STAGE: an Integrated Environment for Statistical Test Script Generation[1]

Bernardo Copstein, Flávio Oliveira, Lucas R. C. Reginato

CPTS/FACIN – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
90.616-900 – Porto Alegre – RS – Brazil

{copstein,flavio}@inf.pucrs.br, reginato@cpts.pucrs.br

Abstract. *This work describes STAGE, an integrated environment for statistical test case generation and scripting, developed at the CPTS (Software Testing Research Center). The key contributions of the system are the use of stochastic automata networks (SANs) for representing the usage model and an intermediate model (called the Interface Event-State Model) to map the abstract usage model into the implemented interface components. SANs allow modular representation of systems with complex non-deterministic behavior, minimizing the state-space explosion found in Markov chains. The use of a separate model for the implementation has the advantage of making changes in the interface and generating the script automatically without affecting the abstract usage model. We implemented the technique into the STAGE environment; the experiments indicate that the performance of test case and script generation with SANs is at least compatible with Markov-based generation.*

1. Introduction

The interest on statistical testing has increased in recent years, because of two main reasons: the growing complexity and non-determinism of the environments on which modern software is expected to operate (the Internet is just the paradigmatic and most popular example) and the higher levels of reliability required for complex mission-critical applications. Indeed, reliability assessment is crucial for systems with safety and fault-tolerance requirements. Such demands are difficult to address with traditional testing techniques. Even when one considers a more deterministic environment, the complexity of the software itself and its interface may require that the test engineer develop the test cases from a complex model describing the application. In both cases, the use of automated tools is inevitable. Such applications are simply too complex to test using only manual tests. Which poses another problem: how to create test scripts with hundreds or thousands of tests in an effective way? Moreover, how to update such scripts efficiently when there is a change in the product? Clearly, there is much to be gained from the possibility of generating the test cases automatically from a high-level model of the application, and then creating the test scripts automatically from the test cases.

Technology for statistical testing with usage models is mostly based in Markov chains. However, a Markov chain for a complex application can easily reach hundreds or thousands of states, which makes the modeling and maintenance a difficult task.

[1] This work is supported by the HP-PUCRS Cooperation Agreement 01/99 - TA 17 (CPTS project).

Moreover, in order to generate the test script, it is necessary to add implementation-related information to the usage model.

We present here STAGE (STAte-based test GEnerator), a test script generation toolkit for statistical testing, developed at CPTS (Software Testing Research Center), a cooperation project involving PUCRS and HP. The key features of the system are the use of stochastic automata networks (SANs) for representing the usage model and an intermediate model (called the Interface Event-State Model - ISEM) to map the abstract usage model into the implemented interface components. SANs allow modular representation of systems with complex non-deterministic behavior, minimizing the state-space explosion found in Markov chains. The use of a separate model for the implementation has the advantage of making changes in the interface and generating the script automatically without affecting the abstract usage model. STAGE also includes features for deterministic testing, based on finite state machines. In this way, one can combine deterministic and statistical testing, using finite-state models as an unifying concept. The choice of state-based testing is motivated by two main characteristics: (a) state diagrams are widely used in software modeling and design; (b) there are state-based formalisms that support both deterministic functional testing and statistical testing. The result is a robust and flexible tool with a unified set of modeling primitives, simplifying the task of the test engineers.

In this paper, we focus on the features introduced in STAGE for statistical testing. In section 2, we describe the issues involved in usage modeling in general and present the SAN and ISEM models. Section 3 describes the STAGE environment. Section 4 comments on related work, while in section 5 we give our conclusions so far and comment on future directions of our research.

2. State-Based Statistical Testing

Statistical Testing is a wide concept. If some aspect of a system, in any level of abstraction, could be represented using events whose occurrence rules could be represented by a statistical model, so we can do statistical testing. The major advantage of using a statistical approach for testing is that in doing so is possible to estimate parameters as, for example, reliability, what are not possible in traditional testing. One of the possible approaches for statistical testing is those based on *usage models*.

