Test Case Generation Using Stochastic Automata Networks: Quantitative Analysis*

Cristiano Bertolini, André G. Farina, Paulo Fernandes and Flávio M. Oliveira
Pontifícia Universidade Católica do Rio Grande do Sul, Brazil
cbertolini@inf.pucrs.br, andre.farina@hp.com, paulof@inf.pucrs.br, flavio@inf.pucrs.br

Abstract

The software engineering community has been using Markov Chains (MC) to describe usage models. We have been working on the use of a more sophisticated discrete state formalism: Stochastic Automata Networks (SAN). SAN is a formalism with the same power of description as MC; however, a system in SAN is described as a collection of subsystems described by local states, transitions and synchronizing events, allowing higher modularity and maintainability. We present a description of SAN *formalism, as well as quantitative analysis of the modeling examples considering the generation time, quality of the test suites.*

1. Introduction

The complexity of software development demands the use of sophisticated formal methods to describe it. This concern with a formal description of software is certainly useful in all phases of a software life-cycle. In this paper, we are particularly interested in formal tools to support the software testing procedures [5]. The technique discussed here may be extended to other phases.

Some years ago, the software engineering community has turned its attention to *usage models* in statistical software testing [8]. It seems natural to consider the description of software as a discrete state model [4]; therefore, Markov Chains (MC) are the simplest way to describe it. MC are one of the oldest formalisms used to describe discrete state models [9]. A MC model describes a system as a set of possible states and transitions among them. Each transition is described by a stochastic process with exponential distribution. MC can be used with a discrete or continuous time scale.

For almost a decade the software engineering community has been using MC to describe usage models [10, 11, 12, 13]. In this paper we suggest the use of a more sophisticated discrete state formalism: Stochastic Automata Networks. SAN is a formalism with the same power of description as the MC; however, a system in SAN is not described as a single system with states and transitions in a single level. A system is described by SAN formalism as a collection of subsystems described by local states and transitions among the states. Each of the subsystem transitions can be affected by the states or transitions of the other subsystems, therefore providing interactions among the subsystems. The composition of all the subsystems is equivalent to a, usually huge, MC model.

Admitting the use of SAN to model applications, the main purpose of this paper is to analyze the differences encountered by generating test cases according to a SAN model in comparison to the same procedure according to a MC model. This seems to be the natural first step to validate the use of SAN as a modeling formalism to the development of usage models for statistical testing. It is not the purpose of this paper to discuss how suitable SAN or MC formalisms are for the development of usage models. Such discussion would require a more profound analysis of usage models characteristics and it will be out of the scope of this study. As a matter of fact, the discussion proposed in this paper is centered in the quantitative analysis of SAN formalism. In contradiction, the modeling adaptation of SAN formalism would be much more based on a qualitative analysis.

In order to present our contribution, this paper

* Our thanks to Lucas Reginato and the CPTS team for the help in development of STAGE. This work was developed in collaboration with HP Brazil R&D - PUCRS (research projects: CPTS - Research Center on Software Test and CASCO - Analysis Center on Concurrent Systems)

presents in the next two sections the basic concepts of statistical testing with usage models (Section 2) and a brief introduction to the Markov chains formalism and its use to describe usage models (Section 3). Section 4 presents SAN formalism in both informal and formal descriptions. The modeling of three small applications using SAN formalism is presented in Section 5. Those examples will be used to present the quantitative analysis of the generation of *test suites*[1] using SAN models or equivalent MC models in Section 6. Finally, the conclusion summarizes the contribution of this paper and it also proposes the future works for this research subject.

2. Statistical Testing

In software testing there are many test case generation techniques trying to find failures (bugs). The main techniques are *functional testing* - that aims to test system behavior based on the system specification -, *structural testing* - that aims to test the system by inspecting the source-code [6] - and *error-based or fault-injection testing* - that aims to identify bugs by generating mutant code. These techniques aim to analyze as many failure situations as possible, by testing the program inputs, system paths and outputs. However, it is not always possible to test it exhaustively due to the multiple possibilities and the high costs.

In this context it is not possible to infer about testing reliability. Statistical software testing makes use of statistical science in order to cope with these problems [8]. The main benefit of statistical testing is to use statistical inference techniques to compute probabilistic aspects of testing [13].

2.1. Usage Models

A usage model characterizes the operational use of a software system. Three aspects can be considered in the construction of a software usage model: *the software use, the user, and the environment of use*. A *software use* can be a working session, a transaction or any service unit limited by a start/finish event that defines a usage instance. The *user* of software can be a person, a hardware device or another software without any loss of generality. The *environment of use* can be defined by the platform (OS, window manager, etc), number of users (single/multiple-user), concurrent applications (distributed, parallel, etc) and other aspects that may affect the software behavior.

The model structure is a set of discrete states and transitions among them. Consequently, it is to use MC formalism mapping. Each user action as a transition that (may) change the current state. The transition probabilities or firing rates (see next Section) represent the expectation on user actions and therefore they configure the usage pattern of the modeled software. Due to the features of usage models, the MC must have at least, two special states representing the beginning and the end of the software use.

