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
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, $P_n$ (resp. $P_{\leq n}$) denotes the set of paths of length $n$ (resp. whose length is $\leq n$) in $G$ from $v_s$ to $v_e$, and $P_{\leq \infty}$ 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 $P_{\leq n}$ 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:

\[
\begin{align*}
f_0(v) &= 1 & \text{if } v = v_e \\
       &= 0 & \text{otherwise} \\
f_i(v) &= \sum_{v \rightarrow v'} f_{i-1}(v') & \text{for } i > 0
\end{align*}
\]

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

\[
\begin{align*}
f_0(0) &= f_1(0) = f_2(0) = f_3(0) = f_4(0) = f_5(0) = f_6(0) = 0 \\
f_1(0) &= 1 \\
f_0(k) &= f_1(k-1) + f_2(k-1) & (k > 0) \\
f_1(k) &= f_3(k-1) & (k > 0) \\
f_2(k) &= f_5(k-1) & (k > 0) \\
f_3(k) &= f_4(k-1) + f_5(k-1) & (k > 0) \\
f_4(k) &= f_6(k-1) & (k > 0) \\
f_5(k) &= f_6(k-1) + f_7(k-1) & (k > 0) \\
f_6(k) &= f_1(k-1) + f_7(k-1) & (k > 0) \\
f_7(k) &= 0 & (k > 0)
\end{align*}
\]

**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
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 addressed 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 $\leq 10$ of the graph of Figure 1.

<table>
<thead>
<tr>
<th>length</th>
<th>words</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>bdk</td>
</tr>
<tr>
<td>4</td>
<td>acfk, bdkj</td>
</tr>
<tr>
<td>5</td>
<td>acegj, acfhj</td>
</tr>
<tr>
<td>7</td>
<td>bdhicfk</td>
</tr>
<tr>
<td>8</td>
<td>acegicfk, acfhicfk, bdhicegj, bdhichfj</td>
</tr>
<tr>
<td>9</td>
<td>acegicegj, acegicfhj, aehicigj, acfhicfhj</td>
</tr>
</tbody>
</table>

Table 2: The 14 paths of length $\leq 10$ from vertex $v_s = 0$ to vertex $v_c = 7$.

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
connected graph $G$ and an edge label $\ell$, a graph $H$ whose set of paths is equal to the set of paths of $G$ which cross edge labelled $\ell$. 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 vertex label $v$ in $G$ becomes $v'$ in $G'$.

2. Suppose that the edge labelled $\ell$ joins vertex $u$ to vertex $v$ in $G$. Then delete this edge and replace it with a new edge labelled $\ell$ between vertex $u$ (in $G$) and vertex $v'$ (in $G'$).

3. set $v'_e$ as the ending vertex, instead of $v_e$ (but $v_e$ remains the start vertex.)

4. Delete all the vertices (and their adjacent edges) to which no path from $v_e$ 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 \( \ell \). 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-Fusse [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

\[
q_{C,N}(D) = 1 - \left(1 - q_{C,1}(D)\right)^N,
\]

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 \( \leq n \)".
below $AP \leq n$, we get the following test quality:

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

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 $\leq 10".

<table>
<thead>
<tr>
<th>$q$</th>
<th>0.9</th>
<th>0.99</th>
<th>0.999</th>
<th>0.9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>32</td>
<td>63</td>
<td>94</td>
<td>125</td>
</tr>
</tbody>
</table>

**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 $\leq 10$ only. Edges $a$ and $c$ appear in 9 such paths. It means that drawing uniformly from $P_{\leq 10}$ leads to a probability of $\frac{5}{14}$ to reach edge $b$, and $\frac{9}{14}$ to reach edge $a$.