2.1 Usage Models

A usage model characterizes the operational use of a software system. Software is used by a user on a specific environment [8, 10], where the user may be a person, a hardware device, other software, or a group of users. A software use may be a working session, transaction or any service unit limited by a start/finish event.

The model structure is composed of a state set and the transitions between these states, constituting a graph. The graph nodes represent the model states, and the graph arcs represent the transitions between states. This structure describes the possible uses of the software. A probability distribution, associated with the model, describes the expected use of the software.

Usage models are represented with some system modeling formalism [8] – discrete Markov chains are the usual choice. In this formalism, the usage states of the software are mapped as chain states, and the user actions are mapped as transitions between these

chain states. There are two other special states: *invoke* – that represents the beginning of the program execution – and *terminate* – that represents the end of the program execution. These are the only start and finish states of the chain, respectively. The transition probabilities or rates represent the user action's probabilities, configuring the typical uses of the software.

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 some event's frequency, project the testing chronogram, estimate software reliability, and so on. We focus here on its application to the testing phase.

When represented with Markov chains, usage models allow the test engineer – a person who has the responsibility of test's 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 software statistically typical usage paths;
- Long-run occupancy, e.g. utilization time percentage for each state;
- Mean number of events per test case;
- Mean number of events between two states, etc.

2.2. Markov Chains

A stochastic process is specified as a family of random variables defined in a probabilities space and indexed by a parameter. Usually this parameter refers to an index set of the process time or time range. If we use discrete time (ex: $T = \{1,2,3,...\}$) we will have a discrete stochastic process, but if we use continuous time (ex: $T = 0 < t < +\infty$) so the stochastic process will be continuous. A markovian process is a stochastic process where the distribution probability function assumes the main property of a markovian process, that is, the process evolution depends only upon its current state, being completely independent of the system previous states.

When the space of states of a markovian process is discrete, the process is called a *Markov Chain*. This kind of chain is classified according to a time scale as Discrete Time Markov Chains (DTMC) and Continuous Time Markov Chains (CTMC).

Using a discrete chain (DTMC) the triggering of transitions from a state to another is ruled by conditional probabilities. These probabilities are denoted by a real number in the [0;1] range and the sum of all transition probabilities from a state to another must be equal to 1.

2.3. Stochastic Automata Networks

We describe here a formal conceptualization of Stochastic Automata Networks (SANs). The treatment given here is somewhat different from the original presentation, given in the works of Fernandes [2]. The goal here is to provide a basis for understanding the approach and data model of STAGE-Test.

Given a set S of states (the local states), let A = { A_1, A_2, … A_n } be a set of *automata*, where each automaton A_i is a subset of S. A **Stochastic Automata Network (SAN)** is a structure

(G,E,Pe,Pt,I)

where:

- G = { G_1, G_2, … G_m } is a set of **global states**, such that each Gi is an element of A_1 X A_2 X … X A_n. In other words, each global state is a combination of local states of the automata.
- E = { E1, E2, … Ek } is a set of **events**. Each event is a function Ei: G → ***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 Ai 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.
- Pe = { P1, P2, … Pk } is a set of **event probability functions**, one for each event. Each function Pi: G → ℝ describes the probability of occurrence of the event at each global state.
- Pt = { Pt1, Pt2, … Ptkm } 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 S and the event is fired, the SAN goes to a state S' which must be an element of Ei(S). The transition probability functions describe the probabilities of the different elements of Ei(S) being selected. Usually, Ei(S) has only one state, so the definition of these probabilities is optional.
- I ⊆ G is a (possibly empty) set of **initial states.** In the original definition, SANs do not have initial states; they are useful for the specification of usage models. Since they are optional, this definition includes the original as a special case.

Any SAN can be converted into an equivalent Markov chain [2]. The states of the Markov chain are the global states of the SAN.

Stochastic automata networks have been shown to have a number of advantages for modeling complex systems, in comparison with Markov chains [4,8]. We believe that this is the case also for statistical testing based on usage models, without loss of generality or information; indeed, any Markov chain can be mapped into a SAN [3,9]. Usage models described with SAN have some interesting characteristics [1]:

- 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 for easier mapping of uses into test cases;

- 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.