The usage model may be applied at many stages of the software life-cycle, in order to refine system specification, evaluate complexity, drive the verification efforts, identify events frequency, estimate the testing effort, estimate software reliability, and so on. We focus here on its application to the testing phase and reliability estimation.

Represented with MC, usage models allow the test engineer, a person who has the responsibility of test creation and management, to predict the critical paths, more susceptible to failure, concentrating the efforts in this context. This analysis is performed over the occurrence probabilities associated to each use of the software.

From the usage model analysis it is possible to extract several interesting properties, such as [11]: number of usage paths, long-run occupancy (e.g. utilization time percentage for each state), mean number of events per test case and mean number of events between two states.

3. Markov Chains

A Markovian process is a stochastic process where the evolution of the process depends exclusively on the current state, not depending on the previously visited states [7].

The evolution of the system is represented by transitions of the process from one state to another. These transitions are assumed to happen in an instantaneous way (without consuming time). When the state space of a Markovian process is discrete, this process is called a Markov Chain (MC). These chains are divided, according to the time scale, in Discrete Time Markov Chain (DTMC) and Continuous Time Markov Chain (CTMC) [9].

In DTMC, we have conditional probabilities of transitions from one state to another. These probabilities of transition of the MC can be represented as p_{ij} (probability of transition from state *i* to state

1 We call a *test suite* a set of test cases.

j) in homogeneous chains - whose transitions do not depend on time -, or as $p_{ij}(n)$ (probability of transition from the *i* state to the *j* state in time *n*) in the non-homogenous chains - whose transitions depend on time. These probabilities are represented by a real number between 0 and 1, and the sum of all the probabilities of transition from one state to each one of the other states must be 1 [7].

In CTMC, we have rates of occurrence associated to each transition. Those rates describe a frequency of firing per time unit. Thus, its value can be any positive Real number (including 0).

The probabilities of transition of the MC are represented through matrices with dimensions $n \times n$, with n being the number of states of this chain. In the discrete time chains, this matrix is called matrix of transition probabilities (p), and in the continuous time chains it is called matrix of transition rates, or infinitesimal generator (Q).

In DTMC, the elements of p are the probabilities p_{ij}. In CTMC, the matrix Q contain all rates to pass from state i to state j ($i \neq j$) in the non-diagonal elements q_{ij} and it completed by the introduction of negative values in the diagonal elements in order to have all rows summing zero [7].

Without any loss of generality discrete-time or continuous-time models can be used to model software usage. If discrete-time scale is adopted the transitions will have probabilities of occurrence in discrete time tics. If continuous-time scale is adopted transitions will have rates describing the frequency in which they occurs [9].

3.1. Limitations of MC

MC is the most used modeling formalism for usage models due to the benefits it takes to the statistical software testing. The chains are intuitive in the way people can recognize the system paths in the diagram, because its structure is very simple. Although all these benefits, MC have some limitations that become critical as much as the chain size grows up.

The stochastic matrix used by the formalism to store the transition probabilities of the models, as the model grows up, causes the *state-space explosion* problem. This problem, sometimes, makes the model computation impossible. The growth of the number of states and transitions between them impacts the diagram readability. Sometimes it is impossible to provide a reasonable visualization of the model because of its size.

Another problem found on using MC for modeling is about maintainability. When adding states to a model, it is hard to find all the transitions that must be included in order to keep the model consistency. The complexity of WEB applications, for example, makes modeling the systems using MC a difficult task. If we have an application running on a browser, all the states that do not block the browser are final state candidates. So, in a MC usage model, all these candidates should be linked to the *Terminate* state by a transition (one for each candidate state). The use of menu frames in the application works the same way. This kind of menu allows all the states of the model that do not block the browser to execute its actions, resulting on multiple transitions to be included for each state. This multiplicity of transitions needed to represent the system complexity can make the model visualization impractical.

4. Stochastic Automata Networks

Stochastic Automata Networks (SAN) formalism was proposed by Plateau [3]. The basic idea of SAN is to represent a whole system by a collection of subsystems with an independent behavior (*local transitions*) and occasional interdependencies (*functional rates* and *synchronizing events*). The framework proposed by Plateau define a modular way to describe continuous and discrete-time Markovian models [3]. SAN formalism has exactly the same application scope as the MC. SAN formalism may cope the state space explosion by a modular way to model and an efficient numerical treatment of the infinitesimal generator matrix. However, SAN keeps the same state space as the corresponding MC.

4.1. Formal Definition

We describe here a formal conceptualization of SAN. The treatment given here is somewhat different from the original presentation, given in the works of Fernandes and Plateau [3].

