes to data values (e.g., assignments).

Data flow is useful to software testers to help them prioritize code paths for testing purposes. The general idea is that paths which define and modify data are higher priority paths to test because manipulating data forces the software to exhibit its functionality. Indeed, this is intuitively very pleasing: since fundamentally software stores and manipulates data, then it makes sense to consider data flow when selecting test cases. As long as the set of paths selected cause all program data to be initialized and used, then one has a certain degree of confidence in the completeness of the test. Untested paths will not contain untested data.

Laski and Corel [7] defined the first testing strategy based on data flow as an alternative to control flow strategies which where prevalent at the time [8]. Later Rapps and Weyuker [13] proposed a number of test selection criteria so that test cases could be chosen based on their ability to define and modify program data. A number of subsequent analyses and case studies have served to mainstream white box, data flow testing [3,5,15,16].

This paper defines a new family of data flow testing criteria inspired by the work cited above. However, we treat the case that source code is unavailable for consideration and test selection must be based only on external information: the interface, inputs and outputs. Thus, we must first define exactly what we mean by *data* and *data flow* from a black box perspective before defining the selection criteria.

Most testers are comfortable with the notion of the *program flow graph* [8] in which source code is modeled as a directed graph where nodes are defined by individual (or blocks of) program statements and edges by control flow changes due to looping and branching structures. White box data flow criteria fit easily into this model by defining, for example, subpaths of the flow graph that begin at a node in which data gets initialized and end at a node in which the same data gets used or modified. Thus, paths through the model define sequences of source statements that can be executed in a target environment. Indeed, the flow graph model defines only paths that can be executed and rules out statement sequences that are not realizable.

We extend this graphical concept to external program behavior by defining a *behavior model* which describes sequences of inputs that can be executed (in order) against the software under test. Like the program flow graph, behavior models describe only sequences that are realizable and rule out impossible input sequencing (like pressing a button on a dialog box before the box appears on the screen). For behavior models we define nodes to be a collection of variables which describe the state of the application under test and the edges to be inputs which can be applied at a particular state. Thus, data flow path selection criteria can be made to apply to black box testing by treating the variables that define software states exactly as program variables are treated in white box testing.

Methods for identifying external data flow and constructing a behavior model are presented in section 0 along with an illustrative example. We conjecture that improvements in overall test quality (coverage of behavior or failure detection) will mirror those of white box data flow. Thus, in cases where the data design is complex and defines much of the application's behavior, we expect data flow criteria will be a good choice for test selection. Our new test selection criteria are presented in section 0.

In order to provide proof-of-concept, In section 4 and 5 we provide a theoretical analysis of a characteristic we call *transition variation*[1] then we present results from laboratory experiments in which software systems with known faults is tested by structurally-identical behavior models using three different test selection strategies: random path selection (including operational profiles), structure-guided path selection (node and edge coverage) and path selection guided by our new criteria. In order to counter the possibility that the random selection was unlucky (i.e., that a path that avoided all the failures was selected), we repeated the random experiment a number of times using different random seeds. We also used three variations of an operational profile: uniform, slightly tilted—to simulate a diverse operational profile—and significantly tilted—to simulate a specialized operational profile.

Our results for these experiments are encouraging. Specifically, they show that testing guided by our new data flow criteria outperforms all three types of random testing and behavior model coverage testing in the number of faults identified. In addition, even if we allow substantially more random testing to be performed, the data flow tests are still more productive. In effect, we are able to find more faults in less time using the data flow criteria than we could using random testing. Thus data flow testing from a black box perspective can achieve favorable results. Experiments with larger applications and production systems are underway.

[1] Transition variation is a quantitative measure that gives an indication about the coverage of all combinations of adjacent transitions of length two or more.

Behavior Models for Software[2]

We define a behavior model for software testing to be the couple (S, δ) where S is the set of all *operational states* of a software system and $\delta{:}S\times I\times[0\ldots1]\rightarrow S$ is the non-deterministic transition function, where I is the set of externally generated inputs to the software. Operational states describe internal or external objects (as long as they are identifiable from an external perspective) which influence the behavior of the software under test. In general, we are interested in objects that affect the way the software reacts to external inputs. We say that software is in state j when a collection of objects have certain values and in state k when they have different values. State j is characterized by which external inputs are allowable and which are prevented (or, at least reacted to in a different manner) by the software. State k will have a different set of allowable inputs that mark it as distinct from state j.

The transition function δ describes how the application of external inputs cause state changes within the software. The optional probability distribution associated with each state can represent the operational profile, i.e., the probability that the corresponding inputs will be applied during typical use, for that state or any other distribution that might be helpful to testers, such as an easy to establish uniform distribution.

Defining S and δ are accomplished by identifying the software's *operational modes*. An operational mode is a variable that abstracts system objects which govern the way the software responds to input. For example the variable "phone status = ringing *or* not ringing" is an operational mode for a phone switch because it governs whether the output for the "take the phone off the hook" input is "connected to caller" or "dial tone," respectively. I.e., if the phone is ringing the behavior will be to connect the two parties, if the phone is not ringing the output will be to emit a dial tone. Since a ringing phone can be detected external to the system (without consulting the actual source code), we say that this variable is black box data. A state $s\in S$ is formed by assigning a specific value to each operational mode of the system.

Operational modes represent situations where: 1) the software treats the same input differently, e.g., it may allow it at a given point during execution of the software and disallow it at others and/or 2) the software produces different output given the same user input. Operational modes thus allow us to model not only which input sequences are allowed and which are prevented (which is important for test case generation), but also how the software will behave given a specific sequence of inputs (which is important for determining expected results).

Operational modes are defined from a software's specification by considering the above situations. For example, in [18] we modeled a clock application using operational modes similar to the following.

- *System* = {not invoked, invoked}
- *Window* = {main form, change dialog, about dialog}
- *Setting* = {analog, digital}
- *Display* = {all, clock only}
- *Cursor* = {time, date}

Let O represent the set of all operational modes relevant to testing a software system. The state set S is formed by taking the cross product of the modes and removing the impossible combinations, i.e., $S\subseteq(o_1\times o_2\times\ldots\times o_m)$ where m is the total number of operational modes. In this manner, the following states are defined for the clock application:

[2] Software testing based on behavior models is supported by a number of commercial tools including Teradyne's TestMaster® (www.teradyne.com) and Q-Lab's toolSET_*Certify*® (www.q-labs.com).

1. {*System* = not invoked, *Window* = any, *Setting* =analog, *Display* =any, *Cursor* = any}
2. {*System* = invoked, *Window* = main form, *Setting* =analog, *Display* =all, *Cursor* = any}
3. {*System* = invoked, *Window* = main form, *Setting* = analog, *Display* = clock-only, *Cursor* = any}
4. {*System* = invoked, *Window* = change dialog, *Setting* = analog, *Display* = all, *Cursor* =time}
5. {*System* = invoked, *Window* = change dialog, *Setting* = analog, *Display* = all, *Cursor* =date}
6. {*System* = invoked, *Window* = about dialog, *Setting* = analog, *Display* = all, *Cursor* = any}
7. {*System* = invoked, *Window* = about dialog, *Setting* = digital, *Display* = all, *Cursor* = any}
8. {*System* = invoked, *Window* = main form, *Setting* = digital, *Display* = all, *Cursor* = any}
9. {*System* = invoked, *Window* = main form, *Setting* = digital, *Display* = clock-only, *Cursor* = any}
10. {*System* = invoked, *Window* = change dialog, *Setting* = digital, *Display* = all, *Cursor* =time}
11. {*System* = invoked, *Window* = change dialog, *Setting* = digital, *Display* = all, *Cursor* =date}
12. {*System* = not invoked, *Window* = any, *Setting* =digital, *Display* =any, *Cursor* = any}

The first execution of the software will be from state 1 and will end in either state 1 or 2. Subsequent runs will begin in either state 1 or 2, exactly where the previous run terminated.