Let $E_D = \{e_1, e_2, ..., e_m\}$ 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

$$q_{G,N}(D) = 1 - (1 - p_{\text{min}})^N,$$

where $p_{\text{min}} = \min\{p_i | i \in (1..m)\}$  \hspace{1cm} (1)

Consequently, the number $N$ of tests required to reach a given quality $q_{G}(D)$ is

$$N \geq \frac{\log(1 - q_{G}(D))}{\log(1 - p_{\text{min}})}$$

8
By definition of the test quality, $p_{\text{min}}$ 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. $p_{\text{min}}$.

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 $P_{\leq n}$, 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 with 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 \( \{1..m\} \),

- \( \alpha_i \): the number of paths of \( P_{\leq n} \) 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 \( \alpha_{i,j} = \alpha_{j,i} \));
- \( \pi_i \): 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 \( \pi_i \); 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

\[
 p_i = \sum_{j=1}^{m} \pi_j \frac{\alpha_{i,j}}{\alpha_j}
\]  \hspace{1cm} (2)

since \( \alpha_{i,i} = \alpha_i \). Note that coefficients \( \alpha_j \) 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

\[
 p_{\text{min}} = \min \{p_i, i \in \{1..m\}\}
\]

This can be stated as a linear programming problem:

Maximize \( p_{\text{min}} \) under the constraints:

\[
\begin{align*}
\forall i \leq m, \quad p_{\text{min}} &\leq p_i ; \\
\pi_1 + \pi_2 + \cdots + \pi_m &\leq 1 ;
\end{align*}
\]
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 \( \alpha_{c,f} = 9 \), i.e. there are exactly 9 paths of length lower or equal to 10 from \( v_a \) to \( v_e \) which cross both edges \( c \) and \( f \) in the graph of Figure 1.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>9</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>f</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>h</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4: Table of the \( \alpha_{ij} \).

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 \( \pi_i \) that we are searching for sum to 1.

Solving this linear program leads to \( \pi_a = \pi_c = \pi_d = \pi_f = \pi_g = \pi_h = \pi_l = \pi_j = 0 \)
while \( \pi_b = \pi_k = \frac{5}{16} \) and \( \pi_e = \frac{6}{16} \), which is the best choice. This gives \( p = \frac{1}{2} \), therefore the optimal quality of test equals \( 1 - \frac{1}{2^m} \), 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
\[ p_{\min} \leq \pi_a + \frac{3}{2}\pi_c + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \frac{3}{4}\pi_h + \pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_b + \frac{1}{4}\pi_c + \pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \frac{3}{4}\pi_g + \frac{3}{4}\pi_h + \frac{1}{2}\pi_i + \frac{3}{4}\pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_b + \frac{1}{2}\pi_c + \pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \frac{3}{4}\pi_g + \frac{3}{4}\pi_h + \frac{1}{3}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ p_{\min} \leq \pi_a + \frac{3}{4}\pi_b + \pi_c + \frac{3}{4}\pi_d + \frac{3}{4}\pi_e + \frac{3}{4}\pi_f + \pi_g + \frac{3}{4}\pi_i + \pi_j + \frac{3}{4}\pi_k \]

\[ 1 = \pi_a + \pi_b + \pi_c + \pi_d + \pi_e + \pi_f + \pi_g + \pi_i + \pi_j + \pi_k \]

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 \textit{Buffer}(Q) and \textit{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:

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig3.png}
\caption{A statechart specification of a buffer with priorities}
\end{figure}

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(); \texttt{ready}[\neg Q.is\texttt{Empty}()]/\_\_; ... 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, Hélène 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 indebted 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


