RAPPORT DE RECHERCHE

L R I

A GENERIC TOOL FOR STATISTICAL TESTING

DENISE A / GAUDEL M C / GOURAUD S D

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

11/2003

Rapport de Recherche N° 1378

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 650
91405 ORSAY Cedex (France)

A Generic Tool for Statistical Testing

A. Denise, M.-C. Gaudel and S.-D. Gouraud

email= {denise,mcg,gouraud}@lri.fr

L.R.I., Université Paris-Sud, bât. 490, 91405 Orsay Cedex, France.

November 24, 2003

Abstract

In the area of software testing, numerous methods have been proposed and used for selecting finite test sets and automating this selection. Among these methods some are deterministic and some are statistical. In the recent years, the general problem of studying and simulating random processes has particularly benefitted from progresses in the area of random generation of combinatorial structures. We explore the idea of using such concepts and tools for statistical software testing. We describe a generic method for using these tools as soon as there is a graphical description of the behaviour of the system under test. Uniform generation is used for drawing paths from the set of execution paths or traces of the system under test, or, more efficiently, among some subsets satisfying some coverage conditions. The paper presents a general method and in the last section, some experimental results on applying it to structural statistical testing.

1 Introduction

In the area of software testing, numerous methods have been proposed and used for selecting finite test sets and automating this selection. Among these methods some are deterministic and some are probabilistic. Depending of the authors, methods of this last class are called statistical testing or random testing.

In the recent years, the general problem of studying and simulating random processes has particularly benefitted from progresses in the area of random generation of combinatorial structures. The seminal works of Wilf and Nijenhuis in the late 70's [28, 23] have

led to efficient algorithms for generating uniformly at random a variety of combinatorial structures. In 1994, Flajolet, Zimmermann and Van Cutsem [11] have widely generalized and systematized the approach. Briefly, their approach is based on a non-ambiguous recursive decomposition of the combinatorial structures to be generated. Their work constitutes the basis of powerful tools for uniform random generation of complex entities, as graphs, trees, words, paths... In the present paper, we explore the idea of using such concepts and tools for random software testing.

Actually, there are several ways to use uniform generation in the area of testing.

A natural idea is to uniformly draw data from the input domain. This approach, generally called random testing, was studied a long time ago for numerical data [9, 10], and turned out to have an uneven detection power, becoming unsatisfactory when applied to realistic complex programs [3, 26].

In this paper, we follow another idea: We describe a generic method for using these tools as soon as there is a graphical description of the behaviour of the system under test. It may be the control graph of the program, or some specification, either directly graphical (Statecharts, Petri nets) or indirectly via some semantics in terms of transition systems, automata, state machines, Kripke structures, etc. Such behavioural graphs can be described as combinatorial structures. Therefore, uniform generation can be used for drawing paths from the set of execution paths or traces of the system under test, or, more efficiently, among some subsets satisfying some coverage conditions.

The paper is organised as follows: Section 2 presents our general method to count and uniformly draw paths in a graph, based on some translation into a combinatorial structure specification; Section 3 recalls some basic notions of statistical testing, coverage criteria and test quality; In Section 4 we present a path generation scheme guided by test quality, and in Section 5 we discuss the issue of deriving test inputs, once a set of paths has been generated. In the two last sections, we recall some experimental results on applying our method to structural statistical testing and we sketch some perspectives.

2 Combinatorial preliminaries

We present here some combinatorial concepts and methods which will be used in the sequel of the paper. Let us consider a connected directed graph G where vertices, as well as edges, are labelled in such a way that any two distinct vertices (resp edges) have

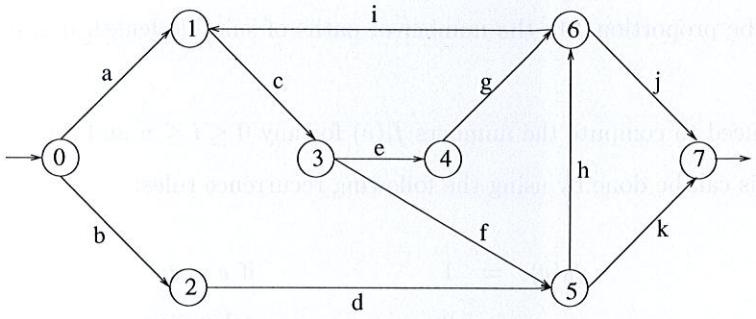

Figure 1 : *A graph with starting and ending vertices*

distinct labels. Furthermore, there exist two vertices v_s (starting vertex) and v_e (ending vertex) such that, for any vertex v, there exists a path from v_s to v and a path from v to v_e in G. Figure 1 presents such a graph, where vertices are labelled with numbers from 0 to 7 and edges are labelled with letters from 'a' to 'k'; vertices 0 and 7 are the starting and ending vertices respectively. If n is a positive integer, $\mathcal{P}_n$ (resp. $\mathcal{P}_{\leq n}$) denotes the set of paths of length n (resp. whose length is $\leq n$) in G from v_s to v_e, and $\mathcal{P}_{\leq \infty}$ denotes the whole (possibly infinite) set of paths from v_s to v_e.

2.1 Uniform random generation of paths in a graph

Our aim is, given an integer n, to generate uniformly at random (u.a.r.) one or several paths of length $\leq n$ from v_s to v_e. Uniformly means that all paths in $\mathcal{P}_{\leq n}$ have the same probability to be generated. At first, let us focus on a slightly different problem: the generation of paths of length n exactly. We will see further that a small change in the graph allows to generate paths of length $\leq n$.