The behavior model is constructed from the state set by defining the transition function δ. A transition from state s_1 to state s_2 is defined by 1) a subset of inputs from I which cause the software to change the value of one or more operational modes represented by s_1 to the values in s_2 and 2) an optional probability that reflects the likelihood of a user submitting the corresponding input to the software. In the event that more than one input causes the transition, each input is assigned a separate probability under the Markovian restriction that the exit probabilities from each state must sum to exactly 1. The behavior model for the clock application appears in figure 1. For readability, we omitted probability labels and self-looping transitions.

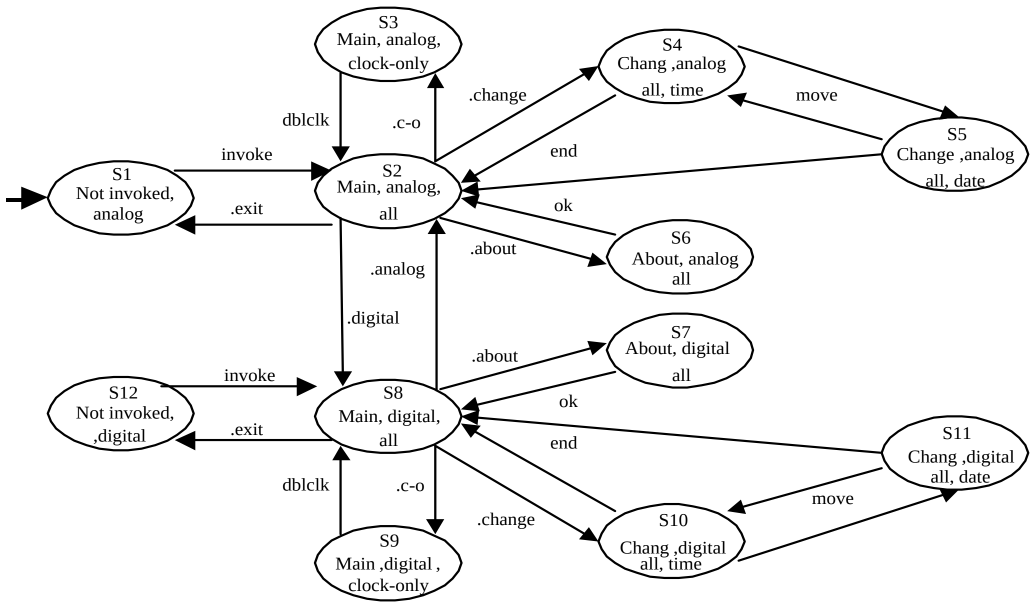

Figure 1. The Behavior Model of the Example Clock Program

Sample Selection

Generating tests from the behavior model is a graph traversal problem. Obvious methods for such traversal include:

- **Random walks** [17]. Beginning in the start state, select the next state according to the discrete probability distribution assigned to the exit transitions. Continue the random walk until any of the path termination states are reached.

  Random test selection is easy to implement and can result in a large number of test cases in a short period of time. One might then argue that there is no need for complex test data adequacy criteria since large numbers of random tests can overcome their individual ineffectiveness. However, test case generation isn't the expensive part of testing. The actual execution and evaluation—checking whether the software's behavior matches its specification—is much more difficult and time consuming. Execution and evaluation are hard to automate and can often be the bottleneck tasks during testing. In addition to spending resources on building a good oracle, it is also productive to consider methods to generate fewer but more effective tests.

- **Structure-guided selection** [14]. Choose paths from the start state to any of the termination states based on algorithms to minimize transition retraversal.

  Variants of the travelling salesman algorithm and the Chinese postman algorithm [14] can be used to achieve this outcome and significantly reduce the size of the sample.

Both of these strategies somewhat miss the point of good path selection: to choose a sequence of transitions, that as an ensemble, force the software to exhibit desired behavior. Random testing, by its very nature, generates significant variation in which transitions are traversed. Our experience has been that despite tilted operational profiles, the paths generated are sometimes nonsensical in that the transitions appearing in a specific sequence have little to do with actually making the software do real work. Imagine a model for a word processor that doesn't generate a sequence in which a document is typed, formatted, spell checked and then printed. Certainly many sequences do each of these things in isolation, but putting them all together is left to chance.

Structure-guided transition selection takes the remedy to the opposite extreme. By minimally covering the model's structure, one also lessens the chance interesting combinations of transitions will occur in the same execution of the software.[3] Another method is needed.

Our motivation is to improve on the manner in which transitions are selected by developing criteria that guide the selection of related transitions. Taking inspiration from white box data-flow criteria, we propose that focusing on data-oriented aspects of the model, instead of model structure, is beneficial. In the case of behavior models, the data of interest are the operational modes that define states. Intuitively, transitions that cause operational modes to change value are "better" than transitions that cause no such change. Our reasoning is similar to Rapps and Weyuker's in that by explicitly forcing data to be initialized and subsequently modified, we are forcing software to manipulate stored data. When software manipulates data, it exercises functionality and could potentially fail.

Our conjecture is that by focusing on model data, we will have a better chance of identifying which transitions are related and should appear together in the same test sequence. If two transitions are structurally unrelated but both manipulate similar data, then they may well be represented by an operational mode (given that the operational modes are defined correctly, of course). If so, then forcing the operational mode to change values may exercise the related transitions.

One can think of the path selection problem as follows:

- Pick a starting point for the path. This is usually a state in the model.
- Pick an ending point for the path (another state).
- Choose the route or routes to take from the starting point to get to the ending point.

We define the basic set of model data criteria as follows:

Given behavior model *M* with operational modes o_1 , o_2 , …, o_m we define one or more starting states for o_i to be the set of states in which o_i first receives its initial value.

DEFINITION: The **initial value** is the first value assigned to a specific operational mode other than the value"any" after invoking the software. Every mode has at least one initial value.

DEFINITION: The **initialization state** is the location at which an operational mode first receives its initial value. Each operational mode has one or more states at which it is initialized.

For the example model of figure 1, the set of *initialization states* are described in table 1.

Table 1
Initialization States for each Operational Mode of the Clock Example

Operational Mode	State(s) where mode is first initialized
System	{System=not invoked, Window=any, Setting=analog, Display=any, Cursor=any} This is the model's starting state.
Window	{System=invoked, Window=main form, Setting=analog, Display=all, Cursor=any}

[3] Other structure-guided path generation methods exist such as *loop-free paths* and *n-iteration-paths* [8] in which traversals through loops and cycles in the graph are allowed. However, the only variation addressed in these criteria are the appearance of transitions that reside within loops and cycles. Thus, the problem remains.

Setting	{System=not invoked, Window=main, Setting=analog, Display=any, Cursor=any} This is the model's starting state.
Display	{System=invoked, Window=main form, Setting=analog, Display=all, Cursor=any}

Cursor	{System=invoked, Window=change, Setting=analog, Display=all, Cursor=time}
	{System=invoked, Window=change, Setting=digital, Display=all, Cursor=time}

As with white-box data flow testing, it is of interest to generate paths that force data to change values. Thus, we can define the ending points for each sequence to be the point at which the mode in question assumes a new value. We can then define the family of criteria below:

Let *Paths*(*M*) represent the infinite set of all connected state sequences (paths) from a model's starting state to any of its terminal states. We are interested in generating a finite subset of paths, $P \subset Paths(M)$, that allow more transition-variation per path than either random or structure-guided walks.

