Markov Analysis of Software Specifications

JAMES A. WHITTAKER
Software Engineering Technology, Inc.

and

J. H. POORE
University of Tennessee

A procedure for modeling software usage with the finite state, discrete parameter Markov chain is described. It involves rigorous analysis of the specification before design and coding begin. Many benefits emerge from this process, including the ability to synthesize a macro level usage distribution from a micro level understanding of how the software will be used. This usage distribution becomes the basis for a statistical test of the software, which is fundamental to the Cleanroom development process. Some analytical results known for Markov chains that have meaningful implications and interpretations for the software development process are described.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Test data generators*; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*; I.6.5 [**Simulation and Modeling**]: Model Development; K.6.3 [**Management of Computing and Information**]: Software Management—*software development*

General Terms: Metrics, software development, software quality assurance, specifications, test-data generators

Additional Key Words and Phrases: Box Structure method, certification, Cleanroom, Markov chain, software specification, statistical test, stochastic process, usage distribution

1. MARKOV CHAINS AS MODELS FOR SOFTWARE USAGE

Cleanroom software engineering [6] results in a certification of the reliability of a software system based on the Certification Model [2]. The Certification Model is itself based on statistical testing [11], which requires random selection of test cases from the input domain of a software system according to the intended usage distribution. Two key aspects of this form of software testing are the usage distribution and the generation of test cases. Previous

Authors' addresses: J. A. Whittaker, Software Engineering Technology, Inc., 2770 Indian River Blvd., Vero Beach, FL 32960; J. H. Poore, Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301.

applications of Cleanroom have used stochastic grammars [5] as the basis for statistical testing. In this paper we explore the closely related idea of Markov analysis [12] of the specification of a software system as the basis for constructing the usage distribution and for generating random test cases based on that distribution.

Cleanroom software engineering produces software systems in a series of increments under statistical quality control [8]. These increments are organized based on information gleaned from a Box Structure [7] analysis and design. This results in a black box view of the system as depicted in Figure 1. Stated differently, the black box view of the system expresses the specification in terms of carefully detailed stimuli, responses, and transition rules. Specifications in this form are amenable to usage modeling of the system so specified. The point of this paper is to demonstrate that the Markov chain is useful in defining the underlying probability system of software usage and in guiding the statistical test.

Statistical software testing is a random experiment, and as such requires the complete characterization of the sample space and its associated probability distribution, the definition of the appropriate event space, and a method of computing properties of descriptive random variables [3]. The sample space is the input domain of the software as indicated by the enumeration of stimuli in the specification document. The selection of points from the sample space is governed by some unknown probability distribution. In statistical testing the events of interest are *sequences* of stimuli that represent an execution of the software. These sequences constitute the event space of the specified software and are obtained by defining an ordering on the points in the sample space. It is the sequences that are ultimately the important attribute of the random experiment, for they represent the test cases for the software. Statistical descriptions of the sequences are desirable in order to gain insight into the makeup of the test cases and of how many are necessary to certify the software. These descriptions are obtained by defining random variables that describe the profile of the entire set of sequences that will be used to certify the software.

The nature of the statistical testing experiment is centered around sequences of events, and as such can be modeled by a stochastic process. Thus, we define a stochastic model to guide test case generation and to compute pertinent usage statistics. In this paper we explore the use of the finite state, discrete parameter Markov chain to model software usage and to conduct statistical testing. The states of the Markov chain represent entries from the input domain of the software. The arcs of the chain define an ordering that determines the event space, or sequences, of the experiment. Furthermore, the ergodic nature of the chain induces a probability distribution on the sample space (i.e., the usage distribution) and facilitates the computation of pertinent random variables that describe the underlying statistics of the experiment.

