A Combination of Model-Based Testing and Random Testing Approaches using Automata

Frédéric Dadeau[1], Pierre-Cyrille Héam[12], and Jocelyn Levrey[1]

[1] LIFC - Université de Franche-Comté – INRIA CASSIS Project
16 route de Gray – F-25030 Besançon cedex
email: {frederic.dadeau,pierre-cyrille.heam}@lifc.univ-fcomte.fr, jlevrey@chu-besancon.fr

[2] LSV - Ens de Cachan – INRIA - CNRS UMR 8643
61 avenue du Président Wilson – F-94235 Cachan cedex
email: heam@lsv.ens-cachan.fr

Abstract. Developing efficient and automatic testing techniques is one of the major challenges facing software validation community. In this paper, we show how a uniform random generation process of finite automata, developed in a recent work by Bassino and Nicaud, is relevant for many faces of automatic testing. The main contribution is to show how to combine two major testing approaches: model-based testing and random testing. This leads to a new testing technique successfully experimented on a realistic case study. We also illustrate how the power of random testing, applied on a Chinese Postman Problem implementation, points out an error in a well-known algorithm. Finally, we provide some statistics on model-based testing algorithms.

1 Introduction

1.1 Motivations and Contributions

Producing secure, safe and bug-free programs is one of most challenging problem of modern computer science. In this context, two complementary approaches address this problem: verification and testing. On one hand, verification techniques mathematically prove that a code or a model of an application is safe. However, complexity bound makes verification difficult to apply on large-sized systems. On the other hand, testing techniques do not provide any proof but are relevant, in practice, in order to produce high quality software. Last years, many works have been done in order to upgrade hand-made (or experience-based) testing techniques to scientific based frameworks.

Since every configuration of a software can not be practically explored, one of the key problem for a validation engineer is to choose a relevant test suite while controlling the number of tests. The crucial question raised is then: “what means *relevant*?”. A frequent answer, in the literature and in practice, is to consider a test suite as relevant if it fulfills some well-known coverage criteria; for

instance, a code coverage criterion, that is satisfied if all the lines of the codes are executed at least once when running the tests. It is important to point out that coverage criteria can be applied on the code (white box or structural testing) or on a model of the implementation (black box or functional testing [4]). Since there are many ways to fulfill covering criteria [15], other criteria can be taken into account, for example based either on computing minimal/maximal length test suites, or on selecting boundary or random values for the test data.

This paper is dedicated to show how a recent result by Bassino and Nicaud [3] may be successfully exploited for several testing techniques. More precisely, this work describes the uniform random generation of automata. Thus, we propose to employ this technique in order to:

- show (in Sect. 2) how it can be used in a purely random testing approach, for generating test data. In this context, we report on a bug on a widely-used Chinese Postman Problem program (ranking second when googling "Chinese Postman Problem") and we show how to fix this bug.
- provide (in Section 3) some statistics on test suites generated in a pure model based testing approach. Such statistics may be relevant in order to help the tester to choose among different existing testing techniques.
- explore (in Sect. 4.1) how to combine random generation of finite automata with model based testing using coverage criteria. To the best of our knowledge, it is the first work making such combination. Our technique was applied (Sect. 4.2) on a non-trivial example of a Demoney model and implementation. This technique turns out to be very efficient, in particular it pointed out two non-conformance between the model and the implementation that were not discovered by other testing techniques.

1.2 Related Works

The work proposed in this paper is based on a random approach [7,11]. Even if such an approach is usually presented as one of the poorest way of generating data, it has been experienced as an efficient way for finding errors, into the software [10]. Random testing can be employed for generating test data, such as in DART [8] or to generate complete test sequences, as in the Jartege tool [16]. A recent work [9] provides an approach combining random-testing and model-cehcking. Our approach focuses on the uniform generation of automata that has been presented in [3].

We are in the context of testing from finite state machines [13]. This consits in describing the system from a labeled transition system on which different algorithms may be used to extract the test cases. This is the principle of SpecExplorer [6] that uses the Chinese Postman algorithm [17] to generate test

sequences. Other approaches, such as TGV [12], consider a product of the initial transition system with a test purpose, also expressed by a labeled transition system, that guides executions of the system.

The test generation technique that we propose differs by proposing a test sequence length-guided approach. This approach can be seen as automated test generation of test purposes, that is motivated by the objective of providing a user-defined number of tests when the test generation process is applied. To the best of our knowledge, this approach has never been targeted before.