Since a given operational mode can have multiple initialization states, we begin with more restrictive criteria.[4] For each criterion, it is of interest to generate a path that satisfies the criterion (start at the starting state and terminated immediately after that criterion under consideration is satisfied) and the set of all satisfying paths (paths in which loops and cycles are traversed only once). For simplicity, we label these *single-satisfying-path* and *all-satisfying-paths*, respectively.

Definition 1. The pair (*M*,*P*) satisfies the *single-initializations/single-values* criterion iff *P* contains a subpath *p* that causes each operational mode $o \in O$ to change from any of that mode's initial values at least once.

The shortest single-initialization/single-value path for the behavior model in figure 1 appears in table 2 with explanation.

Table 2
The Shortest Single-initialization/Single-value Path for the Clock Model

State in Subpath *p*	**Single-initialization/Single-value Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked*. *Setting* mode initialized to *analog*.
{invoked, main, analog, all}	*System* mode changes value to *invoked*. *Window* mode initialized to *main form*. *Display* mode initialized to *all*.
{invoked, main, digital, all}	*Setting* mode changes value to *digital*.
{invoked, main, digital, clock-only}	*Display* mode changes value to *clock-only*.
{invoked, main, digital, all}	
{invoked, change, digital, all, time}	*Cursor* mode initialized to *time*. All modes have now been initialized. *Window* mode changes value to *change*.
{invoked, change, digital, all, date}	*Cursor* mode changes value to *date*. All modes have now changed values at least once.

[4] We adopt the notation of Clarke *et al* [3] as representative of work in this area.

The shortest path may or may not be the best path that satisfies the criterion. Obviously, the reason for considering the shortest path is that, given all transitions are equally expensive to execute, it will be the least expensive path. However, selecting the shortest path also limits the amount of variation in the sequence. We call this *transitional variation* and as the data will show, it can be the difference in locating or missing a failure.

An alternative to the shortest path is to select a set of paths, each of which satisfies the criterion and possesses different transitional variation. The paths we choose are *all-satisfying single-initialization/single-value paths*: all non-looping paths which satisfy the criterion.

Obviously, the single-initialization/single-value criterion misses certain modal values, namely, the *about* value of the *Window* mode. In fact, modal values for any non binary-valued mode may be excluded. The next criterion allows us to cover each value of a mode with the exception of the initial value (i.e., all "other" values).

Definition 2. The pair (M,P) satisfies the *single-initializations/other-values* criterion iff P contains a subpath p that causes each operational mode $o \in O$ to take on each of its other legal values from any of that mode's initial values at least once.

Since only the *Window* mode has more than one other value, we simply need to visit any state where *Window=about dialog*. A simple modification of the path in table 2 can suffice to satisfy this criterion as shown in table 3.

Table 3
The Shortest Single-initialization/Other-values Path for the Clock Model

State in Subpath *p*	**Single-initialization/Other-values Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked.* *Setting* mode initialized to *analog.*
{invoked, main, analog, all}	*System* mode changes value to *invoked.* *Window* mode initialized to *main form.* *Display* mode initialized to *all.*
{invoked, main, digital, all}	*Setting* mode changes value to *digital.*
{invoked, main, digital, clock-only}	*Display* mode changes value to *clock-only.*
{invoked, main, digital, all}	
{invoked, change, digital, all, time}	*Cursor* mode initialized to *time.* All modes have now been initialized. *Window* mode changes value to *change*
{invoked, change, digital, all, date}	*Cursor* mode changes value to *date.*
{invoked, main, digital, all}	
{invoked, about, digital, all}	*Window* mode changes value to *about* All modes have now been assigned each value other than the initial.

An obvious continuation of this line of investigation is to require that each mode change also include returning to the initial value.

Definition 3. The pair (*M*,*P*) satisfies the *single-initializations/all-values* criterion iff *P* contains a subpath *p* that causes each operational mode $o \in O$ to change to each of its legal values (including the initial value) from any of that mode's initial values at least once.

Using the subpath of table 3, we note that both the *Window* and *Display* modes have already been changed back to their initial values, thus, we need to modify the subpath to change *Cursor=time*, *Setting=analog* and *System=not invoked*. The subpath in table 4 is the shortest path to satisfy this criterion.

Table 4
The Shortest Single-initialization/All-values Path for the Clock Model

State in Subpath *p*	**Single-initialization/All-values Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked.* *Setting* mode initialized to *analog.*
{invoked, main, analog, all}	*System* mode changes value to *invoked.* *Window* mode initialized to *main form.* *Display* mode initialized to *all.*
{invoked, main, digital, all}	*Setting* mode changes value to *digital.*
{invoked, main, digital, clock-only}	*Display* mode changes value to *clock-only.*
{invoked, main, digital, all}	*Display* mode changes value to *all.*
{invoked, change, digital, all, time}	*Cursor* mode initialized to *time.* All modes have now been initialized. *Window* mode changes value to *change*
{invoked, change, digital, all, date}	*Cursor* mode changes value to *date.*
{invoked, change, digital, all, time}	*Cursor* mode changes value to *time.*
{invoked, main, digital, all}	
{invoked, about, digital, all}	*Window* mode changes value to *about.*
{invoked, main, digital, all}	*Window* mode changes value to *main form.*
{invoked, main, analog, all}	*Setting* mode changes value to *analog.*
{not invoked, analog}	*System* mode changes value to *not invoked* All modes have now been assigned all values.

We proceed by defining the same criteria but expanding the starting point of the paths to every state in which a mode is initialized. Specifically, we see that the *Cursor* mode has two distinct initialization states.

Definition 4. The pair (*M*,*P*) satisfies the *all-initializations/single-values* criterion iff *P* contains a subpath *p* that causes each operational mode $o \in O$ to change from each of that mode's initial values at least once.

The shortest sequence satisfying this criterion appears in table 5.

Table 5
The Shortest All-initializations/Single-value Path for the Clock Model

State in Subpath *p*	**All-initialization/Single-value Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked.*

	Setting mode initialized to *analog.*
{invoked, main, analog, all}	*System* mode changes value to *invoked.* *Window* mode initialized to *main form.* *Display* mode initialized to *all.*
{invoked, main, digital, all}	*Setting* mode changes value to *digital.*
{invoked, main, digital, clock-only}	*Display* mode change value to *clock-only.*
{invoked, main, digital, all}	
{invoked, change, digital, all, time}	*Cursor* mode initialized to *time.* This is the first initialization of *Cursor*. All modes have now been initialized. *Window* mode changes value to *change*
{invoked, change, digital, all, date}	*Cursor* mode changes value to *date.*
{invoked, main, digital, all}	
{invoked, main, analog, all}	*Setting* mode changes value to *analog.*
{invoked, change, analog, all, time}	*Cursor* mode initialized to *time.* This is the second initialization of *Cursor*. *Window* mode changes value to *change*
{invoked, change, analog, all, date}	*Cursor* mode changes value to *date.* All requirements of the criterion are now satisfied.

Definition 5. The pair (*M*,*P*) satisfies the *all-initializations/other-values* criterion iff *P* contains a subpath *p* that causes each operational mode $o \in O$ to take on each of its other legal values from each of that mode's initial values at least once.

The shortest sequence satisfying this criterion appears in table 6.

