RAPPORT DE RECHERCHE

L R I

COVERAGE-BIASED RANDOM EXPLORATION OF LARGE MODELS AND APPLICATION TO TESTING

DENISE A / GAUDEL M C / GOURAUD S D / LASSAIGNE R / OUDINET J / PEYRONNET

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

06/2008

Rapport de Recherche N° 1494

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Coverage-biased random exploration of large models and application to testing

Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, Sylvain Peyronnet

alain.denise@lri.fr, mcg@lri.fr, gouraud@lri.fr, johan.oudinet@lri.fr, sylvain.peyronnet@lri.fr
LRI, Université Paris-Sud, UMR CNRS 8623
lassaign@logique.jussieu.fr
Equipe de Logique Mathématique, Université Paris VII, UMR CNRS 7056

June 10, 2008

Abstract. This article presents several related methods for drawing traces. First, it is shown how to draw traces uniformly at random in large models composed of several components. Then a method for drawing traces according to a given coverage criterion is presented, together with a notion of randomised coverage satisfaction. These methods rely on combinatorial algorithms, based on a representation of the model by an automaton or by a product of several automata, synchronised or not. We report several experimental results on random generation of traces in large transition systems, and on statistical testing of C programs.

Contents

1 Introduction

Methods based on randomness seem attractive for testing large programs or checking large models. However, designing efficient random methods, i.e. methods that have a good and assessable fault detection power, is far from being obvious: the underlying probability distribution must be carefully designed if one wants to ensure a good coverage of the program or model, or of potential fault locations, and to quantify this coverage.

In this article we describe a set of methods for drawing paths in graphs either uniformly, or with a coverage criterion as target. We present two applications of these methods: a first one to the exploration of very large labelled transition systems (LTS); a second one to structural testing of C programs, where the considered graph is the control graph.

A classical way to explore large graphs at random is random walk.

To perform a random walk in a graph $\mathcal{G}$, it is sufficient to have a representation of it that allows to generate algorithmically, for any vertex v, the set of successors of v.

The following function **Random Walk** uses such a representation to generate a random path of length n and to check if this path leads to the detection of some fault. We make the simplifying assumption that there is a reliable verdict that detects when a fault is reached during the execution of the random walk.

Random Walk
Input: *graph* $\mathcal{G}, vertex(v_0), n$
Output: samples a path π with origin v_0 of length n and detects if there is a fault on π
- if v_0 is faulty then return 1 else 0
- for $i = 0, \ldots, n-1$, v_{i+1} is chosen uniformly among the successors of v_i in $\mathcal{G}$.
  - if v_{i+1} is faulty then return 1 else 0

A drawback of this approach is that the probability distribution that it induces on the paths of the model is a priori unknown: it depends on the topology of the graph. For instance, if we do a classical random walk of length 3 in the model described in Figure 1, then the probability of each path is unbalanced: $P(b;e;j) = 0.5{\times}0.5{\times}0.5 = 0.125$ and $P(a;c;f) = 0.5$. However, a uniform random sampling of length 3 balances the probability of each path: $P(b;e;j) = P(a;c;f) = 0.2$.

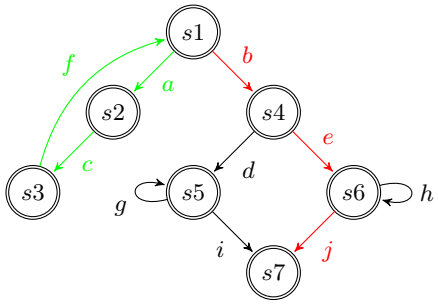

Fig. 1: The case of irregular topology.

By using the method of Karp and al. [22], we could approximate the fault detection probability with approximation techniques for counting problems. However, this method is not practicable for the exploration of paths with very low probabilities, thus for the detection of faults that they contain.

In this paper, we first go one step further by using results of Flajolet and others (see Section 3.1) that make it possible to efficiently draw paths uniformly at random in large graphs.

A second idea is to combine coverage criteria and randomness, introducing a notion of coverage-guided random exploration. What does it means for a random exploration method to take into account a coverage criterion? Let $E_C(\mathcal{G})$ be the set of elements characterised by a coverage criterion C for a given graph $\mathcal{G}$. The satisfaction of this coverage criterion C by some random exploration of the graph $\mathcal{G}$ is characterised by the minimal probability $q_{C,N}(\mathcal{G})$ of covering any element of $E_C(\mathcal{G})$ when drawing N paths. This definition corresponds to a notion of *randomised coverage satisfaction*. It makes it possible to assess and compare random exploration methods with respect to a coverage criterion.

Moreover, it leads to the idea of maximising the minimal probability $q_{C,N}(\mathcal{G})$ of covering any element of $E_C(\mathcal{G})$ when drawing N paths.

The paper is organised as follows. After this introduction, Section 2 recalls briefly some classical definitions, namely labelled transition systems, control flow graphs, and automata.

In Section 3, we present a method for drawing paths uniformly at random in a single automaton (Section 3.1) and then we address the case of the uniform exploration of composed models (Section 3.2), i.e. models that are given

as products of automata, first without and then with synchronisation.

In Section 4 we study the problem of drawing paths at random taking into account some coverage criterion: after some preliminaries, we define in Section 4.2 a notion of random path generation biased towards a coverage criterion. We show how to implement it and how to assess the achieved approximation of the coverage.

In Section 5 we report several experimental results, first on random generation of traces in LTS (in Section 5.1), second on statistical testing of some C programs (in Section 5.2).

Section 6 discusses some related works, and Section 7 gives some conclusions and perspectives.

2 Background

In this paper, we study how to draw paths in two kinds of graphs: labelled transition systems and control graphs of C programs. Considering these graphs as automata makes it possible to use a number of results that are useful for studying random exploration and the notion of randomised coverage.