Figure 1: Login System

For example, let us consider an (quite simple) application consisting of just two dialogues: the first is a login dialog (fig. 1a), where the user is prompted for a username and password; if the username or password is incorrect, the application issues an error message (fig. 1b). The second is a menu (fig. 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:

A_1 = { Start, Password, Menu }

A_2 = { 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) }

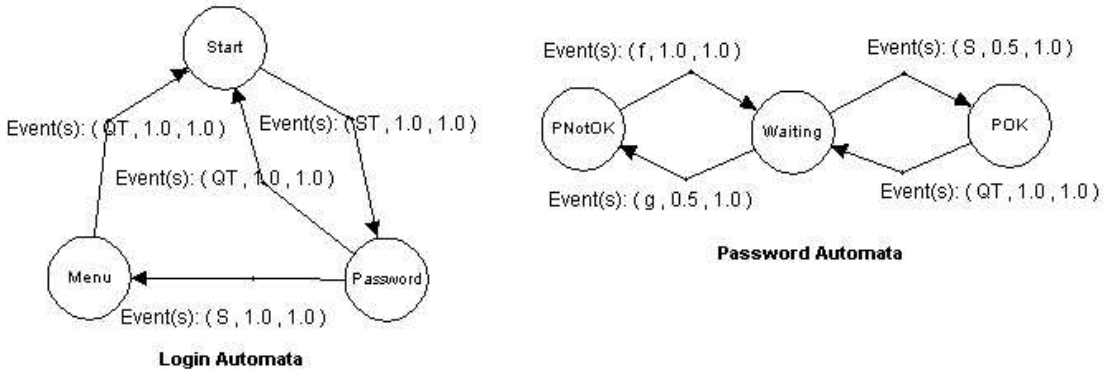

Figure 2: SAN Model

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 figure 2, each event is represented by a triple <event identifier, event probability, transition probability>. For example, the probability of **ST** is 1.0, because it is the only possible event at the state (Start, Waiting), and its transition probability is also 1.0, since there is only one transition assigned to **ST** at that state.

2.4 Interface State-Event Model

STAGE-Test creates the test cases from the SAN usage model. These test cases form the input for STAGE-Script, which creates the scripts for automatic execution of those tests. Since the SAN is an abstract model describing usage of the application to be tested, it does not represent design and/or implementation information, which is needed for script generation. Therefore, we have an **Interface State-Event Model** (**ISEM**) associated with the SAN model. The ISEM contains the information necessary for test script generation.

Let S be a set of local states and A the set of automata. Let (G,E,Pe,Pt) be a SAN as described in section 3. An ISEM consists of a structure

(Ic,SI, EI, Fs, Fe)

Ic is a set of **interface components** (frames, buttons etc.).

SI is a set of **interface states.** Such states represent possible combinations of properties values (values of input fields, window visibility etc.) of the interface components when the system is in that state. Each combination is an interface state.

EI is a set of **interface events**, such that for each possible event in E there is a subset of interface events in EI. The interface events represent which user actions (button clicks, choices etc.) trigger the corresponding event in the model.

Fs: S $\rightarrow$ ***P***(SI) is a mapping from states to sets of interface states, such that for each local state in S there is an associated (possibly empty) subset of states in SI. It is possible having local states with no interface state, representing internal logic of the application. There is an important requirement, though: in order to generate the test script, each reachable global state must have at least one associated interface state.

Fe: E $\rightarrow$ ***P***(EI) is a mapping from events to sets of interface events. For each event in the model there must be at least one related interface event; otherwise, it is not possible to create a script for test cases with that event.

For example, in the SAN model of figure 2, the *Password* state in the **Login** automata corresponds to the pop-up window (a) in figure 1. We have two situations of interest: the fields “user” and “password” are filled with valid values and with invalid values. Therefore, we can define two interface states in the ISEM model of the application, both associated with the **Login**.*Password* state in the SAN model.

