A Markov Chain Model for Statistical Software Testing

James A. Whittaker and Michael G. Thomason, *Senior Member, IEEE*

Abstract— Statistical testing of software establishes a basis for statistical inference about a software system's expected field quality. This paper describes a method for statistical testing based on a Markov chain model of software usage. The significance of the Markov chain is twofold. First, it allows test input sequences to be generated from multiple probability distributions, making it more general than many existing techniques. Analytical results associated with Markov chains facilitate informative analysis of the sequences before they are generated, indicating how the test is likely to unfold. Second, the test input sequences generated from the chain and applied to the software are themselves a stochastic model and are used to create a second Markov chain to encapsulate the history of the test, including any observed failure information. The influence of the failures is assessed through analytical computations on this chain. We also derive a stopping criterion for the testing process based on a comparison of the sequence generating properties of the two chains.

Index Terms— Markov chain, statistical software testing, stochastic process, test case generation

I. INTRODUCTION

THE *black box approach* [19], [28] to the software testing process unfolds as follows. Given a program P with intended function f and input domain d, the objective is to select a sequence of entries from d, apply them to P, and compare the response with the expected outcome indicated by f. Any deviation from the intended function is designated as a failure. It is assumed that f is well defined and completely specified, so that any deviation is unambiguously detected and a failure is explicitly noted. The history of the test at some time n is a sequence of inputs $d_0 d_1 d_2 \cdots d_{n-1}$ and a corresponding sequence of zero or more failures, each of which is uniquely identified with the particular input d_i at which the failure was observed.

Statistical testing follows the black box model with two important extensions. First, sequences from d are stochastically generated based on a probability distribution that represents a profile of actual or anticipated use of the software. Second, a statistical analysis is performed on the test history that enables the measurement of various probabilistic aspects of the testing process. Thus, one can view statistical testing as a sequence generation and analysis problem. A solution to the problem is achieved by constructing a generator to obtain the test input

sequences and by developing an informative analysis of the test history.

This paper describes a sequence generation and analysis technique for statistical testing using Markov chains. We discuss the construction of a Markov chain as a sequence generator for statistical testing and show how analytical results associated with Markov chains can aid in test planning. An innovative aspect of this method is that the test sequences generated and applied to the software are used to create a second Markov chain to encapsulate the history of the test, including any observed failure information. The influence of the failures is assessed through analytical computations on this chain. We also derive a stopping criterion for the testing process based on a comparison of the sequence generating properties of the two chains.

II. A STATISTICAL TESTING MODEL FOR SOFTWARE

The need for testing methods and reliability models that are specific to software has been discussed in various forms in the technical literature [3], [10], [11], [20]. Statistical testing for software is one such method. The main benefit of statistical testing is that it allows the use of statistical inference techniques to compute probabilistic aspects of the testing process, such as reliability [3], [10], [16], [20], mean time to failure (MTTF) [4], [22], and mean time between failures (MTBF) [18].

Current statistical testing techniques model software usage by assigning a single, unconditional probability distribution to individual inputs (or groups of inputs) from the software's input domain [4], [7], [8], [11], [16], [19]. This distribution represents the best estimate of the operational frequency of use for each input. Input sequences are obtained by sampling from the distribution with or without replacement (depending on the application). Obviously, this model is insufficient for many types of software, because the probability of applying an input can change as the software is executed. As software processes inputs, it moves from one *state* or *mode* to the next, depending on any or all prior inputs received. Thus, the probability of an input can change depending on the mode of the software [20]. It is necessary, therefore, to maintain multiple probability distributions for each such mode of a software system.

This paper proposes that statistical testing be carried out with a stochastic model of software usage. We define a stochastic model that is capable of modeling multiple probability distributions corresponding to pertinent software modes and is tractable for the computation of properties of informative random variables that describe its sequence generating capa-

bilities. Ideally, the parameters of the model are established using information obtained from various sources, including the software's intended function and usage patterns of prior versions or prototypes of the software. However, it is often the case that complete information about the probabilities that describe usage is not available from any source; in this case, the stochastic model is based on estimated usage patterns [27].

This *usage model* consists of elements from d, the domain of the intended function, and a probabilistic relationship defined on these elements. A test input is a finite sequence of inputs from domain d probabilistically generated from the usage model. The statistical properties of the model lend insight into the expected makeup of the sequences for test planning purposes.

As the test sequences are applied to the software, the results are incorporated into a second model. This *testing model* consists of the inputs executed in the test sequences, plus any failures discovered while applying the sequences to the software P. In other words, it is a model of what has occurred during testing. The testing model also allows analysis of the test data in terms of random variables appropriate for the application. For example, we may measure the evolution of the testing model and decide to stop testing when it has reached some suitable "steady state."

This paper explores the use of finite state, discrete parameter, time homogeneous Markov chains as the software usage and testing models for program P. For the usage model, the state space of the Markov chain is defined by externally visible modes of the software that affect the application of inputs. The state transition arcs are labeled with elements from the input domain d of the software (as described by the intended function f). Transition probabilities are uniform (across exit arcs from each state) if no usage information is available, but may be nonuniform if usage patterns are known. This model is called the *usage Markov chain*. For the testing model, the state space of the Markov chain is initially the same as the usage chain, but additional states are added to mark each individual failure. This model is called the *testing Markov chain*.

III. THE USAGE MARKOV CHAIN

A usage chain for a software system consists of states, i.e., externally visible modes of operation that must be maintained in order to predict the application of all system inputs, and state transitions that are labeled with system inputs and transition probabilities. To determine the state set, one must consider each input and the information necessary to apply that input. It may be that certain software modes cause an input to become more or less probable (or even illegal). Such a mode represents a state or set of states in the usage chain. Once the states are identified, we establish a start state, a terminate state (for bookkeeping purposes), and draw a state transition diagram by considering the effect of each input from each of the identified states. The Markov chain is completely defined when transition probabilities are established that represent the best estimate of real usage.

Consider the simple selection menu pictured in Fig. 1. The input domain consists of the up-arrow key and the down-arrow

Current Project: [   ]

Select Project
Enter Data
Analyze Data
Print Report
Exit System
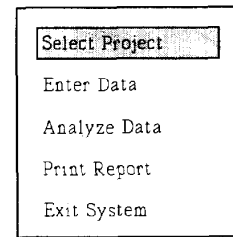
Arrow Keys to Move Cursor   Enter to Select

Fig. 1. An example software system.

key, which move the cursor to the desired menu item, and the "Enter" key, which selects the item. The cursor moves from one item to the next, and wraps from top to bottom on an up-arrow and from bottom to top on a down-arrow. The first item, "Select Project," is used to define a project (the semantics of which are not described here for simplicity). The project name then appears in the upper-right corner of the screen. Once a project is defined, the next three items, Enter Data, Analyze Data, and Print Report, can be selected to perform their respective functions. (These additional screens are also not described.) If no project is defined, selecting these items gives no response.

In this example, there are two items of interest when applying inputs. First, the current cursor location must be maintained to determine the behavior of the "Enter" key. Second, whether a project has been defined must be known to determine which of the menu items are available.

These two items of information are organized as the following *usage variables*:

1) *cursor location* (which is abbreviated CL and takes on values "Sel", "Ent", "Anl", "Prt", or "Ext" for each respective menu item), and
2) *project defined* (which is abbreviated PD and takes on the values "Yes" or "No").

The state set therefore consists of the following: {(CL = Sel, PD = No), (CL = Ent, PD = No), (CL = Anl, PD = No), (CL = Prt, PD = No), (CL = Ext, PD = No), (CL = Sel, PD = Yes), (CL = Ent, PD = Yes), (CL = Anl, PD = Yes), (CL = Prt, PD = Yes), (CL = Ext, PD = Yes)}. In addition, we include states that represent placeholders for the other system screens, as well as start and end states that represent the software in its "not invoked" mode. The state transitions are depicted in Fig. 2 in a graphical format.