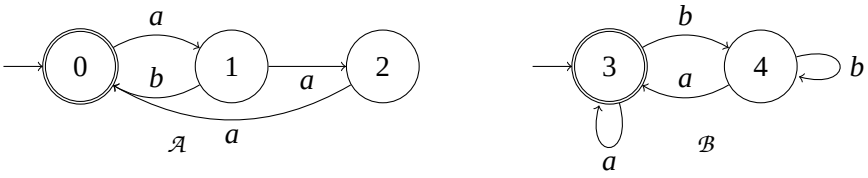
1.3 Notations

We define in this section useful notations. We assume the reader to be familiar with common definitions of *alphabet, letter, word,* etc. A *finite automaton* $\mathcal{A}$ is a quintuplet $(Q, \Sigma, \Delta, I, F)$, where Q a finite set whose elements are called *states*, Σ is a finite alphabet (i.e. a finite set of symbols), $I \subseteq Q$ is the set of *initial states*, $F \subseteq Q$ is the set of *final states* and $\Delta \subseteq Q \times \Sigma \times Q$ is the set of *transitions*. A finite automaton is *deterministic* if for every state p, there exists at most one transition of the form (p, a, q) and if I is a singleton. A finite automaton is *complete* if for every state p there exists at least one one transition of the form (p, a, q).

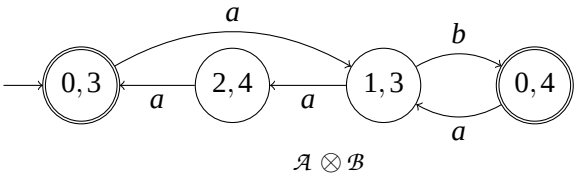
A *path* π in a finite automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ is a a finite sequence $(p_1, a_1, q_1), \ldots, (p_n, a_n, q_n)$ of transitions such that for every $1 \leq i \leq n-1$, $q_i = p_{i-1}$. The transition (p_i, a_i, q_i) is denoted $\pi(i)$. Such a path is called *a path from* p_1 to q_n. A path π *meets* a state p if there exists i such that either $\pi(i) = (p, a, q)$ or $\pi(i) = (q, a, p)$. The finite word $a_1 \ldots a_n$ is called the label of π. The path from an initial state to a final state is called a *successful path*. The set of labels of successful path is *the language accepted* by $\mathcal{A}$. A state p is *accessible* if there exists a path from an initial state to p. A transition (p, a, q) is *accessible* if p is accessible. A path is *accessible* if its first transitions is accessible. A *simple loop* is a path π from a state p to p such that for every i, j, $\pi(i) \neq \pi(j)$.

Given two finite automata $\mathcal{A}_1 = (Q_1, \Sigma, \Delta_1, I_1, F_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \Delta_2, I_2, F_2)$, the automaton $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the automaton $(Q_1 \times Q_2, \Sigma, \Delta, I_1 \times I_2, F_1 \times Q_2)$ where $\Delta = \{((p_1, p_2), a, (q_1, q_2)) \mid (p_1, a, q_1) \in \Delta_1 \mid \text{and } (p_2, a, q_2) \in \Delta_2\}$. Notice that this definition only differs from the definition on classical product of finite automata for final states. If $t = ((p_1, p_2), a, (q_1, q_2)) \in \Delta$, we denote by $\text{Proj}_{\mathcal{A}}(t)$ the transition (p_1, a, q_1) of $\mathcal{A}$. This projection can naturally be extended to paths of $\mathcal{A}_1 \otimes \mathcal{A}_2$ by $\text{Proj}_{\mathcal{A}}(t_1, \ldots, t_n) = \text{Proj}_{\mathcal{A}}(t_1), \ldots, \text{Proj}_{\mathcal{A}}(t_n)$ which is trivially a path of $\mathcal{A}$.

Consider the following illustrating toy example with automata $\mathcal{A}$ and $\mathcal{B}$:



The product $\mathcal{A} \otimes \mathcal{B}$ is the following automaton (we only represent accessible states).



Path $\pi = ((0,3),a,(1,3)),((1,3),b,(0,4)),((0,4),a,(1,3)),((1,3),b,(0,4))$ is a path of $\mathcal{A} \otimes \mathcal{B}$. One has $\text{Proj}_{\mathcal{A}}(\pi) = (0,a,1)(1,b,0),(0,a,1),(1,a,0)$.