The principle of the generation process is simple: Starting from vertex v_s, draw a path step by step. At each step, the process consists in choosing a successor of the current vertex and going to it. The problem is to proceed in such a way that only (and all) paths of length n can be generated, and that they are equiprobably distributed. This is done by choosing successors with suitable probabilities. Given any vertex v, let $f_m(v)$ denote the number of paths of length m which connect v to the end vertex v_e. Suppose that, at any step of the generation, we are on vertex v which has k successors denoted $v_1, v_2, \ldots, v_k$. In addition, suppose that $m > 0$ edges remain to be crossed in order to get a path of length n. Then the condition for uniformity is that the probability of choosing vertex v_i $(1 \leq i \leq k)$ equals $f_{m-1}(v_i)/f_m(v)$. In other words, the probability to go to any successor

of v must be proportional to the number of paths of suitable length from this successor to v_e.

So we need to compute the numbers $f_i(v)$ for any $0 \leq i \leq n$ and any vertex v of the graph. This can be done by using the following recurrence rules:

$$\begin{array}{lcll} f_0(v) & = & 1 & \text{if } v = v_e \\ & = & 0 & \text{otherwise} \\ f_i(v) & = & \sum_{v \to v'} f_{i-1}(v') & \text{for } i > 0 \end{array}$$

where $v \to v'$ means that there exists an edge from v to v'. Table 1 presents the recurrence rules which correspond to the graph of Figure 1.

$$\begin{array}{lcll} f_0(0) & = & f_1(0) = f_2(0) = f_3(0) = f_4(0) = f_5(0) = f_6(0) = 0 & \\ f_7(0) & = & 1 & \\ f_0(k) & = & f_1(k-1) + f_2(k-1) & (k > 0) \\ f_1(k) & = & f_3(k-1) & (k > 0) \\ f_2(k) & = & f_5(k-1) & (k > 0) \\ f_3(k) & = & f_4(k-1) + f_5(k-1) & (k > 0) \\ f_4(k) & = & f_6(k-1) & (k > 0) \\ f_5(k) & = & f_6(k-1) + f_7(k-1) & (k > 0) \\ f_6(k) & = & f_1(k-1) + f_7(k-1) & (k > 0) \\ f_0(k) & = & 0 & (k > 0) \end{array}$$

Table 1 : *Recurrences for the* $f_i(k)$.

Now the generation scheme is as follows:

- Preprocessing stage: Compute a table of the $f_i(v)$'s for all $0 \leq i \leq n$ and all vertices.
- Generation stage: Draw the path according to the scheme seen above.

Note that the preprocessing stage must be done only once, whatever the number of paths to be generated. Easy computations show that the memory space requirement is $n \times |G|$ integer numbers, where $|G|$ stands for the number of vertices in the graph. The number of arithmetic operations needed for the preprocessing stage, as well as for the generation stage, is linear in n.

Now we address the problem of generating paths of length $\leq n$ instead of exactly n. The only change is the following: Add to the graph a new vertex v'_s which becomes the new start vertex, with an edge from v'_s to v_s and a loop edge from v'_s to itself. Each path of length $n+1$ from v'_s to v_e in this new graph crosses k times the loop edge for some

k such that $0 \leq k \leq n$ and once the one from v'_s to v_s. With this path we obviously associate a path of length $n-k$ in the previous graph. It is straightforward to verify that any path of length $\leq n$ can be generated in such a way, and the generation is uniform.

Note that the above developments are a special case of a general method of generation of combinatorial structures, which has been first adressed by Wilf [28] and then generalized and systematized by Flajolet, Zimmermann and Van Cutsem [11]. More precisely, the problem of generating paths of a given length in G is equivalent to the one of uniform random generation of words of so-called *regular languages*, which has first been discussed in [15]. Indeed, a regular language is defined by a particular labelled graph called *finite state automaton*, and any word of the language corresponds to a path in the automaton. We show in Table 2 the set of words which correspond to the paths of length ≤ 10 of the graph of Figure 1.

length	words
3	bdk
4	acfk, bdkj
5	acegj, acfhj
7	bdhicfk
8	acegicfk, acfhicfk, bdhicegj, bdhicfhj
9	acegicegj, acegicfhj, acfhicegj, acfhicfhj

Table 2 : *The 14 paths of length* ≤ 10 *from vertex* $v_s = 0$ *to vertex* $v_e = 7$.

In our implementation, the generation of paths is programmed in MuPAD, using the CS package. MuPAD [24] is a formal and algebraic calculus tool, developed at the University of Paderborn. CS [6, 7], is a package devoted to counting and randomly generating combinatorial structures, based on the general notion of "decomposable structures" defined in [11]. CS is now part of the MuPAD-Combinat package[16] which is freely available at the following address: `http://mupad-combinat.sourceforge.net/`.

2.2 Constraints on paths and graphs transformations

As we will see in Section 4, our method of statistical testing involves counting and random generation of paths subject to additional constraints. In this subsection, we show how to change the graph in order to take into account such constraints.

Let us focus first on a rather simple constraint: we aim to construct, given a labelled

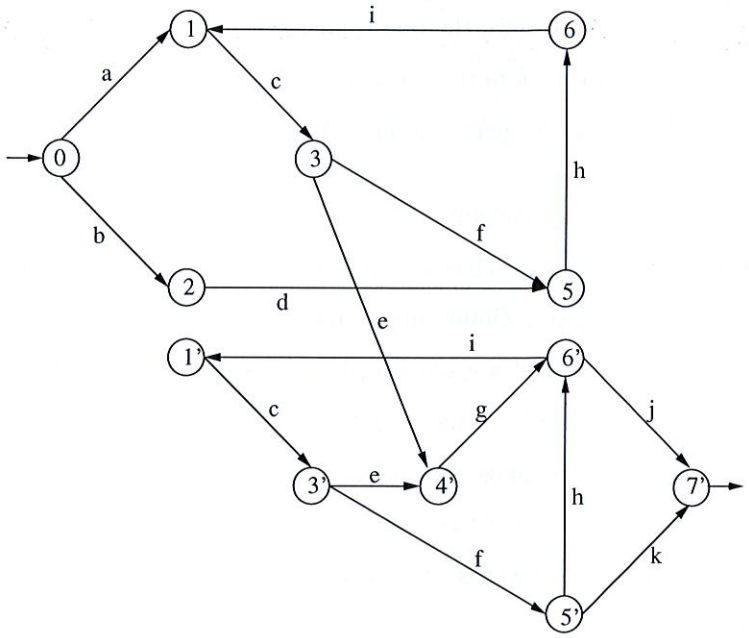

Figure 2 : *Graph which contains only the paths of the graph of Figure 2 which cross edge labelled 'e'.*

connected graph G and an edge label ℓ, a graph H whose set of paths is equal to the set of paths of G which cross edge labelled ℓ. This can be done by using the following procedure:

1. Create a copy G' of graph G, in which the edges are labelled exactly as the edges of G, and in which any vertice label v in G becomes v' in G'.
2. Suppose that the edge labelled ℓ joins vertex u to vertex v in G. Then delete this edge and replace it with a new edge labelled ℓ between vertex u (in G) and vertex v' (in G').
3. set v'_e as the ending vertex, instead of v_e (but v_s remains the start vertex.)
4. Delete all the vertices (and their adjacent edges) to which no path from v_s exists.
5. Delete all the vertices (and their adjacent edges) from which no path to v'_e exists.

This concludes the construction of H. Figure 2 shows the result of the procedure, given the graph of Figure 1 and the edge labelled 'e'. Note that steps 4 and 5 are not mandatory: they are used only to "clean" the final graph by deleting useless elements.

Like in previous subsection, this process can be stated in terms of operations on regular languages: the graph H may be seen as a finite automaton of the regular language which is the intersection of the language of G and the (regular) language of words which contain

at least once the letter ℓ. This approach can be generalized in order to perform more complex transformations of graph G as for example, constructing a graph which contains the paths which cross exactly k times a given edge, or which cross two or more given edges, or which take k times a given cycle in the graph. Roughly, it suffices to be able to express the desired constraint in terms of a regular language, i.e. to design a regular expression or a finite automaton which recognizes the whole set of words which satisfy the constraint. Then a standard algorithm for intersecting regular languages (see e.g. [17]) gives H. In our special case, this general method consists exactly in the procedure described above. Very similar procedures apply if we are given vertices instead of edges.

3 Coverage criteria and test quality

The notion of test quality for statistical testing methods has been defined first by Thévenod-Fosse [25]. We slightly reformulate it for our context, but the notion remains the same.

Let D be some description of a system under test. D may be a specification or a program, depending on the kind of test we are interested in (functional or structural). We assume that D is based on a graph, (or, more generally, on some kind of combinatorial structure.) On the basis of this graph, it is possible to define coverage criteria: all-vertices, all-edges, all-paths-of a certain-kind, etc. More precisely, a coverage criterion C characterises for a given description D a set of elements $E_C(D)$ of the underlying graph (noted E in the sequel when C and D are obvious). In the case of deterministic testing, the criterion is satisfied if every element of the set is exercised by at least one test.

In the case of statistical testing, the satisfaction of a coverage criteria C by a testing method for a description D is characterised by the minimal probability $q_{C,N}(D)$ of covering any element of $E_C(D)$ when drawing N tests. In [25], $q_{C,N}(D)$ is called the test quality of the method with respect to C.

The test quality $q_{C,N}(D)$ can be easily stated if $q_{C,1}(D)$ is known. Indeed, one gets $q_{C,N}(D) = 1 - (1 - q_{C,1}(D))^N$, since when drawing N tests, the probability of reaching an element is one minus the probability of not reaching it N times.

Let us come back to the example of Section 2, where the set of all paths of Figure 1 has been expressed as a specification of some combinatorial structure, and the CS system is used for uniformly drawing among paths of length $\leq n$. Let us note $\mathcal{P}_{\leq n}$ the set of such paths as in Section 2. Considering the coverage criterion "all paths of length $\leq n$", noted

below $AP \leq n$, we get the following test quality:

$$q_{AP\leq n,N} = 1 - (1 - \frac{1}{|\mathcal{P}_{\leq n}|})^N$$

In the example, choosing $n = 10$ allows the coverage of all elementary paths. Since there are 14 paths of length less or equal to 10 (see Table 2) we have:

$$q_{AP\leq 10,N} = 1 - (1 - \frac{1}{|14|})^N$$

Table 3 gives the number of tests required for four values of test quality, for the criterion "all paths of length ≤ 10".

q	0.9	0.99	0.999	0.9999
N	32	63	94	125

Table 3 : *Number of tests N required for a test quality q*

The assessment of test quality is more complicated in general. Let us consider more practicable coverage criteria, such as "all-vertices" or "all-edges", and some given statistical testing method. The elements to be covered generally have different probabilities to be reached by a test. Some of them are covered by all the tests, for instance the initial and terminal vertices v_s and v_e mentioned in Section 2. Some of them may have a very weak probability, due to the structure of the behavioural graph or to some specificity of the testing method. For instance, in our example edges b and d appear in 5 paths of length ≤ 10 only. Edges a and c appear in 9 such paths. It means that drawing uniformly from $\mathcal{P}_{\leq 10}$ leads to a probability of $\frac{5}{14}$ to reach edge b, and $\frac{9}{14}$ to reach edge a.