Table 6
The Shortest All-initializations/Other-values Path for the Clock Model

State in Subpath *p*	**All-initialization/Other-values Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked.* *Setting* mode initialized to *analog.*
{invoked, main, analog, all}	*System* mode changes value to *invoked.* *Window* mode initialized to *main form.* *Display* mode initialized to *all.*
{invoked, main, digital, all}	*Setting* mode changes value to *digital.*
{invoked, main, digital, clock-only}	*Display* mode change value to *clock-only.*
{invoked, main, digital, all}	
{invoked, change, digital, all, time}	*Cursor* mode initialized to *time.* This is the first initialization of the *Cursor*. All modes have now been initialized.
{invoked, change, digital, all, date}	*Cursor* mode changes value to *date.*
{invoked, main, digital, all}	

{invoked, about, digital, all}	*Window* mode changes value to *about.*
{invoked, main, digital, all}	
{invoked, main, analog, all}	*Setting* mode changes value to *analog.*
{invoked, change, analog, all, time}	*Cursor* mode initialized to *time.* This is the second initialization of the *Cursor*. *Window* mode changes value to *change.*
{invoked, change, analog, all, date}	*Cursor* mode changes value to *date.* All requirements of the criterion are now satisfied.

Definition 6. The pair (*M,P*) satisfies the *all-initializations/all-values* criterion iff *P* contains a subpath *p* that causes each operational mode $o \in O$ to change to each of its legal values (including the initial value) from each of that mode's initial values at least once.

The shortest sequence satisfying this criterion appears in table 7.

Table 7
The Shortest All-initializations/All-values Path for the Clock Model

State in Subpath *p*	**All-initialization/All-values Criterion Coverage**
{not invoked, analog}	*System* mode initialized to *not invoked.* *Setting* mode initialized to *analog.*
{invoked, main, analog, all}	*System* mode changes value to *invoked.* *Window* mode initialized to *main form.* *Display* mode initialized to *all.*
{invoked, main, digital, all}	*Setting* mode changes value to *digital.*
{invoked, main, digital, clock-only}	*Display* mode change value to *clock-only.*
{invoked, main, digital, all}	*Display* mode change value to *all.*
{invoked, change, digital, all, time }	*Cursor* mode initialized to *time.* This is the first initialization of the *Cursor*. All modes have now been initialized. *Window* mode changes value to *change.*
{invoked, change, digital, all, date }	*Cursor* mode changes value to *date.*
{invoked, change, digital, all, time }	*Cursor* mode changes value to *time.*
{invoked, main, digital, all}	*Window* mode changes value to *main form.*
{invoked, about, digital, all}	*Window* mode changes value to *about.*
{invoked, main, digital, all}	
{invoked, main, analog, all}	*Setting* mode changes value to *analog.*
{invoked, change, analog, all, time}	*Cursor* mode initialized to *time.* This is the second initialization of the *Cursor*.
{invoked, change, analog, all, date}	*Cursor* mode changes value to *date.*
{invoked, change, analog, all, time }	*Cursor* mode changes value to *time.*
{invoked, main, analog, all}	

{not invoked, analog}	*System* mode changes value to *not invoked* All requirements of the criterion are now satisfied.

Since a given operational mode can have multiple initialization states, more restrictive criteria are introduced here. All the criteria already covered are based on the unique starting state of software. At this point more variation can be added by expanding the starting point of the paths to every state in which a mode is initialized. For example, the clock model has four candidate starting states:

1) S1: at which operational modes *System* and *Setting* are initialized.
2) S3: at which operational modes *Window* and *Display* are initialized.
3) S7 and S9: are different locations that operational mode *Cursor* is initialized.

Thus, three criteria (7, 8,and 9) can be defined as follow:

Definition 7. The pair (*M*,*P*) satisfies the *all-initialization-states/single-values* criterion iff for each initialization state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to change at least once.

Definition 8. The pair (*M*,*P*) satisfies the *all-initialization-states/other-values* criterion iff for each initialization state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to take on each of its other legal values at least once.

Definition 9. The pair (*M*,*P*) satisfies the *all-initialization-states/all-values* criterion iff for each initialization state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to change to each of its legal values (including the values at *s*) at least once.

The test cases that satisfy definitions 7-9 are too large to include but follow directly from the above examples. Simply start in the initial state, select a path that forces each mode to change values and then repeat this for each mode initialization state in the model (S1, S3, S7, and S9). The shortest paths satisfying definitions 7 through 9 are 23 transitions, 44 transitions and 50 transitions, respectively as will be shown in the next section.

The only way left to add variation to the starting state is to require that each and every state act as a start state.

Definition 10. The pair (*M*,*P*) satisfies the *all-states/single-values* criterion iff for each state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to change at least once.

Definition 11. The pair (*M*,*P*) satisfies the *all-states/other-values* criterion iff for each state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to take on each of its other legal values at least once.

Definition 12. The pair (*M*,*P*) satisfies the *all-states/all-values* criterion iff for each state *s*, *P* contains a subpath *p* that start at *s* and causes each operational mode $o\in O$ to change to each of its legal values (including the values at *s*) at least once.

The test cases that satisfy definitions 10-12 are also too large to include but follow directly from the above examples. The shortest path satisfying definitions 10-11 are 76 transitions, 141 transitions and 152 transitions, respectively as will be shown in the next section.

All the definitions covered up to this point are seeking the shortest *single-satisfying* path. Another set of criteria can be introduced here that seeking *all-satisfying* paths. Experimental Results and analysis will be shown in the next section. Definitions of them are as follow:

Definition 13. The pair (M, P) satisfies the *single-initializations/single-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to change from any of that mode's initial values at least once.

Definition 14. The pair (M, P) satisfies the *single-initializations/other-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to take on each of its other legal values from any of that mode's initial values at least once.

Definition 15. The pair (M, P) satisfies the *single-initializations/all-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to change to each of its legal values (including the initial value) from any of that mode's initial values at least once.

Definition 16. The pair (M,P) satisfies the *all-initializations/single-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to change from each of that mode's initial values at least once.

Definition 17. The pair (M,P) satisfies the *all-initializations/other-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to take on each of its other legal values from each of that mode's initial values at least once.

Definition 18. The pair (M,P) satisfies the *all-initializations/all-values/all-satisfying paths* criterion iff P contains all subpaths p that causes each operational mode $o \in O$ to change to each of its legal values (including the initial value) from each of that mode's initial values at least once.

Definition 19. The pair (M,P) satisfies the *all-initialization-states/single-values/all-satisfying paths* criterion iff for each initialization state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to change at least once.

Definition 20. The pair (M,P) satisfies the *all-initialization-states/other-values/all-satisfying paths* criterion iff for each initialization state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to take on each of its other legal values at least once.

Definition 21. The pair (M,P) satisfies the *all-initialization-states/all-values/all-satisfying paths* criterion iff for each initialization state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to change to each of its legal values (including the values at s) at least once.

Definition 22. The pair (M,P) satisfies the *all-states/single-values/all-satisfying paths* criterion iff for each state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to change at least once.

Definition 23. The pair (M,P) satisfies the *all-states/other-values/all-satisfying paths* criterion iff for each state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to take on each of its other legal values at least once.

Definition 24. The pair (M,P) satisfies the *all-states/all-values/all-satisfying paths* criterion iff for each state s, P contains all subpaths p that start at s and causes each operational mode $o \in O$ to change to each of its legal values (including the values at s) at least once.

Proof of Concept

Two proofs of concept are provided here. The first one is a theoretical one through a characteristic, which we call *transition variation.* The second proof of concept is from experimental standpoint through three deferent experiments, which will be shown in the next section.