The Markov chain proves to be a good model for several reasons. From the software engineering point of view, definitions and properties of the model ensure the testability of the software. Once a model has been built, any

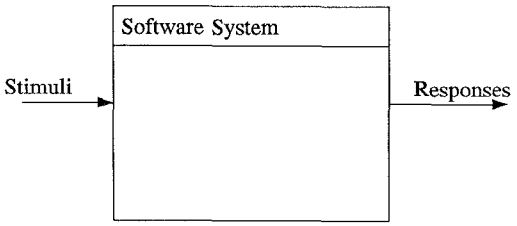

Fig. 1. Black box view of a software system.

number of statistically typical test cases can be obtained from the model. From an analytical point of view, this is a tractable stochastic process and a good basis for statistical testing. There is a rich body of theory, analytical results, and computational algorithms. As will be shown, standard analytical results for Markov chains have important interpretations for software development. Furthermore, the work is based upon an analysis of the specification, which means that all of the information provided by the model, including the usage distribution and test cases, is available before a line of code is written.

2. MARKOV ANALYSIS OF SOFTWARE SPECIFICATIONS

The fundamental step in the Markov analysis of a software specification is to define the underlying probability law for the usage of the software under consideration. This analysis of the specification, performed prior to design and coding, yields an irreducible Markov chain [3] which we call the *usage Markov chain*. This chain has a unique start state S_0 (which represents invocation of the software) a unique final state S_F (which represents termination of the software) and a set of intermediate usage states $\{S_i\}$. Each usage state is labeled with a stimulus from the input domain of the software. The state set $S = \{S_0, \{S_i\}, S_F\}$ is ordered by the probabilistic transition relation $(S \times [0,1] \times S)$. For each arc defined by this relation, the next state is independent of all past states given the present state. This is called the *Markov property*. A chain that possesses this property is said to be a *first order* chain. The usage Markov chain has a two-phase construction. In the *structural phase* the states and arcs of the chain are established, and in the *statistical phase* the transition probabilities are assigned.

At the highest level, software usage can be described by the three-state transition diagram depicted in Figure 2. For application software, the software is invoked, a cycle of usage ensues, and eventually the software is terminated and control is returned to the operating system. (An analogous statement can be made for real-time software using "power on" and "power off".) The usage state is expanded by consulting the specification for all possible actions immediately following invocation. A new state is created for each action with an arc from the invocation state as its incoming arc. Exit arcs from the newly created states are established by determining (from the specification) which actions can be performed following the state in question. These arcs can be directed to preexisting states or they can lead to the creation of new states in the same manner. This method of creating states

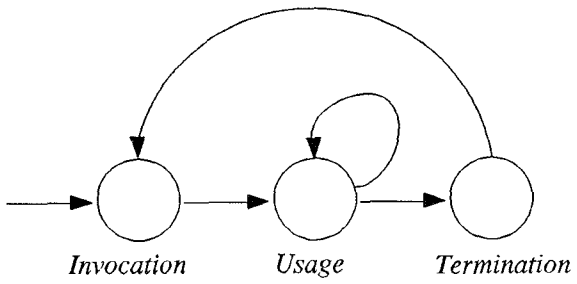

Fig. 2. Top level view of software usage.

yields a directed graph whose states are labeled with entries drawn from the input domain of the software. The arcs of the graph define an ordering that the inputs must obey.

As an example, consider the window pictured in Figure 3 and specified in Table I. Although this is an abbreviated specification, it is sufficient to describe the functionality for the purpose of this example. Invoking the software causes the window to be displayed, thus a state labeled *Window* is created as a neighbor of the invocation state. In order to determine the exit arcs from this new state, each possible action from the *Window* state is considered; *Maximize* (▲), *Minimize* (▼), *Move*, *Size*, and *Close*. The creation of these states yields the transition diagram of Figure 4. Each of these states is analyzed in turn to determine the placement of their respective exit arcs. The state labeled *Maximize* simply returns the user to a fully functional window. Thus, the single exit arc from this state is back to the preexisting *Window* state. The state labeled *Minimize* requires another series of states to be created to model the icon behavior. Establishing the state *Icon* as a neighbor of *Minimize*, we note that only one action is specified from the *Icon* state, namely *Restore* (which simply returns the user to the window). Thus the series, on single transitions with probability one, is *Window*, *Minimize*, *Icon*, *Restore*, and *Window*. *Move* and *Size* require mouse activity to be modeled. When either is selected, a directive to drag the mouse (*Drag Mouse*) is given, followed by the direction (if more detail is desired, a distance could also be supplied). Since the user may choose to change the direction of the mouse movement, the exit arcs from the direction states return to *Drag Mouse* in order to allow continued and varied movement. When all movement is complete, the window is restored (marked by a return to the *Window* state). Finally, the state labeled *Close* simply terminates the window. Thus its single exit arc is to the termination state. The complete transition diagram appears in Figure 5.