2.1 Models of reactive systems: LTS

We consider a rather classical kind of model of reactive systems, namely transition systems where transitions are labelled by symbols of a given alphabet X which represent the set of actions of the reactive system.

Definition 1. A labelled transition system (LTS) is a structure $\mathcal{M} = (S, T, s_0, X)$ where S is a set of states, s_0 the initial state, $T \subseteq S \times X \times S$ a transition relation, X a set of labels.

When a LTS is finite, we note $|\mathcal{M}|$ the size of $\mathcal{M}$, i.e. its number of states.

A path of an LTS $\mathcal{M}$ is a finite or infinite sequence $\sigma = (s_0, a_0, s_1, \ldots, s_i, a_i, s_{i+1}, \ldots)$ of transitions satisfying:

$$\forall i \geq 0, (s_i, a_i, s_{i+1}) \in T.$$

2.2 Control graphs

Control graphs are a classical way of representing programs. They are oriented and connected graphs (S,V,s_0,s_f) where S is a set of states, V is a set of transitions, s_0 is the initial state and s_f is the final state. In control graphs, states are either maximum indivisible blocks of statements of the program, or predicates that appear in conditional or loop statements. Transitions correspond to possible transfers of control between these states.

In this paper, we consider control graphs with two distinguished states named s_0 and s_f: they correspond to the beginning point and the exit point of the program. We consider control graphs with no dead code, i.e., any state is reachable from s_0, and s_f is reachable from any state. Each state (resp. transition) is labelled in order to find easily at which piece of code (resp. branch) of the program it corresponds to. A control path is a path in the control graph which goes from s_0 to s_f.

Given a control path, the valuations of inputs such that this path is followed during program execution are characterised by the *path predicate*. This predicate is the conjunction of the conditions (or of their negations) met when traversing the path, adequately updated in function of the variables assignments (see for instance [15]). Any data satisfying the above predicate is an input executing the path, thus a possible test input for covering this path : Thus test data generation is done by resolution of this predicate, using an adequate constraint solver. The choice of the constraint solver depends on the kind of constraints expressible in the programming language.

2.3 Automata

LTS and control graphs are very close to the notion of automata, that comes with a rich corpus of results that we will use in the sequel.

Definition 2. An automaton A is denoted as follows:

$$A = \langle X, S, s_0, F, T \rangle.$$

where X is an alphabet of labels, S a finite set of states, s_0 the initial state, $F \subseteq S$ a set of final states, and T a transition function $T : S \times X \to S$.

We will consider two special cases for F: the case where F is a singleton $\{s_f\}$ (which is convenient when considering control graphs of programs); the case where $F = S$ (which is

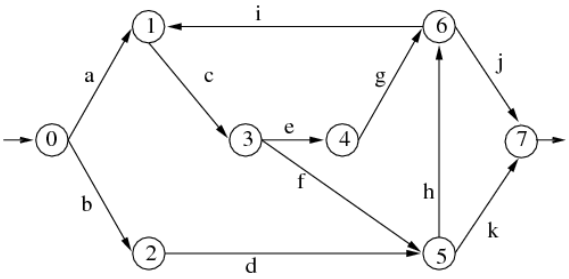

Fig. 2: A finite automaton

convenient when considering models of reactive systems such as LTS).

Figure 2 presents such an automaton, where $S = \{0, 1, 2, 3, 4, 5, 6, 7\}$, $s_0 = 0$, $F = \{7\}$ and $X = \{a, b, c, d, e, f, g, h, i, j, k\}$.

As for LTS, a path of an automaton A is a sequence of transitions.The *trace* of a path in A is the sequence of symbols of X that corresponds to the sequence of transitions. The formal language defined by A, denoted $L(A)$, is the set of traces of all the paths from s_0 to any state of F. Any finite automaton defines a *regular language*. The class of regular languages is the simplest one in Chomsky's hierarchy of formal languages [21].

3 Uniform path random generation

Here we are interested in drawing uniformly at random a set of paths in one or several automata that represents a model, as seen above. In Section 3.1 we suppose that there is only one automaton, and we recall a known algorithm for generating paths uniformly at random. Then, in Section 3.2, we deal with the problem of generating paths uniformly in a very large model which is composed of a (unsynchronised or synchronised) product of much smaller components (modelled by automata). Using combinatorial techniques, we reduce the problem to drawing paths in the components. At first we focus on unsynchronised systems, then we study the case where there is one synchronised transition per component, and all components must synchronise at the same time.

3.1 Single automaton

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 A from s_0 to any state of F.

The aim is, given an integer n, to generate uniformly at random one or several paths of length $\leq n$ from s_0 to any state of F. 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 slight change in the automaton allows to generate paths of length $\leq n$. Remark that generally the number of paths of length n grows exponentially with n.

The principle of the generation process is simple: starting from s_0, 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 state s, let $f_s(m)$ denotes the number of paths of length m which connect s to any state of F. Suppose that, at one given step of the generation, we are on state s, which has k successors denoted $s_1, s_2, \ldots, s_k$. In addition, suppose that $m > 0$ transitions remain to be crossed in order to get a path of length n. Then the condition for uniformity is that the probability of choosing state s_i $(1 \leq i \leq k)$ equals $f_{s_i}(m-1)/f_s(m)$. In other words, the probability to go to any successor of s must be proportional to the number of paths of suitable length from this successor to any state of F.