Transition variation is a quantitative measure that gives an indication about the coverage of all combinations of adjacent transitions of length two or more. Covering high percentage of these transitions imply a high level of transition variation, which means improving the coverage of the testing process. It is argued that the Markov chain-based test data adequacy criteria has a higher transition variation score than both the structural criteria and the random test case generation. To proof this theory, transition variation of the new criteria and the structural coverage (Chinese postman) is studied based on the Markov chain behavior model of the clock application. Three level of comparisons are targeted: states coverage, all single transition coverage (involve two states: the *from* and the *to* states), and all combinations of two adjacent transitions coverage (involve three states: A ->B ->C). Table 8 shows the state coverage of the new criteria satisfied by 24 transitions and above along with the Chinese postman. This data indicate that state coverage is taken care of by both the new criteria and the structural coverage. The second level of comparison is the coverage of single transition.

Table 8

States Coverage Comparison

Criteria	**AIS/**	**AIS/**	**AIS/**	**AS/**	**AS/**	**AS/**	**SI/**	**SI/**	**SI/**	**AI/**	**AI/**	**AI/**	**AIS/**	**AIS/**	**AIS/**	**AS/**	**AS/**	**AS/**	**Chinese**
	SV/	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**OV/**	**SV/**	**OV/**	**AV/**	**Post**
	SP	**SP**	**SP**	**SP**	**SP**	**SP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**Man**
States	**24 t**	**40 t**	**48 t**	**74 t**	**110 t**	**140 t**	**86 t**	**104 t**	**124 t**	**102 t**	**120 t**	**148 t**	**331 t**	**486 t**	**583 t**	**1008 t**	**1363 t**	**1702 t**	**28 t**
Total States Covered Out of 12	7	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12

As table 9 shows, the Chinese postman and most of the new criteria did cover all transitions in the model. However, transition variation cannot be compared using states coverage or only single transitions. Therefore, the third level of comparison (all combinations of two adjacent transitions coverage: A ->B ->C) was needed. A total of 77 combinations of two adjacent transitions were generated from the Markov chain behavior model of the clock application, which represents all possible combinations of adjacent transitions of length two. Table 10 shows how well the criteria did over the Chinese postman. Most of them perform much better than the Chinese postman. In the best case, the criteria *All-states/All-values/All-satisfying-path* covered 66 combination out of 77 and only 22 out of 77 covered by the Chinese postman.

Table 9

Single Transition Coverage Comparison

Criteria	**AIS/**	**AIS/**	**AIS/**	**AS/**	**AS/**	**AS/**	**SI/**	**SI/**	**SI/**	**AI/**	**AI/**	**AI/**	**AIS/**	**AIS/**	**AIS/**	**AS/**	**AS/**	**AS/**	**Chinese**
	SV/	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**AV/**	**SV/**	**OV/**	**OV/**	**SV/**	**OV/**	**AV/**	**Post**
	SP	**SP**	**SP**	**SP**	**SP**	**SP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**AP**	**Man**
Trans.	**24 t**	**40 t**	**48 t**	**74 t**	**110 t**	**140 t**	**86 t**	**104 t**	**124 t**	**102 t**	**120 t**	**148 t**	**331 t**	**486 t**	**583 t**	**1008 t**	**1363 t**	**1702 t**	**28 t**
Total of Single Transition Covered Out of 24	18	20	23	22	22	24	22	24	24	24	24	24	24	24	24	24	24	24	24

Table 10

Two Transition Coverage Comparison

Criteria / Trans.	AIS/ SV/ SP 24 t	AIS/ OV/ SP 40 t	AIS/ AV/ SP 48 t	AS/ SV/ SP 74 t	AS/ OV/ SP 110 t	AS/ AV/ SP 140 t	SI/ SV/ AP 86 t	SI/ OV/ AP 104 t	SI/ AV/ AP 124 t	AI/ SV/ AP 102 t	AI/ OV/ AP 120 t	AI/ AV/ AP 148 t	AIS/ SV/ AP 331 t	AIS/ OV/ AP 486 t	AIS/ OV/ AP 583 t	AS/ SV/ AP 1008 t	AS/ OV/ AP 1363 t	AS/ AV/ AP 1702 t	Chinese Post Man 28 t
The total adjacent transitions of length two Covered out of the 77 possible combinations	11	21	28	22	32	43	33	36	41	32	36	40	41	44	50	61	62	66	22

Laboratory Experiments

Experiment I:

In order to experiment with our criteria and convince industry partners to support full-scale case studies, we performed a modified form of fault seeding [10] to test the ability of the criteria to find faults. We could also have studied the ability of tests to cover requirements but we believe that eliciting failure is more difficult, in general, and would be a better test of the criteria's fitness.

Instead of injecting artificial faults into a program, we injected real faults as follows. We commissioned third party development of the clock program described above and recorded the failures found during the review and testing process. After the program was complete, we reinserted the causal faults back into the software. A total of seven faults were recorded along with the sequence of state transitions required to expose the failure during test. The faults are described in table 8 along with three additional real faults that were unknown at the time of test.[5]

Table 11
Faults Found During Testing of the Clock Application

Failure	**Description**	**Exposing Sequence(s)**
Fault 1	The initial position of the tab stop is incorrectly set to the Date field instead of the Time field on the Change Time/Date form.	First occurrence of : {inv, change form, analog, all, time} or {inv, change form, digital, all, time}.
Fault 2	The digital mode is not unloaded when the user switches to analog mode causing two clocks to appear simultaneously.	The state: {inv, main form, digital, all, any} must occur *before* {inv, main form, analog, all, any} in the same sequence.

[5] Obviously more faults could very well exist in the software but these were the only ones detected by our oracle, a simple comparison of the debugged application with the test application. The oracle was capable of comparing the applications and also detecting gross deviation of actual and specified output (from a written description of input-output pairs). Since every test was monitored by the same oracle and each test history was carefully examined for causal input patterns, we ensured that each test selection strategy had access to the same set of detectable faults.

Fault 3	In the Change Time/Date form, the Tab key moves between fields in the wrong order.	First occurrence of : {inv, change form, analog, all, time} or {inv, change form, digital, all, time} which must be followed by at least two Tab inputs.
Fault 4	When the Change Time/Date form is unloaded in digital mode, the Main form is displayed in analog mode.	First occurrence of : {inv, main form, digital, all, any} immediately preceded by {inv, change form, digital, all, time}.
Fault 5	The text on the About form is misaligned if the Clock-only form was visited prior to the About form.	The state: {inv, main form, analog, clock-only, any} occurred before {inv, about form, analog, all, any} in the same sequence.
Fault 6	When the About form is unloaded in digital mode, the Main form is displayed in analog mode.	First occurrence of : {inv, main form, digital, all, any} preceded by {inv, about form, digital, all, any}.
Fault 7	Analog second hand overlays the digital display on the Clock-only form when both Clock-only forms (analog and digital) are displayed in the same sequence.	Must visit: {inv, main form, analog, clock-only, any} before: {inv, main form, digital, clock-only, any} in the same sequence.
Fault 8	Unable to invoke the clock in digital mode due to incorrect handling of the operating system registry.	Any sequence that begins with: {not inv, any, digital, any, any}.
Fault 9	The Clock-only form is not unloaded.	First occurrence of : {inv, main form, analog, all, any} immediately preceded by {inv, main form, analog, clock-only, any} or {inv, main form, digital, all, any} immediately preceded by {inv, main form, digital, clock-only, any}
Fault 10	Cursor behaves incorrectly when dates are entered.	Either {inv, change form, analog, all, date} or {inv, change form, digital, all, date} is followed by a valid entry for the date field.