It is necessary to note that the construction of the usage chain is a creative *design step* and not an algorithm. Formal, mathematical specification documents lend themselves to more systematic model construction than natural language specification. Good engineering practices are emerging to guide the design of a usage chain from a formal specification.

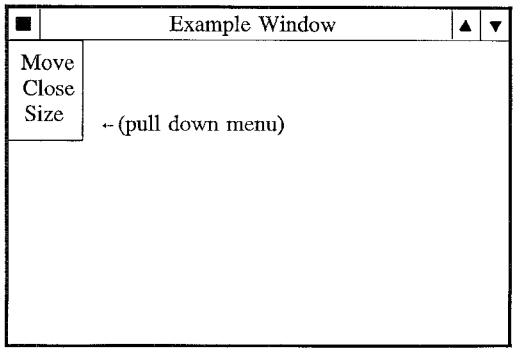

Fig. 3. An example window.

Table I. Example Software Specification

Stimulus	Response
Invocation	Place the window of figure 3.2 on the screen
Select ▲	Expand the window dimensions to cover the entire area of the screen
Select ▼	Remove the window and replace it with its corresponding icon
Select ▪ and choose *Move* from the pull down menu	Move the window as directed by the mouse input (obeying screen boundaries)
Select ▪ and choose *Size* from the pull down menu	Size the window as directed by the mouse input (obeying minimum and maximum limits)
Select ▪ and choose *Close* from the pull down menu	Remove the window from the screen
Select the icon and release	Remove the icon from the screen and restore the window

The structural phase is complete when usage (as defined by the specification) is completely modeled. At this stage the assignment of transition probabilities is the last step toward completion of the chain. This step constitutes the statistical phase of the usage model construction. There are three approaches to the statistical phase.

The first is called the *uninformed* approach. It consists of assigning a uniform probability distribution across the exit arcs for each state. This approach, which maximizes the entropy [1] across the exit arcs, produces a unique model and is useful when no information is available to allow a more informed choice.

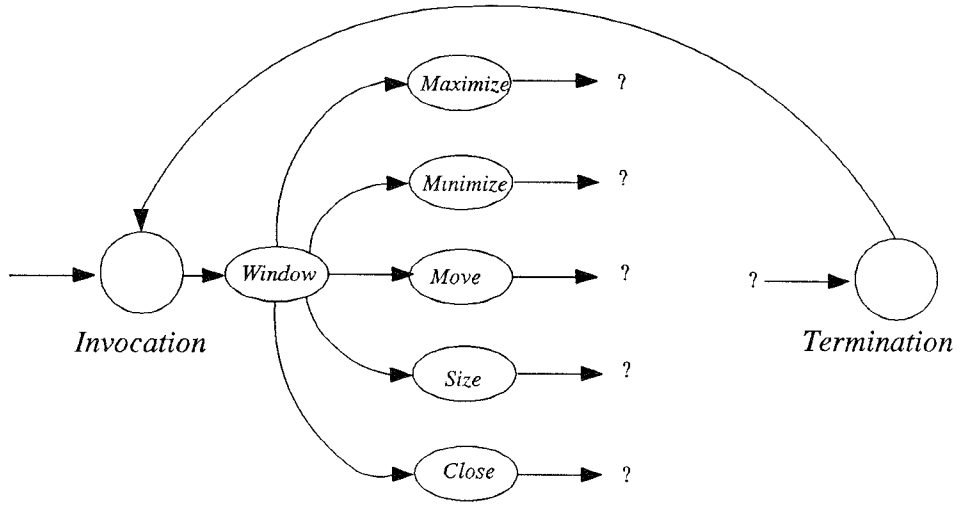

Fig. 4. Expansion of the top level usage diagram.