<table>
<thead>
<tr>
<th>N°</th>
<th>Nom</th>
<th>Titre</th>
<th>Nbre de pages</th>
<th>Date parution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1345</td>
<td>FLANDRIN E LI H WEI B</td>
<td>A SUFFICIENT CONDITION FOR PANCYCLABILITY OF GRAPHS</td>
<td>16 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1346</td>
<td>BARTH D BERTHOMME P LAFOREST C VIAL S</td>
<td>SOME EUCLERIAN PARAMETERS ABOUT PERFORMANCES OF A CONVERGENCE ROUTING IN A 2D-MESH NETWORK</td>
<td>30 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1347</td>
<td>FLANDRIN E LI H MARCZYK A WOZNIAK M</td>
<td>A CHVATAL-ERDOS TYPE CONDITION FOR PANCYCLABILITY</td>
<td>12 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1348</td>
<td>AMAR D FLANDRIN E GANCARZEWICZ G WOJDA A P</td>
<td>BIPARTITE GRAPHS WITH EVERY MATCHING IN A CYCLE</td>
<td>26 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1349</td>
<td>FRAIGNIAUD P GAURON P</td>
<td>THE CONTENT-ADDRESSABLE NETWORK D2B</td>
<td>28 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1350</td>
<td>FAIK T SACLE J F</td>
<td>SOME b-CONTINUOUS CLASSES OF GRAPH</td>
<td>14 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1351</td>
<td>FAVARON O HENNING M A</td>
<td>TOTAL DOMINATION IN CLAW-FREE GRAPHS WITH MINIMUM DEGREE TWO</td>
<td>14 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1352</td>
<td>HU Z LI H</td>
<td>WEAK CYCLE PARTITION INVOLVING DEGREE SUM CONDITIONS</td>
<td>14 PAGES</td>
<td>02/2003</td>
</tr>
<tr>
<td>1353</td>
<td>JOHNEN C TIJEUIL S</td>
<td>ROUTE PRESERVING STABILIZATION</td>
<td>28 PAGES</td>
<td>03/2003</td>
</tr>
<tr>
<td>1354</td>
<td>PETITJEAN E</td>
<td>DESIGNING TIMED TEST CASES FROM REGION GRAPHS</td>
<td>14 PAGES</td>
<td>03/2003</td>
</tr>
<tr>
<td>1355</td>
<td>BERTHOMME P DIALLO M FERREIRA A</td>
<td>GENERALIZED PARAMETRIC MULTI-TERMINAL FLOW PROBLEM</td>
<td>18 PAGES</td>
<td>03/2003</td>
</tr>
<tr>
<td>1356</td>
<td>FAVARON O HENNING M A</td>
<td>PAIRED DOMINATION IN CLAW-FREE CUBIC GRAPHS</td>
<td>16 PAGES</td>
<td>03/2003</td>
</tr>
<tr>
<td>1357</td>
<td>JOHNEN C PETIT F TIJEUIL S</td>
<td>AUTO-STABILISATION ET PROTOCOLES RESEAU</td>
<td>26 PAGES</td>
<td>03/2003</td>
</tr>
<tr>
<td>1358</td>
<td>FRANOVA M</td>
<td>LA &quot;FOLIE&quot; DE BRUNETLESCHI ET LA CONCEPTION DES SYSTEMES COMPLEXES</td>
<td>26 PAGES</td>
<td>04/2003</td>
</tr>
<tr>
<td>1359</td>
<td>HERAULT T LASSAIGNE R MAGNINETTE F PEYRONNET S</td>
<td>APPROXIMATE PROBABILISTIC MODEL CHECKING</td>
<td>18 PAGES</td>
<td>01/2003</td>
</tr>
<tr>
<td>1360</td>
<td>HU Z LI H</td>
<td>A NOTE ON ORE CONDITION AND CYCLE STRUCTURE</td>
<td>10 PAGES</td>
<td>04/2003</td>
</tr>
<tr>
<td>1361</td>
<td>DELAET S DUCOURTHIAL B TIJEUIL S</td>