Given n sets S_i of states (the local states), let the automata be defined by $A = \{A_1, A_2, \ldots A_n\}$ be a set of automata, where each automaton A_i is composed by the states of S_i. A SAN is a structure (G, E, R, P, I), where:

- $G = \{G_1, \ldots, G_m\}$ is a set of global states, such that each G_i is an element of $A_1 \times A_2 \times \cdots \times A_n$. In other words, each global state is a combination of local states of the automata.
- $E = \{E_1, \ldots, E_k\}$ is a set of events. Each event is a function $E_i : G \rightarrow P(G)$. In other

words, each event maps global states into sets of global states. When the event is fired, the SAN can go to any element of the set specified by the function, depending on the probabilities assigned to the event (see below). Since a global state is a list of local states, the function describes, for each automaton A_i in the network, what happens in that automaton when the event is fired. Events can be classified as local and synchronizing. A local event changes the state of only one automaton; a synchronizing event changes the state of two or more automata.

- $R = \{R_1, \ldots, R_k\}$ is a set of event rate functions, one for each event. Each function $R_i : G \rightarrow \mathbb{R}^+$ describes the rate of occurrence of the event at each global state.
- $P = \{P_1, \ldots, P_k\}$ is a (possibly empty) set of transition probability functions, one for each pair (event, global state). As defined above, for an arbitrary event i, when the SAN is in a global state G_i and the event is fired, the SAN goes to a state G_j which must be an element of $E_i(G_i)$. The transition probability functions describe the probabilities of the different elements of $E_i(G_i)$ being selected. Usually, $E_i(G_i)$ has only one state, so the definition of these probabilities is optional.
- $I \subseteq G$ is a (possibly empty) set of initial states. In the original definition, SAN do not have initial states, but a set of reachable states. The definition of initial states is more meaning full to usage models, and its adoption does not change the SAN formalism.

Any SAN can be converted into an equivalent Markov chain [3]. The states of the Markov chain are the global states of the SAN.

SAN have been shown to have a number of advantages for modeling complex systems, in comparison with MC [1, 8]. We believe that this is the case also for statistical testing based on usage models, without loss of generality or information. Usage models described with SAN have some interesting characteristics [2]:

- environment requirements (a critical issue for testing) can be made explicit in the model;
- the representation is modular, improving maintainability and readability;
- an individual use is a sequence of global states in the SAN - thus, its description is more detailed, which allows an easier mapping of uses into test cases;

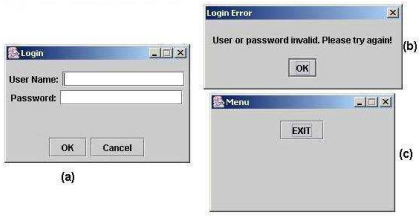


Figure 1. Login System

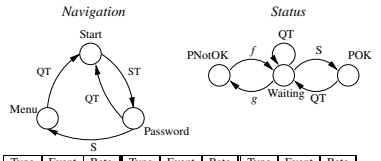
- as we shall see in the Section 4, for complex applications the computational cost of SAN-based test case generation is smaller than Markov-based.

4.2. Informal Definition

For example, let us consider an (quite simple) application consisting of just two dialogues: The first is a login dialog (Figure 1a), where the user is prompted for a username and a password; if the username or the password is incorrect, the application issues an error message (Figure 1b); The second is a menu (Figure 1c), where the user can only terminate the application. This application can be described by the SAN illustrated in Figure 2. The network has the following structure:

$Navigation$ = { Start, Password, Menu }
$Status$ = { Waiting, POK, PNotOk }
E = { ST, QT, S, g, f }
ST = { (Start, Waiting) → (Password, Waiting) }
QT = { (Password, Waiting) → (Start, Waiting), (Menu, Waiting) → (Start, Waiting), (Menu, POK) → (Start, Waiting) }
S = { (Password, Waiting) → (Menu, POK) }
g = { (Password, Waiting) → (Password, PNotOk) }
f = { (Password, PNotOk) → (Password, Waiting) }
I = { (Start, Waiting) }

ST, QT and S are synchronizing events, while g and f are local. Note that, for simplicity, we describe the events as partial functions; if a global state does not have an image defined for some event, it means the event is not enabled at that state (for example, ST is enabled only at global state (Start,Waiting), which is the initial state). In this small example (Figure 2), all probabilities where omitted, since from any given global state the occurrence of an event may lead to a single state.



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
Syn	ST	τ_1	Sync	S	τ_3	Loc	g	τ_5
Sync	QT	τ_2	Loc	f	τ_4			

Figure 2. SAN Model

5. Experiments

The work presented on this paper consists on the analysis of test case samples generated based on Usage Models. The usage models were modeled using the two formalisms described on previous sections: MC and SAN. The use of SAN to construct Usage Models is described in more detail on previous work [2].

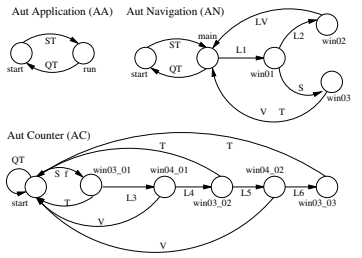
Three applications were chosen as targets for the creation of Usage Models: *Simple Counter Navigation*, *Calendar Manager*, and Form-based Documents Editor (*Docs Editor*). Once the target applications were chosen, a model structure was created for each of them, mapping their system states and transitions. These models are similar to a state chart diagram. Then, the applications were adapted in order to create a *log* about their usage, based on the model structure setup previously. Some users kept using the application creating a sample of the application usage.