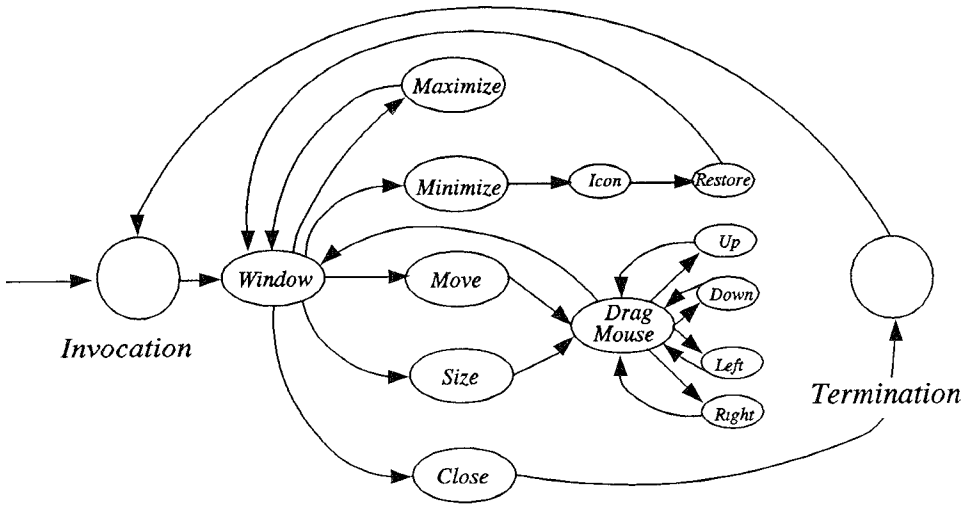

Fig 5. Structural phase—Constructing the usage Markov chain

The second approach, called the *informed* approach, can produce many models, and is used when some actual user sequences are available. These sequences could be captured inputs from a prototype or prior version of the software. Each sequence represents a path through the usage chain that takes the chain from the uninvoked state to the terminate state. Thus a set of structurally complete sequences establishes frequency counts for each arc traversed in the sequence set. The resulting relative frequencies can be used

to estimate the transition probabilities in the usage chain. The informed approach is driven by field data.

The third approach, called the *intended* approach, is similar to the informed approach in that it can lead to many models, but the sequences are obtained by hypothesizing runs of the software by a careful and reasonable user. Relative frequency estimates of the transition probabilities are computed from the symbol transition counts as in the informed approach. An example of the counting technique appears in Table II.

We offer three approaches in order to cover all situations. The informed approach with known sequences for one or more classes of users is best. Next best is the intended approach where general knowledge not supported by field data can be used. As a last resort we use the uninformed approach. The uninformed approach can produce anomalous results in the light of knowledge or intuition. For example, universally accessible "help" leads to an artificially high probability of occurrence. Either of the other two approaches will mitigate such problems.

The structure of the usage chain induces a probability distribution (the usage distribution) on the input domain of the software. That is, each state i has steady-state probability π_i, which can be computed analytically and is a direct consequence of the structure and statistics of the usage Markov chain. Even the uninformed method of assigning transition probabilities induces, in general, a nonuniform usage distribution on the input domain. The usage distribution is given, along with several other analytical results, in the next section.