The exposing sequence for each fault is a good indicator of the difficulty of finding the fault. Faults 1, 2, 8 and 9 will be detected by simply traversing each transition in the behavior model. We term these "easy bugs." Each of the easy bugs were found during manual ad hoc testing. Faults 4, 5, 6 and 10 have more complicated exposing sequences and require more sophisticated tests to detect them. We term these "moderate bugs" and note that two of the three moderate bugs were detected during manual testing. The final category, called "hard bugs" are faults 3 and 7, neither of which were found during manual testing. The exposing sequences are more difficult to arrange in advance.

We then tested the clock application using the behavior model as a test generator. Three categories of tests were independently generated and applied, *criteria-guided tests*, *random tests*, and *structural tests*. In this manner we are ensuring that each set of tests is working on equal footing to each other set: since they are each originating from the same model, they have equal ability to exercise the software and have the capability to elicit the same set of failures (and, thus, find the same faults).

Obviously, testing this small application does not validate our criteria, nor does it bear any statistical significance since it is only a sample of one trial. However, we believe the experiment does show that the new criteria can favorably compete with existing behavioral test selection methods and prove that the concept warrants additional research and, perhaps, limited use in practice.

Criteria-Guided Tests

The twenty-four criteria proposed represents two main sets each with twelve criteria::

- **Single satisfying paths:** sequences satisfy the criterion as quickly as the requirements of that specific criterion satisfied with minimal duplication of subpaths. Obviously, the idea here is that quicker is better. The length of sequences to satisfy each of the criteria ranged from 7 transitions to 140 transitions.
- **All satisfying paths:** sequences represent every possible subpath that will satisfy a specific criterion. This set of paths emphasizes variation in choosing transitions to satisfy the criteria. Obviously, this strategy can require a large number of sequences to satisfy. The length of sequences to satisfy each of the criteria ranged from 86 transitions to 1702 transitions.

Table 12
Number of Transition to Find Each Fault and Satisfy Criteria Guided Tests
(blank entries indicate the fault was not found)

Criterion	**No. Trans. to Satisfy**	**No. Transitions to Find Fault:**									
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Single-init/Single-value/Single-satisfying-path	7	6								5	6
Single-init/Other-value/Single-satisfying-path	9	6								5	6
Single-init/All-value/Single-satisfying-path	11	6								5	6
All-init/Single-value/Single-satisfying-path	10	6	10							5	6
All-init/Other-value/Single-satisfying-path	12	6	11							5	6
All-init/All-value/Single-satisfying-path	15	6	13						14	5	6
All-init-states/Single-values/Single-satisfying-path	24	6	20						22	5	6
All-init-states/Other-values/Single-satisfying-path	40	6	31				29		31	5	6
All-init-states/All-values/Single-satisfying-path	48	6	34				35		48	5	6
All-states/Single-values/Single-satisfying-path	74	6	20				45		70	5	6
All-states/Other-values/Single-satisfying-path	110	6	31			39	29		103	5	6
All-states/All-values/Single-satisfying-path	140	6	34		105	44	35		130	5	6
Single-init/Single-value/All-satisfying-path	86	4	38				47		81	3	5
Single-init/Other-value/All-satisfying-path	104	4	44			28	67		97	3	5
Single-init/All-value/All-satisfying-path	124	4	11		102	32	77		115	3	5
All-init/Single-value/All-satisfying-path	102	4	41				38		94	3	5
All-init/Other-value/All-satisfying-path	120	4	46			29	70		110	3	5

All-init/All-value/All-satisfying-path	148	4	13		98	36	91		136	3	5
All-init-states/Single-values/All-satisfying-path	331	4	38			127	47		81	3	5
All-init-states/Other-values/All-satisfying-path	486	4	44		346	28	67		97	3	5
All-init-states/All-values/All-satisfying-path	583	4	11		102	32	77		115	3	5
All-states/Single-values/All-satisfying-path	1008	4	38	822	823	127	47	265	81	3	5
All-states/Other-values/All-satisfying-path	1363	4	44	1008	346	28	67	318	97	3	5
All-states/All-values/All-satisfying-path	1702	4	11	1235	102	32	77	380	115	3	5

As table 12 shows, all four aspects of a path definition (Single-initialization, All- initialization, All-initialization-states, All-states) matter in finding faults. As the criteria chosen for starting point, ending point and route become harder to satisfy, more faults are exposed. For example, the *All-init/.../Single-satisfying-path* criterion found 4, 4 and 5 faults respectively whereas their counterparts for *Single-init/All-value/Single-satisfying-path* found only 3, 3 and 3 faults, respectively. Another example, the *all-states/../Single-satisfying-path* criterion found 6, 7 and 8 faults respectively whereas their counterparts for *All-init-states/Single-values/Single-satisfying-path* found only 5, 6 and 6 faults, respectively. In fact, the most stringent *Single-satisfying-path* criterion found all but two faults—and did so with only 130 transitions (the criterion was met after 140 transitions). Since each transition amounts to seconds of test time, this is very fast indeed for this particular application.

Random Tests

Random testing has proven very useful in useful in come studies [4,6,11] and using non-uniform operational profiles is an integral part of certain testing methodologies [9,12]. We established three operational profiles for the example behavior model to generate random tests for our experiment:

- The **uniform** profile consists of uniform probability distributions across the exit arcs of each state.
- The **slightly tilted** profile consists of exit arc probability distributions in which the high and low values differ by 0.1 *at most*.
- The **highly tilted** profile consists of exit arc probability distributions in which the highest and lowest probability differs by *at least* 0.5.

Table 13
Number of Trans. to Find Each Fault and Satisfy Random Criteria

Criterion	No. Trans. to Satisfy	No. Transitions to Find Fault:									
		1	2	3	4	5	6	7	8	9	10
uniform	823	26	7			125	332		122	29	95
slightly tilted	823	43	20			102	276			18	109
highly tilted	823	56	43			263	295			38	49

To account for random seeds being "unlucky" we generated 30 samples for each profile type and averaged the outcome. Each sample contained 823 transitions to correspond to the minimum number of states generated to find all ten bugs by the All-states/Single-values/All-satisfying-path criterion.

Structural Tests

Structural tests from a behavior model are analogous to structural coverage of program paths.

- **State coverage** means finding a traversal to cover each state at least once.
- **Transition coverage** means finding a traversal to cover each transition at least once.

Table 14
Number of Trans. to Find Each Fault and Satisfy Structural Criteria

Criterion	No. Trans. to Satisfy	No. Transitions to Find Fault: 1	2	3	4	5	6	7	8	9	10
All-States	17	4			16					3	5
All-Transitions (Chinese Postman)	28	5	16				14		4	27	

Table 15 shows the performance of these additional criteria in locating the same failures as above.

