Test automation for Markov Chain Usage Models

Adriana M. Bettinotti,
adrianabettinotti@yahoo.com.ar
Mauricio Garavaglia
mauriciogaravaglia@gmail.com
Grupo de Desarrollo en Sistemas, Facultad Regional Delta, Universidad Tecnológica Nacional, Campana, Argentina

Abstract
Statistical testing with Markov Chain Usage Models is an effective method to be used by programmers and testers during web sites development, to guarantee the software reliability. The JUMBL software works on this models; it supports model construction with the TML language and analysis, tests generation and execution and analysis of tests results. This paper is targeted at test automation for web sites development with JUMBL and JWebUnit.
Key words: test automation; Markov chain usage models; JUMBL; TML, JWebUnit

1 - Introduction

Model-based software development techniques are becoming more and more attractive in order to master the inherent complexity of real-world applications. Exhaustive testing is not possible, even for a small software under test (SUT). Markov chain usage models (MCUM) are a part of this techniques.
In the prospect to develop a method for testing websites reliability during development, we adopted this model, in the context of the research of Poore et al. (Software Quality Research Laboratory (SQRL), University of Tennessee, Knoxville, USA) [1,6]. Their approach, essentially, transfers the practical operations of a software system into a usage model, which then forms a basis for performing statistical testing and analyzing software reliability. The usage model is a finite state, discrete parameter Markov Chain, assuming that the current state (usage/failure behavior) completely determines the next state.
These models are more accurate than others, because they explicitly model states in the software. The Markov structure model for estimation of software reliability may provide advantages such as: allowing us to avoid common assumptions about failure rate distributions, and incorporating both the operational profile and test coverage explicitly and automatically into reliability computation.
The ready availability of well-established analytical results for Markov chains, which have meaningful interpretations for software testing [1,4,6,7] is a valuable result of treating software testing as a stochastic process.
We use the J Usage Model Builder Library (JUMBL), which is a Java class library and set of command-line tools for working with usage models, developed by the SQRL.[2,3]
In this paper we focus in a main issue of this development, which is to **automate the test suite execution and recording of results,** to allow a quick report to use in the testing process.
The JUMBL takes a "least common denominator" approach to test automation, since there are many different execution platforms, therefore many different test environments**. Each test environment poses unique challenges which test automation must address.** So, we faced this challenge for the websites's development environment.

2 - A usage model: the Markovian Bank.

2.1: Creating a graphical structure

In this model, we represent a navigation structure that covers all the possible navigation paths of a bank's website [Figure 2]. Each node represents a task that may need to be refined to a lower model, (i.e. a "more detailed" usage model.)
The main goal was to automate the test suite generation from use case scenarios, having "statistical usage testing" as its main test objective.
We create initially the graphical structure of our 10-state usage model we are going to test. In the case of this web site, each state is a page we reach, or the result showed after a suite of interactions with the site.
Table 1 shows the names and a description for the states, and Table 2 for the stimuli.

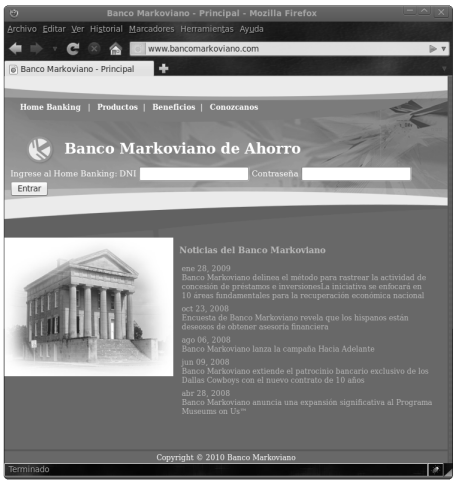


Figure 1. Banking Sample Web Application.

States

S0	Inicio	Start
S1	Principal	Main
S2	Home	Home
S3	Premios	Awards
S4	Premio no Otorgado	Award non accorded
S5	Premio Otorgado	Award accorded
S6	Pagos	Payements
S7	Depósitos	Deposit
S8	Resumen	Account Summary
S9	Salir	Logout

Table 1. States for the example usage model.