This state transition diagram defines the possible input sequences for the software in a formal and concise model. A path, or connected state/arc sequence, from the initial "Uninvoked" state to the final "Terminated" state, represents a single execution of the software. A set of such sequences are used as test cases for the software. Since loops and cycles exist in the
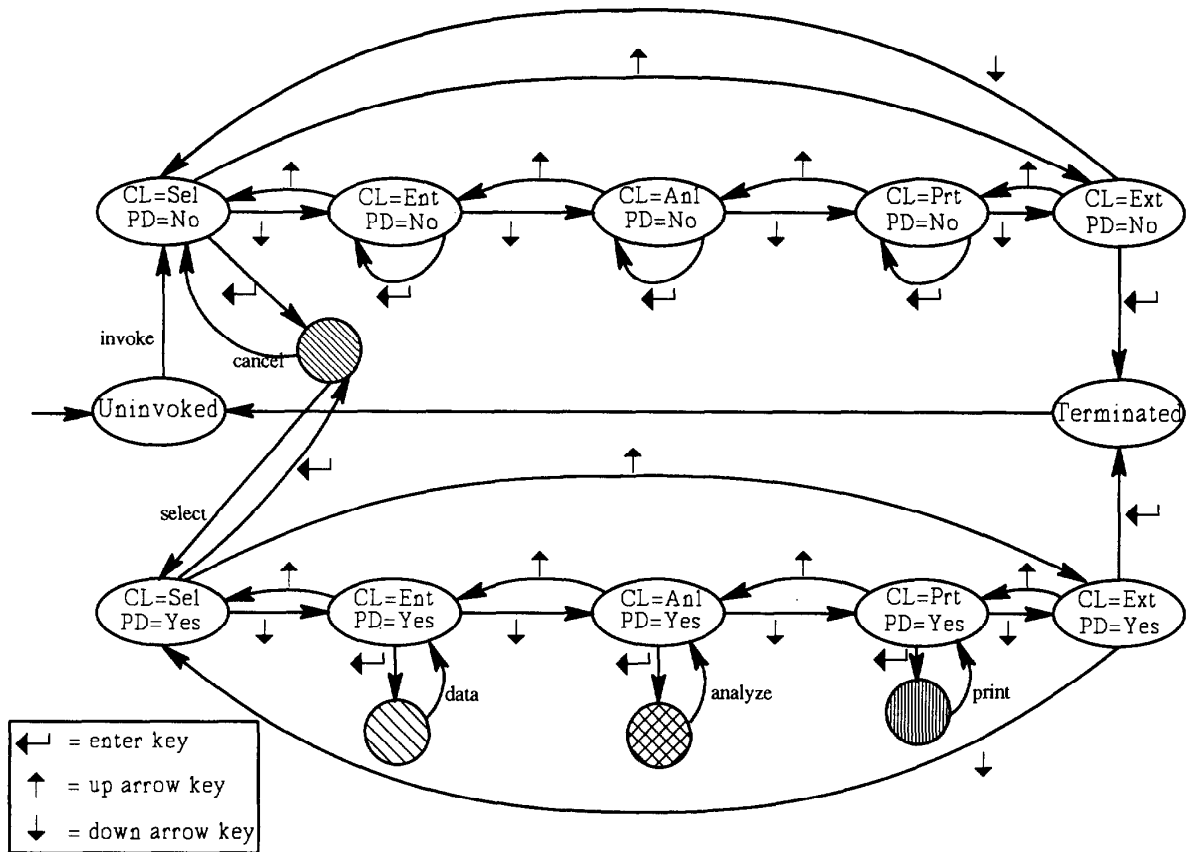
Fig. 2. Usage chain (structure) for the example.

model, an infinite number of sequences are possible. In order to generate sequences statistically, probability distributions are established over the exit arcs at each state that simulates expected field usage. The assignment of these probabilities is discussed below.

Sequences are generated from the model by stepping through state transitions (according to the transition probabilities), from "Uninvoked" to "Terminated," and recording the sequence of inputs on the path traversed. A sample input sequence from the model of Fig. 2 is: **invoke** ↵ **select** ↓↓ ↵ **analyze** ↓↓ ↵. It is readily apparent that the generation of sequences can be automated using a good random number generator and any high-level programming language. Thus, a large number of input sequences can be obtained once a usage chain is constructed.

The construction of the transition diagram identifies the probabilities that need to be estimated, i.e., the state transition probabilities. An investigation into usage patterns of the software should focus on obtaining information about these probabilities. Sequences of use from a prototype or prior version of the software, for example, may be used to estimate these probabilities. These usage sequences, captured as inputs (keystrokes, mouse clicks, bus commands, buffered data, and so forth) from the user, are mapped to states and arcs in the model in order to obtain frequency counts that correspond to state transitions. Normalizing the frequency counts establishes relative frequency estimates of the transition probabilities and completes the definition of the Markov chain.

In the event that no sequences are available to aid in the estimation of the transition probabilities, all probabilities can be distributed uniformly across the exit arcs at each state. In this case, the model building process amounts to establishing only the *structure* of usage sequences without developing any informed statistics. Table I lists each transition for the example chain in Fig. 1 with probabilities assigned both by relative frequency counts and by uniform distributions.

IV. ANALYSIS OF THE USAGE CHAIN

The fact the usage model is a Markov chain allows software testers to perform significant analysis that gives insight how the test is likely to unfold. The details of the underlying mathematics can be found in Feller [9] or Kemeny and Snell [14]; however, we have included Table II to summarize some useful results. This analysis is used to gain insight into how the test will likely unfold so that testers can proceed in an informed manner. The insight gained through the analysis can be used to aid test planning and preparation.

Our experience has been that each result summarized in Table II is useful in practice. It would be too lengthy to

TABLE I
TRANSITION PROBABILITIES FOR THE EXAMPLE USAGE MODEL

From State	Transition Stimuli	To State	Unif. Prob	Est. Prob.
Uninvoked	invoke	{CL=Sel,PD=No}	1	1
{CL=Sel,PD=No}	↓	{CL=Ent,PD=No}	1/3	1/10
	↑	{CL=Ext,PD=No}	1/3	1/10
	⌋	{CL=Sel,PD=No}	1/3	8/10
{CL=Ent,PD=No}	↓	{CL=Anl,PD=No}	1/3	1/3
	↑	{CL=Sel,PD=No}	1/3	1/3
	⌋	{CL=Ent,PD=No}	1/3	1/3
{CL=Anl,PD=No}	↓	{CL=Prt,PD=No}	1/3	1/3
	↑	{CL=Ent,PD=No}	1/3	1/3
	⌋	{CL=Anl,PD=No}	1/3	1/3
{CL=Prt,PD=No}	↓	{CL=Ext,PD=No}	1/3	1/3
	↑	{CL=Anl,PD=No}	1/3	1/3
	⌋	{CL=Prt,PD=No}	1/3	1/3
{CL=Ext,PD=No}	↓	{CL=Sel,PD=No}	1/3	1/3
	↑	{CL=Prt,PD=No}	1/3	1/3
	⌋	Terminated	1/3	1/3
{CL=Sel,PD=Yes}	↓	{CL=Ent,PD=Yes}	1/3	5/9
	↑	{CL=Ext,PD=Yes}	1/3	3/9
	⌋	Select Project	1/3	1/9
{CL=Ent,PD=Yes}	↓	{CL=Anl,PD=Yes}	1/3	4/7
	↑	{CL=Sel,PD=Yes}	1/3	1/7
	⌋	Enter Data	1/3	2/7
{CL=Anl,PD=Yes}	↓	{CL=Prt,PD=Yes}	1/3	3/6
	↑	{CL=Ent,PD=Yes}	1/3	1/6
	⌋	Anlyz Data	1/3	2/6
{CL=Prt,PD=Yes}	↓	{CL=Ext,PD=Yes}	1/3	2/6
	↑	{CL=Anl,PD=Yes}	1/3	1/6
	⌋	Prnt Report	1/3	3/6
{CL=Ext,PD=Yes}	↓	{CL=Sel,PD=Yes}	1/3	1/6
	↑	{CL=Prt,PD=Yes}	1/3	2/6
	⌋	Terminated	1/3	3/6
Select Project	cancel	{CL=Sel,PD=No}	1/2	1/8
	select	{CL=Sel,PD=Yes}	1/2	7/8
Enter Data	data	{CL=Ent,PD=Yes}	1	1
Anlyz Data	analyze	{CL=Anl,PD=Yes}	1	1
Prnt Report	print	{CL=Prt,PD=Yes}	1	1
Terminated	null	Uninvoked	1	1

describe each result in detail; however, two examples of the analytical results are given to illustrate their usefulness.