Once having the model structure and usage samples, the usage model was created by transforming the transition counter (obtained from the *logs*) on transition rates, updating the model structure with statistical data. The models created for each application using both formalisms (MC and SAN) are used as input for the automated test case generation process described later.

Simple Counter Navigation

The MC model is composed of 9 states and 24 transitions. The SAN model is composed of 3 automata $(2 \times 5 \times 6)$, having 9 global reachable states (of 60 total states). The application represents a Web window system and its main characteristics are the navigability between the windows, and a counter used to navigable for the windows. The scope this application is the navigability.

Figure 3 show SAN model to this application. Such model is composed by 3 automata: Automaton Application (AA): representing the invocation and termination of usage; Automaton Navigation (AN): representing the navigation among the windows; Automaton Counter (AC): representing the counter system used by the application to navigate.



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
Sync	ST	τ_1	Loc	L_1	τ_6	Loc	L_4	τ_9
Sync	QT	τ_2	Loc	L_2	τ_7	Loc	L_5	τ_{10}
Loc	LV	τ_{12}	Loc	L_3	τ_8	Loc	L_6	τ_{11}
Sync	S	$\tau_3 * f$	Sync	V	τ_5	Sync	T	τ_4

$f = [st\ \ AA == run]$

Figure 3. Simple Counter Navigation

Calendar Manager

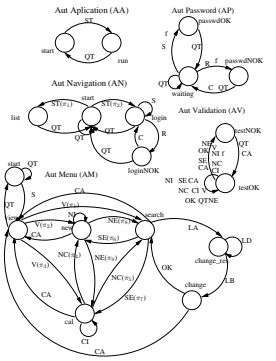
The MC model is composed of 16 states and 67 transitions. The SAN model is composed of 5 automata $(2 \times 3 \times 4 \times 2 \times 7)$, having 16 global reachable states (of 336 total states)). This application is a Web system to calendar manager and it is used to create and to organize the event dates, holidays and new schedules.

Figure 4 show SAN model and the 5 automata defined are: Automaton Application (AA): representing the invocation and termination of usage; Automaton Navigation (AN): representing the navigation among the windows; Automaton Password (AP): representing the user validation as a subsystems that authentic the password and user name; Automaton Validation (AV): representing the forms validation; Automaton Menu (AM): representing all action possibilities of application as insertion, deletion, alteration and searching.

Form-based Documents Editor (Docs Editor)

The MC model is composed of 417 states and 2593 transitions. The SAN model is composed of 6 automata $(2 \times 2 \times 2 \times 3 \times 3 \times 10)$, having 417 global reachable states (of 720 total states).

The model is composed by 6 automata: Automaton Server (AS): representing the server active or not active; Automaton Application (AA): representing the invocation and termination of us-

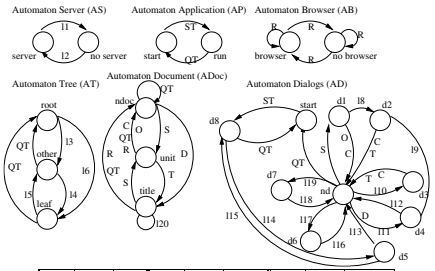


Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
Sync	ST	τ_1	Sync	NC	$\tau_6 * f$	Sync	NI	$\tau_{11} * f$
Sync	QT	τ_2	Sync	NE	$\tau_7 * f$	Sync	CI	$\tau_{12} * f$
Sync	C	τ_3	Sync	SE	$\tau_8 * f$	Sync	OK	$\tau_{13} * f$
Sync	R	τ_9	Sync	S	$\tau_4 * f$	Loc	LA	τ_{14}
Loc	LB	τ_{15}	Sync	CA	$\tau_{10} * f$	Sync	V	$\tau_5 * f$
Loc	LD	τ_{16}						

$f = [st \;\; AA == run]$

Figure 4. Calendar Manager

age; Automaton Browser (AB): representing the browser visualization active or not; Automaton Tree (AT): representing the navigability tree used to change of document; Automaton Document (ADoc): representing a document open, no save or neither document open; Automaton Dialogs (AD): representing all action dialogs of application.



Type	Event	Rate	Type	Event	Rate	Type	Event	Rate
Sync	ST	τ_1	Sync	C	τ_{10}	Loc	l_6	τ_{19}
Sync	QT	τ_2	Sync	T	τ_{11}	Loc	l_7	τ_{20}
Sync	R	τ_9	Sync	D	τ_{12}	Loc	l_8	τ_{21}
Sync	S	τ_4	loc	l_3	$\tau_{13} * f$	Loc	l_9	τ_{22}
Loc	l_1	τ_5	Sync	l_4	τ_{14}	Loc	l_{10}	τ_{23}
Loc	l_2	τ_6	Sync	l_5	τ_{15}	Loc	l_{11}	$\tau_{24} * f$
Loc	l_{12}	τ_7	Sync	l_{13}	τ_{16}	Loc	l_{14}	$\tau_{25} * f2$
Loc	l_{15}	τ_8	Sync	l_{16}	τ_{17}	Loc	l_{17}	τ_{26}
Loc	l_{18}	τ_9	Sync	l_{19}	τ_{18}			

$f = [st \;\; Documents == Unititled]$;
$f2 = [st \;\; Tree == Root \&\& Documents == NoDoc]$;

Figure 5. Docs Editor

6. Test Case Generation and Analysis

After building the usage models, the next step of the Statistical Testing process was to generate *test suites* (set of test cases) from these models and then analyzing them. Next section describes the process of generation of test suites by characterizing the automated test case generator used on this experiment. The structure of the test cases is shown as well.