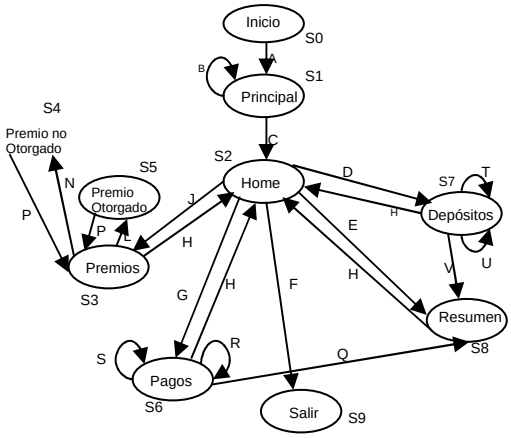


Figure 2. Graphical representation of the usage model

A	"Ir a Principal"
B	"Clave Mal"
C	"Clave Bien"
D	"Ir a Depósitos"
E	"Ir a Resumen de Cuenta"
F	"Salir"
G	"Ir a Pagos"
H	"Ir a Home"
J	"Ir a Premios"
L	"Elegir Premio y Alcanzan Puntos"
N	"Elegir Premio y No Alcanzan Puntos"
P	"Ir a Premios"
Q	"Ingresa Descripcion y Montos Correctos"
R	"Ingresa Descripción Vacía"
S	"Ingresa Monto Erróneo" (from S6)
T	"Ingresa Cuenta Vacía"
U	"Ingresa Monto Erróneo" (from S7)
V	"Ingresa Monto Correcto"

Table 2. Stimuli for the example usage model

The Markov chain usage model should be completed with a distribution attached to each arc, i.e. the transition probabilities. We further say something about this matter.

2.2: Describing and analyzing Markov chain usage models with TML and JUMBL

We now create from the graphical representation of the usage model, the **banking.tml** file, using the TML (The Model Language) , a description language for Markov chain usage models, developed by the SQRL (Software Quality Research Laboratory, University of Tennessee, Knoxville . [3,10,11]. Figure 3 shows a snippet of the banking.tml file. It can be written with any word processor.

```
model banking
source [Inicio]
        Ir a Principal" [Principal]

[Principal]
        a:|$        estoyEnPrincipal($s);
        "Clave Bien"
                        [Home]
                "Clave Mal"
                        [Principal]

. . .
[Premios]
  "Elegir Premio y alcanzan puntos"
                        [Premio Otorgado]
  "Elegir Premio y no alcanzan puntos"
                        [Premio no Otorgado]
  "Ir a Home"
                        [Home]
. . .
[Pagos]
   "Ingresa Descripcion Vacia"
                        [Pagos]
   "Ingresa Monto Erroneo"
                        [Pagos]
   "Ingresa Descripcion y Montos Correctos"
                        [Resumen]
   "Ir a Home"
                        [Home]
[Resumen]
   "Ir a Home"
                        [Home]
sink [Salir]
 end
```

Figure 3: a snippet of the banking.tml file.

This file features all the states and the stimuli of the usage model. Probability distribution may be associated to the models with TM. The default distribution assumed by TML and JUMBL is the uniform, the one that makes all the outgoing arcs from a state equally likely. For instance, since S2 has 4 outgoing arcs, the transition probability for all of them is 0.25.
The JUMBL has the command **Analyze**, which can be applied to banking.tml to create an analytical report of the model and use statistics that can be used to validate model correctness, plan for testing, investigate characteristics of expected use, and compare different modelling approaches. We show only for illustration a snippet of the report in fig. 4.