3. STAGE

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 process of designing tests for an application using STAGE consists of three steps: (a) building a model of the application to be tested; (b) generating test cases from the model; (c) generating test scripts from the test cases. The architecture of STAGE is organized in three packages, corresponding to the steps above. We call these packages STAGE-Model, STAGE-Test and STAGE-Script, respectively (figure 3). STAGE was developed in the Java language.

STAGE-Model is a toolkit for creating and editing state-based models of the application under test, using one of four types of state-based models - finite state machines, finite state machines with variables, Markov chains or SANs. We focus here on the SAN models. The core tool in the toolkit is SCE (State Chart Editor), a graphic editor where the user (normally the test engineer) defines the model structure – states, transitions, events and rates, depending on the model type. SCE can also import the states and transitions from a spreadsheet file. This is a useful feature when you want to test a new version of an application previously documented outside SCE. For example, when developing a Web site, many teams describe the pages and links in a spreadsheet. In that case, SCE reads the sheet and creates a first drawing of the model.

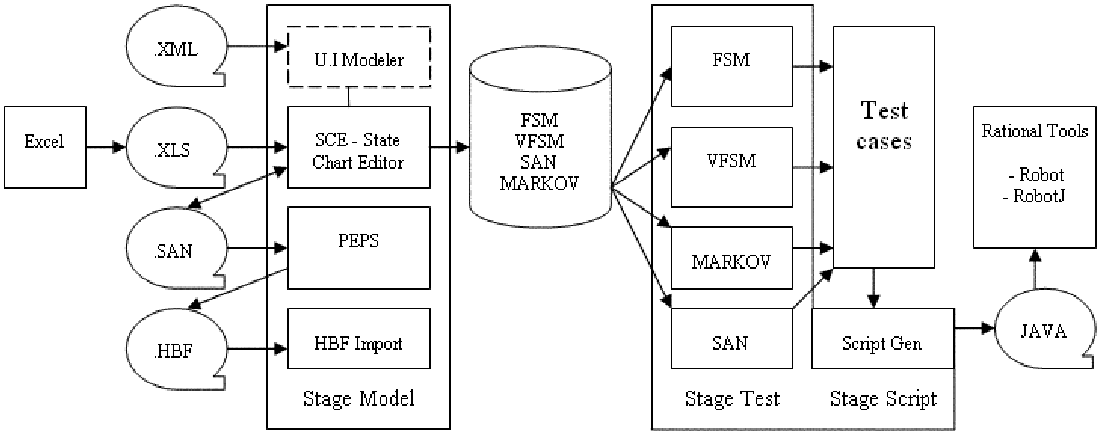

Figure 3: Architecture of STAGE

When the model is a SAN, one can perform some preliminary static analysis of the model before generating the tests, using the PEPS tool [4]. SCE 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 Markov chain 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.

The U.I. Modeler (dotted box on figure 3) is a user interface modeling tool that generates an XML description of the user interface and its relationships with the states and events in the model. This information is necessary for the script generation process in STAGE-Script. The format of the description is an implementation of the ISEM model described in the section 3. The UI Modeler is currently under implementation. At this moment the XML format is already specified but must be created using a text editor.

STAGE-Test is a toolkit for generating test cases from stated-based models built using SCE. The core tool in this case is TCG (Test Case Generator), an interactive tool where the user can create test suites. 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.

For each type of model there are test case generation techniques that can be applied. For FSM or VFSM models (VFSM are properly converted to FSM before test case generation) there are two techniques available: a simple state coverage technique and the Wp algorithm [9]. For SAN and Markov models there are specific random test case generation algorithms. Each technique has some user-defined parameters. For all of them it is possible to define the maximum test case length; for Markov and SAN models, it is necessary to inform the number of test cases to be generated (the sample size).

In terms of test case generation for Markov chain models, the generation tool walks thru 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 Markov chain process. According to the models developed, each test case should start on the **ST** event and end at the **QT** event. So, the generation tool analyzes the current global state of the 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.