We now need to precisely define what is a *coverage criterion*. It is a function that maps each automaton and each set of accessible paths of this automaton to 0 or 1. The *state coverage* criterion φ_s is defined by: $\varphi_s(\mathcal{A},\Pi) = 1$ if and only if for every accessible state q of $\mathcal{A}$ there exists a path π in Π meeting q. The *transition coverage* criterion φ_t is defined by: $\varphi_t(\mathcal{A},\Pi) = 1$ if and only if for every accessible transition (p,a,q) there exists a path π in Π such that $\pi(i) = (p,a,q)$. The *consecutive transition coverage* criterion φ_{ct} is defined by: $\varphi_{ct}(\mathcal{A},\Pi) = 1$ if and only if for every pair of accessible transitions of the form $(p,a,q),(q,b,r)$ there exists a path π in Π such that $\pi(i) = (p,a,q)$ and $\pi(i+1) = (q,b,r)$. The *simple loop transition coverage* criterion φ_{sl} is defined by: $\varphi_{sl}(\mathcal{A},\Pi) = 1$ if and only if for every accessible simple loop π_{loop}, there exists a path π in Π and path π_1, π_2 in $\mathcal{A}$ such that $\pi = \pi_1,\pi_{\text{loop}},\pi_2$.

2 Random Testing of a Chinese Postman Problem Implementation

The main idea of random testing is that randomness is not influenced by the tester. In this context, a crucial issue is to perform uniform generation, i.e. every element has the same chance to be selected by the algorithm. Otherwise, selected values are related to the chosen algorithms, what is precisely opposed to the main idea. The result [3] is therefore deep results, opening many possibilities to test algorithms that manipulate labeled graphs (e.g. the traveling salesman algorithm).

As an example, we use random generation of finite automata in order to test a well-known freely down-loadable[3] implementation [17]of the Chinese

[3] `http://web4.cs.ucl.ac.uk/uclic/harold/cpp/`

states	alphabet	tests	(1) fails	(2) fails	(2) is better	(1) is better
17	3	10	0	0	1	0
30	5	8	0	0	3	0
30	20	8	1	0	1	0
30	5	10	1	0	2	0
100	5	10	0	0	1	0

Table 1. Test results for the two versions of the algorithm (bugged, fixed)

Postman Problem [**?**]: given a labeled graph G, the question is to find a path in G of smallest length using all transitions.

We randomly generate strongly connected finite automata and we ask the tested program to provide a minimal path starting from the initial state and using all transitions. Generating 8 deterministic automata with 30 states on a 20 letters alphabet, we point out an automaton making the program fail (this automaton is provided in appendix A). It is not the purpose of the paper to discuss why there is a problem in the tested code. Just note it is an array overflow. We fixed this bug and we did several random tests on both the implementations, that did not reveal other errors on the original program nor side-effects introduced when fixing the bug.

Table 1 shows several experiments. Column *states* shows the size of generated automata, Column *alphabet* the size of the alphabet, Column *tests* the number of randomly generated automata, Column *(1) fails* the number of tests making the initial program failed, Column *(2) fails* the number of tests making the new implementation fail, Column *(2) better* the number of test providing a better (smaller) path with the new implementation and Column *(1) better* the number of test providing a better (smaller) path with the old implementation.

As we see, random generation does not provide any test that makes the new implementation fail. Moreover, several times, the new implementation provides better results than the old implementation (whose result is therefore not optimal and thus false).

3 Statistics for Model-Based Testing

3.1 Motivations

The goal of this section is to show how random generation of finite automata can provide fruitful statistics in a model-based testing context. However a not difficult (but quite long) work has to be done in order to compute and compare these statistics.

The testing phase is a crucial point for industrial development. For this purpose many works have been done in order to automatically achieve this point.

Methods	# states	# tests	av. length	max. length	length std. dev
FDS	20	2.4	16.03	17.4	1.29
CPP	20	11.04	22.34	131.4	37.25
FDS	23	2.64	18.59	20.1	1.49
CPP	23	12.06	23.11	146.2	40.02
FDS	26	3.04	20.93	22.74	1.91
CPP	26	13.68	23.1	167.2	43.74
FDS	30	3.24	24.05	23.38	2.2
CPP	30	15.46	23.58	195.32	48.34

Table 2. Statistical Results

Since testing process may be long and expensive, it is useful to provide some help to the tester to choose a testing technique. For instance, the time spent on testing directly depends on the number and on the sizes of test sequences. We propose in this section to employ random automata generation for building statistics for evaluating the result of different test generation algorithms.