Node	Occupancy	Probability of occurrence
[Depositos]	100,E-3	0,5
[Home]	0,25	1,0
[Inicio]	50,E-3	1
[Pagos]	100,E-3	0,5
[Premio No Otorgado]	50,E-3	0,333333333
[Premio Otorgado]	50,E-3	0,333333333
[Premios]	0,15	0,5
[Principal]	0,1	1
[Resumen]	0,1	0,666666667
[Salir]	50,E-3	1

Node	Mean occurrence/ Variance (visits per case)		Mean First Passage/ Variance (cases)	
[Depositos]	2	10	2	2
[Home]	5	20	1	0
[Inicio]	1	0	1	0
[Pagos]	2	10	2	2
[Premio No Otorgado]	1,0	4	3	6
[Premio Otorgado]	1,0	4	3	6
[Premios]	3	24	2	2
[Principal]	2	2	1	0
[Resumen]	2	6	1,5	0,75
[Salir]	1	0	1	0

Figure 4 : A snippet of the Analytical report of the usage model.

(Attaching different distributions to the model, and analyzing the model are beyond the scope of this paper). See [2,3,6,7,10,11,12]

2.3: Introducing labels for Test Automation

Since this paper focuses on test automation, we now describe the use of labels that will be introduced in banking.tml. A "label" is arbitrary data that is attached to a model, state, or arc. Their primary use is to attach automated testing information to the model.

For this example, test automation was built using JAVA language, due to its frequent use by web sites developers, and its multi-platform capacity. We use JUnit and JWebUnit as frameworks for test automation, available for Java language. (Nevertheless, these operations may be done using any kind of framework for test automation, like TestNG, or Selenium.)
Figure 5 shows a snippet of the new version of banking .tml, with the attached labels.

```
model banking
a:|$
  |$   public class TestHomebk {\n
  |$                  @Test\n
  |$                  public void test() {\n
source [Inicio]
                    "Ir a Principal" [Principal]

[Principal]
          a:|$       estoyEnPrincipal($s);
          "Clave Bien"
               a:|$  escribirClaveBien($s);
          [Home]

          "Clave Mal"
               a:|$  escribirClaveMal($s);
          [Principal]

. . .
[Premios]
      "Elegir Premio y alcanzan puntos"
           a:|$       irAPremioQueAlcance($s);
       [Premio Otorgado]

           "Elegir Premio y no alcanzan puntos"
           a:|$       irAPremioQueNoAlcance($s);
       [Premio no Otorgado]

       "Ir a Home"
           a:|$       irAHome();
        [Home]

[Pagos]
        "Ingresa Descripcion Vacia"
      a:|$ dejarDescripcionEnBlancoYAceptarPago($s);
        [Pagos]
        "Ingresa Monto Erroneo"
       a:|$  dejarMontoEnBlancoYAceptarPago($s);
        [Pagos]
        "Ingresa Descripcion y Montos Correctos"
       a:|$ completarFormularioYAceptarPago($s);
        [Resumen]
        "Ir a Home"
       a:|$ irAHome($s);
        [Home]

[Resumen]
         "Ir a Home"
           a:|$ irAHome($s);
            [Home]

sink [Salir]
a:|$                    }\n
  |$                }\n

end
```

Figure 5: A snippet of banking.tml with labels.

We attached labels that produce the necessary stimuli to perform the transition between states, and validate that the correct state has been reached. Using JWebUnit, each "estoyEn…" (I am in…) validates the latter situation.

For test automation, each generated test must simulate the interactions between the user and the software. Each user's action has two main goals:

1) To stimulate the system in some way that makes the system perform the desired action.
2) To obtain the expected result from the system.

To achieve this goals, we use the methods provided by the testing framework., adapting them to the states and stimuli built with TML. We therefore simulate a user interacting with the web site pages. We now describe some examples.
We use the assertTextPresent(String a) method, that checks whether the page actually being visited contains the text that we passed as a parameter in the method's argument. If the page does not contain the text, the test fails, generating an error condition.
To simulate that the user is checking his account report, we use the method

```
protected void estoyEnResumen(int step) {
    this.step = step;
```

```
    assertTitleEquals("Banco Markoviano - Imprimir Resumen");
   assertTextPresent("Detalle de la cuenta");
}
```

The estoyEnResumen method gets an integer number as a parameter, which we keep as "step". This parameter will be eventually read by JUMBL, and will be used to report in which state there was an error during the test.
The assertTitleEquals y assertTextPresent methods get the text we should find en the title and text page, to be sure that we are visiting the account report page.