Let $E_C(D) = \{e_1, e_2, ..., e_m\}$ and for any $i \in (1..m), p_i$ the probability for the element e_i to be exercised during the execution of a test generated by the considered statistical testing method. Then

$$q_{C,N}(D) = 1 - (1 - p_{min})^N, \text{ where } p_{min} = min\{p_i | i \in (1..m)\} \quad (1)$$

Consequently, the number N of tests required to reach a given quality $q_C(D)$ is

$$N \geq \frac{log(1 - q_C(D))}{log(1 - p_{min})}$$

By definition of the test quality, p_{min} is just $q_{C,1}(D)$. Thus, from the formula above one immediately deduces that for any given D, for any given N, maximising the quality of a statistical testing method with respect to a coverage criteria C reduces to maximising $q_{C,1}(D)$, i. e. p_{min}.

4 Generation of paths guided by the QoT

4.1 General scheme.

Now we describe a methodology in order to maximize the quality of test for any given coverage criterion C. As a preliminary remark, note that the set of elements $E_C(D)$ must be finite, otherwise the quality of test would be zero. This implies, in particular, that the coverage criterion "all paths" is irrelevant as soon as there is a cycle in the description, like in our example (figure 1). Thus, this criterion has to be bounded by additional conditions, for example "all paths of length $\leq n$", "all paths of length between given n_1 and n_2", or "all paths which take at most m times each cycle in the graph". For the sake of simplicity, we consider in the following that paths are generated within $\mathcal{P}_{\leq n}$, the set of paths of length $\leq n$ that go from v_s to v_e.

Now, we are given a suitable coverage criterion C, and we aim to optimize the quality of test. We consider two cases, according to the nature of the elements of $E_C(D)$.

If $E_C(D)$ denotes a set of paths in the graph, we immediately state that the quality of test is optimal if the paths of $E_C(D)$ are generated uniformly, i.e. any path has the same probability $1/|E_C(D)|$ to be generated. Indeed, if the probability of one or several paths was greater than $1/|E_C(D)|$, then there would exist at least one path whith probability less than $1/|E_C(D)|$, therefore the quality of test would be lower. We saw in Section 2.1 how to generate uniformly random paths of given length n in a graph, and how to modify the graph in order to fit with the criterion "all paths of length $\leq n$". The method easily applies to other criteria that involve paths, as those given above, by ways similar to the ones seen in Section 2.2.

Now, we consider the case where the elements of $E_C(D)$ are not paths, but are constitutive elements of the graph as, for example, vertices, edges, or cycles. Here, we have to maximize the minimal probability of taking each element when drawing a path. Clearly, uniform generation of paths does not ensure optimal quality of test in this case. So we

generate a path in two steps:

1. pick at random one element e of $E_C(D)$, according to a suitable probability distribution (which will be discussed in Section 4.2);
2. generate uniformly at random one path of length $\leq n$ that goes through e.

Algorithms for achieving the second step are detailed in Section 2. The next subsection deals with the first step.

4.2 Probability distribution for an optimal quality of test.

The problem consists in choosing the suitable probability distribution over $E_C(D)$ in order to maximize the quality of test. Given $E_C(D) = \{e_1, e_2, \ldots, e_m\}$, with $m > 0$, we denote, for any i and j in $(1..m)$,

- α_i the number of paths of $\mathcal{P}_{\leq n}$ which takes element e_i;
- $\alpha_{i,j}$ the number of paths which take both elements e_i and e_j; (note that $\alpha_{i,i} = \alpha_i$ and $\alpha_{i,j} = \alpha_{j,i}$);
- π_i the probability of choosing element e_i during step 1 of the above process.

Now, let us compute, for any i, the probability p_i for the element e_i to be reached by a path:

$$p_i = \pi_i + \sum_{j \in (1..m)-\{i\}} \pi_j \frac{\alpha_{i,j}}{\alpha_j},$$

Indeed, the probability of choosing element e_i in step 1 is π_i; and the probability of reaching e_i by drawing a random path which goes through another element e_j is $\frac{\alpha_{i,j}}{\alpha_j}$. The above equation simplifies in

$$p_i = \sum_{j=1}^{m} \pi_j \frac{\alpha_{i,j}}{\alpha_j} \tag{2}$$

since $\alpha_{i,i} = \alpha_i$. Note that coefficients α_j and $\alpha_{i,j}$ are easily computed by ways given in Section 2.

Now we have to determine probabilities $\{\pi_1, \pi_2 \ldots, \pi_m\}$ with $\sum \pi_i = 1$, which maximize $p_{min} = \min\{p_i, i \in [1..m]\}$. This can be stated as a linear programming problem:

$$\text{Maximize } p_{min} \text{ under the constraints: } \begin{cases} \forall i \leq m, \quad p_{min} \leq p_i \, ; \\ \pi_1 + \pi_2 + \cdots + \pi_m = 1 \, ; \end{cases}$$

where the p_i's are computed as in Equation (2). Standard methods lead to a solution in time polynomial according to m.

Let us illustrate this with our example. Given the coverage criterion "all the edges" and given $n = 10$, Table 4 presents the coefficients $\alpha_{x,y}$, where x and y denote letters from 'a' to 'k'. For example, the value '9' in row 'f' and column 'c' means that $\alpha_{\mathrm{c,f}} = 9$, i.e. there are exactly 9 paths of length lower or equal to 10 from v_s to v_e which cross both edges c and f in the graph of Figure 1.

	a	b	c	d	e	f	g	h	i	j	k
a	9	0	9	0	5	7	5	5	6	6	3
b	0	5	3	5	1	2	1	4	3	3	2
c	9	3	12	3	6	9	6	8	9	8	4
d	0	5	3	5	1	2	1	4	3	3	2
e	5	1	6	1	6	3	6	3	5	5	1
f	7	2	9	2	3	9	3	7	7	5	4
g	5	1	6	1	6	3	6	3	5	5	1
h	5	4	8	4	3	7	3	9	7	7	2
i	6	3	9	3	5	7	5	7	9	6	3
j	6	3	8	3	5	5	5	7	6	9	0
k	3	2	4	2	1	4	1	2	3	0	5

Table 4 : *Table of the α_{ij}.*

The corresponding linear program is shown in Table 5. Each line, but the last one, is an inequation which corresponds to a row in Table 4. The first term of the inequation is p_{min}, the value to be maximized. The second term is one of the p_i's, computed according to Formula 2. The first one, for example, means that p_{min} must be lower or equal to p_{a}, the probability of reaching edge 'a' with a random path. By maximizing p_{min}, one maximizes the lowest p_i, so that the quality of test is optimal. The last line ensures that the probabilities π_i that we are searching for sum to 1.