Computing the numbers $f_s(i)$ for any $0 \leq i \leq n$ and any state s of the graph can be done by using the following recurrence rules:

$$\begin{aligned} f_s(0) &= 1 && \text{if } s \in F \\ &= 0 && \text{otherwise} \\ f_s(i) &= \textstyle\sum_{s \to s'} f_{s'}(i-1) && \text{for } i > 0 \end{aligned} \qquad (1)$$

where $s \to s'$ means that there exists a transition from s to s' (note that s' may be equal to s if loops are allowed in the automaton). Table 1 presents the recurrence rules which correspond to the automaton of Figure 2.

Now the generation scheme is as follows:

- Preprocessing stage: Compute a table of the $f_s(i)$'s for all $0 \leq i \leq n$ and any state s.
- 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

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

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

memory space requirement is $n \times |S|$ integer numbers. The number of arithmetic operations needed for the preprocessing stage is in the worst case in $O(nd|S|)$, where d stands for the maximum number of transitions from a state, and the generation stage is $O(nd)$.

For generating paths of length $\leq n$ instead of exactly n, the only change is the following: Add to the automaton a new state s'_0 which becomes the new initial state, with a (void) transition from s'_0 to s_0 and a (void) loop transition from s'_0 to itself. Each path of length $n+1$ from s'_0 to a state of F in this new automaton crosses k times the new loop transition for some k such that $0 \leq k \leq n$ and exactly once the one from s'_0 to s_0. 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.

The above developments are a special case of a general method of generation of combinatorial structures, which has been first addressed by Wilf [34] and then generalised and systematised by Flajolet, Zimmermann and Van Cutsem[10]. More precisely, our problem is equivalent to the one of uniform random generation of words of regular languages, which has first been discussed by Hickey and Cohen in [19]. We show in Table 2 the set of words which correspond to the paths of length ≤ 10 of the graph of Figure 2.

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

Table 2. The 14 words that correspond to the 14 paths of length ≤ 10 from vertex $s_0 = 0$ to $F = \{7\}$.

3.2 Composed automata

3.2.1 Without synchronisation

Here we focus on the problem of uniformly (that is equiprobably) generating traces of a given length n in a system of r modules represented by automata. In a first step, we consider that there is no synchronisation between the r modules. Each one is represented by a finite state automaton

$$A_i = \langle X_i, S_i, s_i^0, F_i, T_i \rangle.$$

where the X_i's are pairwise disjoint. Each of the A_i's defines a regular language L_i whose words correspond to the traces within the corresponding module.

Now, the following automaton recognises the language L that represents the set of traces in the whole system: $A = \langle X, S, s_0, F, T \rangle$, where

- $X = X_1 \cup X_2 \cup \ldots \cup X_r$;
- $S = S_1 \times S_2 \times \ldots \times S_r$;
- $s_0 = (s_1^0, s_2^0, \ldots, s_r^0)$;
- $F = F_1 \times F_2 \times \ldots \times F_r$;
- $T((s_1, \ldots, s_i, \ldots, s_r), x)) =$

$$
\begin{array}{l}
(T_1(s_1, x), \ldots, s_i, \ldots, s_r) \text{ if } x \in X_1 \\
\ldots \\
(s_1, \ldots, T_i(s_i, x), \ldots, s_r) \text{ if } x \in X_i \\
\ldots \\
(s_1, \ldots, s_i, \ldots, T_r(s_r, x)) \text{ if } x \in X_r
\end{array}
$$

We call this automaton a *shuffling automaton* of $L_1, L_2, \ldots, L_r$ because the language L can be also described by the *shuffling* operation on languages. The *shuffling* of two words w, w', denoted $w \sqcup\!\sqcup w'$ is the set $w \sqcup\!\sqcup w'$ defined as follows:

$$
\begin{array}{l}
\{w_1 w'_1 \cdots w_m w'_m \mid w_i, w'_i \in X^* \wedge \\
\qquad w = w_1 \cdots w_m \wedge w' = w'_1 \cdots w'_m\}.
\end{array}
$$

For example, $ab \sqcup cde = \{abcde, acbde, acdbe, acdeb, cabde, cadbe, cadeb, cdabe, cdaeb, cdeab\}$. The shuffle of two languages L_1 and L_2 is the set

$$L_1 \sqcup L_2 = \bigcup_{\substack{w_1 \in L_1, \\ w_2 \in L_2}} w_1 \sqcup w_2$$

This easily generalises to any finite number r of languages.

Since there is no synchronisation in the system, clearly there is a one-to-one correspondence between the set of its traces and the words of $L = L_1 \sqcup L_2 \sqcup \ldots \sqcup L_r$. Thus the problem reduces to uniformly generating words of length n in L. We present two different approaches for this problem and we discuss their complexity issues.

Brute force method. This first approach consists in constructing the shuffling automaton seen above for $L = L_1 \sqcup L_2 \sqcup \ldots \sqcup L_r$. Then classical algorithms for randomly generating words of a regular language can be processed, as described in Section 3.1.

Let $C_1 = \sum_{0 \leq i \leq r} |X_i|$ and $C_2 = \prod_{0 \leq i \leq r} |S_i|$. The worst-case complexities of the two main steps of the algorithm are the following.

1. Constructing the automaton: This step is performed only once, whatever the number of traces to be generated. Its worst-case complexity is $C_1 C_2$ in time and space requirements.
2. Generating traces: Using classical algorithms, generating one word requires nC_1 time requirement, after a preprocessing stage having worst-case complexity nC_1C_2 in time and space. This preprocessing stage is performed once, whatever the number of traces to be generated.

Hence the worst case complexity for generating m traces of length n is $O(nC_1C_2 + mnC_1)$ in time and $O(nC_1C_2)$ in space. This is linear in n, in m, in the total size of the alphabets. However, since $C_2 = \prod_{0 \leq i \leq r} |S_i|$, the complexity is exponential according to the number of modules. Thus the algorithm will be efficient only for a small number of modules.