3. TEST CASE GENERATION AND ANALYTICAL RESULTS

The usage Markov chain is the source of test sequences for the software. A *statistical test case* is any connected state sequence of the usage chain that begins with the invocation state and ends with the termination state. The process of generating statistical test cases is easily automated using a random number generator and any high-level language. One has simply to step through the states of the chain based upon the transition probabilities. The sequence of states visited becomes the test case. Any number of test cases can be obtained automatically from the model.

A major advantage of using the Markov chain as the stochastic model of software usage is that analytical descriptions of the set of test cases can be acquired before any testing begins. One such description is the *usage distribution*, denoted π, of the usage chain. This vector is computed by solving the system of equations

$$\pi = \pi P \tag{1}$$

where P is the *transition matrix* of the usage chain. (The state transition diagram of a Markov chain can be encoded as a 2-D matrix with the state labels as indices and the arc probabilities as entries. Note that the transition matrix is square and each of its rows sums to one.) The strong law of large numbers allows us to interpret each entry π_i as the expected appearance rate

Table II. Statistical Phase—Assigning the Transition Probabilities

From-State	To-State	Frequency	Probability
Invocation	*Window*	6	1
Window	*Maximize*	1	1/12
Window	*Minimize*	1	1/12
Window	*Move*	2	1/6
Window	*Size*	2	1/6
Window	*Close*	6	1/2
Maximize	*Window*	1	1
Minimize	*Icon*	1	1
Icon	*Restore*	1	1
Restore	*Window*	1	1
Move	*Drag Mouse*	2	1
Size	*Drag Mouse*	2	1
Drag Mouse	*Window*	4	4/15
Drag Mouse	*Up*	1	1/15
Drag Mouse	*Down*	5	1/3
Drag Mouse	*Left*	3	1/5
Drag Mouse	*Right*	2	2/15
Up	*Drag Mouse*	1	1
Down	*Drag Mouse*	5	1
Left	*Drag Mouse*	3	1
Right	*Drag Mouse*	2	1
Close	*Termination*	6	1
Termination	*Invocation*	-	1

Captured or hypothesized sequences:

1. *<Invocation><Window><Maximize><Window><Close><Termination>*
2. *<Invocation><Window><Minimize><Icon><Restore><Window><Close><Termination>*
3. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
4. *<Invocation><Window><Size><Drag Mouse><Left><Drag-Mouse><Up><Drag Mouse><Left><Drag Mouse><Window><Close><Termination>*
5. *<Invocation><Window><Move><Drag Mouse><Down><Drag-Mouse><Left><Drag Mouse><Down><Drag Mouse><Window><Close><Termination>*
6. *<Invocation><Window><Size><Drag Mouse><Down><Drag-Mouse><Right><Drag Mouse><Window><Close><Termination>*

of state i in the long run. In terms of the test cases, this is the expected appearance rate of state i asymptotically. Since each state is ultimately associated with some part of the actual software, this information allows testers to determine which parts of the software will get the most attention from the test cases. Furthermore, the entries in the usage distribution for states related in some manner (for example, states in the same window) can be summed to obtain a long-run value for that group of states. This enables the comparison of the usage of subsections of the software that involve multiple states.

Properties of other random variables of interest to software testers may be derived from the usage distribution. An interesting statistic for testers is the number of states necessary until one can expect to generate state i. This value, denoted x_i, is computed by

$$x_i \pi_i = 1.$$

Solving for x yields

$$x_i = \frac{1}{\pi_i}. \tag{2}$$

When the value of x_i is computed for i equal to the termination state, the result is the expected number of states until termination of the software. For software testers, this translates to the *expected test case length* for the usage model.

The expected number of *sequences* necessary until state i occurs can be derived as

$$s_i = \frac{x_i}{x_{TERM}} = \frac{\pi_{TERM}}{\pi_i}. \tag{3}$$

The largest entry in the vector s identifies the amount of expected testing until all usage states are encountered at least once. The values for π, x, and s, computed using the informed probabilities, appear in Table III for the example.

