Annals of Software Engineering Vol. 4 (1997) 115-131

Stochastic Software Testing

James A. Whittaker
Florida Institute of Technology
150 West University Boulevard
Melbourne, Florida 32901
(321)674-7638
(321)674-7046 fax
jw@cs.fit.edu

Abstract.This paper presents a method for test case selection that allows a formal approach to testing software. The two main ideas are (1) that testers create stochastic models of software behavior instead of crafting individual test cases and (2) that specific test cases are generated from the stochastic models and applied to the software under test. This paper describes a method for creating a stochastic model in the context of a solved example. We concentrate on Markov models and show how non-Markovian behavior can be embedded in such models without violating the Markov property.

Annals of Software Engineering Vol. 4 (1997) 115-131

1. INTRODUCTION

It is impossible to test most modern software systems completely. That is, the application of every input combination or scenario (generally called *exhaustive testing*) is infeasible for all but the most simple software systems. The reasons for this are outlined in Table 1, but more importantly, the implication for testing is that only a small subset of the possible input space can be applied before release. In other words, no matter what test selection criteria is used, testers are *sampling* from the input population of the software under test.

Table 1. Some Reasons Why Exhaustive Testing is Impractical

Reason	Comments
Software allows variable input	Variable input, i.e., integers, real numbers, character strings and so forth, creates such a large range of possible values that could be tested that it is impractical to supply all those values during testing. Since a failure could occur on any specific value, complete testing is impractical to achieve.
Software inputs can be sequenced	The order that inputs can be applied can, in general, be rearranged into an infinite number of combinations. Thus, the number of input sequences that could occur are, for all practical purposes, infinite.
Software can accept multiple and interacting streams of input	It is not uncommon for software to handle simultaneous input from multiple sources. Potentially, developers must consider the entire cross product of multiple input domains when creating software. Testers are faced with the formidable task of choosing which subset of the cross product to apply during testing.

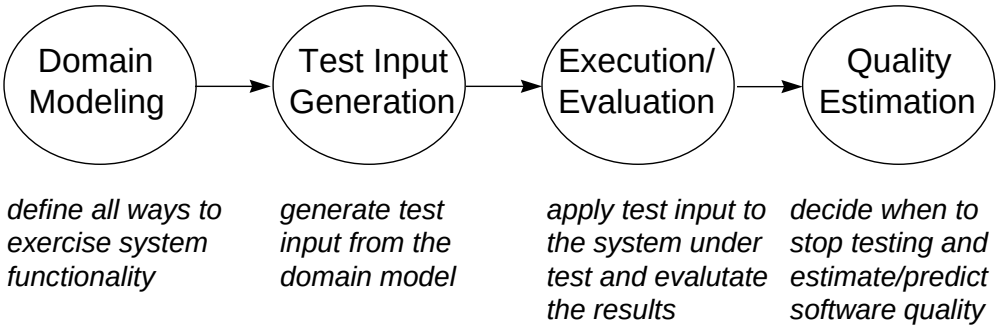
Sampling is not a new idea for testing. Duran and Wiorkowski [1984] show how sampling can be used to quantify the validity of software and Poore *et al.* [1993] discuss sampling as a way to plan and certify system reliability. Indeed, there are many examples in the literature showing how to exploit sampling results but there is little published on how to generate the sample itself. However, obtaining the sample in the first place is arguably the most difficult part and perhaps the most important as well since the measures from the sample are only valid when the sample is statistically unbiased and representative of the population from which it is drawn.

This paper presents a method for sampling that requires modeling the input population for the software under test as a stochastic process and generating sample test cases from the model. The model considered is the finite-state, discrete-parameter Markov chain. We present a method to define an input population by identifying a set of Markov chains that, as an ensemble, encompass the behavior of the system under test. Next, a process to construct the Markov chains is presented in the context of a solved example. Finally, we show how Markov chains are used to generate the sample test cases.

2. TESTING AS SAMPLING

The fact that testing is a sampling problem creates three separate sub-problems to be considered when testing. Each problem and the solution artifacts created are summarized in Figure 1.

Fig. 1. The Sampling Problem for Software Testing



The first step is to model the input domain as a stochastic process. Generally, this involves analysis to decompose the input domain into subpopulations. Decomposition can be based on input source (i.e., where inputs originate) and/or functionality (i.e., what the system under test does). A stochastic model is then constructed for each subpopulation. Next, testers develop a procedure to generate sample input (test cases) from each model. The sample input is then executed against the system under test and the resulting behavior is compared to the specification in order to determine whether the software is functioning correctly. Finally, testers measure the progress of testing during the execution phase by considering the stochastic properties of the generated samples.

Only the first problem, modeling the input population, is considered in this paper; we present only a general discussion of sampling, execution and measurement and intend to publish more in future papers.

Many existing testing methodologies focus on the sampling problem by using heuristics to determine which test cases to construct. An example heuristic is to choose a sample so that it executes all the code statements in the target software. In such methodologies, no real attempt is made to either define the population or to measure whether usage of the software that falls outside the sample will succeed. Some notable exceptions are *partition testing* [Hamlet and Taylor 1990; Ostrand and Balcer 1988] in which inputs are partitioned into equivalence classes and *statistical testing* [Avritzer and Weyuker 1995; Duran and Ntafos 1984; Thevenod-Fosse and Waeselynck 1993; Whittaker and Thomason 1994] in which inputs are randomly sampled based on a probability distribution representing expected field use.