Example 1: In some automated real-time test execution environments there is a physical limit on the number of inputs in a single test case [1]. It is useful, therefore, to know the "expected length and standard deviation of the input sequences," so that overloading the test execution environment can be controlled. Using (4) with i = Uninvoked and j = Terminated, this expectation for the example chain with relative frequency estimated probabilities is 20.1, with standard deviation of 15.8. If these results were unacceptable (i.e., outside the range of the test environment), then modification of the transition probabilities would be necessary to obtain more suitable results.

Example 2: In practice, software testers are often concerned with the *coverage* of some specific attribute of the software under test. For example, Myers [19] relates coverage criteria concerning the percentage of source code executed by a set of test cases. When testing is performed from the black box point of view, coverage of elements of the input domain is often of interest [19].

Equations (6) and (8) are used to estimate the coverage of usage chain states and arcs. This measure goes beyond input domain coverage, because the software modes are represented (i.e., as states) as well as inputs (i.e., as arcs). The information is organized into percentages of states and arcs in Table III. For example, Table III indicates that 81.25% of the states have expectation of seven sequences or less until they appear in the test sequences.

The information from each of these examples is used to make the following estimates of expected effort to achieve full coverage (each result is rounded up to the nearest integer). Suppose testers determine that it takes 5 s, on average, to apply an input to the example selection menu software. An average sequence will then take $21 \times 5 = 105$ s (1 min, 45 s) to execute. Further, it will take $12 \times 105 = 1260$ s (21 min), on average,

TABLE II
SOME STANDARD ANALYTICAL RESULTS FOR MARKOV CHAINS*

Result	Equation for Prob. or Mean	Interpretation of Mean
Recurrent Chain		
stationary distribution, π	$$\pi_j = \sum_i \pi_i U_{ij} \quad (1)$$	π_j is the asymptotic appearance rate of state j in a large number of sequences from U.
recurrence time for state j	$$m_{jj} = \frac{1}{\pi_j} \quad (2)$$	The mean number of state transitions between occurrences of state j in a large number of sequences from U.
no. of occurrences of state i between occurrences of state j	$$m_{jj}\pi_i = \frac{\pi_i}{\pi_j} \quad (3)$$	The mean number of occurrences of state i between occurrences of state j.
first passage times	$$m_{ij} = 1 + \sum_{k \neq j} U_{ik} m_{kj} \quad (4)$$	The mean number of state transitions until state j occurs from state i.
Absorbing Chain (for initial state i)		
single sequence prob. for state j	$$y_{ij} = U_{ij}^a + \sum_{k \in \tau} U_{ik}^a y_{kj} \quad (5)$$	The probability that state j occurs in a single sequence (i.e., from the initial state to the absorbing state).
no. of sequences to occurrence of state j	$$h_j = \frac{1}{y_{ij}} \quad (6)$$	The mean number of sequences until state j occurs.
single sequence prob. for arc j,k	$$z_{jk} = y_{ij} U_{jk} \quad (7)$$	The probability that arc j,k occurs in a single sequence (i.e., from the initial state to the absorbing state).
no. of sequences to occurrence of arc j,k	$$h_{jk} = \frac{1}{z_{jk}} \quad (8)$$	The mean number of sequences until arc j,k occurs.
no. of occurrences of state j in a single sequence	$$m(j\mid i) = \sum_{k \in \tau} U_{ik}^a m(j\mid k) + \begin{cases} 1 & if\ i=j \\ 0 & if\ i \neq j \end{cases} \quad (9)$$	The mean number of occurrences of state j in a single sequence.

*Each measure in this table is based on the usage model encoded as a *transition matrix*, U, with states as indices and transition probabilities as entries. U is called the *recurrent model* because the arc from Terminated to Uninvoked occurs with probability 1, causing a new sequence to begin each time the previous sequence ends. The *absorbing model*, U, is achieved by redirecting the arc from Terminated to Uninvoked back to Terminated; thus, this is a model representing only single executions of the software. In this case, the state Terminated is called *absorbing*, and the other states are called *transient*. (The set of transient states is denoted τ.)

to execute enough sequences so that every state is covered and $36 \times 105 = 3780$ s (1 hr, 3 min) so that every arc is covered. In addition to these results, each measure in Table II has been used in practice to analyze some aspect of software usage or testing. More detail is presented elsewhere [1], [26].

V. THE TESTING MARKOV CHAIN

When the usage chain is complete, a series of input sequences is stochastically generated and applied to software P. The application of the test sequences can be manual or automatic, depending on the testing environment and the

availability of suitable automated support. We assume the presence of an oracle that is capable of comparing the output of P with the intended behavior, f, and correctly classifying success or failure. Thus, the history of the test at some time n is a series of input sequences (and usage chain states) $d_0 d_1 \cdots d_{n-1}$ and a corresponding sequence of failures, each of which is uniquely identified with the particular sequence and specific input d_i with which the failure was observed.

As failures are discovered and the software's internal faults repaired, the software evolves, becoming more or less reliable, depending on the success of the fixes. Each change to the

TABLE III
EXPECTATIONS FOR STATE AND ARC COVERAGE FROM THE EXAMPLE USAGE CHAIN

No. of Input Sequences	Expected Percentage of States Covered	Expected Percentage of Arcs Covered
1	18.75	86.32
2	62.50	87.89
3	68.75	89.45
4	75.00	91.01
5	75.00	91.79
6	75.00	92.18
7	81.25	92.57
8	87.50	92.57
9	87.50	93.35
10	87.50	94.92
11	87.50	95.31
12-19	100.00	95.31
20-22	100.00	96.48
23-33	100.00	97.65
34-35	100.00	98.82
36	100.00	100.00

software creates a new software version. Corresponding to each such version is a subset of the test history that represents the testing experience for that particular version. Thus, if one is interested in quantifying the behavior of a specific, homogeneous software version, then the applicable data to use as a basis for measuring this is the corresponding subset of the test history [20]. In addition, if one is interested in studying the rate at which failures are identified and how this rate varies during the complete testing process, then the applicable data are the entire test history over successive software versions. Although the entire test history pertains to no specific software version, it does represent the entire testing experience for a software project and can be helpful in analyzing the underlying software process used to create the software. The following discussion applies to either view of the testing history. In fact, both analyses can be performed simultaneously for any given project.

The test history (or any meaningful subset thereof) is a realization of a stochastic process and is appropriately analyzed by a stochastic model. In this paper, we use a stochastic model of a test history to identify the length of the test sequence that will be a suitable stopping point for testing the software, and to analyze the effect of the failures on the testing stochastic process. For these purposes, the test history is encoded as another Markov chain, the *testing Markov chain*, T. This section describes construction of the testing chain from a test history and derives an analytical stopping criterion. In addition, analytical results associated with Markov

chain theory are used to quantify the impact of the failures on the testing process.

A set of test input sequences is a realization of the usage chain U and has certain characteristics imposed by U; e.g., states and transitions appear with known probabilities in the long run. The development of these characteristics occurs probabilistically; i.e., given a new random seed, a different set of sequences could be obtained in which states and arcs are generated in a different order. Detailed analysis of the testing process therefore requires a model that itself evolves as specific testing is carried out.