STAGE-Script is the script generation tool. It is integrated in the TCG interface and allows the user to generate automatic test scripts from the generated test cases. The current version generates scripts for the Rational playback testing tools (Robot and RobotJ) [7]. Users can generate 3 types of scripts: navigation, duration and performance. The first one is just the translation of the generated test cases to the script language. In the duration script, the user defines a time interval of test execution. The script will apply all the generated tests and, if there is time remaining, it will select tests at random to be applied until time is over. The performance script is specific for testing web servers and generates series of web page requests, simulating multiple users requests over a predefined time span. It is important to note that the script generation technology adopted applies only to systems with GUIs; that is a limitation of the current version. We are studying other execution engines and languages, in order to generate scripts for different types of applications.

3.1 Examples and experiments

Let us see an example of test script generation using a SAN model. We will use the login application presented in section 3. Figure 4 shows the main window of SCE, with the SAN model of the login application. SCE opens one window for each of the automata in the model. Note that using SCE we edit just the high-level automata. The relationships between the interface states and the SAN states are declared on a XML file as mentioned above. On figure 5 we can see a sample of the corresponding XML file.

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 6 shows the main interface of TCG 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.

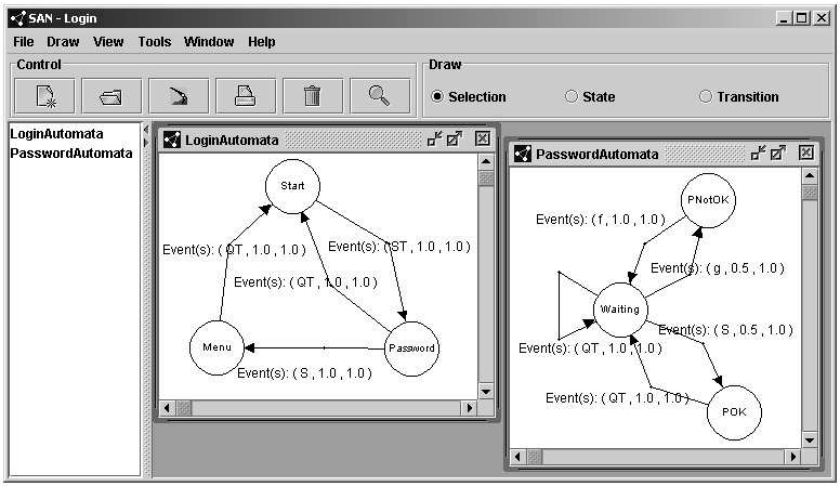

Figure 4: SCE interface

```
<isem>
    <interface_structure>
        <structure>
            <component name="MenuDialog">
                <object id="1" objectClass="JButton" text="EXIT" />
             </component>
         </structure>
        .
        .
    </interface_structure>
    <interface_states>
    <state id = "v1" modelState="Login Automata.Passoword" property="PasswordDialog">
        <components>
            <object id="Text1" value="lucas"/>
            <object id="Text2" value="123456"/>
        </components>
    </state>
    .
    .
    </interface_states>
    <interface_events>
        <event map="S">
            <transition>
                <source> valid 1 </source>
                <eventAction id="OK" action="Click" />
                <target> MenuDialog </target>
            </transition>
        </event>
    </interface_events>
</isem>
```

Figure 5: XML file ISEM model

The test script is also generated using TCG. In the current version, the generated scripts can be for Rational RobotJ or Rational Robot. Figure 7 shows a snippet of the script for testing the login application, based from the test suite. Note that it is necessary to combine the information in the test cases with the information declared on the XML file in order to generate a test script.