Solving this linear program leads to $\pi_{\mathrm{a}} = \pi_{\mathrm{c}} = \pi_{\mathrm{d}} = \pi_{\mathrm{f}} = \pi_{\mathrm{g}} = \pi_{\mathrm{h}} = \pi_{\mathrm{i}} = \pi_{\mathrm{j}} = 0$, while $\pi_{\mathrm{b}} = \pi_{\mathrm{k}} = \frac{5}{16}$ and $\pi_{\mathrm{e}} = \frac{6}{16}$, which is the best choice. This gives $p = \frac{1}{2}$, therefore the optimal quality of test equals $1 - \frac{1}{2^N}$, according to Formula 1.

5 From paths to input data

So far we have presented a generic method for generating execution paths in a way that maximizes test quality. This method relies on existing algorithms and tools and can be

$$
\begin{array}{lccccccccccc}
p_{min} \leq & \pi_a & & +\frac{3}{4}\pi_c & & +\frac{5}{6}\pi_e & +\frac{7}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{5}{9}\pi_h & +\frac{2}{3}\pi_i & +\frac{2}{3}\pi_j & +\frac{3}{5}\pi_k \\
p_{min} \leq & & \pi_b & +\frac{1}{4}\pi_c & +\pi_d & +\frac{1}{6}\pi_e & +\frac{2}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{4}{9}\pi_h & +\frac{1}{3}\pi_i & +\frac{1}{3}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \pi_a & +\frac{3}{5}\pi_b & +\pi_c & +\frac{3}{5}\pi_d & +\pi_e & +\pi_f & +\pi_g & +\frac{8}{9}\pi_h & +\pi_i & +\frac{8}{9}\pi_j & +\frac{4}{5}\pi_k \\
p_{min} \leq & & \pi_b & +\frac{1}{4}\pi_c & +\pi_d & +\frac{1}{6}\pi_e & +\frac{2}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{4}{9}\pi_h & +\frac{1}{3}\pi_i & +\frac{1}{3}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{1}{5}\pi_b & +\frac{1}{2}\pi_c & +\frac{1}{5}\pi_d & +\pi_e & +\frac{1}{3}\pi_f & +\pi_g & +\frac{1}{3}\pi_h & +\frac{5}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{1}{5}\pi_k \\
p_{min} \leq & \frac{7}{9}\pi_a & +\frac{2}{5}\pi_b & +\frac{3}{4}\pi_c & +\frac{2}{5}\pi_d & +\frac{1}{2}\pi_e & +\pi_f & +\frac{1}{2}\pi_g & +\frac{7}{9}\pi_h & +\frac{7}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{4}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{1}{5}\pi_b & +\frac{1}{2}\pi_c & +\frac{1}{5}\pi_d & +\pi_e & +\frac{1}{3}\pi_f & +\pi_g & +\frac{1}{3}\pi_h & +\frac{5}{9}\pi_i & +\frac{5}{9}\pi_j & +\frac{1}{5}\pi_k \\
p_{min} \leq & \frac{5}{9}\pi_a & +\frac{4}{5}\pi_b & +\frac{2}{3}\pi_c & +\frac{4}{5}\pi_d & +\frac{1}{2}\pi_e & +\frac{7}{9}\pi_f & +\frac{1}{2}\pi_g & +\pi_h & +\frac{7}{9}\pi_i & +\frac{7}{9}\pi_j & +\frac{2}{5}\pi_k \\
p_{min} \leq & \frac{2}{3}\pi_a & +\frac{3}{5}\pi_b & +\frac{3}{4}\pi_c & +\frac{3}{5}\pi_d & +\frac{5}{6}\pi_e & +\frac{7}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{7}{9}\pi_h & +\pi_i & +\frac{2}{3}\pi_j & +\frac{3}{5}\pi_k \\
p_{min} \leq & \frac{2}{3}\pi_a & +\frac{3}{5}\pi_b & +\frac{2}{3}\pi_c & +\frac{3}{5}\pi_d & +\frac{5}{6}\pi_e & +\frac{5}{9}\pi_f & +\frac{5}{6}\pi_g & +\frac{7}{9}\pi_h & +\frac{2}{3}\pi_i & +\pi_j & \\
p_{min} \leq & \frac{1}{3}\pi_a & +\frac{2}{5}\pi_b & +\frac{1}{3}\pi_c & +\frac{2}{5}\pi_d & +\frac{1}{6}\pi_e & +\frac{4}{9}\pi_f & +\frac{1}{6}\pi_g & +\frac{2}{9}\pi_h & +\frac{1}{3}\pi_i & & +\pi_k \\
1 = & \pi_a & +\pi_b & +\pi_c & +\pi_d & +\pi_e & +\pi_f & +\pi_g & +\pi_h & +\pi_i & +\pi_j & +\pi_k
\end{array}
$$

Table 5 : *The linear program.*

fully automated. A last step is to generate, for every path, input values that will cause its execution.

5.1 The trivial case of finite models

First let us consider the case where the graphical description corresponds to a finite model of the system under test (Finite State Automata, Finite State Machine, etc) [5]. Every edge of the description is labelled by some symbol of a finite alphabet that represents some input or event. This symbol may be coupled with some other symbol indicating some expected reaction (output, action). Here, the test data for executing a given path is just the sequence of inputs labelling its edges, possibly followed by some additional inputs in order to observe that the system under test is in the expected state [18, 4].