This paper reports on a method we call *stochastic testing* that takes a different approach to sampling than partition testing and statistical testing. First, we define the input population by building a stochastic model (or set of models) that defines the structure of how the target system is stimulated by its environment. Multiple models may be constructed by decomposing the population into subpopulations. In order to allow for statistical inference, we use probabilities in the models; although it is possible to ignore the probabilities if statistical inference is not desired. Next, we define a sampling algorithm that produces test cases directly from the models. Finally, when samples are generated randomly it is possible to use standard reliability analysis to infer population characteristics based on the randomly generated sample.

We attempt to define and formalize current practice of stochastic testing by organizing techniques used in a number of industry projects. This paper is written from the viewpoint of a solved example and necessarily suffers from the simplicity of the example. Variations of these techniques have been used on much larger projects with encouraging results [Dahle 1995; Houghtaling 1996; Rautakorpi 1995].

3. DEFINING THE INPUT POPULATION

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

> **Definition 1**. The *operational environment* is the set of all systems, components and people that interact with the system under test or affect the system under test in any manner.

Informally the operational environment is the "environment in which the software operates." The process of understanding the operational environment and dividing it into subpopulations is called *domain decomposition*. This is the first activity testers pursue when treating testing as sampling.

As an example, consider a typical word processor. We identify three sources of input: human users who submit keystrokes through the user interface, API users (i.e., other applications) which bypass the user interface to invoke specific subfunctions, and the operating system which supplies data files, resource allocation and error messages. In addition, the printer destination will also affect how the word processor behaves. Thus, the following subpopulations make up the operational environment.

- Human users: file manipulation, basic features, advanced features, etc.
- API users: custom applications.
- Operating system: initialization/customization files, native binary files, import files, text files, unsupported file formats, memory/storage resources, system messages, etc.
- Printer: local printer, network printer, postscript file, etc.

The operational environment indicates the factors that we need to consider when testing. In this example, we need to execute the system under varying human and API input, with data files of various formats and with printers of varying capabilities in order to explore how the system behaves on the entire input population.

Once the operational environment is fully defined, the next step is to consider possible interaction between input sources. For example, we may want to test what happens when a file is deleted from the operating system while it is being manipulated by a user. We generally use an additional subpopulation category called *combination usage* to handle these situations.

- Combination usage: files deleted while open, extremely large files, disabled printers/drivers, etc.

The operational environment defines the possible sources of input scenarios. Testers can then turn to defining the practical considerations to effecting such scenarios.

> **Definition 2**. The *test environment* is the set of equipment, tools and oracles necessary to execute, evaluate and measure the test inputs.

Understanding the test environment is less methodical but no less important than understanding the operational environment. It is also highly domain specific and requires detailed trade studies of available tools that might be used in testing. This topic is pursued no further in this paper.