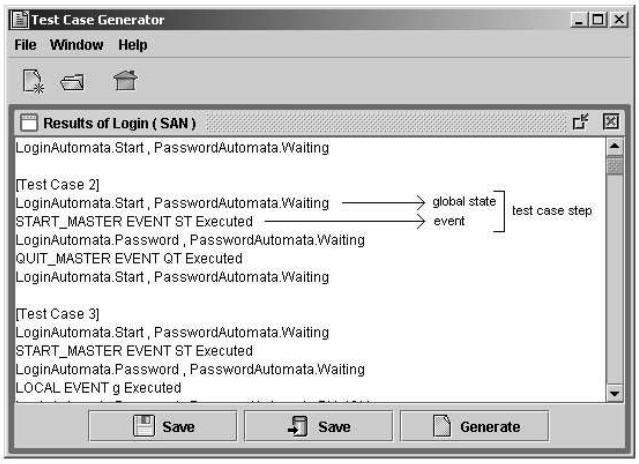

Figure 6: TCG interface

```
...
logInfo("Executing Test Number 3");
  try{
    startApp("Main");
    passwordtextText().setProperty("text", "123456");
    textText().setProperty("text", "lucas");
    OKButton().click();
    EXITButton().click();
  }catch(Exception e){
    logTestResult("Error generate = "+e.toString(), false);
  }
logInfo("Executing Test Number 4");
  try{
    ...
  }catch(Exception e){
    logTestResult("Error generate = "+e.toString(), false);
  }
...
```

Figure 7: Script for Login System

We developed a set of experiments with the environment, considering the following applications:

- the login application described above (2 automata, 9 global states),
- a Web calendar tool (5 automata, 420 global states),
- a forms-based documents editor (6 automata, 648 global states), and
- a Web-based bug tracking tool (8 automata, 9000 global states).

For each application, we developed a SAN model and generated the equivalent Markov chain, using the PEPS tool. For these models, we generated test suites from 100 to 5000 test cases, varying the maximum test case size from 10 to 40 steps. For eample, Figure 8 shows the execution times for the Calendar tool, considering 40 steps of maximum test case size both for the Markov Chain and SAN models.

The first information observed from these results is the small time necessary to the test case generation. Even the generation of 5000 test cases with 40 steps as limit stays under 8 seconds. It is possible to observe an almost linear behavior of the generation times for the Markov models. This phenomenon is probably due to the simplicity of the

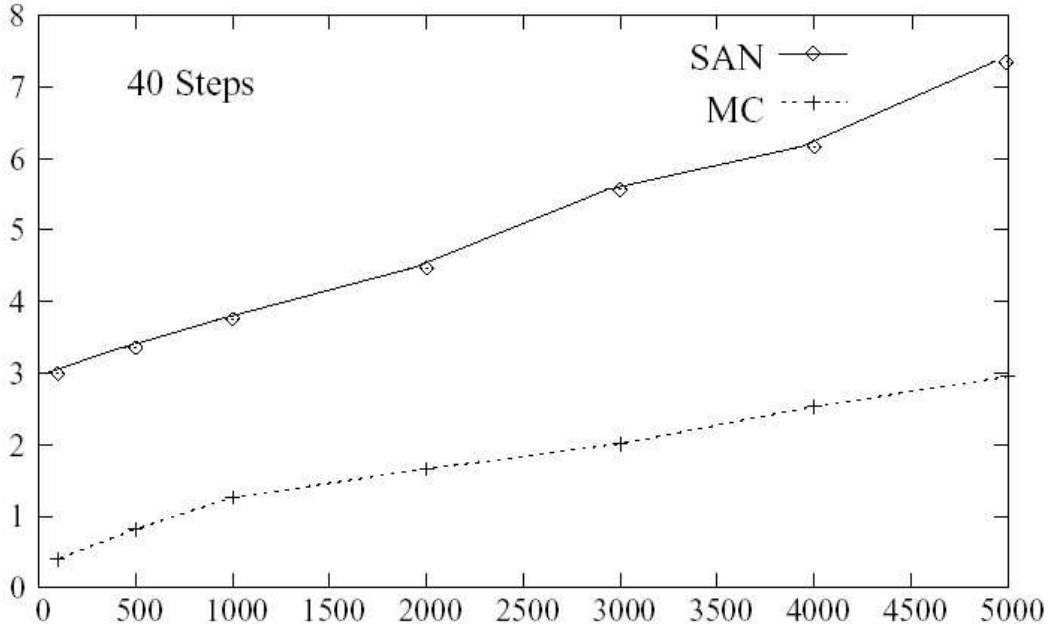

Figure 8: execution times for Calendar tool

Markov models, which describe all possible states in one single automaton. The algorithm for test case genration from the SAN models has a higher complexity, because of the synchronizing events. We observed that the number of synchronizing events in the model has an impact on the test case generation times - for example, the Calendar SAN model has a large number of synchronizing events (13 events).