3.2 Examples of Statistics

For computing above statistics, we proceeded as follows: we uniformly generate 30 trim (all states are both accessible and co-accessible) deterministic complete automata with n states on a 10-letters alphabet. Then, we compute a test suite using a first-depth search algorithm that covers *all states*. It is reported on Table 2 by the *FDS method*. We count the number of tests, that is the number of leafs of the covering tree. We also compute the average length of tests (the length of a test is the length of the path from the root to the corresponding leaf in the covering tree). Finally, we compute the maximal length of obtained tests and the length standard deviation. The same work is done using the covering criterion *all transitions* obtained by the Chinese Postman Problem, reported on Table 2 by the *CPP method*.

For each final state of each automaton, we add a *reset* transition, that is a transition labeled by a reserved character indicating that the system has to be re-initialized.

Table 2 provides some experimental results that are incomplete and gave to illustrate what kind of statistics can be done. Many experimental results are left to done in order to obtain experimental laws. One can give here some first interesting experimental results. First, one can notice that for the CPP methods, the number of tests is about the half of the number of states. This observation is still right if we vary the alphabet size. However, a deeper investigation shows that with the CPP method, the number of tests is approximately equal to the number of final states of the generated automaton. Obviously, there is a connection

between the average number of final states of randomly generated automaton and the size of the automaton.

3.3 The AUTIST Tool

Statistics presented above were obtained using an automatic tool called **AU**tomaton **T**est**I**ng **ST**atistics. This C++ based tool can be downloaded on
`http://lifc.univ-fcomte.fr/~dadeau/tools/#AUTIST`

For instance, the command line (on a linux computer)

```
$ autist -n9 -s8 -k4 -v -g
```

computes 9 automata (`-n9`) of size 8 (`-s8`) on a 4 letters alphabet (`-k4`). Option `-v` prints the transition table of all generated automata. The `-g` option computes a `.dot` file in the folder `output/dot` encoding a graphical representation of the automaton. Each `.dot` file is translated into a `.jpeg` file. This conversion may require a very long time, so option `-g` has to be used carefully and only on few small automata.

In order to compute statistics, it is also possible to use the `autist-grid` command. For instance, the command

```
$ autist-grid -n30 -mins8 -nmaxs24 -k4
```

computes statistics for automata from sizes 8 to 24, each on 30 4-letters automata.

The AUTIST-tool has to be improved in order to handle others testing algorithms and to compare them each other.

4 Combining Random Testing and Model Based Testing

In this section, we assume the tested implementation is modeled by a finite automaton $\mathcal{A}$, which is a frequently used abstraction. We first present in Sect. 4.1 a generic approach to combine model based testing and random testing and we expose experimental results in Sect. 4.2.

4.1 Testing Process

The main purpose is to generate a test suite that fulfills a given coverage criteria while integrating a random process. Given a testing procedure and a model $\mathcal{A}$ to be tested, the idea is to randomly generate an automaton $\mathcal{B}$, to compute a direct-product like $\mathcal{A} \otimes \mathcal{B}$ and to apply to testing procedure to this product.

The basic idea of this procedure is to guide the test generation by a coverage criterion, but also by the expected number of tests, that a validation engineer may require. Indeed, it is a current practice, especially in the industry, to consider that the more tests are run on the system, the more confidence we may

have in it. This approach is thus dedicated to drive the test generation so as to obtain, from a given automaton and a given a coverage criterion implemented into an test generation algorithm, a given number of tests. This approach reuses the results of previous section on the statistics of test suites.

Proposition 1. *Let $x \in \{s,t,ct\}$. If $\mathcal{B}$ is a complete accessible finite automaton and if $\varphi_x(\mathcal{A} \otimes \mathcal{B}, \Pi) = 1$, then $\varphi_x(\mathcal{A}, \mathrm{Proj}_{\mathcal{A}}(\Pi)) = 1$. If $\mathcal{B}$ is a complete strongly connected finite automaton and if $\varphi_{sl}(\mathcal{A} \otimes \mathcal{B}, \Pi) = 1$, then $\varphi_{sl}(\mathcal{A}, \mathrm{Proj}_{\mathcal{A}}(\Pi)) = 1$.*

Proof. We give here the proof for the φ_t criterion. Others proofs are similar.