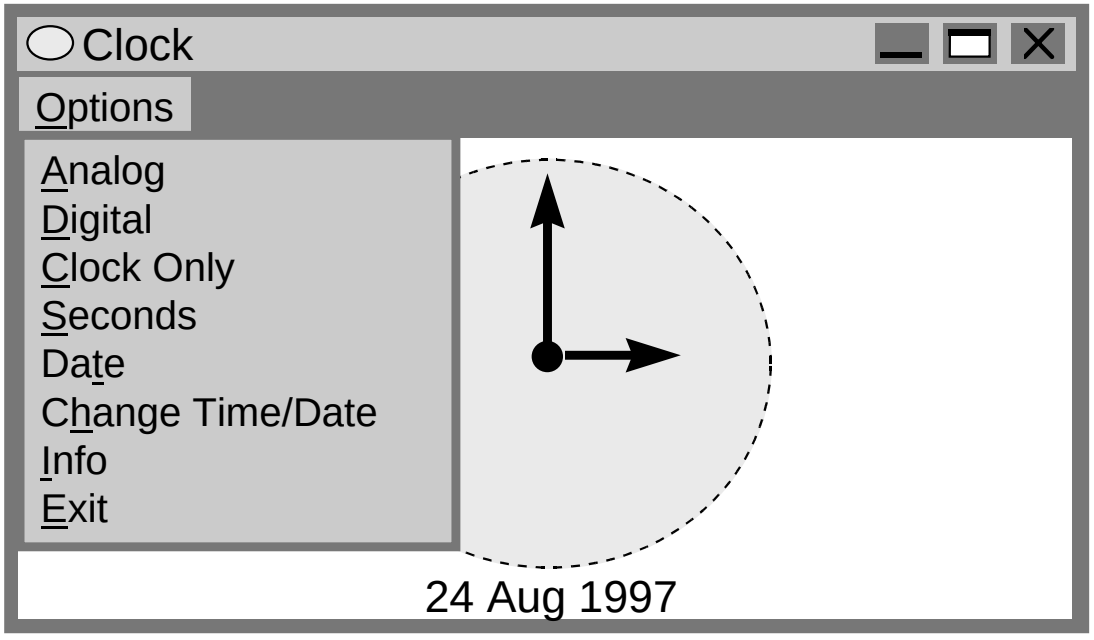
4. MODELING SUBPOPULATIONS

Once domain decomposition is complete, a stochastic model is created for each subpopulation that has been determined to be necessary for testing. The model described here is the finite-state, discrete parameter Markov chain. Markov chains have three components: states, arcs and transition probabilities. The states must obey the *Markov property*, i.e., the information necessary to determine the next state in the process must be wholly contained in the current state. This is traditionally stated as: *the future is independent of the past given the present.* Markov chains can be represented by either a matrix with the states as indices and the transition probabilities as entries, a directed graph with the states as nodes and the arcs (labeled with the transition probabilities) as edges or, finally, a table of from-state/to-state/probability entries. We will depict Markov chains as directed graphs in this paper.

4.1 An Example

In order to illustrate the decomposition process and model construction, we will describe a testing scenario for a hypothetical clock program for a graphical user interface. Most graphical interfaces for operating systems allow a clock to be displayed which indicates current system time and date in either an analog or digital format. We will model the clock software shown in Figure 2.

Fig. 2. Example Clock Software



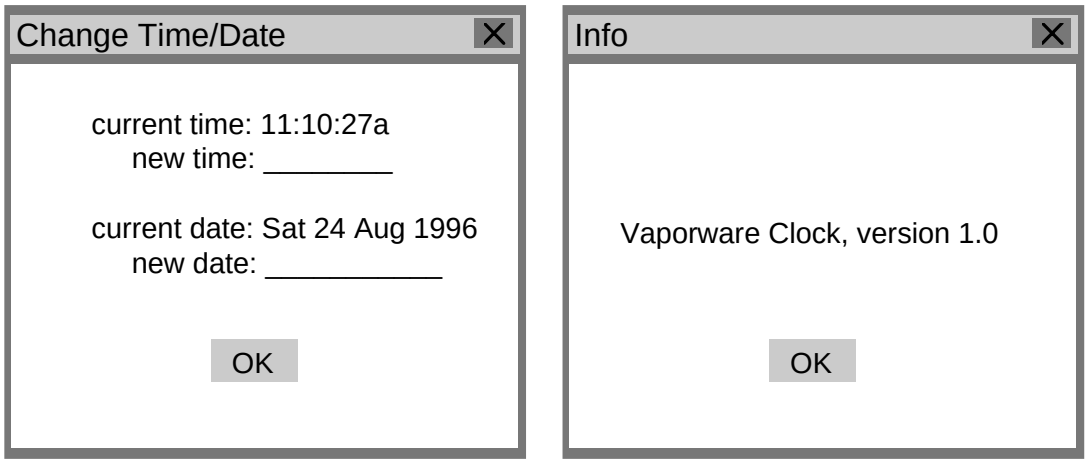
The items under the *Options* menu perform the following functions:

- *Analog*: changes the display to analog format (as pictured in Figure 2). If the display is already analog then this option is not selectable.

- *Digital*: changes the display to digital format from analog. If the display is already digital then this option is not selectable.
- *Clock Only*: makes the title bar, menu and border disappear (and thereby become inaccessible) so that only the face of the clock is visible. To return the face to the original window appearance, users can double click anywhere on the face of the clock.
- *Seconds*: is always available and allows the user to toggle between a visible or invisible second hand (or second counter in digital mode).
- *Date*: is always available and allows the user to toggle between a visible or invisible date (which appears below the clock face).
- *Exit*: unconditionally exits the clock program, causing it to be removed from the screen. Note that the current settings (e.g., analog/digital format, second hand, and so forth) are saved and that subsequent execution of the clock will reflect the most recent values.

The options *Change Time/Date* and *Info* cause other windows to be displayed that allow the time and date to be changed or display information about the product, respectively. These windows are shown in Figure 3. Illegal input on the *change window* simply causes the current input field to be blanked out and the cursor to be set at the beginning of the editable area.

Fig. 3. Modal Windows for the Example Clock



We begin by decomposing the input domain into subpopulations:

- Human users: keystrokes, mouse clicks
- System clock: time/date input
- Combination usage: time/date changes from the OS while the clock is executing

We create one Markov chain to model the input from the user. The input from the clock can be incorporated in this model too, or a separate model can be constructed. Model construction begins by first defining the *operational modes* of the system and then developing the model from them.

Definition 3. An *operational mode* is a formal characterization (specifically a set) of the status of one or more internal data objects that affect system behavior.

A "mode" is characterized by the changes in system behavior that it causes when it is active. Modes are distinguished from one another by one of the following properties:

Definition 4. Two modes *a* and *b* are distinguished by the *Property of User Choice* if mode *a* should produce different input choices than mode *b*.

Definition 5. Two modes *a* and *b* are distinguished by the *Property of External Behavior* if the system should produce a different output in mode *a* than in mode *b* given the same user input.

In general, we consider only *composite outputs* such as screens, reports or displays and not single output items that do not affect long term behavior of the software under test.

Definition 6. Two modes *a* and *b* are distinguished by the *Property of Input Likelihood* if input *i* has a different probability of being applied in mode *a* than in mode *b*.