4. Related Work

UMMs (Unified Markov Models) [6] are an extension of traditional Markov based models. UMM models capture the same kind of information that a SAN or a Markov chain does, but this information is represented as a set of hierarchical Markov chains. A top-level Markov chain represents the high-level states and transitions. At any level, more detail can be associated with an individual state using sub-models, creating a hierarchical model. Notice that in some of these Markov chains the sum of probabilities for transitions out from a given state may be less than one because the external destinations are omitted to keep the model simple. The implicit understanding is that the missing probabilities go to external destination (a model in a different level).

SANs are not hierarchical; instead, they allow for many different types of relatioships among system components, while keeping the description modular. From a computational viewpoint, SANs also have higher scalability: for example, it is possible to model environment aspects as, for example, number of servers and clients, with little impact in the computational effort to compute model properties. Such flexibility is useful for both statistical and load testing. Most current products in the testing industry (such as Rational TeamTest or Mercury TestRunner) do not generate test suites based on a usage model. Implemented systems use Markov chains, which impose severe limitations on the complexity of the models.

5. Conclusions and Future Work

Statistical testing is a technique with its own history of success. Nevertheless, the size of state-space and the need for automated execution have imposed restrictions on the application of statistical testing to more complex systems. STAGE is a framework that supports state-based modeling and script generation for complex systems in a modular

way. The productivity of the script development and script maintenance tasks also increased significantly; when there is a change in the interface, one only needs to edit the state-based model and re-generate the test cases and scripts. The key feature in the script generation is the mapping from the abstract model to the interface model.

A limitation of the current version of STAGE is that the ISEM model must be generated manually, using a text editor; we are already building the UI Modeler to help this task, which can be very time-consuming for complex interfaces. More than that, some parts of the ISEM model (components description) could be automatically extracted from the interface code (Java or HTML), leaving to the test engineer the task of defining states and assigning links to the high-level model (FSM, VFSM, Markov or SAN). We are currently investigating this possibility. We are also planning to incorporate state-based code analysis for object-oriented software components. This will allow, in the future, to integrate structural statistical testing into STAGE framework.

References

[1] Farina, A.G.; Fernandes, P H L; Oliveira, F.M.. "Representing Usage Models With Stochastic Automats Networks". In: *14th International Conference On Software Engineering And Knowledge Engineering* - SEKE02, 2002, Ischia, Italy.

[2] Fernandes, P. *Méthodes numériques pour la solution de systèmes Markoviens à grand espace d' états*. INPG, Grenoble, 1998. (PhD Thesis)

[3] Fernandes, P., Plateau, B. and Stewart, W.J. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, volume 45, no. 3, 1998, pp. 281-414.

[4] J.M.Fourneau, B.Plateau. PEPS: a package for solving complex Markov models of parallel systems. In: Proceedings of the Fourth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Palma, Spain, 1988.

[5] Fujiwara, S.; Bochmann G. V.; Khendek F.; Amalou M. and Ghedemsi A. Test Selection Based on Finite State Model. IEEE Transactions On Software Engineering, vol 17(6):591-603, 1991.

[6] Kallepalli, C.; Tian J. - Measuring and Modeling Usage and Reliability for Statistical Web Testing. IEEE Transactions on Sotware Engineering, vol 27(11):1023-1036, 2001.

[7] Rational products. Description available at URL http://www.rational.com/products/. Accessed 9/19/2003.

[8] Sayre, K. Improved Techniques for Software Testing Based on Markov Chain Usage Models. PhD thesis. University of Tennessee, Knoxville, December 1999.

[9] Shehady, R.K.; Siewiorek, D.P.; A method to automate user interface testing using variable finite state machines Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 24-27 June 1997. Page(s): 80 -88

[10] Trammel, C. Quantifying the Reliability of Software: Statistical Testing Based on a Usage Model. *Proceedings of the Second IEEE International Symposium on Software Engineering Standards*, Canada, August 1995