A. Constructing the Testing Chain

Usage chain U has stationary transition probabilities; i.e., they do not change throughout the test. However, probabilities in testing chain T are updated, and tracking T's evolution is an inherent part of monitoring the statistical testing process. Let $s_1, s_2, \cdots, s_m$ denote the set of test sequences in the order generated by U and applied to software P. The corresponding series of testing chains $T_0, T_1, \cdots, T_m$ describes the evolution of T during testing and is constructed as follows.

Before any sequence is input to P, the test history is empty. The initial chain T_0 is a copy of usage chain U, with all arc probabilities set to 0. Assume first that no software failures occur. T_1 is obtained from T_0 by incrementing arc frequencies along the path of states from "Uninvoked" to "Terminated" in s_1. Similarly, T_2 is obtained from T_1 by sequence s_2, and, in general, T_i is obtained from T_{i-1} by sequence s_i. In

this way, frequency counts on arcs in T_i are always obtained from specific sequences applied to software P. These arc frequencies are converted to relative frequency probabilities whenever computation with T_i's state transition probabilities is required.

The testing chain's arc counts are reset when fixes are applied to P. Thus, as the software changes, a new testing chain is created to model only the sequences applied on that version. In this manner, the testing chain remains an accurate model of the testing experience of the current software version. An additional formulation is to maintain a testing chain that is not reset between fixes and incorporates testing experience across different software versions. This latter testing chain is really a model of the *process* of error discovery and fault removal, whereas the former series of chains represents each successive version of the software *product*. Either interpretation can provide valuable feedback about software development activity.

What can be said about the series $T_0, T_1, \cdots, T_m$? If no failures are detected, the evolution of T is dictated solely by sequences from U. The Strong Law of Large Numbers for Markov chains [6] guarantees (with probability 1) that these sequences $s_1, \cdots, s_m$ will become statistically typical of U when enough are generated. This means that convergence of T to U is certain, because the relative frequencies on T's arcs will converge to the probabilities on U's arcs. A key point is that the test history T is statistically typical of the usage chain U if and only if convergence is achieved.

In other words, U is a fixed reference toward which T_i evolves at an expected rate with statistical variation that depends on factors such as the source entropy of U [26]. This evolution is well controlled and predictable in statistical terms.

B. Incorporating Failure Data

Suppose now that failures do occur and that the jth failure f_j is detected during input of sequence s_i to P. To incorporate this failure event into the test history, a new state labeled f_j is placed in Markov chain T_i exactly as it was ordered in s_i. The arcs to and from the new state f_j have frequency count 1. If f_j is a catastrophic failure, then the run of software P is aborted, and the arc from f_j goes to "Terminated"; otherwise, the test sequence can continue, and the arc from f_j goes to the next state in s_i. In this way, T_i is maintained as a Markov chain that incorporates both the underlying structure of the source of test sequences, U, and the frequency count history of sequences-plus-failures as testing evolves.

Convergence of T to U is adversely affected by failures of software P during testing. To achieve convergence when failures have been observed, the relative frequency probabilities on arcs to failure states in T_i must approach 0. In this way, the probabilities on the nonfailure arcs are still forced to converge to the corresponding (nonzero) values in U. If even one failure occurs, this can be accomplished only when P responds to more test sequences without exhibiting failures. Thus, failures automatically impose additional testing to overcome their adverse impact on the convergence of T to U.

When no failures occur in the test history, convergence will ultimately be achieved. Intuitively, comparison of the actual evolution of T (including failures) with its expected evolution (without failures) supports statistical estimation of P's characteristics based on the software's actual performance. At any point in the testing process, the most recent test history T_i is available for analysis. Because T_i itself is a well-defined Markov chain, computations are based on the theory of Markov chains.

The testing chain, T, is a model of the current test history and is useful for computing properties of descriptive random variables as shown in the next section. An alternative would be to obtain statistics directly from the set of sequences executed; however, T incorporates explicitly the *structure* of the usage chain, which is only implicit in the sequences. In other words, each sequence is accorded different status according its specific attributes; e.g., sequences can vary in length and probability, and thus contribute a different amount of information to the statistical testing experiment. The testing chain incorporates each event of each sequence, recognizing the probabilistic relationship between states and arcs established in the usage chain. Any computation based on T incorporates this information as well. Thus, T is an important model for the identification and derivation of measures that describe the statistical testing process. See [26] for proofs concerning specific attributes of testing chains.

To illustrate testing chain construction, consider the example usage chain of Fig. 2. The initial testing chain, before any sequences are executed, is a copy of this chain, with each arc frequency initialized at zero. A randomly generated sequence is then obtained and executed against the software. The testing chain is updated to reflect the states and arcs traversed in that sequence. For example, the following sequence causes the corresponding transition arcs in the testing chain to be updated. (States are included in the sequence for reference; individual inputs are indented.)

Uninvoked
   invoke   update transition: (Uninvoked,
{CL = Sel, PD = No}) {CL = Sel, PD = No}
                              from 0 to 1
   Enter key   update transition: ({CL = Sel,
                              PD = No}, Select Project)
                              from 0 to 1

Select Project Screen
   select   update transition: (Select Project,
                              {CL = Sel, PD = Yes})
                              from 0 to 1

{CL = Sel, PD = Yes}
   Dn Arrow key   update transition:
                              ({CL = Sel, PD = Yes},
                              {CL = Ent, PD = Yes})
                              from 0 to 1

{CL = Ent, PD = Yes}
   Dn Arrow key   update transition:
                              ({CL = Ent, PD = Yes},
                              {CL = Anl, PD = Yes})
                              from 0 to 1

{CL = Anl, PD = Yes}
    Dn Arrow key   update transition:
                        ({CL = Anl, PD = Yes},
                        {CL = Prt, PD = Yes})
                        from 0 to 1

{CL = Prt, PD = Yes}
    Enter key   update transition:
                        ({CL = Prt, PD = Yes}, Print
                        Report) from 0 to 1

Print Report Screen
    print   update transition: (Print Report,
                        {CL = Prt, PD = Yes})
                        from 0 to 1

{CL = Prt, PD = Yes}
    Enter Key   update transition: ({CL = Prt,
                        PD = Yes}, Print Report)
                        from 1 to 2

Print Report Screen
    print   update transition: (Print Report,
                        {CL = Prt, PD = Yes})
                        from 1 to 2

{CL = Prt, PD = Yes}
    Dn Arrow key   update transition:
                        ({CL = Prt, PD = Yes},
                        {CL = Ext, PD = Yes})
                        from 0 to 1

{CL = Ext, PD = Yes}
    Enter key   update transition:
                        ({CL = Ext, PD = Yes},
                        Terminated) from 0 to 1
Terminated   update transition: (Terminated,
                        Uninvoked) from 0 to 1

Suppose now that a failure appeared during printing that caused the system to halt execution. This same sequence, under these circumstances, would achieve the following updates in the testing chain.

Uninvoked
    invoke   update transition: (Uninvoked,
                        {CL = Sel, PD = No})
                        from 0 to 1

CL = Sel, PD = No
    Enter key   update transition:
                        ({CL = Sel, PD = No},
                        Select Project) from 0 to 1

Select Project Screen
    select   update transition: (Select Project,
                        {CL = Sel, PD = Yes})
                        from 0 to 1

{CL = Sel, PD = Yes}
    Dn Arrow key   update transition:
                        ({CL = Sel, PD = Yes},
                        {CL = Ent, PD = Yes})
                        from 0 to 1

{CL = Ent, PD = Yes}
    Dn Arrow key   update transition:
                        ({CL = Ent, PD = Yes},
                        {CL = Anl, PD = Yes})

                                      from 0 to 1
{CL = Anl, PD = Yes}
    Dn Arrow key   update transition:
                        ({CL = Anl, PD = Yes},
                        {CL = Prt, PD = Yes})
                        from 0 to 1
{CL = Prt, PD = Yes}
    Enter key   update transition:
                        ({CL = Prt, PD = Yes},
                        Print Report) from 0 to 1
Print Report Screen
    print   add state: Failure State j
                        update transition: (Print Report,
                        Failure State j) from 0 to 1
Failure State i update transition: (Failure State
                        j, Terminated) from 0 to 1
Terminated   update transition: (Terminated,
                        Uninvoked) from 0 to 1

Thus, the testing chain is updated with frequency counts that reflect the actual events that occurred when the sequence was executed. If the failure had not caused the system to halt, then the testing chain would be updated with the failure state followed by the remaining sequence parts. Whenever computation is desired, the frequency counts are normalized to probabilities.