6.1. Automated Test Case Generator

STAGE (STAte-based test GEnerator) is an integrated environment for computer-supported generation of test cases and test scripts using state-based techniques. Our goal in building this environment was to provide a framework where we can easily apply different testing techniques unified by the approach of state-based modeling.

The architecture of STAGE is organized in three packages, called STAGE-Model, STAGE-Test and STAGE-Script. STAGE-Model is a toolkit for creating and editing state-based models of the application under test, using a state-based formalism; the current version supports four model types: finite state machines, finite state machines with variables, MC and SAN. Here we focus on MC and SAN models.

When the model is a SAN, one can perform some preliminary static analysis of the model before generating the tests, using the PEPS tool [1]. STAGE-Model exports the model into a file with the .san extension, containing the model description in the PEPS input format. One of the features of PEPS is to compute the equivalent MC for a given SAN, and output it into a .hbf file. STAGE-Model may import this equivalent chain into the system using the HBF-Import tool.

STAGE-Test is a toolkit for generating test cases from state-based models built using STAGE-Model. To create a test suite, the user must select a model and one of the test case generation techniques. The generated tests can be stored with the test suite on the database or exported into a text file. There are specific random test case generation algorithms for each type of model (MC and SAN). The input for the generator is the model, the maximum test case length and the sample size (number of test cases).

In terms of test case generation for MC models, the generation tool walks through the model states, starting on the initial state. At each state, transitions are selected according to their probability distribution. The test case ends when the terminate state of the chain is reached or in the case that the maximum test case size is reached.

The generation of test cases based on SAN usage models works quite different of the MC pro-

cess. According to the models developed, each test case should start on ST event and end at QT event. So, the generation tool analyzes the current global state of SAN, enumerating the candidate events to be fired, according to the current local state of each automaton, and an event is selected according to their rate distribution, such that events with higher rates have higher probability of being selected.

The test cases are generated on the same database as the models, making references to the automata, source and target states, transitions and events that constitute each test case step. The input parameters for the generator are the model and the expected size of the test suite.

Figure 6 shows the main window of STAGE-Model, with the SAN model of the login application. STAGE-Model opens one window for each automata in the model. Note that using STAGE-Model we edit just the high-level automata. The relationship between the interface states and SAN states are declared on a XML file.

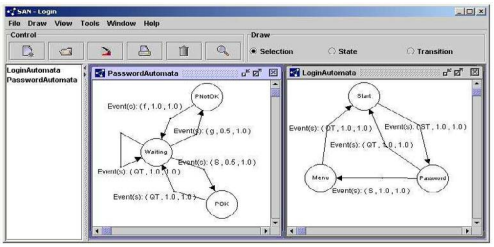


Figure 6. STAGE-Model Interface

With STAGE-Test, we create the test suite. Depending on the model type, a different set of test case generation techniques is available. Each technique has its own parameters. Regarding SAN models, the test case generation technique is the usage-model statistical testing algorithm. In this case, for each test suite we must define the number of test cases to be generated and the maximum test case size (number of steps). Figure 7 shows the main interface of STAGE-Test with a sample of test cases. Note that each test case step is composed by a SAN global state followed by the event that triggers the state change.

7. Quantitative Analysis

In order to analyze some numerical issues of test case generation through SAN models, we draw a comparison with the test case

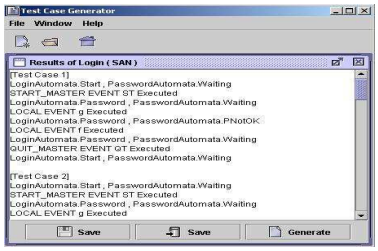


Figure 7. STAGE-Test Interface

generation using MC models. The first comparison shows the time needed to generate sets of test cases for the three study cases presented in the previous section. The second comparison shows different behavior of the convergence to a satisfactory set of test cases by using SAN or MC.

7.1. Generation Time Analysis

We generated five test suites from SAN and MC models of Simple Counter Navigation, Calendar Manager and Docs Editor applications. Each test suite is a set of test cases and the samples were generated with 100, 500, 1000, 2000, 3000, 4000, and 5000 test cases. For these test suites we perform four generations procedures limiting the test cases sizes to 10, 20, 30 and 40 steps. Therefore, when generating a test suite limited to 10 steps, we only consider test cases where the largest test case pass by 10 model states, and so on.