Table 15
Number of Faults Found While Satisfying Each Criterion

Criteria	Number of Transitions																	
	7	9	11	12	15	18	30	40	48	74	110	150	331	486	583	1008	1363	1702
SI/SV/SP	3	**(Three faults found by this criterion after seven transitions)**																
SI/OV/SP	3	3	**(Three faults found by this criterion after seven trans. and five faults after nine trans.)**															
SI/AV/SP	3	3	3															
AI/SV/SP	3	3	4															
AI/OV/SP	3	3	4	4														
AI/AV/SP	3	3	3	3	5													
AIS/SV/SP	3	3	3	3	3	3	5											
AIS/OV/SP	3	3	3	3	3	3	4	6										
AIS/AV/SP	3	3	3	3	3	3	4	5	6									
AS/SV/SP	3	3	3	3	3	3	4	4	5	6								
AS/OV/SP	3	3	3	3	3	3	4	6	6	6	7							
AS/AV/SP	3	3	3	3	3	3	3	5	6	6	7	8						
SI/SV/AP	3	3	3	3	3	3	3	4	5	5	6							
SI/OV/AP	3	3	3	3	3	3	3	4	5	6	7							
SI/AV/AP	3	3	4	4	4	4	4	5	5	5	7	8						
AI/SV/AP	3	3	3	3	3	3	3	3	5	5	6							
AI/OV/AP	3	3	3	3	3	3	3	4	5	6	7	7						
AI/AV/AP	3	3	3	3	4	4	4	5	5	5	7	8						
AIS/SV/AP	3	3	3	3	3	3	3	4	5	5	6	7						
AIS/OV/AP	3	3	3	3	3	3	4	4	5	6	7	7	7	8				
AIS/AV/AP	3	3	4	4	4	4	4	5	5	5	7	8	8	8	8			
AS/SV/AP	3	3	3	3	3	3	3	4	5	5	6	7	7	8	8	10		
AS/OV/AP	3	3	3	3	3	3	4	4	5	6	7	7	7	7	9	10	10	
AS/AV/AP	3	3	4	4	4	4	4	5	5	5	7	8	8	8	9	10	10	10

Uniform	1	1	1	1	1	1	3	3	3	3	4	6	6	6	7	7		
Slightly tilted	0	0	0	0	0	1	2	2	3	3	5	5	5	5	6	6		
Highly tilted	0	0	0	0	0	0	0	1	2	4	4	4	4	6	6	6		
All-States	3	3	3	3	3	4												
All-Transitions	2	2	2	2	3	4	5											

For example, after 15 transitions only six criteria have been satisfied. However, one of the criteria (involving *all-values* and indicated by shaded entries) have found a whopping five faults. That is fifty percent of the known faults in what constitutes only seconds of testing. At this point in testing, none of the other criteria found more than four faults including the random and the structural criteria. After 50 transitions, many of the criteria gain ground and are closely matched; eighteen criteria (indicated by shaded entries) have found five or more faults. Neither random nor structural criteria could catch up with the content criteria. Closer analysis reveals that the more values of operational modes are forced to change, the more faults are uncovered. The *all-values* criteria have been always much better than both *single-value* and *other-value* criteria.

The *all-transitions* criterion is the best performer of the structural and random criteria in the early stages but is satisfied too early to find many of the more complex faults. The reason for the “poor” performance of the structural criteria is that many of the failures in the clock program are associated with multiple transitions. The strength of these criteria, on the other hand, is in covering single transitions as quickly as possible. They are, by definition, ill equipped to find complicated multi-transition failures: they lack the ability to generate sequences with significant variation in the transitions.

Random testing does indeed generate sequences with substantial transition variation. However, the variation that occurs is strictly by chance and not by design. From the data in Table 12 it is obvious that leaving such variation to chance can cause a failure to go unobserved for a very long time. Each of the random tests has streaks of tests in which no failures are found despite the fact that many failures remain. Certainly over the long run, the random criteria catch up but if large testing budgets are problematic then this may not be an effective means of testing software to find faults.

The new criteria remove this element of chance and replace it with careful design. The criteria are based on the premise that transitions related to a single value of an operational mode should appear in the same test sequence. This means that transition variation is directly addressed in the satisfaction of the criteria.

Experiment II:

The goal of this experiment is to confirm the findings from the first experiment and to provide more understanding about the detection ability of this new family of test data adequacy criteria. The first experiment intended to be relatively small to get high level of control over the application of these new criteria. However, this second experiment is totally real life problem and much bigger in size. It started when a team of testers assigned to a contract with Microsoft for testing some of the models in MS Windows-CE loaded in the Pocket PCs. During the testing activates, bugs found were documented to be used as a measure for the detection ability of the new criteria. These bugs are not artificial or injected bugs, they are real bugs delivered with Microsoft Windows-CE. A total of ten faults were recorded along with the sequence of state transitions required to expose them. The Markov model of the first experiment contain 12 states, however, the Markov model of the second experiment contain 49 states. Just like what has been done in the first experiment, three test sets are generated: random tests, structural tests, and criteria-guided tests using the new family of test data adequacy criteria. Table 13 shows all 24 criteria along with two structural criteria and three random operational profiles.

Table 16
Number of Transition to Find Each Fault and Satisfy a Criteria

Guided Selection Results											
Criterion	**N0. Trans. To Satisfy**	**No. Transitions to Find Fault:**									
		1s	**2s**	**3**	**4**	**5**	**6**	**7**	**8**	**9s**	**10**
Single-init/Single-value/Single-satisfying-path	17			3				9			
Single-init/Other-value/Single-satisfying-path	19			3				10			
Single-init/All-value/Single-satisfying-path	20			3				10			
All-states/Single-values/Single-satisfying-path	275			3		4	34	9	238		
All-states/Other-values/Single-satisfying-path	351			3		175	47	10	273		153
All-states/All-values/Single-satisfying-path	375			3		41	53	10	291		187
Single-init/Single-value/All-satisfying-path	328	2	277	19		18		9	283		26
Single-init/Other-value/All-satisfying-path	342	2	340	21		23		10	313		28
Single-init/All-value/All-satisfying-path	358	2	316	22		25		10	327		29
All-states/Single-values/All-satisfying-path	13807	2	277	19	12013	18	931	9	283	3971	26
All-states/Other-values/All-satisfying-path	16475	2	340	21	14391	23	1015	10	313	2402	28
All-states/All-values/All-satisfying-path	20711	2	316	22	18310	25	1178	10	327	5561	29
Structural Coverage Results											
All-States Coverage Depth First Traversal	86					58		6			28
All-Transitions / Chinese postman	584	580	91			77		7		552	438
Random Coverage Results											
Uniform Random	13807	8	3120	97		351		76	312	585	
slightly tilted Random	13807	16	4231	75		459		274	1034	678	
highly tilted Random	13807	34		189		1103	1211	384	941	2354	

Data in table 16 are consistent with the data of the first experiment shown in table 9. Two more finding should be highlighted here:

1) **Self loops:** The Markov chain model used in the first experiment has no self loops, however the Markov model of the second experiment has it. This gave the opportunity of analyzing

which criterion covers the self-loops and it turned to be the criteria associated with All-satisfying-paths does cover them.

2) **The new criteria vs. the random:** comparing the performance of the new criteria vs. the random confirms that the bug detection rate of the random goes down over time. For example, in the experiment 13807 transitions are generated to satisfies both All-states /Single-values/All-satisfying-path and the random coverage. In the case of the random the last bug detected was after 4231 transitions. On the other hand, the last bug detected by the All-states/Single-values/All-satisfying-path criterion was after 12013 transitions. This incident draw attention to an important feature of this new family, which I call “ *Transition Variation*”. In the random case transition variation is not targeted and traversal of the model is random where redundancy is maximum. However, the transitions generated by the new criteria focus in maximizing the transition variation where the sequence of transition and states are always guided by the criteria satisfaction requirements and not left to chances. This decreases the probability that a specific sequence of states or transitions is cover more than one time.

Experiment III:

The purpose of this experiment is to measure the scalability of this new family of test data adequacy criteria. The metrical unit of testing in our case is the *transition.* The cost of generating, executing and evaluating a transition is the unit cost used to estimate the cost of satisfying specific criterion. Table 14 show the cost of each criterion based in number of transitions needed to satisfy it. Three different experiments sizes are used to assess the difference of cost, as software gets bigger. The table shows the MS Clock model with 12 states, the Inbox model with 49 states, and the Inbox model with 864 states. The new criteria are compared against two structural criteria: *All-states* coverage and *All-transition* coverage (Chinese postman). Data shows that as software under test gets bigger, more of the new criteria become less expensive than the structural criteria. (All operational modes of the second and third experiments are initialized in the model starting state, which means that any criterion involves more than initial state is not applicable (N/A)).