"On line" shuffling method. Here we describe an alternative method which avoids constructing the above automaton. At fist we need some additional notation. Let $\ell(k)$ (resp. $\ell_i(k)$) be the number of words of length k belonging to the language L (resp. L_i). The number of words of length n belonging to L is:

$$\ell(n) = \sum_{k_1 + \cdots + k_r = n} \binom{n}{k_1, \ldots, k_r} \ell_1(k_1) \cdots \ell_r(k_r)$$

The method consists first in choosing at random, with a suitable probability, the length n_i of each word w_i of L_i which will contribute to the word w of L to be generated. Then each w_i is generated independently. Finally, the shuffling operation is processed. That is:

1. Choose at random a r-uple $(n_1, \ldots, n_r)$ with probability $\Pr(n_1, \ldots, n_r)$ such that
$$\Pr(n_1, \ldots, n_r) = \frac{\binom{n}{n_1, \ldots, n_r} \ell_1(n_1) \cdots \ell_r(n_r)}{\ell(n)} \quad (2)$$
2. For each $1 \leq i \leq r$, draw uniformly a random word w_i of length n_i in L_i, using the classical algorithm for generating words of a regular language.
3. Shuffle the r words. This can be done with the following algorithm:

Shuffling r words
Input: r words $w_1, \ldots, w_r$, of length $n_1, \ldots, n_r$ respectively
Output: one word w of length $n = \sum_i n_i$, drawn uniformly among the set of shuffles of $w_1, \ldots, w_r$.
$w \leftarrow \varepsilon$
$n \leftarrow \sum_i n_i$
while $n > 0$ do
  choose an integer i between 1 and r with probability $\frac{n_i}{n}$
  add the first letter of w_i at the end of w
  remove the first letter of w_i
  $n_i \leftarrow n_i - 1$
  $n \leftarrow n - 1$

The word w has been generated equiprobably among all the words of L of length n. Regarding complexity issues, clearly the complexity of step 3 is linear in n. The complexity of step 2 is linear in n, in the maximum of $|X_i|$ and in the maximum of $|S_i|$, in time as well as in space requirements. The main contribution to the total worst-case time complexity is the computation of the suitable probabilities by Formula (2). The space requirement is $O(1)$ but the number of terms in $\ell(n)$ is exponential in n. However, there is a way of drastically

simplifying the computation of $\ell(n)$, by using *asymptotic approximates*, as explained below.

According to a well known result (see *e.g.* [23, Chap. 7] or [9, Section IV.5.1]), there exist an integer N_1, a finite set of complex numbers $\omega_1, \omega_2, \ldots, \omega_k$ and a finite set of polynomials $R_1(n)$, $R_2(n)$, ..., $R_k(n)$ such that

$$n \geq N_1 \;\rightarrow\; \ell(n) = \sum_{j=1}^{k} R_j(n)\omega_j^n. \qquad (3)$$

The number N_1, as well as the ω_j's and the R_j's, can be computed from any automaton of L, with an algorithm of polynomial complexity according to the size of the automaton.

If the automaton of L satisfies certain conditions (see below), then there is an unique i such that $|\omega_i| > |\omega_j|$ for any $j \neq i$, and $R_i(n)$ has degree zero, that is $R_i(n) = C$ for any n, where C is a constant. Thus, if we define $\omega = \omega_i$, the following formula holds, asymptotically:

$$\ell(n) \sim C\omega^n. \qquad (4)$$

This gives a very good estimation of $\ell(n)$ even for rather small n since, according to Formulas (4) and (3), $C\omega^n/\ell(n)$ converges to 1 at an exponential rate.

A simple sufficient condition for Formula (4) to hold is: the automaton is *aperiodic* and *strongly connected*. An automaton is aperiodic if, for any sufficiently large n, $l(n) \neq 0$.

Now, as stated in Section 2.3, this is true for labelled transition systems (LTS) because all the states are final states. Concerning strong connectivity, it is a sufficient yet not mandatory condition. For instance, for satisfying Formula (4), it suffices to have some unique biggest strongly-connected component in the automaton.

Now, assuming that all the L_i's are such that

$$\ell_i(k) \sim C_i\omega_i^k \qquad (5)$$

where C_i and ω_i are two constants, we have:

$$\ell(n) \sim C_1 \cdots C_r \sum_{k_1+\cdots+k_r=n} \binom{n}{k_1, \ldots, k_r} \omega_1^{k_1} \cdots \omega_r^{k_r} \qquad (6)$$

$$= C_1 \cdots C_r(\omega_1 + \cdots + \omega_r)^n \qquad (7)$$

However, if the L_i's satisfy the hypothesis of Formula (5), then, by Formula (6):

$$\Pr(n_1, \ldots, n_r) \;\sim\; \frac{\binom{n}{n_1,\ldots,n_r}\omega_1^{n_1}\omega_2^{n_2}\ldots\omega_r^{n_r}}{(\omega_1+\omega_2+\ldots+\omega_r)^n}. \qquad (8)$$

There is an easy algorithm for choosing $n_1, \ldots, n_r$ with this probability without computing it: take the set of integers $\{1, \ldots, r\}$ and draw a random sequence by picking independently n numbers in this set in such a way that the probability to choose i is $\Pr(i) = \frac{\omega_i}{\omega_1+\omega_2+\ldots+\omega_r}$. Then take n_i as the number of occurrences of i in this sequence.

Well, one could argue that Formula (8) only provides an asymptotic approximation of $\Pr(n_1, \ldots, n_r)$ as n tends to infinity. However, as noticed above, the rate of convergence is exponential, so Formula (8) is precise enough even for rather small n. And for really small n (at least when $n < N_1$ in Formula (3)), $\Pr(n_1, \ldots, n_r)$ can be computed exactly by Formulas (1) and (2).