Assume that $\varphi_t(\mathcal{A} \otimes \mathcal{B}, \Pi) = 1$. Let (p,a,q) be an accessible transition of $\mathcal{A}$. By definition, there exists a path π_1 in $\mathcal{A}$ from an initial state to q. Since $\mathcal{B}$ is complete and accessible, there exists a path π_2 in $\mathcal{B}$ from an initial state of $\mathcal{B}$ with the same label as π_1. Therefore, by a direct induction, there exists a path π in $\mathcal{A} \otimes \mathcal{B}$ from an initial state of $\mathcal{A} \otimes \mathcal{B}$ and ending in state of the form (p,r) where r is the last state of π_2. Since $\mathcal{B}$ is complete, there exists in $\mathcal{B}$ a transition of the form (r,a,s). Consequently the transition $((p,r),a,(q,s))$ is an accessible transition of $\mathcal{A} \otimes \mathcal{B}$. By hypotheses, there exists a path π' in Π and a positive integer i such that $\pi'(i) = ((p,r),a,(q,s))$. To finish, it suffices to note that $\mathrm{Proj}_{\mathcal{A}}(\pi')(i) = (p,a,q)$. It follows that $\varphi_t(\mathcal{A}, \mathrm{Proj}_{\mathcal{A}}(\Pi)) = 1$

For the φ_{sl} criterion, notice that the REGAL-tool [2] implementing [3] allows the uniform random generation of strongly connected deterministic automata.

Now one can consider the paths on automata $\mathcal{A}$ and $\mathcal{B}$ provided in Section 1.3.
$\pi_1 = ((0,3),a,(1,3)),((1,3),b,(0,4)),((0,4),a,(1,3)),((1,3),b,(0,4))$ and
$\pi_2 = ((0,3),a,(1,3)),((1,3),a,(2,4)),((2,4),a,(0,3))$.
One has $\varphi_t(\mathcal{A} \otimes \mathcal{B}, \{\pi_1,\pi_2\}) = 1$. Now $\mathrm{Proj}_{\mathcal{A}}(\pi_1) = (0,a,1)(1,b,0),(0,a,1),(1,a,0)$ and $\mathrm{Proj}_{\mathcal{A}}(\pi_2) = (0,a,1)(1,a,2),(2,a,0)$. One can easily check that

$$\varphi_t(\mathcal{A}, \mathrm{Proj}_{\mathcal{A}}(\{\pi_1,\pi_2\})) = 1.$$

4.2 Experimentation on the Demoney Case Study

Demoney, a Demonstrative Electronic Purse, is a specification of electronic purse that has been developed by Trusted Logics [14]. Even though Demoney is meant to be used for research purposes and does not aim at being embedded on a smart card, it recreates all the mechanisms of a smart card application, and thus, it can be considered as a realistic case study. This electronic purse is protected by two pin codes, one for the user and one for the bank. As for every smart cards, it has a life cycle, that begins with a personalization phase,

in which the different parameters of the card are being set (maximal amount on the card, maximal debit authorized, pin and bank codes). Then the card reaches the *use* phase in which different operations can be performed, such as crediting or debiting. The credit operation requires an authentification of the user, though the verification of his pin code. If he fails to authentificate (a given number of times) the card becomes blocked. It is then up to the bank to authentify in order to change/unblock the card. If the bank also fails to authenticate, the card is definitively dead.

From previous research and teaching experiments we had a formal model of the system, written as a B abstract machine [1] and a Java implementation, along with a number of mutants of this implementation. Each mutant is a variation of the original implementation, in which a mistake has been introduced on purpose. This technique is used to evaluate the quality of a test suite: the more mutant are killed, the more efficient is the test suite.

The experiment we designed is summarized in Fig. 1. We manually designed an automaton representing the Demoney specification. This automaton is displayed in Appendix B. We used an abstraction of the states, for which we abstracted the value of the balance of the purse to two possible values: 0 and > 0. We labelled the transitions by the different "behaviors" of the operations. A behavior is a subpart of an operation (for example, operation VERIFY_PIN(type,value) contains 4 behaviors depending on the input values of the parameters and the expected output of the command: correct –or incorrect– verification of a user –or bank– pin). Then, we ask the algorithm to generate a given number of tests, that will thus consist of sequences of operation behaviors. In order to generate full operation calls with instantiated values of parameters, we replay these sequences in a B symbolic animator [5]. We then only have to reuse our existing material to produce the JUnit tests cases that can be applied on the Java implementation and the mutants. When running the tests we are looking for a non-conformance between the results obtained by the implementation, and the expected results given by the model. This conformance relationship is based on