The banking.tml file contains the stimuli needed to make transitions between sates. These stimuli, from the standpoint of test automation, indicate the interaction with the page, either clicking on some actual link, or filling a form and sending it to be processed. In the following example, we simulate the action of signing in the Banco Markoviano's site with the correct user name and password.

```
protected void escribirClaveBien(int step) {
this.step = step;
assertTitleEquals("Banco Markoviano - Principal");
setTextField("dni", _DNI);
setTextField("password", _PASSWORD);
clickButtonWithText("Entrar");
}
```

Again, the integer got as a parameter is used by JUMBL to determine the last step executed by the test before the occurrence of a failure.
The setTextField method completes with its first parameter the text field whose name is the second parameter, thus simulating a user typing a word.
The clickButtonWithText method simulates the clicking on a link with a determined text.

With methods to evaluate whether the user has reached a particular state, and methods to simulate a stimulus to move among the states, we are ready to introduce the test instructions as labels in banking.tml. In a simple way, we added to each state and each stimulus the needed instructions to reach the state and then validate the intended state has been reached.
Now, we will describe more in detail the way labels work. (The whole text is shown in Fig. 5)

```
[Principal]
        a:|$ estoyEnPrincipal($s);
          "Clave Bien"
        a:|$ escribirClaveBien($s);
                [Home]
        "Clave Mal"
        a:|$ escribirClaveMal($s);
                [Principal]
```

In this example, as we told before, when the system gets state [Principal], the label contains the instruction to validate that the expected state has been reached, that is, the Main Page of the Home banking. The $s is a special JUMBL placeholder which is replaced by a number corresponding to the step of the test. If a failure occurs, this number can be used to know the exact place where the failure took place. The JUMBL can read and save this information, as we will explain in 2.4.

The stimulus "Clave Bien" implies that the user typed the correct name and password so he is expected to reach the state that represents the Home page.
The estoyEnPrincipal() and escribirClaveBien, as well as escribirClaveMal() methods have already been created and will be used from a library.

2.4: Generating and executing tests, saving and analyzing results.

Once the TML archive is completed, the JUMBL tool is used to generate a battery of tests.
JUMBL supports four automated test generation methods:
1) Generating the Coverage set: it is a collection of test cases which visit every arc in the model with the minimum number of test steps.
2) Generating Random test: Tests are generated randomly, in the number desired, based on the probabilities of the model.
3) Generating Weighted tests: each test case may a weight associated with it. This weight can be of two forms:
3.a) The probability of the test case generated. One can generate a number of test cases, in order of decreasing probability, with the highest probability test case first.
3.b) The sum of weights traversed in the test case. Under this approach, one needs to introduce a distribution in the TML file that assigns each arc a nonnegative "weight" (or "cost"), and test cases are generated in order starting with the lowest sum (from least costly to most costly)
4) Generating Tests Manually: The JUMBL has a GUI application to design, save and modify test cases.

Next, we show some examples of random tests.
With the command:

```
$ jumbl GenTest -n 20 banking.tml
```

JUMBL generates 20 random tests from the graph defined in the banking.tml file. This tests are saved in a file named banking.str. JUMBL organizes tests into Save Test Records (STR file), herein called simply *test records*.
This file may be read by JUMBL again for analysis purposes, through the Test Case Analysis report. (It is a report generated *before* test case execution, in which JUMBL assumes that test cases have been executed to completion without failure.)
We need now to *export* these tests, to get the code generated from the text attached to the labels, in our case, in JAVA language.
The following command:

```
$ jumbl ManageTest Export banking.str --key=a --extension=.java
--start=/* --end=*/ --suffix=_utn
```

writes the test cases from the banking.str test record as 20 single files, named banking_0_utn.java, banking_1_utn.java,…, banking_19_utn.java with the code we wrote inside the label "a" inside the TML file.
Thus we get 20 ".java" files containing the code to execute each particular test.
A snippet of banking_0_utn.java is shown in fig. 6.

```
/* ====================== */
/* Trajectory: 0 */
/* Model: banking */
/* Key: */
/* Method: random */
/* */
/* Events: 11 */
/* Including failure information. */
/* ======================== */
/* Step: 1, Trajectory: 0 */
import org.junit.Test;
public class TestHomebk {
@Test
public void test() {
/* Step: 2, Trajectory: 0 */
estoyEnPrincipal(2);
escribirClaveBien(2);
/* Step: 3, Trajectory: 0 */
estoyEnHome(3);
irAPagos(3);
...
```

Figure 6: A snippet of banking_0_utn.java.