In conclusion, for any large enough n, the algorithm generates traces of length n almost uniformly at random. Its overall complexity is linear according to n, polynomial according to the maximum of $|X_i|$ and to the maximum of $|S_i|$, in time as well as in space requirements.

3.2.2 With one synchronisation

Now we suppose that each module contains exactly one synchronised transition, denoted α. Thus, in the global system all modules must take α at the same time.

Let $A_1, \ldots, A_r$ be r automata, with alphabets $X_1, \ldots, X_r$, all containing a common synchronisation symbol α, such that

$$\forall i, j \in 1 \ldots r, i \neq j, X_i \cap X_j = \{\alpha\}.$$

Let $L_1, \ldots, L_r$ be the respective languages recognised by $A_1, \ldots, A_r$. Here, any trace can be represented by a word belonging to the language L defined as follows: L is the set of words $w \in X_1 \cup \ldots \cup X_r$ such that

$$w = w_0\alpha w_1\alpha \ldots w_{m-1}\alpha w_m$$

where the projection of w onto any X_i belongs to L_i. The number m is the number of synchronisations during the process: each of the projections contains exactly m letters α (and, equivalently, there is no α in any of the w_i.)

Again the brute force approach. Here the approach consists in constructing the *synchronised* product of $A_1, A_2, \ldots, A_r$, as follows. Let $X_{i,\alpha} = X_i \setminus \{\alpha\}$. The synchronised product [1] of $A_1, A_2, \ldots, A_r$ with $\{\alpha\}$ as synchronisation

set is the finite automaton $A =< X, S, s_0, F, T >$, where

- $X = X_1 \cup X_2 \cup \ldots \cup X_r$;
- $S = S_1 \times S_2 \times \ldots \times S_r$;
- $s_0 = (s_1^0, s_2^0, \ldots, s_r^0)$;
- $F = F_1 \times F_2 \times \ldots \times F_r$;
- T is as follows:

$$T((s_1, \ldots, s_i, \ldots, s_r), x)) = \begin{array}{ll} (T_1(s_1, x), \ldots, s_i, \ldots, s_r) & \text{if } x \in X_{1,\alpha}, \\ \ldots & \\ (s_1, \ldots, T_i(s_i, x), \ldots, s_r) & \text{if } x \in X_{i,\alpha}, \\ \ldots & \\ (s_1, \ldots, s_i, \ldots, T_r(s_r, x)) & \text{if } x \in X_{r,\alpha}. \end{array}$$

$$T((s_1, \ldots, s_i, \ldots, s_r), \alpha)) = T_1(s_1, \alpha), \ldots, T_i(s_i, \alpha), \ldots, T_r(s_r, \alpha))$$

This automaton accepts the language L of synchronised traces. Once it has been built, the generation process is exactly as in 3.2.1. The construction easily generalises to these cases where there are several synchronisations $\alpha_1, \ldots, \alpha_k$ in each automaton. If k is small, the size of the synchronised product is of the same order as for the non-synchronised case (see 3.2.1). However, in presence of numerous synchronisations the brute force method turns out to be exploitable (see Section 5.1.4) since the reachable state space of the synchronised product remains of reasonable size.

"On line" generation of synchronised traces. Here we sketch an algorithm for almost uniformly generating random synchronised traces of length n, avoiding the construction of the synchronised product. The approach is similar to the one we described in Section 3.2.1, although we must be more careful because of the synchronisations. Given that each automaton A_i contains a unique transition labeled by α (the synchronised transition), let $s_{i,1}$ and $s_{i,2}$ be the states just before and just after this transition, respectively. Now let us define, for each L_i, the four following languages:

- The *beginning language*: B_i is the set of words corresponding to the paths which start at the initial state of A_i, which do not cross the α transition, and which stop at $s_{i,1}$.
- The *central language*: C_i is the set of words corresponding to the paths which start at $s_{i,2}$, which do not cross the α transition, and which stop at $s_{i,1}$.
- The *ending language*: E_i is the set of words corresponding to the paths which start at $s_{i,2}$, which do not cross the α transition, and which stop anywhere.
- The *non-synchronised language*: D_i is the set of words which start at the initial state of A_i, which never cross the α transition, and which stop anywhere.

For any i, the language L_i can be defined according to B_i, C_i, E_i and D_i:

$$L_i = B_i.(\alpha.C_i)^*.\alpha.E_i \ \cup \ D_i \, .$$

Thus, if we define $B = \sqcup\!\sqcup_{i=1}^{r} B_i$ (resp. $C = \sqcup\!\sqcup_{i=1}^{r} C_i$, $E = \sqcup\!\sqcup_{i=1}^{r} E_i$, and $D = \sqcup\!\sqcup_{i=1}^{r} D_i$), we have:

$$L = B.(\alpha.C)^*.\alpha.E \ \cup \ D \, . \tag{9}$$

Now let $\ell(n)$ (resp. $\ell_i(n)$, $b(n)$, $b_i(n)$, $c(n)$, $c_i(n)$, $e(n)$, $e_i(n)$, $d(n)$, $d_i(n)$) be the number of words of length n in L (resp. L_i, B, B_i, C, C_i, E, E_i, D, D_i). Additionally, let $\ell(n, m)$ be the number of words of L of length n which contain α exactly m times. Let w be one of these words. If $m > 0$, then w writes $w = w_0.\alpha.w_1.\alpha.\ldots.\alpha.w_m$ where $w_0 \in B$, $w_i \in C$ for any $1 \leq i < m$, and $w_m \in E$. Finally, let $\ell(n, m, i_0, i_m)$ be the number of such words such that the length of w_0 equals i_0 and the length of w_m equals i_m. Then we have