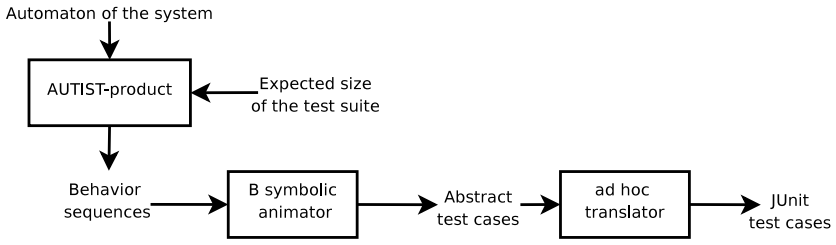


Fig. 1. Process of the experiment

Test suite	# tests	av. length	max. length	comp. time	mutants killed
ChinesePostman	3	91	175	3.8s	24/31 (77%)
ChineseAug2F_10a	9	126	618	25.23s	28/31 (90%)
ChineseAug2F_10b	9	109	547	25.32s	27/31 (87%)
ChineseAug2F_10c	9	127	632	32.94s	27/31 (87%)
ChineseAug2F_10d	9	121	668	24.88s	28/31 (90%)
ChineseAug2F_10e	9	117	618	26.26s	28/31 (90%)
ChineseAug2F_12	13	135	929	2min18s	31/31 (100%)
ChineseAug2F_15	15	130	973	5min32s	31/31 (100%)
ChineseAug2F_18	19	133	1388	18min38s	31/31 (100%)
ChineseAug2F_20	21	125	1264	23min28s	31/31 (100%)
ChineseAug2F_25	25	132	1675	1h1min28s	31/31 (100%)
ChinesePostman4F	5	61	180	3.8s	24/31 (77%)
ChineseAug4F_10	9	74	390	8.8s	25/31 (80%)
ChineseAug4F_12	13	73	517	15.5s	31/31 (100%)
ChineseAug4F_15	13	68	495	13.4s	31/31 (100%)
ChineseAug4F_18	17	71	573	31.8s	31/31 (100%)
ChineseAug4F_20	21	72	771	1min24s	31/31 (100%)
ChineseAug4F_25	25	74	869	2min12s	31/31 (100%)
ChinesePostman5F	6	52	121	4.1s	24/31 (77%)
ChineseAug5F_10	11	48	124	6.5s	25/31 (80%)
ChineseAug5F_12	11	48	124	6.6s	25/31 (80%)
ChineseAug5F_15	16	55	206	12.9s	31/31 (100%)
ChineseAug5F_18	16	53	200	11.6s	31/31 (100%)
ChineseAug5F_20	21	55	220	31.7s	31/31 (100%)
ChineseAug5F_25	26	58	403	1min10s	31/31 (100%)
LTG	59	2.66	8	2min 34s	19/31 (61%)

Table 3. Results on the Automata Augmentation

observing the outputs of the different commands, that are supposed to return a status code indicating if the command succeeded or failed, and why. A test fails if the codes do not correspond at a given step of the execution of the test.

The results of this experiment is given in Tab. 3. This table displays the following informations: the list of test suites that we have generated (col. *test suite*), the number of tests for each suite (col. *# tests*), the average length of the tests (col. *av. length*), the maximal length of a test (col. *max. length*), the computation time (col. *time*), and the number and percentage of mutants killed.

The *ChinesePostman* lines shows the results obtained by applying the bug-fixed version of the Chinese Postman algorithm on the automaton of Demoney (with initially 2 final states). The *ChineseAug2F_10L* (with $L \in a..e$) lines show the result on 5 runs of the automaton augmentation algorithm in order to reach 10 tests when applying the Chinese Postman algorithm. The goal of this set of test sequences is to see if the randomly generated automaton for the product may have an influence on the resulting tests. Globally, we notice that their average

and maximal length may vary but this has no serious influence on the efficiency of the test cases. Generally, the *ChineseAugNF_S* lines represent the application of the automaton augmentation on the Demoney automaton having *N* final states, and aiming at producing *S* tests. Finally the last line, *LTG* compares the results obtained by a commercial fully-automated test generation tool, Leirios Test Generator from the Smartesting company[4]. This tool generates tests from a B model and using model structural coverage criteria.

These results shows that the average length of the tests suites is relatively similar for a given number of final states. The computation time decreases with the increase of the number of final states. Intuitively, this is due to the fact that adding final states add backward transitions simplify the search for the optimal path in the Chinese Postman algorithm. However, the resulting test cases are longer, and more efficient as the number of final states decreases, even with a small number of tests. In general, the maximal length of the tests sequences increases with the number of tests. We can notice that the efficiency of the test suite is related to the length of the test cases. A deeper study of the results (not represented here) shows that the longest test cases find the largest number of errors. In addition, we have to mention that, before starting the mutation experiments, we found two non-conformances on the original version our B model and implementation, whereas these were supposed to conform to each other, according to extensive test campaigns that had been performed before.