Figure 8 shows the processing time (in seconds) for all the samples generation. The test cases were generated in a Pentium III 550MHz with 256 MB of RAM accessing a SQL Server Data Base.

The first information observed from these results (Figure 8) is the small time necessary to the test case generation of the small models. Even the generation of 5000 test cases with 40 steps limit stays under 5 and 8 seconds respectively. The Docs Editor model generation times also remain quite reasonable (under 210 seconds) considering its size (417 states).

It is possible to observe an almost linear behavior of the generation times for the MC models. This phenomenon is probably due to the simplicity of the MC models, which describe all possible states by one single automaton. The SAN models have a more complex representation based on local and synchronizing events. Such greater complexity in the SAN models structure is the reason that explains the more significant difference between

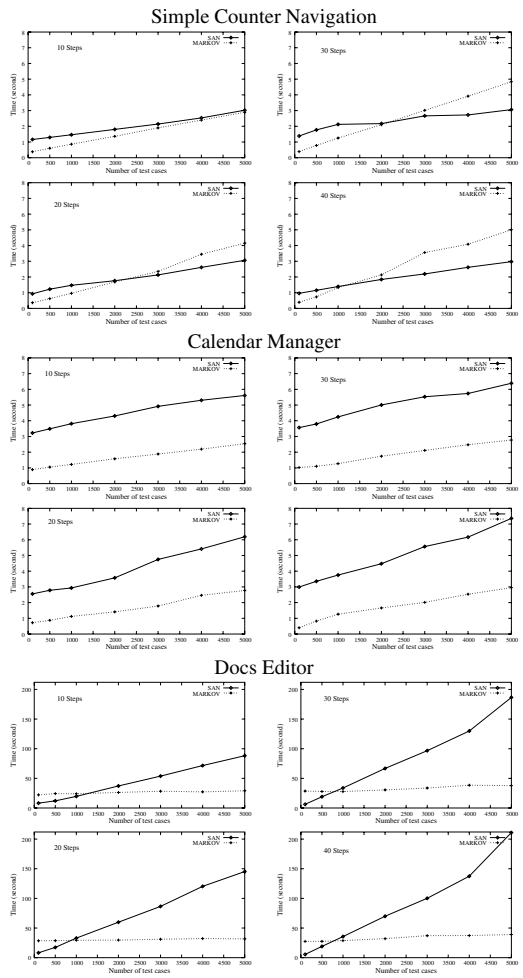


Figure 8. Generation Times

the generations times encountered in the Calendar Manager models.

The Calendar Manager SAN model has a large number of synchronizing events (13 events) and it increases the generation times. A similar phenomenon can be observed to Docs Editor SAN model where the number of synchronized events is also relevant (12 events).

The Simple Counter Navigation SAN model has a small number of synchronizing events (5 events), and, therefore, the generation time for the SAN model actually becomes lesser than the time for the MC model. In fact, for larger (generating more than 2000 cases) and longer (samples with 20 or more steps) test cases the SAN model of the Simple Counter Navigation application shows a quite consistent gain in comparison to MC model.

In fact, the size of the model, as well as its complexity, seems to amplify tho growing times in the generations of larger test suites (3000 and over). The MC model seems to be the much more simple to handle, since the time increase is negligible. This quite particular behavior must be subject of a further study.

7.2. Quality of Test Suites

The second comparison proposed in this paper tries to numerically express how fast the test case generation provides a *satisfactory test suite*. We mean by *satisfactory* a test suite that represents a good trade-off between the total number of generated test cases, and the number of distinct test cases in each sample. To evaluate the convergence of the test case generation for a model, we observe how many new test cases are added to the sample as the generation procedure goes on. To that matter we consider the number of different test suites generated with 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500 and 5000 test cases. For this test suites we also perform four generations procedures limiting the test cases sizes to 10, 20, 30 and 40 steps.

In Figures 9, we indicate the absolute number of new test cases generated for the SAN and MC models of Calendar Manager and Docs Editor applications. We can observe from these curves a better quality of the generation for the SAN models. For the Calendar Manager example, the complexity of the SAN model is probably the cause for such irregular curve. For the Docs Editor models the size (417 states) must be responsible for the improved results obtained by the SAN model.

The next results (Tables 1, 2, 3 and 4) present the quality trade-off for Calendar Manager and Docs Editor examples. Each of the tables indicates the absolute number of different test cases generated for 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500 and 5000 sized test suites. Respectively, the tables present results for test cases limited in 10, 20, 30 and 40 steps. The column percent indicates the percentual rate of distinct test cases considering the size of the test suite.

Observing these values, we notice a considerable fluctuation of values which is expected due to the random nature of the generation process. The pure comparison between SAN and MC shows a higher number of new test cases, attesting the better quality of SAN generated test suites.