$$\ell(n) = \sum_{i=0}^{n} \ell(n, i) \, , \tag{10}$$

where

$$\ell(n, m) = \begin{cases} d(n) & \text{if } m = 0, \\ \displaystyle\sum_{i_0 + i_m = 0}^{n-m} \ell(n, m, i_0, i_m) & \text{otherwise,} \end{cases} \tag{11}$$

and, for $m > 0$,

$$\ell(n, 1, i_0, i_1) = b(i_0) \ e(i_1), \tag{12a}$$

$$\ell(n, m, i_0, i_m) = b(i_0) \ e(i_m) \sum_{\substack{i_1 + \ldots + i_{m-1} = \\ n-m-i_0-i_m}} c(i_1)c(i_2)\ldots c(i_{m-1}). \tag{12b}$$

Now suppose that all the B_i's, the C_i's, the E_i's and the D_i's satisfy Formula (4), that is:

$$b_i(k) \sim C_{b,i}\omega_{b,i}^k \, ,$$
$$c_i(k) \sim C_{c,i}\omega_{c,i}^k \, ,$$
$$e_i(k) \sim C_{e,i}\omega_{e,i}^k \, ,$$
$$d_i(k) \sim C_{t,i}\omega_{t,i}^k \, .$$

Then, similarly to Formula (6), we have:

$$b(k) \sim C_{b,1} \dots C_{b,r} (\omega_{b,1} + \dots + \omega_{b,r})^k \,, \quad (13)$$
$$c(k) \sim C_{c,1} \dots C_{c,r} (\omega_{c,1} + \dots + \omega_{c,r})^k \,, \quad (14)$$
$$e(k) \sim C_{e,1} \dots C_{e,r} (\omega_{e,1} + \dots + \omega_{e,r})^k \,, \quad (15)$$
$$d(k) \sim C_{t,1} \dots C_{t,r} (\omega_{t,1} + \dots + \omega_{t,r})^k \,. \quad (16)$$

Consequently, for $m = 1$,

$$\ell(n, 1, i_0, i_1) \sim (C_{b,1} \cdots C_{b,r})(C_{e,1} \cdots C_{e,r}) (\omega_{b,1} + \cdots + \omega_{b,r})^{i_0} (\omega_{e,1} + \cdots + \omega_{e,r})^{i_1}, \quad (17a)$$

and for $m > 1$,

$$\ell(n, m, i_0, i_m) \sim \binom{n-i_0-i_m-2}{m-2} (C_{b,1} \cdots C_{b,r})(C_{c,1} \cdots C_{c,r})^{m-1}(C_{e,1} \cdots C_{e,r}) (\omega_{b,1} + \cdots + \omega_{b,r})^{i_0} (\omega_{c,1} + \cdots + \omega_{c,r})^{n-m-i_0-i_m} (\omega_{e,1} + \cdots + \omega_{e,r})^{i_m} \,. \quad (17b)$$

Now let us evaluate the complexity of computing $\ell(n, m, i_0, i_m)$ for all pairs (i_0, i_m) such that $0 \leq i_0 + i_m \leq n$ and for all m such that $1 \leq m \leq n - i_0 - i_m$. At first, suppose that i_0 and i_m are fixed. Computing the $\ell(n, m, i_0, i_m)$'s for all m involves computing all the binomial coefficients in Formula (17b). Each of them involves only a constant number of arithmetic operations, using the binomial recurrence formula. In the same way, each of the $(C_{c,1} \cdots C_{c,r})^{m-1}$'s, each of the $(\omega_{b,1} + \cdots + \omega_{b,r})^{i_0}$'s, each of the $(\omega_{c,1} + \cdots + \omega_{c,r})^{n-m-i_0-i_m}$'s, and each of the $(\omega_{e,1} + \cdots + \omega_{e,r})^{i_m}$'s can also be computed with a constant number of operations. Altogether, computing the $\ell(n, m, i_0, i_m)$'s for all $1 \leq m \leq n - i_0 - i_m$ can be achieved in $O(n + r)$ arithmetic operations. Thus, computing $\ell(n, m, i_0, i_m)$ for all pairs (i_0, i_m) such that $0 \leq i_0 + i_m \leq n$ and for all m such that $1 \leq m \leq n - i_0 - i_m$ needs $O(n^3 + (n + r)) = O(n^3 + r)$ arithmetic operations.

Now we can sketch the algorithm for generating a trace of length n.