The mean first passage times are useful in many applications. These values are computed by

$$m_{jk} = 1 + \sum_{i \neq k} p_{ji} m_{ik}. \tag{4}$$

Each m_{jk} is interpreted as the expected number of usage states encountered from state j until the first occurrence of state k. This indicates the extent to which states j and k are encountered within the same sequence. For example, if m_{jk} is greater than the expected test case length, then the implication is that the occurrence of state j followed by state k is expected to require multiple sequences. The mean first passage values appear in Figure 6 for the previous example. Note that the vector x is the diagonal of this matrix, the mean first passage from a state back to itself.

The last result discussed in this paper is the source entropy of the usage chain. Intuitively, the source entropy quantifies the uncertainty present in a stochastic source. This value is computed by [1]

$$H = -\sum_i \pi_i \sum_j p_{ij} \log p_{ij}. \tag{5}$$

where π is the usage distribution and the p_{ij} values are the transition probabilities. This single number is related (exponentially) to the number of sequences that are "statistically typical" of the Markov chain. That is, a

Table III. Analytical Results for the Example Usage Model

State	π	x	s
Invocation	0.093750	10.7	1
Window	0.187500	5.3	0.5
Maximize	0.015625	64	6
Minimize	0.015625	64	6
Icon	0.015625	64	6
Restore	0.015625	64	6
Move	0.031250	32	3
Size	0.031250	32	3
Drag Mouse	0.234375	4.3	0.4
Up	0.015635	64	6
Down	0.078125	12.8	1.2
Left	0.046875	21.3	2
Right	0.031250	32	3
Close	0.093759	10.7	1
Termination	0.093750	10.7	1

	Invocation	*Window*	*Maximize*	*Minimize*	*Icon*	*Restore*	*Move*	*Size*	*Drag Mouse*	*Up*	*Down*	*Left*	*Right*	*Close*	*Term.*
Invocation	11	1	64	62	63	64	26	26	11	74	22	31	42	9	10
Window	10	5	63	61	62	63	25	25	10	73	21	30	41	8	9
Maximize	11	1	64	62	63	64	26	26	11	74	22	31	42	9	10
Minimize	13	3	66	64	1	2	28	28	13	76	24	33	44	11	12
Icon	12	2	65	63	64	1	27	27	12	75	23	32	43	10	11
Restore	11	1	64	62	63	64	26	26	11	74	22	31	42	9	10
Move	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Size	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Drag Mouse	16	7	70	68	69	70	31	31	4	63	12	20	31	14	15
Up	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Down	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Left	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Right	17	8	71	69	70	71	32	32	1	64	13	21	32	15	16
Close	2	3	66	64	65	66	28	28	13	76	24	33	44	11	1
Termination	1	2	65	63	64	65	27	27	12	75	23	32	43	10	11

Fig. 6. The mean first passage matrix for the example usage model (entries are rounded).

Markov chain has a set of *typical sequences* whose ensemble statistics closely match the statistics of the chain. Thus, higher source entropy implies an exponentially greater number of typical sequences, i.e., more sequences exist because of the uncertainty present in the model.

The implication of a high source entropy in applications where a Markov chain is used as a sequence generator is that more sequences must be generated in order to accurately describe the Markov source. One can see from Eq. 5 that low entropy can be attributed to a biased usage distribution or from uneven distributions over the exit arcs of the states of the chain.

The source entropy serves as a comparative measure for usage chains with the same structure but different transition probabilities. For example, suppose one develops usage chains U_1 and U_2 using the same structural analysis and assigns U_1's transition probabilities using the uninformed statistical analysis and U_2's using the informed statistical analysis. Let H_1 and H_2 be the source entropies for U_1 and U_2, respectively. As explained earlier, when $H_1 > H_2$ one should expect to generate an exponentially greater number of sequences using U_1 to obtain asymptotic behavior than using U_2. U_1 often serves as a good basis for determining how much the informed approach has biased the usage chain. The values for the example are $H_1 = 1.0884$ bits for the uninformed chain and $H_2 = .8711$ bits for the informed chain. Thus the U_2 source will require fewer sequences to reach asymptotics than U_1.