VI. ANALYTICAL RESULTS FOR THE TESTING CHAIN

In this section, the testing chain, T, is used to obtain analytical results to answer two questions. First, at what point does the test history become representative of usage (as defined by U); second, how does each failure impact the testing process?

A. An Analytical Stopping Criterion

Stopping criteria for statistical software testing can be as simple as choosing some target reliability [3], [4], [10], [18], [20], [22], [24], and testing until the estimate of the reliability meets or exceeds the target. However, the usage-to-testing-chain approach suggests an analytic stopping criterion based directly on the statistical properties of the usage and testing chains. The usage chain is a model of ideal testing of the software; i.e., each arc probability is established with the best estimate of actual usage, and no failure states are present. The testing chain, on the other hand, is a model of a specific test history, including failure data. Thus, the usage chain represents what *would* occur in the statistical test in the absence of failures, and the testing chain represents what *has* occurred. Dissimilarity between the two models is therefore a useful measure of the progress of testing. When the dissimilarity is small, the test history is an accurate picture of the usage model.

Failure states are introduced into the testing chain by actual observations of software failure during testing. Since the usage chain does not have these failure states, they have an implied long-run probability of zero in U. In order to match the stochastic characteristics of the testing chain T in which the failure states may exist, enough nonfailure sequences from U
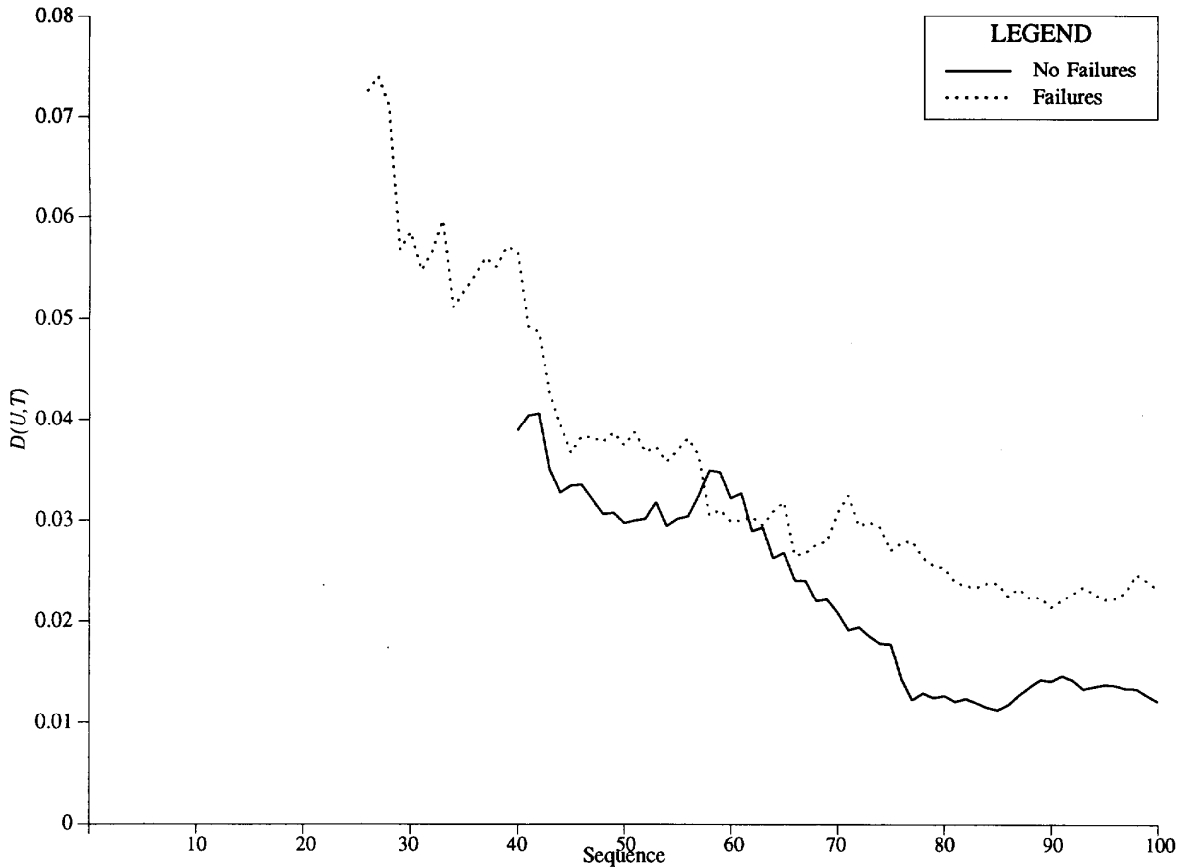
Fig. 3. Two plots of $D(U, T)$ for the example usage chain.

must end in correct termination to push the long-run occupancy of all failure states in T close to zero. Thus, failures observed during testing tend to increase the number of sequences that must be applied to P.

Regardless of whether failures are encountered, we are seeking to identify the point at which the stochastic properties of the usage chain and the testing chain are indistinguishable within some acceptable tolerance. In order to measure this, one could compute, for example, the stationary distribution of each chain and use a goodness of fit criterion (e.g., Chi-squared [5]) to measure their similarity. However, this approach takes into account only a single (albeit very important) attribute of the two models. If we want to measure the difference of the ensemble characteristics of each chain, then another approach is desirable.

Consider each chain as an ergodic stochastic source. Each chain has a set of typical sequences that accurately characterize it as a sequence generator. If both chains have the same set of typical sequences, we may draw the conclusion that the two chains are indistinguishable as sequence generators. Stated differently, it should be extremely difficult, if not impossible, to determine whether a long concatenation of sequences $d_0 d_1 \cdots d_{n-1}$ was generated by U or T.

The *log likelihood ratio* [15] is a fundamental computation in measuring the evidence an observation provides for or against a hypothesis. In this case, the hypothesis is "stochastic process U is equivalent to stochastic process T," and an observation is a large number of sequences generated by recurrent chain U. We define a measure for two stochastic processes as the expected value of the log likelihood ratio, called the *discriminant* [15]. This value is computed for two arbitrary ergodic stochastic processes λ_0 and λ_1 [13] as follows:

$$D(\lambda_0, \lambda_1) = \lim_{n \to \infty} \frac{1}{n} [\log_2 p(d_0 d_1 \cdots d_{n-1} | \lambda_0) - \log_2 p(d_0 d_1 \cdots d_{n-1} | \lambda_1)], \qquad (10)$$

where $p(d \cdots | \lambda)$ denotes the probability with which stochastic process λ generates sequence d. Although $D(\lambda_0, \lambda_1)$ cannot be directly computed for arbitrary processes λ_0 and λ_1, it can be computed for Markov chains U and T [26] as follows:

$$D(U, T) = \sum_{ij} \pi_i p_{ij} \log_2 \frac{p_{ij}}{\hat{p}_{ij}}, \qquad (11)$$