This last property does not distinguish modes in a structural sense. In other words, the software is not forcing behavior changes on the users. Instead, these are logical modes based on how the user interacts with the system. Indeed, one user of a system may distinguish such modes and another may not.

Modes are represented as sets. To determine operational modes, one considers each input from the user (i.e., the user represented by the subpopulation in question) and determines which (if any) of the above properties apply for that input. For the clock example the input options.analog distinguishes the mode *Setting* = {analog, digital} by the property of user choice because it is selectable in the digital setting but not in analog. That is, the user is presented with input choices based on whether or not the clock is in analog or digital mode. One possible encoding of the operational modes of the clock is:

- *Window* = {main window, change window, info window}. This mode models which window has the focus and is distinguished by the property of user choice because the user has different input options on each window.
- *Setting* = {analog, digital}. This mode models the format of the clock and is distinguished by both the property of external behavior and the property of user choice.
- *Display* = {all, clock only}. This mode models the disposition of the border, i.e., whether the full window is displayed or just the clock face. It is also distinguished by both the property of external behavior and the property of user choice.
- *Cursor* = {time, date, none}. This mode tracks which input field (on the change window) the cursor presently occupies. It is distinguished by the property of user choice.

Obviously the software can be in more than one mode at the same time. Thus, we define states to represent the status of each mode at any given time during execution of the system under test.

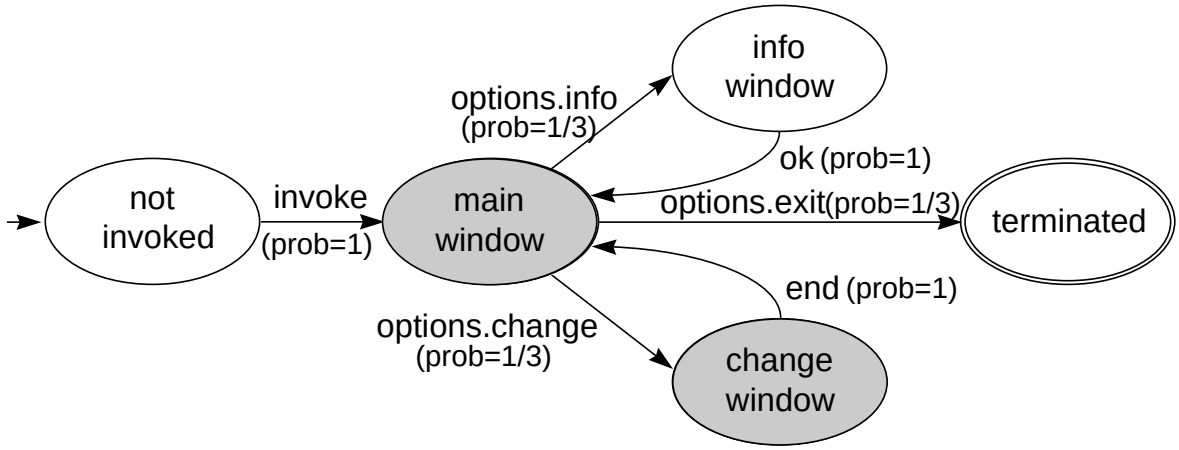
Definition 7. A *state* of the system under test is an element of the set *S*, where *S* is the cross product of the operational modes (removing the impossible combinations).

In the clock example we define the following states:

1. {main window, analog, all, none}
2. {main window, analog, clock-only, none}
3. {main window, digital, all, none}
4. {main window, digital, clock-only, none}
5. {change window, analog, all, time}
6. {change window, analog, all, date}
7. {change window, digital, all, time}
8. {change window, digital, all, date}
9. {info window, analog, all, none}
10. {info window, digital, all, none}

In practice we rarely build (without automation) a model of complex systems with the entire state set because the number of states can grow very large. Instead, we develop the Markov chain in a hierarchical fashion by selecting a primary modeling mode, creating a Markov chain for it (which becomes the top-level in the hierarchy) and then adding the remaining operational modes by expanding states in the top-level model. In this example, we choose the *Window* operational mode as the primary modeling mode and create the Markov chain pictured in Figure 4.

Fig. 4. The Top-level (Level 1) Markov Chain



The arc labels are expanded in the data dictionary entries in Table 2.

Table 2. Data Dictionary Entries for the Top-Level Model

Arc Label	Input to be Applied	Comments/Notes for Tester
invoke	Invoke the clock software	Main window displayed in full Tester should verify window appearance, setting, and that it accepts no illegal input
options.change	Select the “Change Time/Date...” item from the “Options” menu	All window features must be displayed in order to execute this command The change window should appear and be given the focus Tester should verify window appearance

		and modality and ensure that it accepts no illegal input
options.info	Select the “Info...” item from the “Options” menu	The title bar must be on to apply this input The info window should appear and be given the focus Tester should verify window appearance and modality and ensure that it accepts no illegal input
options.exit	Select the “Exit” option from the “Options” menu	The software will terminate, end of test case
end	Choose any action (cancel or change the time/date) and return to the main window	The change window will disappear and the main window will be given the focus
ok	Press the ok button on the info window	The info window will disappear and the main window will be given the focus

Although this model describes the clock at a very high level of abstraction, it encompasses the behavior from invocation of the software to termination and can be used to generate a sample simply by assigning a probability distribution to the set of outgoing arcs at each state. In Figure 4 we have added uniform probabilities to illustrate.