This first part is written in almost legal executable JAVA code, which may be recognised by JWebUnit as a test. The generated class extends the MarkovTest class, which contains the auxiliary code to get, in case of failures, *where* they occurred: the numbers passed as parameters to the methods estoyEnPrincipal, escribirClaveBien, and estoyEnHome will be used to determine the step in which a failure occurred. (The $s parameter is automatically replaced by the number of step in each test, as fig. 6 shows).

Now we need to pass the source file through a precompilation stage before we can compile and execute it by using a JwebUnit runner.

The precompilation stage will fix the source code by changing the class declaration to follow the Java convention, which states the public class name that must match the file name

We created a GUI [graphical user interface] to perform these operations automatically. The "Markov Runner" as we call it, will perform the following operations:

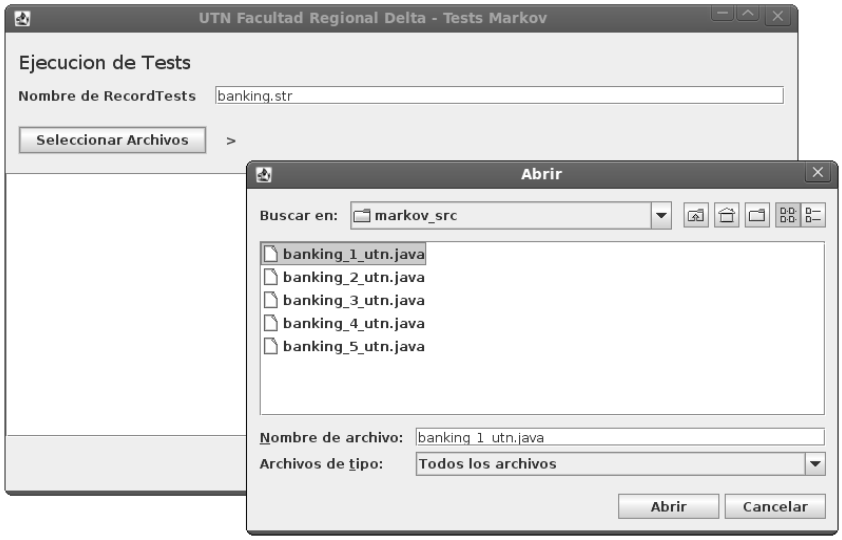
1) Source code fixing to follow Java specification.
2) Java Compiler invocation with JwebUnit Libraries to produce executable Test class files.
3) Test execution as JWebUnit tests cases. Each time a failure occurs in a test execution, the step in which failure occurred is saved..
4) Display of the test results using the Jumbl format.

Here is an example of the use of the Markov Runner:

1)We introduce the name of the .str file, such as banking.str

2)With a click in "Seleccionar Archivos" we choose the .java files containing the tests produced by the Jumbl ManageTest Export. It is also possible to choose a whole folder containing all the .java file, to produce automated execution of all the tests.

Figure 7: the Markov Runner.



After clicking Seleccionar Archivos, test execution begins. Then, after a short time, we will see the screen shown in Fig. 8.

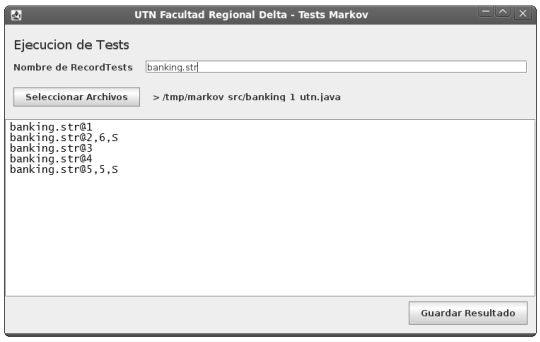


Figure 8: Testing results with the Markov Runner.