where π is the stationary distribution of U, p_{ij} is the probability of a transition from i to j in U, and $\hat{p}_{ij}$ is the corresponding probability in T. Each $\hat{p}_{ij}$ that corresponds to a nonzero p_{ij} must be greater than zero in order for $D(U, T)$ to be defined. $D(U, T)$ is non-negative and equal to zero if and only if $p_{ij} = p_{ij}$ for all i, j [15].
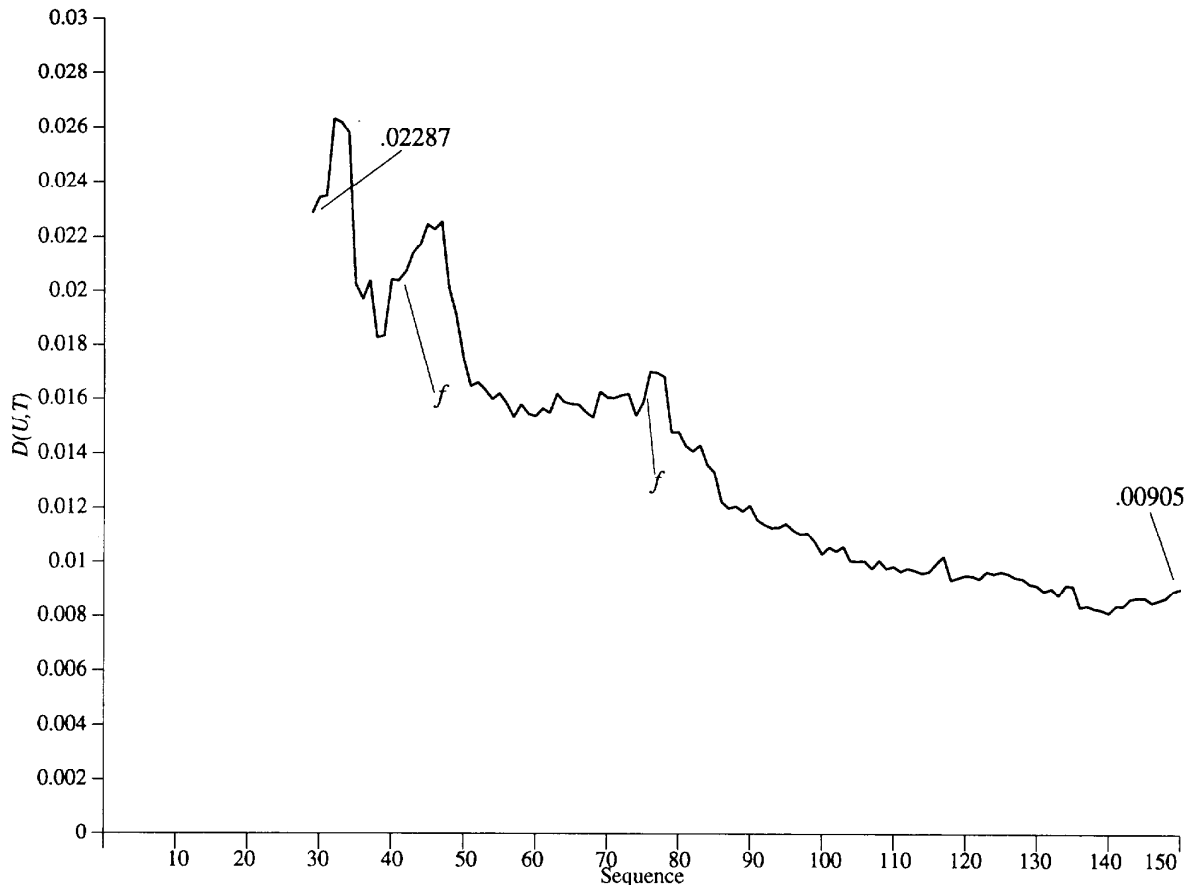
Fig. 4. Plot of $D(U, T)$.

The two sources U and T are likely to generate a similar set of typical sequences only if the value for $D(U, T)$ is very small. A value of $D(U, T)$ approaching zero has several implications for software testing. First, it ensures that each usage state appears in the test history in the correct proportion, as computed in the stationary distribution of U, and that the sequencing properties of the test history closely match those of U. Second, it forces the probability of occurrence of the failure states in T to be pushed toward zero. This means that confidence must be gained in every path through the testing chain before $D(U, T)$ will be acceptably small. Third, it recognizes the limitations of the usage chain for testing the software. When the statistics of the testing chain and usage chain match, the usage chain is unlikely to generate a sequence that adds any additional information to the statistical testing experiment.

To monitor the testing process, $D(U, T)$ can be computed with each sequence applied to the software after T becomes fully defined. A downward trend in the values of $D(U, T)$ signifies growing similarity of the two models. Usage chain U never changes; however, $D(U, T)$ reflects the impact of each additional sequence on the stochastic characteristics of the testing chain. $D(U, T)$, for example, can rise when no failures are observed if a sequence reinforces some low-probability event. Of course, a rise is expected when a failure

occurs. When the discrimination drops below some predefined threshold and experiences little change for an extended period, it is implied that additional test sequences will not significantly impact the statistics of the testing model, and testing can stop.

Fig. 3 shows two plots of $D(U, T)$ that depict typical behavior of the function. Each plot represents a separate series of sequences from the example usage chain with relative frequency estimated transition probabilities. The solid line depicts behavior of $D(U, T)$ with no failures. The dotted line depicts a sequence with three failure states. There are several interesting features of this figure. First, note that $D(U, T)$ becomes computable at sequence 40 for the first plot and at sequence 26 for the second plot. Since $D(U, T)$ is computable only when every arc in T has been initialized (i.e., generated by U and applied to the software), its first occurrence is a random variable and depends on the specific sequences generated by U. Second, the failure states cause T to converge to U more slowly than without failures. The general trend of the failure-free plot is toward significantly smaller values than the plot with failures. Third, the fluctuation of $D(U, T)$, even in the absence of failures, can be seen in both plots. When states and arcs occur in a sequence which reinforces low probability events, $D(U, T)$ can rise significantly. This is made explicit in the plot at sequence 56, where the failure-free plot rises significantly and even surpasses the plot with failures
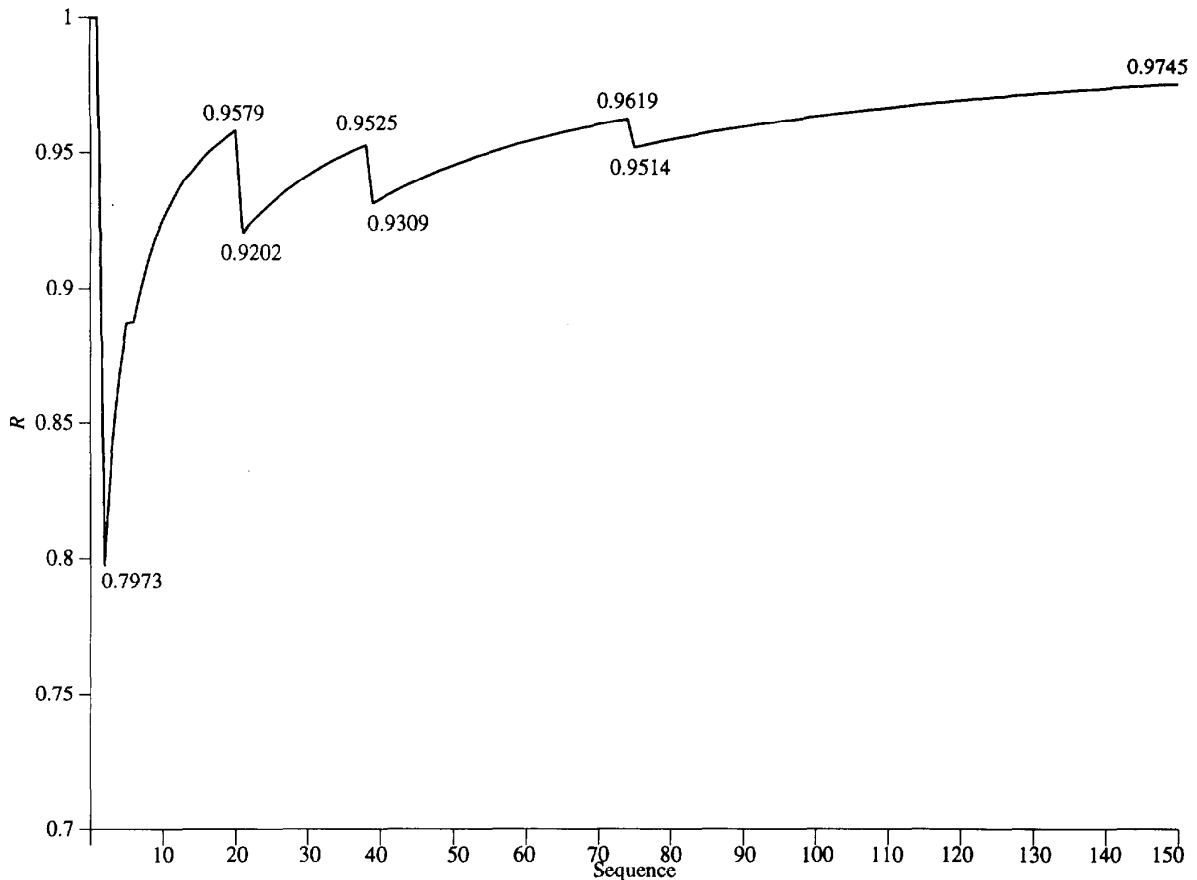
Fig. 5. Plot of R.