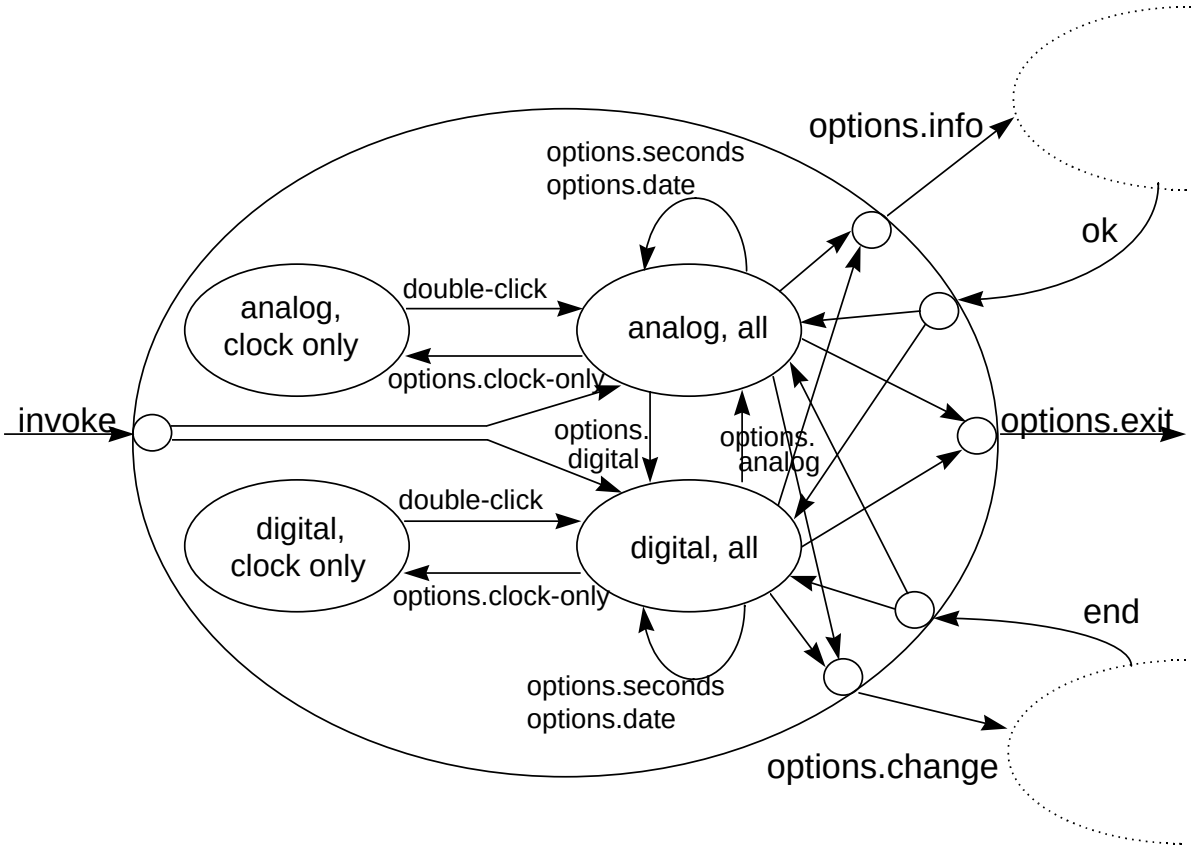
For Markov chains the rules are:

- each arc is assigned a probability between 0 and 1 inclusive,
- the sum of the exit arc probabilities from each state is exactly 1.

Test cases are random walks [Feller, 1950] through paths of the model beginning in the *not invoked* state and ending with the first occurrence of the *terminated* state. Each time a state or arc is traversed then the corresponding label or data dictionary entry is logged in the test case.

In order to add more detail to both the model and the generated sample, the states are expanded by including additional operational modes; in effect creating a submodel within each shaded state in the top-level model. Figure 5 shows the submodel for the main window state (probabilities omitted for clarity). At this level, we include arcs that represent “any other input” by installing “loopback” arcs to and from the state same. For clarity, we only installed one such arc in Figure 5 at the state {*analog, clock-only*}.

Fig. 5. Level 2 Submodel for Main Window



In this submodel, two new operational modes, *Setting* and *Display*, have been added in addition to six new "boundary states" that represent transitions across the boundary to/from the higher level. The data dictionary for this submodel appears in Table 3.