5.2 The general case of infinite models

The problem is more difficult as soon as the model underlying the description is not finite [18]. It is the case of various sorts of Extended Finite State Machines, State-charts, or the control graph of pieces of code, namely any description including non-trivial data types and guards.

An example is partially given in Figure 5.2, using a notation close to UML state charts. A very similar example is completely presented in [19]. In this example, the M variable is of a given type *Message*; every message has a priority. The Q variable is of type *PriorityQueue*. The *get* operation returns the oldest message of the queue with the best

priority. The boxes labelled by $Buffer(Q)$ and $ClientReady(Q)$ denote infinite classes of states, (as many states as possible values for the Q variable). A possible trace (or more exactly class of traces) is:

$$\_/Q.init(); ?M/Q.add(M); ?ready[\neg Q.isEmpty()]/\_; !Q.get()/Q.remove()$$

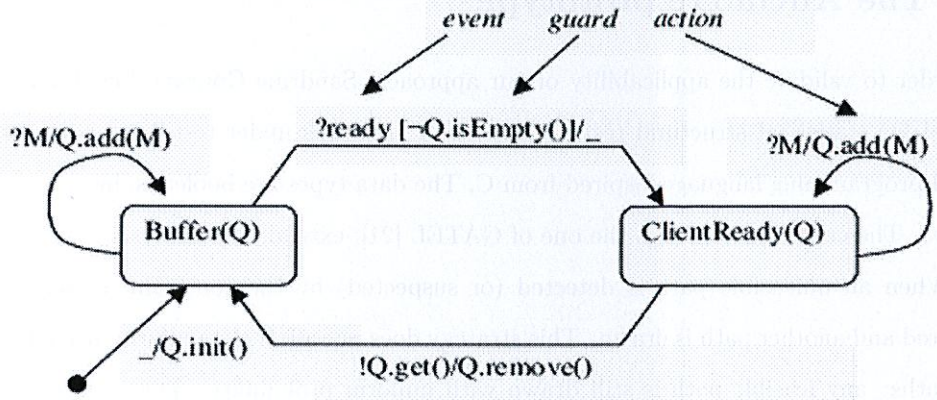

Figure 3 : *A statechart specification of a buffer with priorities*

Given some path or some trace made of conditioned statements or guarded commands, how to find some inputs triggering its execution? It is a classical issue in structural testing, or in functional testing based on specifications with data types. Constructing, via symbolic evaluation techniques, the predicate characterizing the input domain of the path, can solve it: This predicate is the conjunction of the guards (or conditions) encountered on the path, adequately updated in function of the variables assignments (see for instance [14]). Then the problem reduces to a constraint-solving problem. Any data satisfying the above predicate is an input covering the path. At this stage, the tool to be used is highly dependent on the kind of guards and data types allowed in the description: There exist a lot of specialised constraint solvers for various types of variables and constraints, which ensure termination and completeness. However, in full generality the problem is only semi-decidable and a general-purpose solver may not terminate when searching for a solution. Lately, significant advances have been achieved with the introduction of powerful heuristics and randomisation techniques, such as those used in the LOFT and GATEL tools [20, 21] or the BZ-tools in [1]. Other uses of constraint solvers for test generation are reported in [2, 12, 22]. A classical difficulty at this stage is that unfeasible paths may arise. For instance, in the example of Figure 3, all paths beginning

by $\_/Q.init(); ?ready[\neg Q.isEmpty()]/\_; \ldots$ are unfeasible since the *init* method assigns an empty state to the Queue. In the next section we show how we cope with this problem in the prototype that we have developed for structural statistical testing.

6 The AuGuSTe prototype

In order to validate the applicability of our approach, Sandrine Gouraud has developed a tool for statistical structural testing [13]. The programs under test are written in a small programming language inspired from C. The data types are booleans, integers, and arrays. The constraint solver is the one of GATEL [21], extended to arrays.

When an unfeasible path is detected (or suspected) by the constraint solver, it is rejected and another path is drawn. This strategy does not affect the uniform distribution on paths: any feasible path is still drawn with uniform probability. This ensures that, if the coverage criterion involves paths only (like e.g. "all paths of length $\leq n$"), the quality of test stays optimal. However, in other cases, it may decrease with regard to its theoretical value, depending on the distribution of unfeasible paths in the graph.

Actually, our first experiments with AuGuSTe show that the difference may be significant in presence of big numbers of unfeasible paths.

We are currently investigating methods for improving this experimental quality of test. For example, in some cases a number of infeasible paths can be detected by static analysis of the description of the system. Then the combinatorial specification of the graph can be modified in order to avoid these paths.

This tool has been used for testing the same set of four C programs as in [27], where Pascale Thévenod-Fosse, Héléne Waeselinck and Yves Crouzet presented the first experimental evaluation of the detection power of statistical structural testing. Thanks to them, it was possible to reuse the same sets of mutants and to replay almost the same set of experiments. In [27], the statistical method is different from here, since it is based on the explicit construction of a distribution on the input domain, either analytically, or empirically (when there is some loop). In our case, we draw paths and then use constraint-solving tools to produce inputs. Of course, this induces a distribution on the input domain. As this distribution is highly dependent on the implementation of the constraint solver, it remains implicit. Despite of this fundamental difference, the results of the experiments are quite similar, with the advantage that our new approach is fully automated.

The fourth program, the one named FCT4 in [27], was the most difficult and the most interesting. It contains a huge number of unfeasible paths. The coverage criterion was "all the edges with maximal path length of 234", in number of edges of the control graph (thus much more in number of statements). Consequently, the predicates to be solved were rather long too, since at least one out of two vertices on a path corresponds to a decision point, thus to a condition to be added to the predicate. AuGuSTe was successfully used to automatically produce several test sets for this program (see [13] for details). These first experiments let think that the method scales up well.