The conclusion on this case study let us think that the technique of augmenting the test suite can be efficient in practice, for a given test generation algorithm.

5 Conclusion and Future Works

We have presented in this paper the use of uniform random generation of automata, as a basis for test generation-related works. First, we have illustrated how random testing can be employed to detect bugs, even on a well-known and widely-spread algorithm. Second, we have provided some experimental data and statistics on several test generation algorithms based on automata. Third and finally, we have explored an original combination of random and model based testing, through a technique that makes it possible to augment the size of a test suite.

For the future, we plan to experiment on large-scale examples. In this paper, we have limited the statistical study to 2 test generation algorithms. We think it would be fruitful to improve the statistical study by analysing other graph-based test generation algorithms. We also intend to compare our combination approach

[4] `http://www.smartesting.com`, formerly Leirios Technologies

with other automated testing techniques. Finally, another prospect would be to extend our approach to take properties, expressed as labeled transitions (equivalent to logical formulae, regular expressions, etc.), into account.

References

1. J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. F. Bassino, J. David, and C. Nicaud. REGAL: a library to randomly and exhaustively generate automata. In Jan Holub and Jan Žďárek, editors, *12th (CIAA'07)*, volume 4783, pages 303–305, Prague, Czech Republic, July 2007.
3. F. Bassino and C. Nicaud. Enumeration and random generation of accessible automata. *Theorerical Computer Science.*, 381(1-3):86–104, 2007.
4. B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.
5. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157, August 2004.
6. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *International Symposium of Formal Methods Europe (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 542–547, Newcastle, UK, July 18-22 2005. Springer.
7. Joe W. Duran and Simeon Ntafos. A report on random testing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.
8. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.
9. Alex Groce and Rajeev Joshi. Random testing and model checking: building a common framework for nondeterministic exploration. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 22–28, New York, NY, USA, 2008. ACM.
10. D. Hamlet and R. Taylor. Partition testing does not inspire confidence (program testing). *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990.
11. Richard Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
12. C. Jard and T. Jéron. Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.
13. David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, pages 1090–1123, 1996.
14. R. Marlet and D. Le Metayer. Security properties and java card specificities to be studied in the secsafe project, 2001.
15. A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for Generating Specification-Based Tests. In *5th International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–, Las Vegas, NV, USA, Oct 1999. IEEE Computer Society.
16. Catherine Oriat. Jartege: A tool for random generation of unit tests for java classes. In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2005.

17. Harold W. Thimbleby. The directed chinese postman problem. *Software Practice and Experience*, 33(11):1081–1096, 2003.

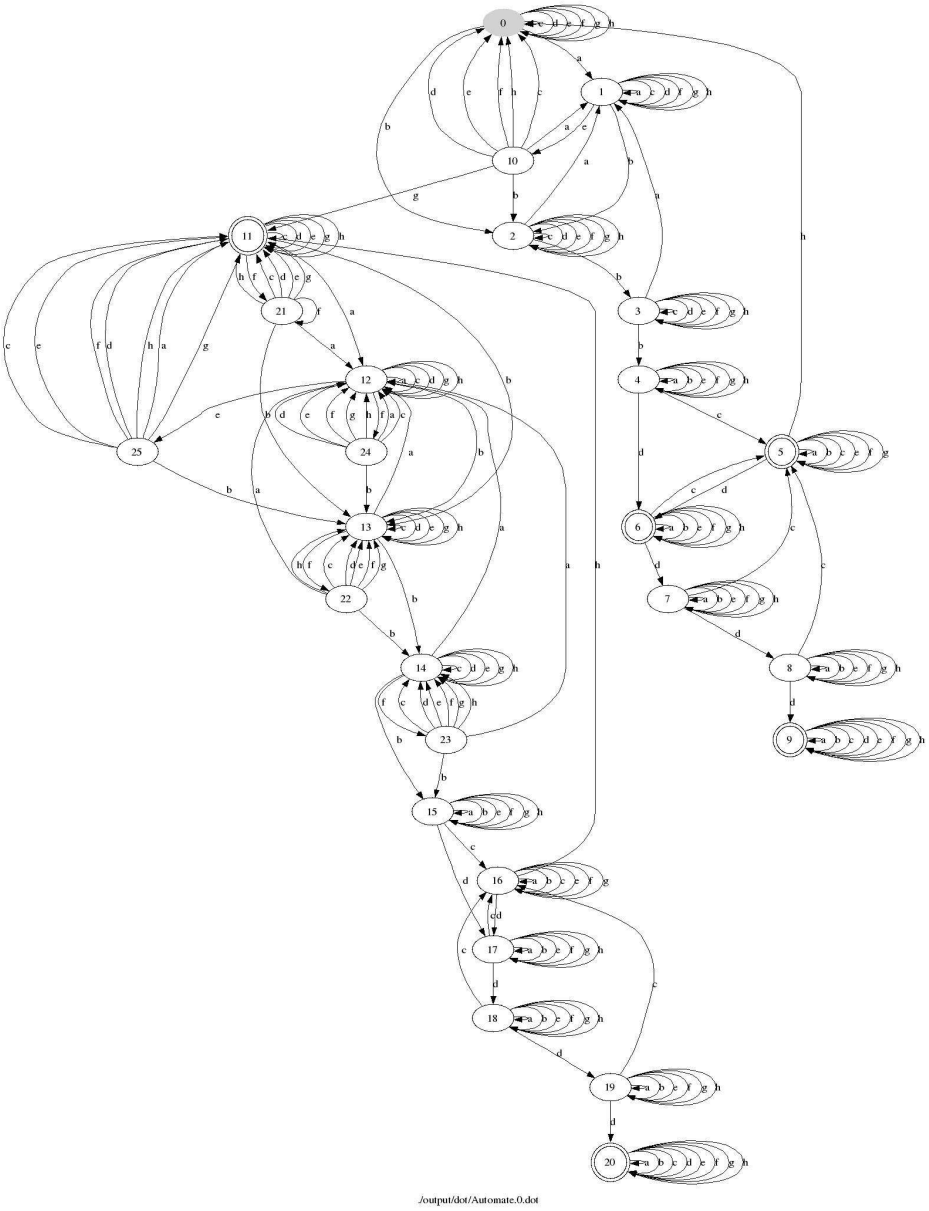
A Automaton to find the Bug in the Chinese Postman