Table 17
Criteria Cost Comparison

Criterion	Number of Transitions to Satisfy		
	MS Clock 12 States	Pocket PC 49 States	Pocket PC 864 States
Single-init/Single-value/Single-satisfying-path	7	17	35
Single-init/Other-value/Single-satisfying-path	9	19	35
Single-init/All-value/Single-satisfying-path	11	20	38
All-init/Single-value/Single-satisfying-path	10	N/A	N/A
All-init/Other-value/Single-satisfying-path	12	N/A	N/A
All-init/All-value/Single-satisfying-path	15	N/A	N/A
All-init-states/Single-values/Single-satisfying-path	24	N/A	N/A
All-init-states/Other-values/Single-satisfying-path	40	N/A	N/A
All-init-states/All-values/Single-satisfying-path	48	N/A	N/A
All-states/Single-values/Single-satisfying-path	74	275	9182
All-states/Other-values/Single-satisfying-path	110	351	11215

All-states/All-values/Single-satisfying-path	140	375	15743
Single-init/Single-value/All-satisfying-path	86	328	19004
Single-init/Other-value/All-satisfying-path	104	342	20062
Single-init/All-value/All-satisfying-path	124	358	22186
All-init/Single-value/All-satisfying-path	102	N/A	N/A
All-init/Other-value/All-satisfying-path	120	N/A	N/A
All-init/All-value/All-satisfying-path	148	N/A	N/A
All-init-states/Single-values/All-satisfying-path	331	N/A	N/A
All-init-states/Other-values/All-satisfying-path	486	N/A	N/A
All-init-states/All-values/All-satisfying-path	583	N/A	N/A
All-states/Single-values/All-satisfying-path	1008	13807	16963320
All-states/Other-values/All-satisfying-path	1363	16475	17333568
All-states/All-values/All-satisfying-path	1702	20711	19168704
Structural Coverage			
All-States Coverage	17	86	2449
All-Transitions / Chinese postman	28	584	22704

Summary and Future Work

We presented a family of black box data flow criteria that are useful to practitioners who use behavior models or similar finite-state machines to model and generate test inputs. The criteria focus on exercising software functionality by forcing value changes in data objects (operational modes) that affect system behavior. Our objective is to make the software do more work in less time, hopefully resulting in more thorough testing. Our initial laboratory experiments comparing criteria-guided testing to random testing and simulated operational profile testing are encouraging.

Since graph-based techniques for software testing are abundant [2,14] and supported by popular commercial tools [1], we believe that practitioners will readily find uses for these new criteria. We provided experimental results that are based on different software system sizes and range from a somewhat-larger-than-a-toy example to big application. Note that new experiments are necessary to determine more precisely the effectiveness of the criteria. Granted, the examples was simple and many of the faults fairly easy to find, however, the new criteria competed on even terms with established test case selection techniques and performed impressively.

We also derived and code algorithms that automate each criterion for testing from arbitrary behavior models. Such a tool allowed us to perform additional laboratory experiments and field trials to further study the effectiveness of testing using the criteria on real software. We have begun such trials in cooperation with Microsoft but no data is yet available.

In the current study, we compared our criteria only to random testing and obvious graph coverage criteria. Our reasons were that these are currently the techniques espoused by practitioners of behavioral testing [1,14]. The overall efficiency of the criteria will be investigated in future work. One such interesting investigation is to compare criteria based on code structure and data flow. This is a interesting step in determining the usefulness of the proposed criteria and might provide insight into the white box vs. black box debate.

One final area of research is to expand the criteria to force value changes on pairs, triplets, etc, of related operational modes. It may be that we may more easily expose subtle faults which manifest as a function of multiple operational modes value changes.

Acknowledgments

We thank the many students of software engineering at Florida Tech who have scrutinized this work and used our criteria in their own testing experiments. Also, we acknowledge the helpful discussions with H. Robinson, S. Rosaria and J. Tierney of Microsoft and M. Houghtaling of IBM who are experimenting with our criteria on their products.

References

1. L. Apfelbaum, “Specification-based Tests Make Sure Telecom Software Works,” *IEEE Spectrum*, Vol. 34, No. 11, pp. 77-83, Nov. 1997.
2. B. Beizer, “Software Testing Techniques,” Van Nostrand Reinhold, New York, 1990.
3. L. A. Clarke, A Podgurski, D. J. Richardson and S. J. Zeil, “A Formal Evaluation of Data Flow Path Selection Criteria,” *IEEE Trans. Software Eng., vol 15*, no. 11, pp. 1318-1332, Nov. 1989.
4. J. W. Duran and S. C. Ntafos, “An Evaluation of Random Testing.” *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 438-444, July 1984.
5. P. Frankl and E. Weyuker, “An Applicable Family of Data Flow Testing Criteria,” *IEEE Trans. Software Eng.*, vol. 14, no. 10, pp.1483-1498, Oct. 1988.
6. D. Hamlet and R. Taylor, “Partition Testing Does Not Inspire Confidence,” *IEEE Trans. Software Eng.*, vol. 16, no. 12, pp. 1402-1411, Dec. 1990.
7. J. Laski and B. Korel, “A Data Flow Oriented Program Testing Strategy,” *IEEE Trans. Software Eng.*, vol. 9, no. 3, pp.347-354, May 1983.
8. E. Miller and W. E. Howden, *Tutorial: Software Testing and Validation Techniques*, IEEE Computer Society Press, 1981.
9. J. D. Musa, “Software Reliability Engineered Testing,” *IEEE Software,* vol. 29, no. 11, pp. 61-68, Nov. 1996.
10. G. Myers, *The Art of Software Testing*, Wiley, New York, 1979.
11. D. L. Parnas, “An Evaluation of Safety-Critical Software,” *Comm. of the ACM*, vol. 23, no. 6, pp. 636-648, June 1990.
12. J. H. Poore, H. D. Mills and D. Mutchler, “Planning and Certifying Software System Reliability,” *IEEE Software*, vol. 10, no. 1, pp. 88-99, Jan. 1993.
13. S. Rapps and E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information,” *IEEE Trans. Software Eng.*, vol. 11, no. 4, pp. 367-375, Apr. 1985.
14. H. Robinson, “Model-Based Testing on a Shoestring Budget,” to appear in *Proceedings of the Software Testing Analysis and Review Conference*, San Jose, CA, Nov. 1999.
15. E. Weyuker, “More Experience with Data Flow Testing,” *IEEE Trans. Software Eng.*, vol. 19, no. 9, pp. 912-919, Sept. 1993.
16. E. Weyuker, “The Cost of Data Flow Testing: An Empirical Study,” *IEEE Trans. Software Eng.*, vol. 16, no. 2, pp. 121-128, Feb. 1990.
17. J. A. Whittaker and M. G. Thomason, “A Markov Chain Model for Statistical Software Testing,” *IEEE Trans. Software Eng.,* vol. 20, no. 10, pp. 812-824, Oct. 1994.

18. J. A. Whittaker, “Stochastic Software Testing,” *Annals of Software Eng.*, Vol. 4, pp. 115-131, Oct. 1997.
19. J. A. Whittaker and M. A. Al-Ghafees, “Markov chain-based test data adequacy criteria,” 2000 IRMA International Conference, Alaska: IDEA Group Pub.