1. Using Formulas (17a) and (17b), compute $\ell(n, m, i_0, i_m)$ for all pairs (i_0, i_m) such that $0 \leq i_0 + i_m \leq n$ and for all m such that $1 \leq m \leq n - i_0 - i_m$. As seen above, this requires $O(n^3 + r)$ arithmetic operations. Then compute $\ell(n, m)$ for all m such that $1 \leq m \leq n$, using Formula (11) and, additionally, Formula (16) when $m = 0$. Finally compute $\ell(n)$ by Formula (10). It is worth noticing that this preliminary stage has to be done only once, whatever the number of traces of length n to be generated. Its overall arithmetic complexity is $O(n^3 + r)$.
2. Choose m, the number of synchronisations, with probability
$$\Pr(m) = \frac{\ell(n, m)}{\ell(n)}.$$
Computing these probabilities requires $O(n)$ arithmetic operations in the worst case.
3. If $m = 0$, then generate uniformly at random a word of length n in T, with the "on line" shuffling method of Section 3.2.1.
4. If $m > 0$, then:
   (a) Choose the length of w_0 and the length of w_m by picking at random a pair (i_0, i_m) with probability
   $$\Pr(i_0, i_m) = \frac{\ell(n, m, i_0, i_m)}{\sum_{k_0+k_m=0}^{n-m} \ell(n, m, k_0, k_m)} \,.$$
   Computing these probabilities requires $O(n^2)$ arithmetic operations in the worst case.
   (b) Choose the lengths of $w_1, w_2, \dots, w_{m-1}$ by picking at random a $(m-1)$-uple $(i_1, i_2, \dots i_{m-1})$ with probability
   $$\Pr(i_1, \dots i_{m-1}) = \frac{c(i_1)c(i_2)\dots c(i_{m-1})}{\sum_P c(k_1)c(k_2)\dots c(k_{m-1})} \,.$$
   where P stands for :
   $$\{k_1, \dots, k_{m-1} \mid k_1 + \cdots + k_{m-1} = n - m - i_0 - i_m\}.$$
   By Formula (14) this leads to
   $$\Pr(i_1, \dots, i_{m-1}) \sim \frac{(\omega_{c,1} + \cdots + \omega_{c,r})^{n-m-i_0-i_m}}{\sum_P (\omega_{c,1} + \cdots + \omega_{c,r})^{n-m-i_0-i_m}} = \frac{1}{\sum_P 1}$$
   The denominator equals the number of distinct ways to choose $(k_1, k_2, \dots, k_{m-1})$ in such a way that they sum to $n - m - i_0 - i_m$. This means that the sequence $(i_1, i_2, \dots, i_{m-1})$ is to be picked uniformly among all sequences such that $k_1 + k_2 + \cdots + k_{m-1} = n - m - i_0 - i_m$. Let $Q = n - m - i_0 - i_m$ and $q = m - 1$. The number of ways to choose q numbers greater or equal to zero that sum

to Q equals $\binom{Q+q-1}{q-1}$, for any positive integers Q and q. Hence

$$\Pr(i_1, \dots i_{m-1}) \sim \frac{1}{\binom{n-2-i_0-i_m}{m-2}}.$$

And there is an easy algorithm to generate uniformly at random q numbers $i_1, i_2, \dots, i_q \geq 0$ that sum to Q: pick uniformly at random $q-1$ distinct numbers $j_1 < j_2 < \dots < j_{q-1}$ between 1 and $Q+q$, then set $i_1 = j_1 - 1$, $i_2 = j_2 - j_1 - 1$, $\dots$, $i_{q-1} = j_{q-1} - j_{q-2} - 1$, $i_q = Q - j_{q-1}$. Clearly, this simple algorithm is linear according to Q and q, hence to n and m.

(c) Now we have got the whole sequence $(i_0, i_1, ..., i_m)$ with a suitable probability. It remains to generate the words $w_0 \in B$, $w_1, w_2, \dots, w_{m-1} \in C$ and $w_m \in E$, each w_k having length i_k. Each of these words is simply a shuffle of the r languages $(B_i)_{i=1\dots r}$ if $k = 0$, $(C_i)_{i=1\dots r}$ if $1 \leq k < m$, $(E_i)_{i=1\dots r}$ if $k = m$. For each of the w_k's, the shuffling algorithm given in Section 3.2.1 can be used.

As noticed above, the first step of the algorithm, in $O(n^3 + r)$ operations, has to be done only once. Thereafter, the overall complexity of generating any random trace of length n is quadratic according to n. And, as in Section 3.2.1, it is polynomial according to the maximum of $|X_i|$ and to the maximum of $|S_i|$, in time as well as in space requirements. Thus we have defined an efficient way for approximating the uniform coverage in presence of one synchonisation for any sufficiently large n.

4 Randomised coverage criteria

4.1 Coverage criteria and randomness

The idea of combining coverage criteria and random testing aims at overcoming some drawbacks of both approaches.

Applying coverage criteria corresponds to a decomposition of the input domain into some (very often non disjoint) sub-domains: Each element to be covered defines a sub-domain that is the set of inputs that cause its execution. The main drawback here is that these sub-domains are generally not homogeneous, i.e. some of their inputs may result in a failure, and some others may yield correct results.

Random testing lessens this drawback since it allows intensive test campaigns where the same element of the program may be executed several times with different data. However, in its pure uniform version it induces a bad coverage of cases corresponding to small input subdomains.

In [31,32], Thévenod-Fosse and Waeselynck developed what they called a statistical testing method where the input distribution takes into account some coverage criteria in order to avoid the existence of low probability cases. They have reported several experiments, which led to the conclusion that their approach has a better fault detection power than uniform random testing and deterministic testing based on classical coverage criteria. However, the construction of the input distribution is difficult since it requires the resolution of as many equations as paths in the program (or traces in the specification). For large programs, or in presence of loops, the construction is empirical, based on preliminary observations of the behaviour of the program [32].

Here, we avoid the explicit construction of the distribution by using the techniques presented in Section 3 for drawing paths, and then randomised constraint solving for generating inputs that exercise these paths. Before presenting the approach in detail, let us precisely state what it means for a random testing method to take into account a coverage criteria.

A notion of test quality for statistical testing methods has been defined first in [30]. We slightly reformulate it for our context.

Let A be an automaton describing a system under test. On the basis of this automaton , it is possible to define coverage criteria: all-states, all-transitions, all-paths-of a certain-kind, etc. More precisely, a coverage criterion C characterises for a given description A a set of elements $E_C(A)$ of the underlying automaton (denoted E in the sequel when C and A 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 random testing, the satisfaction of a coverage criteria C by a testing method for a description A is characterised by the minimal probability $q_{C,N}(A)$ of covering any element of $E_C(A)$ when drawing N tests.

q	0.9	0.99	0.999	0.9999
N	32	63	94	125

Table 3. Number of tests N required for a given test quality q

In [30], $q_{C,N}(A)$ is called the test quality of the method with respect to C.