In order to estimated how good an application was covered by a test suite we assume that lower

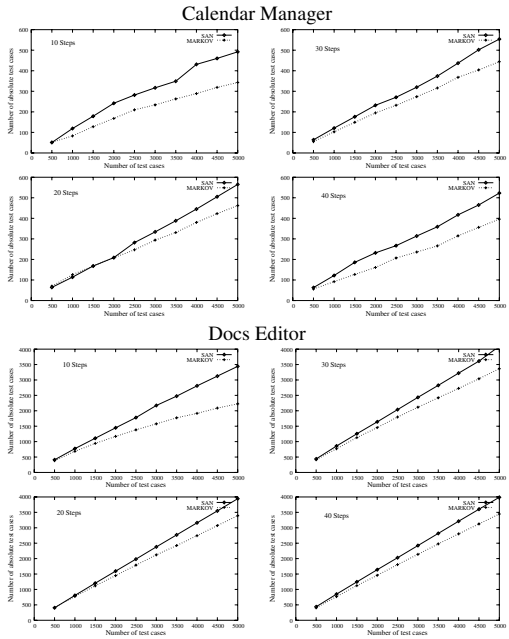


Figure 9. Quality of test suites

percentages indicates a good coverture. When an application was satisfactory covered, *i.e.*, the test suite has enough different test cases, the number of test cases replications is high. Note that this analysis is not valid when comparing percentages of different models, *e.g.*, between SAN and MC.

Trying to visualize the evolution of coverture for the models, despite of the fluctuation, we believe that Calendar Manager models have low percentage in comparison with Docs Editor models caused by the difference of complexity between these examples. Thus, even for 500 sized test suites the coverture of Calendar Manager example seems satisfactory. The same phenomenon could the explain a better coverture of Docs Editor example for the table representing test cases limited at 10 steps (Table 1).

Size of Test Suites	Calendar Manager				Docs Editor			
	Absolute		Percent		Absolute		Percent	
	SAN	MC	SAN	MC	SAN	MC	SAN	MC
500	51	52	10.2%	10.4%	410	382	82.0%	76.4%
1000	119	83	11.9%	8.3%	773	685	77.3%	68.5%
1500	179	128	11.9%	8.5%	1109	942	73.9%	62.8%
2000	242	168	12.1%	8.4%	1448	1170	72.4%	58.5%
2500	282	210	11.3%	8.4%	1779	1383	71.2%	55.3%
3000	317	234	10.6%	7.8%	2173	1576	72.4%	52.5%
3500	349	263	9.9%	7.5%	2476	1773	70.7%	50.6%
4000	431	289	10.7%	7.2%	2808	1916	70.2%	47.9%
4500	460	319	10.2%	7.1%	3125	2088	69.4%	46.4%
5000	492	343	9.8%	6.9%	3442	2226	68.8%	44.5%

Table 1. Test suites with 10 steps

Size of Test Suites	Calendar Manager				Docs Editor			
	Absolute		Percent		Absolute		Percent	
	SAN	MC	SAN	MC	SAN	MC	SAN	MC
500	64	70	12.8%	14.0%	399	389	79.8%	77.8%
1000	114	126	11.4%	12.6%	796	731	79.6%	73.1%
1500	168	169	11.2%	11.3%	1188	1071	79.2%	71.4%
2000	209	208	10.4%	10.4%	1585	1404	79.2%	70.2%
2500	282	247	11.3%	9.9%	1982	1743	79.3%	69.7%
3000	334	294	11.1%	9.8%	2373	2094	79.1%	69.8%
3500	388	331	11.1%	9.4%	2760	2429	78.9%	69.4%
4000	445	380	11.1%	9.5%	3143	2748	78.6%	68.7%
4500	505	423	11.2%	9.4%	3530	3089	78.4%	68.6%
5000	565	462	11.3%	9.2%	3917	3389	78.3%	67.8%

Table 2. Test suites with 20 steps

Size of Test Suites	Calendar Manager				Docs Editor			
	Absolute		Percent		Absolute		Percent	
	SAN	MC	SAN	MC	SAN	MC	SAN	MC
500	56	55	12.8%	11.0%	439	414	87.8%	82.8%
1000	121	103	12.1%	10.3%	857	769	85.7%	76.9%
1500	176	149	11.7%	9.9%	1254	1131	86.3%	75.4%
2000	232	195	11.6%	9.7%	1642	1455	82.1%	72.7%
2500	271	232	10.8%	9.3%	2038	1798	81.5%	71.9%
3000	320	274	10.7%	9.1%	2437	2116	81.3%	70.5%
3500	374	316	10.7%	9.1%	2826	2422	80.7%	69.2%
4000	437	368	10.9%	9.2%	3222	2726	80.5%	68.1%
4500	502	404	11.2%	8.9%	3611	3040	80.2%	67.5%
5000	554	444	11.1%	8.9%	4094	3365	81.9%	67.3%