Table 3. Data Dictionary Entries for the Main Window Submodel

Arc Label[1]	**Input to be Applied**	**Comments/Notes**
invoke	Invoke the clock software	Main window displayed in full Invocation may require that the software be calibrated by issuing either an *options.analog* or an *options.digital* input Tester should verify window appearance, setting, and ensure that it accepts no illegal input
options.change	Select the "Change Time/Date..." item from the "Options" menu	All window features must be displayed in order to execute this command The change window should appear and be given the focus

1. Labels that appear in italics represent inputs from the higher-level model.

		Tester should verify window appearance and modality and ensure that it accepts no illegal input
options.info	Select the “Info...” item from the “Options” menu	The title bar must be on to apply this input The info window should appear and be given the focus Tester should verify window appearance and modality and ensure that it accepts no illegal input
options.exit	Select the “Exit” option from the “Options” menu	The software will terminate, end of test case
end	Choose any action (cancel or change the time/date) and return to the main window	The change window will disappear and the main window will be given the focus Note: this action may require that the software be calibrated by issuing either an *options.analog* or an *options.digital* input
ok	Press the ok button on the info window	The info window will disappear and the main window will be given the focus Note: this action may require that the software be calibrated by issuing either an *options.analog* or an *options.digital* input
options.analog	Select the “Analog” item from the “Options” menu	The digital display should be replaced by an analog display
options.digital	Select the “Digital” item from the “Options” menu	The analog display could be replaced by a digital display
options.clock-only	Select the “Clock Only” item from the “Options” menu	The clock window should be replace by a display containing only the face of the clock, without a title, menu or border
options.seconds	Select the “Seconds” item from the “Options” menu	The second hand/counter should be toggled either on or off depending on its current status
options.date	Select the “Date” item from the “Options” menu	The date should be toggled either on or off depending on its current status
double-click	Double click, using the left mouse button, on the face of the clock	The clock face should be replaced by the entire clock window

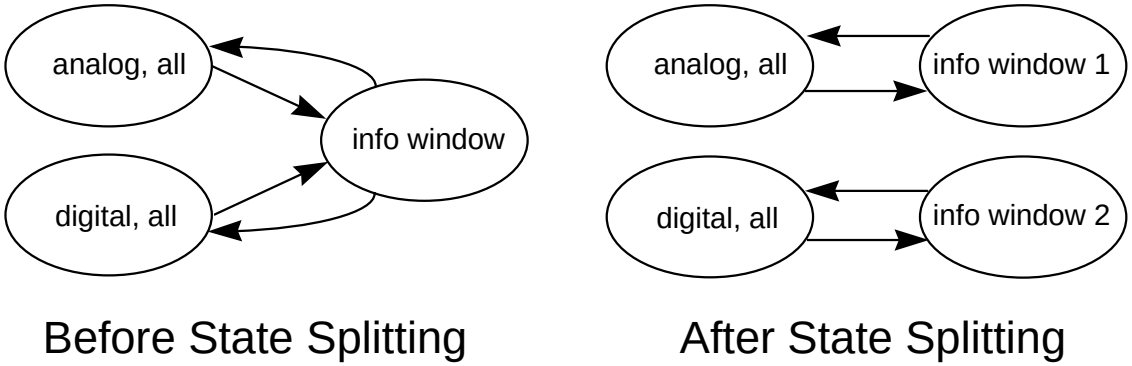
In several places in this model, history is being stored in order to handle events that depend on prior occurrences of certain inputs. Some information is being stored in each state, e.g., we know whether or not the clock is in digital or analog mode by checking the value of the operational mode *Setting*. However, other history information is not tracked on a state-by-state basis. In particular, whenever we traverse an arc from the lower level to the top level, we have lost the information necessary to determine which top level state the software is in. For example, when we traverse the arc *options.info* from the state {*analog, all*} then upon return to the main window from the info window we will not know whether to go back to {*analog, all*} or {*digital, all*} because we will have lost the history. There are three possible solutions to this problem which are outlined below.

First, we could simply allow the model to make random traversals even when pertinent history is not available. In this case, it is possible that the software and the model will become out of sync during actual testing.

Whenever this occurs in practice, we reset the software so that it reflects the current model state. Using the above scenario, if a departure from the *info window* causes the model to be in state {*digital, all*} and the software to be in state {*analog, all*} then the tester would issue the input *options.digital* to return the software to the state specified by the model. These actions are documented in the data dictionary in Table 3. It is important to note that since it is the software that is reset and not the model, this calibration does not violate the Markov property.

Second, a technique called *state splitting* [Whittaker 1992] can be used to keep the model and software in the same state. Essentially, state *i* is split into multiple states, one for each dependency that exists at state *i*. As an example, consider the dependency that exists at the state *info window*. When this state is exited, it can either go to {*analog, all*} or {*digital, all*} depending on the previous state. State splitting creates two copies of the *info window* state to separately handle the analog and digital cases. This technique is illustrated in Figure 6. Although this technique creates models with more states, it is the most common solution for maintaining history in practice.

Fig. 6. An Illustration of State Splitting



Third, we can install an external memory structure which is accessed whenever a boundary state is traversed. This structure, usually a stack, stores the last state traversed (i.e., it is pushed on the stack) whenever a level change occurs. Then when that boundary state is once again reached a pop operation determines which arc to traverse. This storage mechanism obviously violates the Markov property. In reality, the combination state machine and stack is a *multitype branching process*. This model is very useful in practice but has less analysis capability than the Markov chain. These models are not discussed further in this paper and are the subject of ongoing research.

The next step is to continue to build submodels until we have completed the hierarchy. The only remaining submodel for this example is for the *change window* state. This model appears in Figure 7 and the data dictionary is shown in Table 4.