The analytical results obtained from the usage chain serve to give testers advance knowledge of the time and resources it will take to conduct prescribed testing for the software in question.

4. INITIAL FIELD EXPERIENCE

In this section we give an overview of the experience we have had in building and analyzing Markov usage models. Each project presents new challenges, each of which must be met within theoretical constraints. These solutions might be characterized fairly as arcane technical details. While we do not catalog tips to the practitioner here, we touch on the practical lessons learned for each project mentioned below.

The method of usage modeling and statistical testing described in this paper has been used with success in a demonstration project [9]. The demonstration was to show how Cleanroom ideas could be applied in an environment [10] of mixing reuse of existing code units with newly developed units to achieve planned reliability levels. A system was designed to interact with the user through five screens in a spreadsheet style. One screen managed project files, another managed data entry and editing, and all others provided for parameter set-up, initiation, and display of results for curve-fitting, matrix calculations, and similar mathematical chores. The certification aspect of the demonstration project was based on Markov usage modeling as described here. This project involved approximately 24,000 lines of Ada, new and reused, and was developed and certified in four increments. A Markov chain that modeled the entire system was developed and subsets were extracted for

the certification of each increment (the increments were cumulative so that the parts of the model used in testing grew at each subsequent increment). The entire Markov chain had 90 states and 237 arcs (3% nonzero cells in the transition matrix). A "blunder state" was created from each state to model illegal usage. Whenever this state appeared in a test case, a random keystroke or series of keystrokes was applied to the software. Similarly, the model also contained states that represent data values. For each data value, two states were installed in the chain for legal and illegal configurations of the data, respectively. The appearance of a data state that modeled, for example, a string of characters, caused the generation of an appropriate set of data. In the case of a filename, this data would consist of a string of 1–8 randomly selected alphanumeric characters. In this case study the data values were generated beforehand and then used when directed by the Markov chain. In retrospect, it would have been just as easy to incorporate the additional states into the Markov chain. Thus, we were able to randomize both the control flow through the software and the data that was entered.

We are using Markov chains to generate test cases and certify a comprehensive Cleanroom CASE tool [4] that is under development at the University of Tennessee with IBM sponsorship. These tools are being developed for the OS/2 environment in C. Tools for building, editing, and analyzing Markov models are included. Since the inception of the project in 1989, eight increments have been completed in more than 20,000 lines of C, and work continues. Two separate usage models have been built by two different persons. Just as two programmers can write two very different programs to meet one and the same specification, so too can different Markov usage models be constructed that model one specification. The first model, evolved over seven increments, had reached some 400 states, and as a result of the pressures of complexity and computation time, new insights led to an improved model with fewer than 350 states and a better way to manage the subtleties involved with the multitasking features of the software.

A Markov usage model has been constructed for IBM's DB2 software product; however, the model has not yet been used in statistical testing. The model has approximately 2,000 states. In the course of building this model we created a special notation for representing large models, complete with macros. This, in turn, has presented the concept of translating models expressed in this special notation into the usual arc-probability representation in order to facilitate working with large models.

We know of about 20 Cleanroom projects that are underway in several companies and government agencies. Markov usage models are being used to support certification in three or more of them. The environments range from real-time, embedded code on a bare machine to sophisticated software engineering environments. Code-size estimations range from 75,000 lines of C to 40,000 lines of Ada. These are small systems, but well beyond the scale of academic exercises.

We do not yet have field experience with every class of software. However, we have become relaxed about several issues. First, building usage models is

not essentially more difficult than writing specifications or designing code. As an activity that improves specifications, gives an analytical description of the specification, and quantifies the testing costs, it is well worth the effort. Constructing models is a creative process, but good engineering practices are emerging to constrain the process.

Second, the size of the model is a function of the usage states and arcs implicit in the specification and not of the input space. Just as a small program can have an enormous input space or an enormous internal program state space, a small usage model can be the source for a very complex stochastic process.