temporarily. The general trend of both curves is downward; however, each is affected by the appearance of atypical events in the sequences. It is important to stress that analysis of $D(U, T)$ should involve *trends* in the values of the function over time rather than any single value at some specific point in time.

Fig. 4 depicts a graph of $D(U, T)$ from a real usage chain used to test a graphical user interface [26]. $D(U, T)$ is computed after each sequence beginning at sequence 29 when the last arc is generated by the usage chain. This graph illustrates typical behavior of $D(U, T)$ during both failure and nonfailure sequences. When no failure occurs, $D(U, T)$ either falls or rises, depending on whether the current sequence causes the testing chain to become more or less similar to the usage chain. Note that the general trend is downward under this circumstance. When a failure is observed, a rise in $D(U, T)$ occurs that is sustained over several subsequent failure-free sequences. The effect of the failure starts to diminish only after multiple failure-free sequences are incorporated that reinforce the paths that avoid the failure state.

B. Measuring the Impact of Failures

The testing chain represents the test history of the software, P, during correct functioning and during software failure. Thus, it is possible to define random variables that characterize

the relationship of failure states to nonfailure states in order to describe the impact of failures on the testing stochastic process. We compute two characteristics of the testing chain that give insight into the effect of the failures. The first is the probability of a failure free realization of the testing chain, denoted R, computed by using a standard result from Markov chain theory. The second is the expected number of steps between failure states, denoted M, which requires a new computation.

R and M can be computed directly from the testing chain T at any time during the testing of software P, even when only a single sequence has been input to P. It must be emphasized that R is a probability and M is an expected value *conditioned* on the test history encoded as T. These values gain credibility as statistical measures as the discrimination $D(U, T)$ becomes relatively small, for this indicates that T is becoming statistically typical of software P's response to the input sequences from usage chain U.

The *probability, R, of a failure-free realization* of the testing chain is the probability that a realization of T beginning with "Uninvoked" and ending with the first occurrence of "Terminated" will not contain a failure state. To compute R, each failure state and "Terminated" are made absorbing states. R is the probability that absorption occurs at "Terminated,"
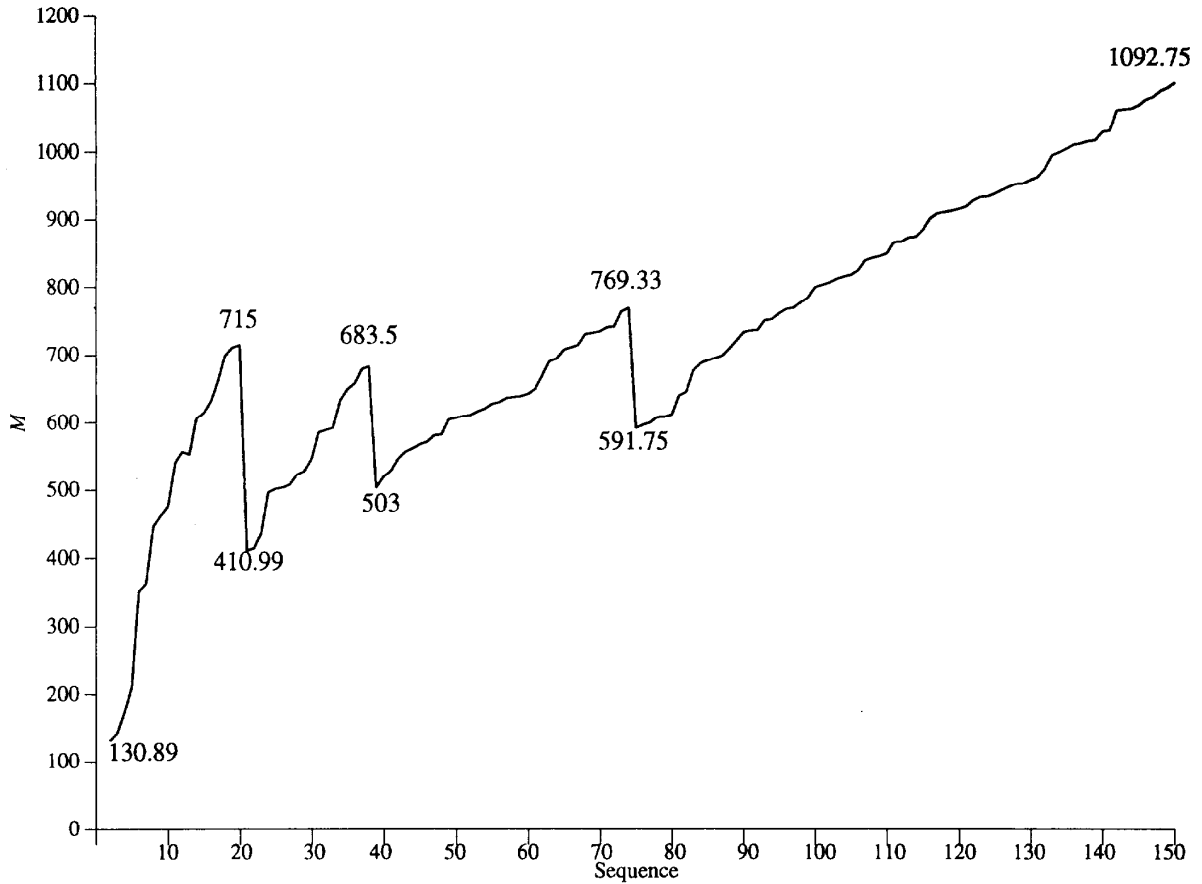
Fig. 6. Plot of M.

given "Uninvoked" as the start state [9], [14]; namely, as follows:

$$R_{\text{Unin,Term}} = \hat{p}_{\text{Unin,Term}} + \sum_{j \in \tau} \hat{p}_{\text{Unin},j} R_{j,\text{Term}}, \qquad (12)$$

where τ is the set of transient (nonabsorbing) states.

Fig. 5 depicts a plot of R for 150 sequences generated from the data of Fig. 4. The smoothness of the curve is due to the fact that the measurement is obtained by multiplying probabilities, and thus the effect of any one sequence is small. Failures on high-probability paths will cause a sharper decrease in R, because the failures are probability-weighted according to their location in the chain. Note that $R = 1$ when no failure states exist in T. Because it is a conditional probability, R gains credibility as $D(U, T)$ gets small. See Miller *et al.* [16] or Parnas *et al.* [20] for an alternative formulation for this probability.