Fig. 7. Level 2 Submodel for Change Window

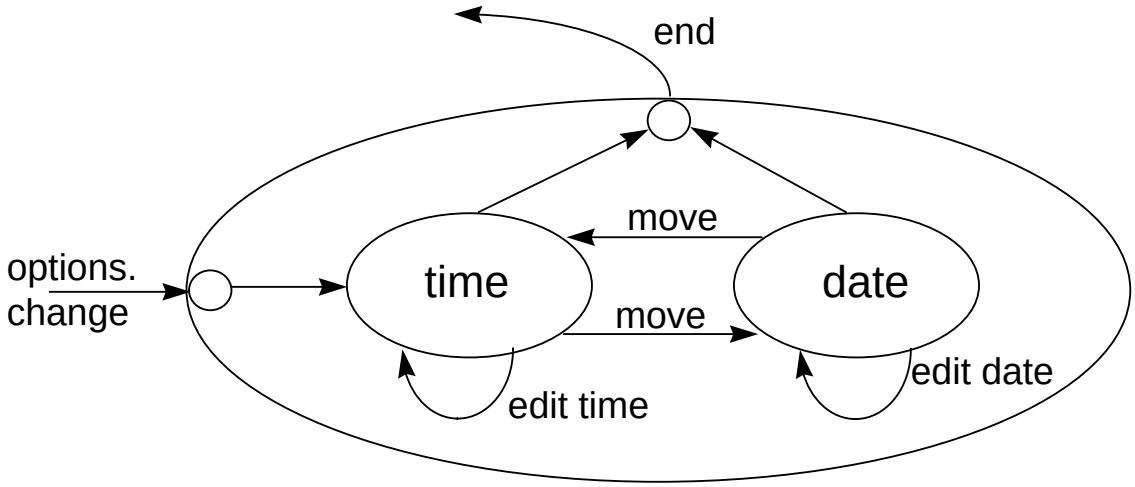


Table 4. Data Dictionary Entries for Change Window Submodel

Arc Label	Input to be Applied	Comments/Notes for Tester
options.change	Select the "Change Time/Date..." item from the "Options" menu	All window features must be displayed in order to execute this command The change window should appear and be given the focus Tester should verify window appearance and modality and ensure that it accepts no illegal input
end	Choose either the "Ok" button or hit the cancel icon and return to the main window	The change window will disappear and the main window will be given the focus Note: this action may require that the software be calibrated by issuing either an *options.analog* or an *options.digital* input
move	Hit the tab key to move the cursor to the other input field or use the mouse to select the other field	Tester should verify cursor movement and also verify both options for moving the cursor
edit time	Change the time in the "new time" field or enter an invalid time	The valid input format is shown on the screen
edit date	Change the date in the "new date" field or enter an invalid date	The valid input format is shown on the screen

At this point we need to decide how to handle combination usage (time and date changes from outside the system while the clock is being executed). We could build a separate model in the manner described above or we could add the appropriate inputs to the existing model. In the former case, we would perform the same process on the operating system inputs that we used to model the keyboard input, i.e., define the operational modes and construct the state diagram. In the latter case, we could simply install arcs in our current model that dictated to the tester to go outside the clock and change the time and date. In this case, these arcs would be "loopback" arcs (arcs

from/to the same state) installed in each state of the existing model; since changing the time and date does not cause a state change in this model.

4.2 Managing Model Size

For extremely large or complex systems, the operational modes may define a set of states that is too large to be practically constructed, even using hierarchies of models. There are several techniques to deal with this complexity. Each technique has the effect of removing states from consideration during testing. Thus, it is important to reach consensus among project stakeholders that the techniques do not significantly reduce the effectiveness of testing. The techniques for reducing model size are outlined below.

- *Exclusion*

  Limit the values of an operational mode. Let's say that we are modeling the operation of embedded automobile software and that we have identified the operational mode: *Gear Status* which can assume the values: {1, 2, D, N, R, P}. We may decide the operation of the vehicle in neutral and park does not impact the software being tested and, therefore, we would remove the values N and P.

- *Equivalence*

  Define equivalence classes for the values of an operational mode. Using the same automobile example as above, imagine an operational mode *Speed* = {0, 1, 2, 3, 4, ..., 120}; which may be rewritten as four equivalence classes {0, 1-29, 30-55, 55-120}. It is important to define the classes so that each value in a class is essentially the same in terms of user choice, external behavior and input likelihood (as defined earlier in this paper).

- *Abstraction*

  Create abstract inputs by combining physical inputs with the information stored in one or more operational mode. Using the automobile example one last time, suppose that we have the inputs *press accelerator pedal* and *press brake pedal*. We can combine the *Speed* operational mode with these physical inputs and create the abstract inputs: *accelerate to 1-29 mph*, *slow to 0 mph*, *accelerate to 30-55 mph* and so forth. Now when we take the cross product of the remaining operational modes, we can leave out the *Speed* mode because the abstract inputs have the information we need embedded in them.

No matter which method of model simplification is used, it is useful to determine the complete set of physical operational modes, even though some equivalences and abstractions may be immediately obvious. Determining the modes in advance has never proven overly difficult or costly, and it can guide abstraction and partitioning so that testers have a better feel for what is given up with each decision to move away from the completely physical point of view. Trade-offs between test effectiveness and ease of model construction can be more informed and their implications more completely understood.