The output we get is used to give feedback to JUMBL about test execution results. These results can be exported to a text file making click on Guardar Resultado and loaded on JUMBL to continue with the Markov Analysis. An exemple of the output after testing is shown in fig. 8. JUMBL detected that tests 1, 3 and 4 were run from start to finish without failure, test 2 was run with a failure at step 6 and test 5 was run with a failure at step 5. The JWebUnit libraries make a test stop when a failure occurs. This last fact is shown by the S beside the number of step.

Once testing has been performed, we can generate with JUMBL a comprehensive report based on testing experience, the Test Case Analysis. Many product and process measures are available, including estimated product field reliability, and indicators of where unreliability is most significant, and stopping criteria. Test Case Analysis reports considerable information in different levels of granularity:

- The model section reports overall counts and coverage statistics
- The stimuli section reports stimuli counts and reliabilities
- The arcs section report arc counts and reliabilities
- The reliabilities section reports overall (end-to-end) reliabilities and stopping criteria.

In the arcs section we can see that the two failures occurred in “Ir a Home” from [Resumen].
As an example, we show in table 3 only the line corresponding to the failures in the Stimulus test Statistics:

Stimulus	Generated	Executed	Failed
Ir a Home	9	5	2

Stimu-lus	Reliability Variance	Optimum Reliability/ Variance	Prior Successes/ Failures	
Ir a Home	0,538461538 0,017751479	0,764705882 0,009996155	4	4

Table 3: A line of the Stimulus report

The arc was generated 9 times into the five tests, but, since 2 of them stopped before the end, it only was executed 5 times, with 2 failures. Thus, the optimum reliability does not match the reliability. The optimum values assume that all generated events are executed without failure.
We will not give details about the values of priors, we will only say that the tests were ran assuming “non prior information”, according to Bayesian model, as Prowell and Poore propose in [6] and [4].In the last columns, since we ran the tests without using prior information, the table’s line shows that there are 4 arcs in the model having the stimulus “Ir a Home”. We will not give details about the priors, we only say that value 1 is the “non prior information” of successes and failures for arcs. Since the stimulus labels 4 arcs, the priors for this stimulus are 4 and 4. The priors can be controlled by the user through the JUMBL.
If we want to fix the error before running more tests, we can see the following lines of the Arcs Stimulus test Statistics (this is only a part of the table). On the bottom of the table we can see that the 2 failures occurred in “Ir a Home” from [Resumen].
In table 4 (next page) we can see that the arcs labeled with “Ir a Home” come from the states [Depósitos], [Pagos] and [Resumen]; but the failure occurred from state Resumen, whose line we highlight, where the reliability 0,25 decreases with respect to the optimum value , 0,833333333.

Single Event Reliability	11,4767585E-3
Single Event Variance	819,946137E-9
Single Event Optimum Reliability	13,6317191E-3
Single Event Optimum Variance	586,098345E-9
Single Use Reliability	0,295446364
Single Use Variance	0,592367625
Single Use Optimum Reliability	0,42639844
Single Use Optimum Variance	0,399250525
Arc Source Entropy	26,9033447E-3 bits
Kullback Discrimination	0,14003805 bits
Relative Kullback Discrimination	520,523 %
Optimum Kullback Discrimination	79,7825301E-3 bits
Optimum Relative Kullback Discrimination	296,552 %

Table 5: The overall reliabilities

Table 5 shows some overall measures. The *single event reliability* is introduced to provide an estimate of the probability that a single user action, (a single state transition in the usage model), will occur without failure. The *single-use reliability*, defines the reliability as the probability of the software executing a randomly selected use without a failure; in our example, from state [Home] to state[Salir]. All these values are based upon Markov Chain results. Again, the values are lower than the optimum ones, because of the failures occurred.
The arc source entropy is the measure of long-run average uncertainty in choosing the next arc, for the distribution used in the usage model, before testing. The five tests executed define a new distribution, whose arc source entropy is different from the former. The Kullback discrimination is a measure of the difference between these two source entropies, and approaches zero as the two distributions became similar. In our case, the usage model’s arc source entropy is 26,9033447E-3 and the Kullback discrimination is 0,14003805. To better appreciate their relationship, the relative Kullback discrimination: (Kullback discrimination / arc source entropy) . 100 is shown. Its value, considering the optimum case (that is, if the tests had been executed without failure), is 296,552 %, and higher: 520,523 % when the errors occurred. We conclude that the generated distribution from the 5 tests is far from the true distribution.
The arc source entropy and Kullback discrimination derive from Theory Information. See [4] and [7].