A 10 states automaton making the codes of [17] fail can be build with the following Java instructions that are easily readable: the instruction `G.addArc("x",p,q,1);` add in the grapg `G` a transition from p to q labelled by x.

```
OpenCPP G = new OpenCPP(10);
G.addArc("a",0,1,1);
G.addArc("b",0,0,1);
G.addArc("c",0,5,1);
G.addArc("d",0,4,1);
G.addArc("e",0,5,1);
G.addArc("a",1,2,1);
G.addArc("b",1,7,1);
G.addArc("c",1,8,1);
G.addArc("d",1,5,1);
G.addArc("e",1,0,1);
G.addArc("a",2,3,1);
G.addArc("b",2,5,1);
G.addArc("c",2,1,1);
G.addArc("d",2,2,1);
G.addArc("e",2,1,1);
G.addArc("a",3,3,1);
G.addArc("b",3,4,1);
G.addArc("c",3,7,1);
G.addArc("d",3,4,1);
G.addArc("e",3,5,1);
G.addArc("a",4,0,1);
G.addArc("b",4,4,1);
G.addArc("c",4,5,1);
G.addArc("d",4,4,1);
G.addArc("e",4,9,1);
G.addArc("a",5,1,1);
G.addArc("b",5,1,1);
G.addArc("c",5,6,1);
G.addArc("d",5,3,1);
G.addArc("e",5,2,1);
G.addArc("a",6,6,1);
G.addArc("b",6,5,1);
G.addArc("c",6,5,1);
```

```
G.addArc("d",6,5,1);
G.addArc("e",6,7,1);
G.addArc("a",7,3,1);
G.addArc("b",7,5,1);
G.addArc("c",7,1,1);
G.addArc("d",7,8,1);
G.addArc("e",7,7,1);
G.addArc("a",8,8,1);
G.addArc("b",8,9,1);
G.addArc("c",8,5,1);
G.addArc("d",8,8,1);
G.addArc("e",8,1,1);
G.addArc("a",9,9,1);
G.addArc("b",9,6,1);
G.addArc("c",9,5,1);
G.addArc("d",9,1,1);
G.addArc("e",9,3,1);
G.addArc("Backward closure",0,0,1);
G.addArc("Backward closure",1,0,1);
G.addArc("Backward closure",5,0,1);
```

B Automaton of the Demoney Case Study



a VERIFY_PIN(holder,_) → OK
b VERIFY_PIN(bank,_) → OK
c VERIFY_PIN(holder,_) → KO
d VERIFY_PIN(bank,_) → KO
e INITIALIZE_TRANSACTION(credit,_) → OK
f INITIALIZE_TRANSACTION(debit,_) → OK
g COMMIT_TRANSACTION()
h PIN_CHANGE_UNBLOCK(_)