4.3 Partitioning Variable Input

Whenever a software system is being modeled that accepts variable input like character strings or integer data, the possible set of values must be partitioned. Stochastic models support this partitioning by creating separate arcs for each possible partition created. For example, in the case of an integer input, we might create the partitions:

negative, *zero*, *positive* and *illegal*. Thus the state that models the input field will have four exit arcs labeled with one each of the four partitions, respectively. It follows that boundary values, e.g., *illegal negative*, *smallest negative*, *other negative*, -1, *zero*, 1, *other positive*, *largest positive* and *illegal positive* are treated similarly.

In fact, we can use this technique in our clock example by partitioning the inputs *edit time* and *edit date* (Figure 7). Generally, input partitioning is performed in the data dictionary but one could develop a submodel to achieve the same results.

Table 5 depicts an expansion of the data dictionary in Table 4 which partitions both time and date input.

Table 5. Partition of the Time and Date Inputs

Arc Label	Input to be Applied	Comments/Notes for Tester
edit time	LEGAL:	
	• 11:59:59a	• make sure date doesn't change
	• 11:59:59p	• make sure date changes
	• omit seconds	• should default to ":00"
	• omit a/p	• should default to "a"
	• use 24 hour clock (military time)	• should be converted to 12 hour clock
	• any other legal time	• entry should be accepted
	ILLEGAL:	
	• hour field > 24	• entry should be erased, cursor reset
	• minute field > 60	• entry should be erased, cursor reset
	• second field > 60	• entry should be erased, cursor reset
	• character data in any field	• entry should be erased, cursor reset
	• 24 clock with a/p attached	• entry should be erased, cursor reset
edit date	LEGAL:	
	• 29 Feb, leap year	• entry should be accepted
	• 31 Dec 1997	• confirm year change
	• 31 Dec 1999	• confirm year change
	• omit day of week	• entry should be accepted
	• include day of week	• entry should be accepted
	ILLEGAL:	
	• character data in any field	• entry should be erased, cursor reset
	• invalid month	• entry should be erased, cursor reset
	• day exceeds maximum limit for month	• entry should be erased, cursor reset

4.4 Special Purpose States/Arcs

It is sometimes useful to insert special purpose states and arcs into the model to accomplish a particular task or satisfy a testing goal. The purpose of these states can vary but usually they represent some sort of "outside knowledge" about how the system under tests behaves that cannot be captured in the operational modes.

For example, in some systems, failures may only manifest themselves after long sequences of input without termination. If this is the case, then we may want to add a second "terminated" state that represents the end of a test case, but not termination of the software. Thus, we can account for the desired behavior directly in the model.

5. FUTURE WORK

This paper shows how stochastic modeling can be applied to the software testing problem. Although the choice for a model could have been any number of stochastic processes, Markov chains were used because they have been shown to be successful in practice and because of their potential to provide valuable analytical feedback.

Follow-on papers will discuss how analytical results associated with Markov chains are computed and used to validate the stochastic model against known user behavior. In addition, we will further discuss the generation and analysis of the sample. Sample analysis is concerned with estimating two specific events: when to stop testing and software reliability assessment.

Future research will focus on three areas:

First, we are investigating methods to mechanically enumerate the state space of the model from the operational modes. We envision the development of the operational modes and a set of constraints defining possible states to be the task of a human tester and then an algorithm would generate the full state space. It remains to be seen how general such an algorithm could be and whether we could embed arc information so that the entire Markov chain could be constructed.

Second, work is in progress on the use of other types of stochastic processes as the population model, replacing the Markov chain. This paper identified multitype branching process as the next logical progression of this idea.

Third, the issue of generating test cases randomly from the model is an important one. Currently, we generate test cases based only on the probabilities assigned to each arc. An extension of this idea would be to dynamically change the probabilities as new information surfaces during test. For example, we might decide that a particularly buggy section of code needs additional testing and then raise the probabilities associated with that part of the model. Thus the model would adapt to the demands of testing by learning failure patterns and adjusting probabilities to get better coverage of specific parts of the model.

Acknowledgments

I thank Lina Khatib of Florida Tech for her review and comments on an earlier version of this paper. Also, I thank the editor and anonymous reviewers for their insights that helped improve this paper a great deal.

References

A. Avritzer and E. Weyuker (1995), "Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Transactions on Software Engineering*, 21, 9, 705-716.

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

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

Annals of Software Engineering Vol. 4 (1997) 115-131

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

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

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

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

T. Ostrand and M. Balcer (1988), "The Category-Partition Method for Specifying and Generating Functional Tests," *Communications of the ACM*, 31, 6, 676-686.

J. H. Poore, H. D. Mills and D. M. Mutchler (1993), "Planning and Certifying Software System Reliability, *IEEE Software*, 88-99.

M. Rautakorpi (1995), *Application of Markov Chain Techniques in Certification of Software*, Master's Thesis, Department of Mathematics and Systems Analysis, Helsinki University of Technology, Helsinki, Finland.

P. Thevenod-Fosse and H. Waeselynck (1993), "STATEMATE Applied to Statistical Software Testing," In *Proceedings of the International Symposium on Software Testing and Analysis*, ACM Press, Cambridge, MA, pp.99-109.

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

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