<td>SELF-STABILIZATION WITH r-OPERATORS IN UNRELIABLE DIRECTED NETWORKS</td>
<td>24 PAGES</td>
<td>04/2003</td>
</tr>
<tr>
<td>1362</td>
<td>YAO J Y</td>
<td>RAPPORT SCIENTIFIQUE PRESENTE POUR L'OBTENTION DUNE HABILITATION A DIRIGER DES RECHERCHES</td>
<td>72 PAGES</td>
<td>07/2003</td>
</tr>
<tr>
<td>1363</td>
<td>ROUSSEL N EVANS H HANSEN H</td>
<td>MIRRORSPACE : USING PROXIMITY AS AN INTERFACE TO VIDEO-MEDIATED COMMUNICATION</td>
<td>10 PAGES</td>
<td>07/2003</td>
</tr>
<tr>
<td>N°</td>
<td>Nom</td>
<td>Titre</td>
<td>Nbre de pages</td>
<td>Date parution</td>
</tr>
<tr>
<td>------</td>
<td>------------------------</td>
<td>-----------------------------------------------------------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>1364</td>
<td>GOURAUD S D</td>
<td>GENERATION DE TESTS A L'AIDE D'OUTILS COMBINATOIRES : PREMIERS RESULTATS EXPERIMENTAUX</td>
<td>24 PAGES</td>
<td>07/2003</td>
</tr>
<tr>
<td>1365</td>
<td>BADIS H</td>
<td>DISTRIBUTED ALGORITHMS FOR SINGLE AND MULTIPLE-METRIC LINK STATE QoS ROUTING</td>
<td>22 PAGES</td>
<td>07/2003</td>
</tr>
<tr>
<td>1366</td>
<td>AL AGHA K</td>
<td>WHY : A MULTI-LANGUAGE MULTI-PROVER VERIFICATION TOOL</td>
<td>20 PAGES</td>
<td>09/2003</td>
</tr>
<tr>
<td>1367</td>
<td>FILLIATRE J C</td>
<td>A THEORY OF MONADS PARAMETERIZED BY EFFECTS</td>
<td>18 PAGES</td>
<td>09/2003</td>
</tr>
<tr>
<td>1368</td>
<td>FILLIATRE J C</td>
<td>HASH CONSING IN AN ML FRAMEWORK</td>
<td>14 PAGES</td>
<td>09/2003</td>
</tr>
<tr>
<td>1369</td>
<td>FILLIATRE J C</td>
<td>DESIGN OF A PROOF ASSISTANT : COQ VERSION 7</td>
<td>16 PAGES</td>
<td>09/2003</td>
</tr>
<tr>
<td>1370</td>
<td>HERMAN T</td>
<td>A DISTRIBUTED TDMA SLOT ASSIGNMENT ALGORITHM FOR WIRELESS SENSOR NETWORKS</td>
<td>32 PAGES</td>
<td>09/2003</td>
</tr>
<tr>
<td>1371</td>
<td>RIGAUX P</td>
<td>GENERATION AND SYNDICATION OF LEARNING OBJECT METADATA</td>
<td>32 PAGES</td>
<td>10/2003</td>
</tr>
<tr>
<td>1372</td>
<td>APPERT C</td>
<td>CONTEXT MATTERS : EVALUATING INTERACTION TECHNIQUES WITH THE CIS MODEL</td>
<td>14 PAGES</td>
<td>10/2003</td>
</tr>
<tr>
<td></td>
<td>BEAUDOUIN-LAFON M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MACKAY W E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1373</td>
<td>BLANCH R</td>
<td>SEMANTIC POINTING : IMPROVING TARGET ACQUISITION WITH CONTROL-DISPLAY RATIO ADAPTATION</td>
<td>12 PAGES</td>
<td>10/2003</td>
</tr>
<tr>
<td></td>
<td>GUIARD Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEAUDOUIN-LAFON M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1374</td>
<td>FORGE D</td>
<td>COVERING OF THE VERTICES OF A GRAPH BY SMALL CYCLES</td>
<td>16 PAGES</td>
<td>10/2003</td>
</tr>
<tr>
<td></td>
<td>KOUIDER M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>