The *expected number of steps between failure* is the expected number of state transitions encountered between occurrences of failure states in the testing chain. This value is computed [26] as follows:

$$M = \sum_{i \in f_1, \cdots, f_m} v_i \left(\sum_{j \in u_1, \cdots, u_n} \hat{p}_{ij}(m_j + 1) \right), \qquad (13)$$

where v_i is the conditional long-run probability for failure state f_i, given that the process is in a failure state, m_j is the mean number of steps until the first occurrence of any failure state from j, $u_1, \cdots, u_n$ is the set of usage chain states, and $f_1, \cdots, f_m$ is the set of failure states.

Fig. 6 is a plot of M for 150 sequences generated from the data of Fig. 4. Since M counts the number of steps between failure states, which could grow significantly when arcs are traversed for the first time, the increase tends to be more pronounced than the measure for R. Thus, when new paths are established by traversing arcs for the first time, the increase can be quite large. However, as the testing chain becomes complete, the changes are less dramatic.

The analytical results computed for the testing chain have several beneficial features. First, they are based on actual occurrences of failures. No assumptions about the distributions of failures are required in order to measure these quantities. Second, each state generated is accounted for in the computations. Each sequence of states contributes to the model in proportion to its length and probability of occurrence. The computations on the model take into account the facts that the sequences are not equally likely and that some have more impact than others. Third, each failure is probability weighted according to its location in the testing chain. Failures

attached to relatively high-probability paths will impact the testing stochastic process more than failures attached to lower-probability paths. Thus, the testing chain delivers results that are based on the usage patterns described in the usage model.

VII. CONCLUSION AND PROSPECTS FOR FUTURE WORK

The finite state, discrete parameter, time homogeneous Markov chain represents a practical option for software test engineers in the development and analysis of usage models and automatic test input generation. There have been several successful applications of Markov chain usage models to date [1], [27], involving both real-time embedded systems and user-oriented applications. Our experience has shown that Markov chain usage models can be constructed in a diverse set of application domains, and are useful for driving statistical tests.

It is sometimes the case that model size (i.e., the number of states) becomes unwieldy for large and complex systems. However, in such cases, many states are duplicates of other states, because certain inputs can be applied in different software modes. Thus, maintaining large chains often becomes a library problem that can be automated. We have also found it useful to model usage of such systems in a more abstract form. For example, the software system of Fig. 1 could be modeled with only the PD = {Yes, No} usage variable by creating the abstract inputs "choose the Select Project option," "choose the Enter Data option," "choose the Analyze Data option," and "choose the Print Report option." Thus, the usage variable for cursor location has been effectively removed by including the necessary information in the abstract inputs. We are investigating the details of these more abstract models, including the gain/loss in test effectiveness and rules for when it is or is not beneficial.

The analysis of the testing chain is currently intended as a supplement to the many reliability models that exist in the literature. The testing chain represents a new perspective on test data and bypasses assumptions concerning anticipated rates of failure appearance. However, it is not yet offered as a complete reliability model for software. Our current research is directed toward this end.

REFERENCES

[1] K. Agrawal and J. A. Whittaker, "Experiences in applying statistical testing to a real-time embedded software system," *Proc. Pacific Northwest Software Quality Conf.*, 1993, pp. 154–170.
[2] R. Ash, *Information Theory and Coding.* New York: McGraw-Hill, 1963.
[3] R. C. Cheung, "A user-oriented software reliability model," *IEEE Trans. Software Eng.*, vol. SE-6, Mar. 1980.
[4] P. A. Currit, M. Dyer, and H. D. Mills, "Certifying the correctness of software," *IEEE Trans. Software Eng.*, vol. SE-12, no. 1, pp. 3–11, Jan. 1986.
[5] H. Cramer, *The Elements Probability Theory.* Huntington, NY: Robert E. Krieger, 1955.
[6] J. L. Doob, *Stochastic Processes.* New York: Wiley, 1953.
[7] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Software Eng.*, vol. SE-10, no. 4, pp. 438–444, July 1984.
[8] J. W. Duran and J. J. Wiorkowski, "Quantifying software validity by sampling," *IEEE Trans. Reliability*, vol. R-29, no. 2, pp. 141–144, June 1980.
[9] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1. New York: Wiley, 1950.
[10] R. Hamlet, "Testing software for software reliability," Tech. Rep. TR-91-2, rev. 1, Dept. of Comput. Sci., Portland State Univ., Portland, OR, USA, Mar. 1992.
[11] R. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. Software Eng.*, vol. 16, pp. 1402–1411, Dec. 1990.
[12] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory.* Reading, MA: Addison-Wesley, 1979.
[13] B. H. Juang and L. R. Rabiner, "A probabilistic distance measure for hidden Markov models," *AT&T Tech. J.*, vol. 64, no. 2, pp. 391–408, Feb. 1985.
[14] J. G. Kemeny and J. L. Snell, *Finite Markov Chains.* New York: Springer-Verlag, 1976.
[15] S. Kullback, *Information Theory and Statistics.* New York: Wiley, 1958.
[16] K. W. Miller, L. J. Morrell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas, "Estimating the probability of failure when testing reveals no failures," *IEEE Trans. Software Eng.*, vol. 18, pp. 33–43, Jan. 1992.
[17] H. D. Mills, "The new math of computer programming," *Commun. ACM*, vol. 18, no. 1, pp. 43–48, Jan. 1975.
[18] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312–321, Aug. 1975.
[19] G. J. Myers, *The Art of Software Testing.* New York: Wiley, 1979.
[20] D. L. Parnas, A. J. Van Schouwen, and S. P. Kwan, "An evaluation of safety-critical software," *Commun. ACM*, vol. 23, pp. 636–648, June 1990.
[21] E. Parzen, *Stochastic Processes.* San Francisco, CA: Holden-Day, 1962.
[22] M. L. Shooman, *Software Engineering: Design, Reliability, and Management.* New York: McGraw-Hill, 1983.
[23] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.
[24] K. Siegrist, "Reliability of systems with Markov transfer of control," *IEEE Trans. Software Eng.*, vol. 14, pp. 1049–1053, July 1988.
[25] M. G. Thomason, "Generating functions for stochastic context-free grammars," *Int. J. Patt. Recognition Art. Intell.* , vol. 4, pp. 553–572, Apr. 1990.
[26] J. A. Whittaker, "Markov chain techniques for software testing and reliability analysis," Ph.D. dissertation, Dept. of Comput. Sci., Univ. of Tennessee, Knoxville, USA, 1992.
[27] J. A. Whittaker and J. H. Poore, "Markov analysis of software specifications," *ACM Trans. Software Eng. Methodology*, vol. 2, pp. 93–106, Jan. 1993.
[28] D. M. Woit, "Realistic expectations of random testing," CRL Rep. 246, McMaster Univ., Hamilton, ON, Canada, May 1992.

J. A. Whittaker received the B.A. degree from Bellarmine College, Louisville, KY, USA, in 1987, and the M.S. and Ph.D. degrees from the University of Tennessee, Knoxville, TN, USA, in 1990 and 1992, respectively.

He currently works as a computer scientist for Software Engineering Technology, Inc., and is an Adjunct Assistant Professor of Computer Science at the University of Tennessee, Knoxville, TN, USA. His research interests include methodical and statistical techniques for software testing and software reliability engineering.

M. G. Thomason (S'63–M'65–SM'83) received the B.S. degree from Clemson University, Clemson, SC, USA, in 1965, the M.S. degree from Johns Hopkins University, Baltimore, MD, USA, in 1970, and the Ph.D. degree from Duke University, Durham, NC, USA, in 1973.

He worked at the Westinghouse Defense and Space Center, Baltimore, MD, USA, has been a consultant for Perceptics Corp., Knoxville, TN, USA, as well as other companies, and is currently Professor of Computer Science at the University of Tennessee, Knoxville, TN, USA. His research interests include structural pattern analysis and stochastic processes in computer science.

Dr. Thomason is a member of Sigma Xi, Tau Beta Pi, and the ACM.