7 Conclusion and perspectives

In this paper we have shown how uniform generation of combinatorial structures can be used for statistical testing as soon as some graphical description of the program under test is available. If the description is at the program level (control flow graph), our method applies to structural statistical testing. If the description is at the specification level, it applies to functional statistical testing.

Our approach was sketched in a previous paper[14], where we presented a first example for structural statistical testing. This paper presents the method in its generality, and avoids the heuristic used in [14] for the implementation of the all-statements and all-branches criteria: Indeed, in Section 4, we show how to build a probability distribution on the elements to be covered for any given criteria. This distribution ensures an optimal quality of test when associated with the methods of Section 2.2 for randomly generating paths constrained to traverse these elements.

As mentioned in Section 6, this approach has been validated on realistic examples. It seems that it can provide a basis for a new class of tools in the domain of testing.

Moreover, some interesting perspectives are still open. The CS tool can deal with languages more complex than regular languages (for instance, with cardinality constraints such as *paths with the same number of iterations in loop 1 and loop 2*). Practically, it means that it could be possible to compile behavioural graphs into more elaborated combinatorial structures, taking into account some knowledge on the system under test, or some results of static analysis. This could improve significantly the efficiency of the tools, by eliminating some major sources of infeasible paths.

Another possibility worth to explore is the use of the new approach proposed recently

by Flajolet & al. [8] for random generation of combinatorial structures : It is based on Boltzmann models and could avoid the introduction on a bound on the length of the considered paths.

Acknowledgements

We are undebted to Claire Kenyon for some fruitful discussions on the optimisation part of this work. For our experiments we wish to acknowledge : the testing group in LAAS for providing us the library of mutants, Sylvie Corteel for the combinatorial structures library and Bruno Marre for his help in extending and using the constraint solver. This work was partially funded by the European community (IST Project 1999-11585: DSoS).

References

[1] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacele. BZ-testing-tools: A tool-set for test generation from Z and B using constraint logic programming. In *FATES'02, Formal Approaches to Testing of Software, Workshop of CONCUR'02*, pages 105–120, Brnö, Czech Republic, August 2002.

[2] F. Barray, P. Codognet, D. Diaz, and H. Michel. Code-based test generation for validation of functionnal processor descriptions. In *Proceedings TACAS 2003, Springer Verlag*, pages 569–584. january 2003. LNCS 2619.

[3] P.G. Bishop, D.G. Esp, M. Barnes, P. Humphreys, G. Dahll, and J. Lahti. PODS-A project on diverse software. In *IEEE Transactions on Software Engineering*, volume SE-12, pages 929–941, 1986.

[4] E. Brinskma and J. Tretmans. Testing transition systems: An annotated bibliography. In *LNCS*, volume 2067, pages 187–195, 2001.

[5] T.S. Chow. Testing software design modeled by finite-state machines. In *IEEE Transactions on Software Engineering*, volume SE-4,n° 3, pages 178–187, 1978.

[6] S. Corteel, A. Denise, I. Dutour, F. Sarron, and P. Zimmermann. CS web page. http://dept-info.labri.u-bordeaux.fr/~dutour/CS/.

[7] A. Denise, I. Dutour, and P. Zimmermann. CS: a package for counting and generating combinatorial structures. *mathPAD*, 8(1):22–29, 1998. http://www.mupad.de/mathpad.shtml.

[8] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Random sampling from Boltzmann principes. In P. Widmayer et al, editor, *ICALP 2002*, LNCS 2380, pages 501–513. Springer-Verlag Berlin Heidelberg.

[9] J.W. Duran and S.C. Ntafos. A report on random testing. In *5th IEEE International Conference on Software Engineering*, pages 179–183, San Diego, march 1981.

[10] J.W. Duran and S.C. Ntafos. An evaluation of random testing. In *IEEE Transactions on Software Engineering*, volume SE-10, pages 438–444, july 1984.

[11] Ph. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132:1–35, 1994.

[12] A. Gottlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Proceedins CL2000, Springer Verlag*, pages 399–413. july 2000. LNAI 1891.

[13] S.-D. Gouraud. Génération de test à l'aide d'outils combinatoires : premiers résultats expérimentaux. Technical report, LRI, Université Paris II, Orsay, France, 2003. RR No1364.

[14] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Automated Software Engineering Conference, IEEE*, pages 5–12, 2001.

[15] T. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM. J. Comput*, 12(4):645–655, 1983.

[16] F. Hivert and N.M. Thiéry. *MuPAD-Combinat, an open-source package for research in algebraic combinatories.* Preprint available at http://mupad-combinat.sourceforge.net.

[17] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[18] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, août 1996.

[19] G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *13th IEEE International Symposium on Software Reliability Engineering (ISSRE-2002)*, pages 3–14, Annapolis, 2002.

[20] B. Marre. LOFT, a tool for assisting selecting of test data sets from algebraic specifications. In *LNCS*, number 915. Springer-Verlag, 1995.

[21] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATEL. In *15h IEEE International Conference on Automated Software Engineering*, pages 229–237, 2000.

[22] C. Meudec. Atgen : automatic test data generation using constraint logic programming and symbolic execution. *Software Testing, Verification and Reliability*, 11(2):81–96, june 2001.

[23] A. Nijenhuis and H.S. Wilf. *Combinatorial algorithms.* Academic Press Inc., 1979.

[24] The MuPAD Group (Benno Fuchssteiner et al.). *MuPAD User's Manual - MuPAD Version 1.2.2 Multi Processing Algebra Data Tool.* John Wiley and sons, 1996. http://www.mupad.de/.