Arcs	Proba-bility	Generated	Executed	Failed	Reliability/ Variance		Optimum Reliability/Variance		Prior Successes /Failures	
[Depositos]										
"Ingresa Cuenta Vacia"	0,25	3	3	0	0,8	0,026666667	0,8	0,0266666667	1	1
"Ingresa Monto Correcto"	0,25	2	0	0	0,5	0,083333333	0,75	0,0375	1	1
"Ingresa Monto Erroneo"	0,25	2	2	0	0,75	0,0375	0,75	0,0375	1	1
"Ir a Home"	0,25	2	1	0	0,66666667	0,055555556	0,75	0,0375	1	1
[Pagos]										
"Ingresa Descripcion Vacia"	0,25	0	0	0	0,5	0,083333333	0,5	0,083333333	1	1
"Ingresa Monto Correcto"	0,25	0	0	0	0,5	0,083333333	0,5	0,083333333	1	1
"Ingresa Monto Erroneo"	0,25	1	0	0	0,5	0,083333333	0,5	0,083333333	1	1
"Ir a Home"	0,25	2	1	0	0,66666667	0,055555556	0,75	0,0375	1	1
[Resumen]										
"Ir a Home"	1	4	2	2	0,25	0,0375	0,8333	0,01984127	1	1

Table 4: The arcs in the Arcs Report, where the stimulus "Ir a home" appears.

3. Conclusions and future work

Our goal is to develop a method to test the reliability of software. We keep working based on the approach presented here in both the theoretical domain and in developing good engineering practices. We specially intend to develop a practical way to decide when to stop testing, a sensitive point of statistical testing.

4. References:

[1] J.H. Poore, G.H. Walton, J.A. Whittaker: "A constraint-based approach to the representation of software usage models", Information and Software Technology, vol. 42 (2000) p. 825–833.

[2] S. J. Prowell: "JUMBL: A Tool for Model-Based Statistical Testing", Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03) (2002).

[3] Software Quality Research Laboratory (University of Tennessee, Knoxville): "JUMBL 4.5 User's Guide", 28th July 2003.

[4] S. J. Prowell: "Computations for Markov Chain Usage Models", Technical Report UT-CS-03-505.

[5] K.W. Miller, L.J.Morell et al.: "Estimating the probability of failure when testing reveals no failures", IEEE Transactions on Software Engineering, Vol. 18, Nº 1, January 1992.

[6] S.J. Prowell , J.H. Poore: "Computing system reliability using Markov chain usage models", The Journal of Systems and Software 73 (2004) 219–225.

[7] Kirk Sayre: "Improved techniques for software testing based on Markov Chain Usage Models" ,Ph. D. thesis. The University of Tennessee, Knoxville.December 1999.

[8] K. Sayre, J.H. Poore : "Stopping criteria for statistical testing", Information and Software Technology 42 (2000) 851–857.

[9] S.J. Prowell: "TML: a description language for Markov chain usage models", Information and Software Technology 42 (2000) 835–844.

[10] S.J. Prowell: "A TML Tutorial", Software Quality Research Laboratory (University of Tennessee, Knoxville) October 27, 2000.

[12] G. H. Walton, J. H. Poore: **"**Generating transition probabilities to support model-based software testing", Software –Practice and Experience. 2000; 30:1095–1106.

[13] J.A.Whittaker, Michael Thomason: "A Markov Chain model for Statistical Software Testing" ", IEEE Transactions on Software Engineering, Vol. 20, Nº 10, October 1994.

Received: March 2011. Accepted: Aug. 2011.