Table 3. Test suites with 30 steps

Size of Test Suites	Calendar Manager				Docs Editor			
	Absolute		Percent		Absolute		Percent	
	SAN	MC	SAN	MC	SAN	MC	SAN	MC
500	63	56	12.6%	11.2%	438	404	87.6%	80.8%
1000	122	92	12.2%	9.2%	848	768	84.8%	76.8%
1500	186	127	12.4%	8.8%	1245	1124	83.0%	74.9%
2000	232	161	11.6%	8.1%	1640	1458	82.0%	72.9%
2500	267	207	10.7%	8.3%	2029	1805	81.2%	72.2%
3000	314	236	10.5%	7.8%	2423	2138	80.7%	71.3%
3500	359	266	10.3%	7.6%	2818	2478	80.5%	70.8%
4000	417	315	10.4%	7.8%	3213	2803	80.3%	70.1%
4500	465	356	10.3%	7.9%	3602	3126	80.1%	69.5%
5000	523	396	10.5%	7.9%	3989	3447	79.8%	68.9%

Table 4. Test suites with 40 steps

8. Conclusions

The use of SAN formalism to develop usage models seems naturally a better option than the classic MC. The possibility to describe an application by interacting modules (automata) seems an unavoidable advantage. The complexity of the real world applications with distributed code, web-based interfaces and databases furnishes very little room to a monolithic model, such as a straightforward MC. Nevertheless, the advantages of a canonical stochastic representation provided by the MC formalism is quite desirable for the generation of test cases (or any other procedural approach). The alternative of using SAN formalism seems to be, at least, just as good as the MC formalism, since the SAN models keep all stochastic characteristics of the equivalent MC model.

We discovered with the quantitative analysis presented in this paper that the SAN formalism can have few disadvantages and important advantages in comparison to the use MC formalism. The higher time cost to generate test suites with SAN models do not seem to be prohibitive. In fact, this

higher cost is justified by more effective result, *i.e.*, test suites with more distinct test cases (Section 7.1). Additionally, SAN models seem to converge more rapidly to satisfactory test suite (Section 7.2).

The set of experiences developed for this paper was relatively small, since larger models could be easily imagined. However we believe our point was not to explore all the power of description of the SAN formalism, but to be restricted to small examples in which the equivalent MC model was not too difficult to describe. In fact, we develop some larger examples in the SAN formalism, but the development of an equivalent MC model was too complex, and, therefore, it was not possible to manually verify the equivalence with the SAN model. A natural future work will then be the study of automatic procedures to compare large SAN and MC models. Such work would require a further study of algorithms for models structural analysis.

A natural future work to validated the proposed test case generation is to complete the testing process with script execution. However this practical work would provide evidences of success, we believe that SAN generated test suites will always be more effective due to the larger number of distinct test cases. In fact, this future work also waits for larger examples, because only to really large applications the number of bugs founded by script execution could be statistically significant to be analysed.

The most appealing future work on this subject seems to be the evolution of the test case generation algorithm. The current procedure, for both SAN and MC models, is based on the generation of test cases regardless their statistical relevance. With the complete SAN usage model, it is quite easy, computationally speaking, to calculate the statistical relevance of a specific test case. If, instead of generating test cases and analyzing the convergence to a satisfactory set as performed in Section 7.2, we guide the generation procedure by the generation of only the most statistically relevant test cases, the savings in overall generation time could be enormous. This future work needs an efficient solver for the SAN model to compute transient solutions [9].

Finally, we believe that the use of SAN formalism to model usage models has all the reasons to be a new standard in the statistical test community. However, the future works for this paper may also include the use of SAN formalism to develop other kinds of models of software, *e.g.*, data structures access models or, more generally, any resource utilization restriction. This work may result in a complete framework to formally describe virtually every aspect of an application and automatically generate reliable and efficient test suites.

References

[1] A. Benoit, L. Brenner, P. Fernandes, B. Plateau, and W. J. Stewart. The PEPS Software Tool. In *Computer Performance Evaluation / TOOLS 2003*, volume 2794 of *LNCS*, pages 98–115, Urbana, IL, USA, 2003. Springer-Verlag.

[2] A. G. Farina, P. Fernandes, and F. M. de Oliveira. Representing software usage models with stochastic automata networks. In *14th international conference on Software engineering and knowledge engineering*, pages 401–407. ACM Press, 2002.

[3] P. Fernandes, B. Plateau, and W. J. Stewart. Efficient Descriptor - Vector Multiplication in Stochastic Automata Networks. *Journal of the ACM*, 45(3):381–414, 1998.

[4] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Transactions on Software Engineering*, 27(11), 2001. Pages 1023-1036.

[5] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, New York, USA, 1998.

[6] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, Ontario, Canada, 1979.

[7] J. R. Norris. *Markov Chains*. Cambridge University Press, New York, USA, 1998.

[8] K. Sayre. *Improved techniques for software testing based on Markov chain usage models*. PhD thesis, University of Tennessee, Knoxville, USA, 1999.

[9] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, New Jersey, USA, 1994.

[10] J. Tian and E. Lin. Unified Markov Models for Software Testing, Performance Evaluation, and Reliability Analysis. Seatle, USA, aug 1998.

[11] C. Trammell. Quantifying the Reliability of Software: Statistical Testing Based on a Usage Model. Montreal, Canada, aug 1995.

[12] G. H. Walton and J. H. Poore. Generating transition probabilities to support model-based software testing. *Software Practice and Experience*, 30(10), 2000. Pages 1095-1106.

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