[25] P. Thévenod-Fosse. Software validation by means of statistical testing: Retrospect and future direction. In *International Working Conference on Dependable Computing for Critical Applications*, pages 15–22, 1989.

[26] P. Thévenod-Fosse and H. Waeselynck. An investigation of software statistical testing. *The Journal of Software Testing, Verification and Reliability*, 1(2):5–26, july-september 1991.

[27] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: deterministic versus random input generation. *21st IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS'21)*, pages 410–417, 1991.

[28] H.S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1345	FLANDRIN E LI H WEI B	A SUFFICIENT CONDITION FOR PANCYCLABILITY OF GRAPHS	16 PAGES	01/2003
1346	BARTH D BERTHOME P LAFOREST C VIAL S	SOME EULERIAN PARAMETERS ABOUT PERFORMANCES OF A CONVERGENCE ROUTING IN A 2D-MESH NETWORK	30 PAGES	01/2003
1347	FLANDRIN E LI H MARCZYK A WOZNIAK M	A CHVATAL-ERDOS TYPE CONDITION FOR PANCYCLABILITY	12 PAGES	01/2003
1348	AMAR D FLANDRIN E GANCARZEWICZ G WOJDA A P	BIPARTITE GRAPHS WITH EVERY MATCHING IN A CYCLE	26 PAGES	01/2003
1349	FRAIGNIAUD P GAURON P	THE CONTENT-ADDRESSABLE NETWORK D2B	26 PAGES	01/2003
1350	FAIK T SACLE J F	SOME b-CONTINUOUS CLASSES OF GRAPH	14 PAGES	01/2003
1351	FAVARON O HENNING M A	TOTAL DOMINATION IN CLAW-FREE GRAPHS WITH MINIMUM DEGREE TWO	14 PAGES	01/2003
1352	HU Z LI H	WEAK CYCLE PARTITION INVOLVING DEGREE SUM CONDITIONS	14 PAGES	02/2003
1353	JOHNEN C TIXEUIL S	ROUTE PRESERVING STABILIZATION	28 PAGES	03/2003
1354	PETITJEAN E	DESIGNING TIMED TEST CASES FROM REGION GRAPHS	14 PAGES	03/2003
1355	BERTHOME P DIALLO M FERREIRA A	GENERALIZED PARAMETRIC MULTI-TERMINAL FLOW PROBLEM	18 PAGES	03/2003
1356	FAVARON O HENNING M A	PAIRED DOMINATION IN CLAW-FREE CUBIC GRAPHS	16 PAGES	03/2003
1357	JOHNEN C PETIT F TIXEUIL S	AUTO-STABILISATION ET PROTOCOLES RESEAU	26 PAGES	03/2003
1358	FRANOVA M	LA "FOLIE" DE BRUNELLESCHI ET LA CONCEPTION DES SYSTEMES COMPLEXES	26 PAGES	04/2003
1359	HERAULT T LASSAIGNE R MAGNIETTE F PEYRONNET S	APPROXIMATE PROBABILISTIC MODEL CHECKING	18 PAGES	01/2003
1360	HU Z LI H	A NOTE ON ORE CONDITION AND CYCLE STRUCTURE	10 PAGES	04/2003
1361	DELAET S DUCOURTHIAL B TIXEUIL S	SELF-STABILIZATION WITH r-OPERATORS IN UNRELIABLE DIRECTED NETWORKS	24 PAGES	04/2003
1362	YAO J Y	RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION D'UNE HABILITATION A DIRIGER DES RECHERCHES	72 PAGES	07/2003
1363	ROUSSEL N EVANS H HANSEN H	MIRRORSPACE : USING PROXIMITY AS AN INTERFACE TO VIDEO-MEDIATED COMMUNICATION	10 PAGES	07/2003

RAPPORTS INTERNES AU LRI - ANNEE 2003

N°	Nom	Titre	Nbre de pages	Date parution
1364	GOURAUD S D	GENERATION DE TESTS A L'AIDE D'OUTILS COMBINATOIRES : PREMIERS RESULTATS EXPERIMENTAUX	24 PAGES	07/2003
1365	BADIS H AL AGHA K	DISTRIBUTED ALGORITHMS FOR SINGLE AND MULTIPLE-METRIC LINK STATE QoS ROUTING	22 PAGES	07/2003
1366	FILLIATRE J C	WHY : A MULTI-LANGUAGE MULTI-PROVER VERIFICATION TOOL	20 PAGES	09/2003
1367	FILLIATRE J C	A THEORY OF MONADS PARAMETERIZED BY EFFECTS	18 PAGES	09/2003
1368	FILLIATRE J C	HASH CONSING IN AN ML FRAMEWORK	14 PAGES	09/2003
1369	FILLIATRE J C	DESIGN OF A PROOF ASSISTANT : COQ VERSION 7	16 PAGES	09/2003
1370	HERMAN T TIXEUIL S	A DISTRIBUTED TDMA SLOT ASSIGNMENT ALGORITHM FOR WIRELESS SENSOR NETWORKS	32 PAGES	09/2003
1371	RIGAUX P SPYRATOS N	GENERATION AND SYNDICATION OF LEARNING OBJECT METADATA	32 PAGES	10/2003
1372	APPERT C BEAUDOUIN-LAFON M MACKAY W E	CONTEXT MATTERS : EVALUATING INTERACTION TECHNIQUES WITH THE CIS MODEL	14 PAGES	10/2003
1373	BLANCH R GUIARD Y BEAUDOUIN-LAFON M	SEMANTIC POINTING : IMPROVING TARGET ACQUISITION WITH CONTROL-DISPLAY RATIO ADAPTATION	12 PAGES	10/2003
1374	FORGE D KOUIDER M	COVERING OF THE VERTICES OF A GRAPH BY SMALL CYCLES	16 PAGES	10/2003