Third, we are not concerned about computation time for model analysis. Matrices for large models are inherently very sparse and have the property that each row sums to one. These insure that we can use certain iterative sparse matrix techniques which will keep computation time modest even for models with thousands of states (should we encounter such large models).

5. CONCLUSIONS AND FUTURE WORK

We are currently investigating Markov chains to model the execution of the test cases. We hypothesize that it is possible to develop a Markov chain that will "evolve" as the testing process unfolds. The chain must model software failures as well as software usage. Such a model will lead to analytical stopping criteria and a data-driven, discrete software reliability model. We are also working toward a reliability prediction model.

Engineering process is being developed as well as theoretical understanding. This includes the process of model construction, techniques for handling special situations, representation of complex models in simplified form, transformation from one model to an equivalent model, expansion of states to submodels, compression of submodels to states, tools for writing and editing models, computation tools for analysis, and comparison and evaluation of models.

Usage modeling focuses attention and resources on understanding the customer and the product: What will the user likely do with the software? What is the software to be capable of doing? Once an acceptable usage model is in hand, there are no further assumptions. Everything flows analytically and probabilistically from the model. Changes to specifications can be accommodated by structural changes to the model. Furthermore, as new information is learned about users or classes of users, it can be reflected in the parameters of the model. Statistical testing for software certification is supported directly.

ACKNOWLEDGMENTS

The authors acknowledge the helpful conversations and interactions with several persons, including H. D. Mills, Software Engineering Technology, Inc., J. Hudepohl, Northern Telecom, Inc., R. Drake, R. C. Linger and M. Pleszkoch, IBM, and M. G. Thomason, University of Tennessee. We also appreciate the assistance of E. Ploedereder and the anonymous referees.

REFERENCES

1. ASH, R. *Information Theory*, Wiley, New York, 1966.
2. CURRIT, P. A., DYER, M., AND MILLS, H. D. Certifying the reliability of software. *IEEE Trans. Softw. Eng., SE-12*, 1, 3–11 (Jan. 1986), 3–11.
3. FELLER, W. *An Introduction to Probability Theory and its Applications* Vol 1, Wiley, New York, 1950.
4. FUHRER, D., MAO, H , AND POORE, J. H. OS/2 Cleanroom environment: A progress report on a Cleanroom tools development project. In *Proceedings of the 25th Hawaii International Conference on Systems Science*, IEEE Computer Society Press, Vol. 2, 1992, pp. 449–458.
5. LINGER, R. C., AND MILLS, H. D. A case study in Cleanroom software engineering: The IBM Cobol restructuring facility. In *Proceedings of COMPSAC '88*, IEEE, 1988
6. MILLS, H. D., DYER, M., AND LINGER, R. C. Cleanroom software engineering. *IEEE Software* (Sept. 1987), 19–24.
7. MILLS, H. D., LINGER, R. C., AND HEVNER, A. R. Box structured information systems. *IBM Syst. J. 26*, (1987).
8. MILLS, H. D., AND POORE, J. H. Bringing software under statistical quality control. *Quality Progress*, (Nov. 1988), 52–55.
9. POORE, J. H., MILLS, H. D., HOPKINS, S. L., AND WHITTAKER, J. A. Cleanroom reliability manager: A case study using Cleanroom with box structures ADL. Software Engineering Technology, Inc., IBM STARS CDRL 1940, May 1990.
10. POORE, J. H., MUTCHLER, D., AND MILLS, H. D. STARS-Cleanroom reliability: Cleanroom ideas in the STARS environment. Software Engineering Technology, Inc., IBM STARS CDRL 1710, Sept. 1989.
11. SEXTON, B. C. Statistical testing of software, Master's thesis, Dept. of Computer Science, Univ. of Tennessee, 1988.
12. THOMASON, M. G. Generating functions for stochastic context-free grammars. *Int. J. Pattern Recogn. Artif. Intell. 4*, 4 (April 1990), 553–572.