The test quality $q_{C,N}(A)$ can be easily stated if $q_{C,1}(A)$ is known. Indeed, one gets $q_{C,N}(A) = 1-(1-q_{C,1}(A))^N$, 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 3.1, where we generate uniformly random paths among the set $\mathcal{P}_{\leq n}$ of paths of length $\leq n$. 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".

The assessment of test quality is more complicated in general. Let us consider more practicable coverage criteria, such as "all-states" or "all-transitions", and some given random testing method. Generally, the elements to be covered have different probabilities to be reached by a test. Some of them are covered by all the tests. Some of them may have a very weak probability, due to the structure of the behavioural automaton or to some specificity of the testing method. For instance, in the example transitions b and d appear in 5 paths of length ≤ 10 only. Transitions 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 transition b, and $\frac{9}{14}$ to reach transition a.

Let $E_C(A) = \{e_1, e_2, ..., e_m\}$ 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 random testing method. Then

$$q_{C,N}(A) = 1 - (1 - p_{min})^N \qquad (18)$$

where $p_{min} = min\{p_i | i \in (1..m)\}$. Consequently, the number N of tests required to reach a given quality $q_C(A)$ is

$$N \geq \frac{log(1 - q_C(A))}{log(1 - p_{min})} \qquad (19)$$

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

In the case of random testing based on a given distribution, p_{min} characterizes, for a given coverage criterion C, the approximation of the coverage induced by the distribution. However, maximizing p_{min} must not lead to give up the randomness of the method. This may be the case when there exists a path traversing all the elements of $E_C(A)$: one can maximize p_{min} by giving a probability 1 to this path, going back to a deterministic testing method. Thus, another requirement must be combined to the maximization of p_{min}: All the paths traversing an element of $E_C(A)$ must have a non null probability and the minimal probability of such a path must be as high as possible. Thus designing an optimal random testing method for a given coverage criterion turns out to be a difficult multi-criteria problem:

1. maximising p_{min}, that is the minimal probability to any element of $E_C(\mathcal{A})$ to be reached by a path
2. maximising the minimal probability of any path traversing an element of $E_C(\mathcal{A})$

In most cases, these criteria are antagonist and some compromises must be found. In the next section, we present a solution that favorises (1) and weakens (2) into: "any path traversing an element of $E_C(\mathcal{A})$ must have a non-null probability."

4.2 Path generation biased towards a coverage criterion

Now let us consider a given coverage criterion C. As a preliminary remark, note that the set of elements $E_C(A)$ 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 the example (Figure 2).

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 automaton". 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.

Two cases must be considered, according to the nature of the elements of $E_C(A)$. If $E_C(A)$ denotes a set of paths in the automaton, the quality of test is optimal if the paths of $E_C(A)$ are generated uniformly, i.e. any path has the same probability $1/|E_C(A)|$ to be generated. Indeed, if the probability of one or several paths was greater than $1/|E_C(A)|$, then there would exist at least one path with probability less than $1/|E_C(A)|$, therefore the quality of test would be lower. Section 3.1 presented how to generate uniformly random paths of given length n in an automaton, and how 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 4.3.

In the case where the elements of $E_C(A)$ are not paths, but are constitutive elements of the automaton as, for example, states, transitions, or cycles, uniform generation of paths does not ensure optimal quality of test in this case. Ideally, the distribution on paths should ensure conditions (1) and (2) above. In Gouraud *and al.* [5,15], some of the authors of the present paper propose a practical solution in two steps:

1. pick at random one element e of $E_C(A)$, according to a suitable probability distribution (which is discussed below),
2. generate uniformly at random a path of length $\leq n$ that goes through e. This ensures a balanced coverage of the set of paths which cover e.

Now let us compute the probability p_i for the element e_i (for any i in $[1..m]$) to be reached by a path generated with the above process. Let

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

The probability of reaching e_i by drawing a random path which goes through another element e_j is $\frac{\alpha i,j}{\alpha_i}$. Thus the probability p_i for the element e_i (for any i in $[1..m]$) to be reached by a path is

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

which simplifies to

$$p_i = \sum_{j=1}^{m} \pi_j \frac{\alpha_{i,j}}{\alpha_j} \tag{20}$$

since $\alpha_{i,i} = \alpha_i$.

We will see in the following Subsection how to compute the α_j's and the $\alpha_{i,j}$'s, and how to generate paths that cross a given transition. For now, let us suppose that the α_j's and the $\alpha_{i,j}$'s have been computed. The problem of computing the probabilities $\{\pi_1, \pi_2 \ldots, \pi_m\}$ with $\sum \pi_i = 1$, which maximise $p_{min} = \min\{p_i, i \in [1..m]\}$ can be stated as a linear programming problem:

Maximise p_{min} 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 Formula 20. Standard methods lead to a solution in time polynomial according to m.

However, it may happen that some paths traversing an element to be covered have a null probability (see the example below). In this case, the solution we have chosen is to redefine the probability distribution on $E_C(\mathcal{A})$ with the additional requirements that each element has a non-null probability greater or equal than some small positive value ε[1].

Starting with the principle of a two-step drawing strategy as seen above, this method ensures a maximal minimum probability of reaching the elements to be covered and, once one element chosen, a uniform coverage of the paths traversing this element. For a given number of tests, it makes it possible to assess the approximation of the coverage, and conversely, for a required approximation, it gives a lower bound of the number of tests to reach this approximation (cf. Formula 19).

[1] This solution has the advantage of being general, i.e. applicable for any coverage criterion. For certain simple criteria, there exist simpler solutions. For instance, for "all-states", it is sufficient to add the requirement that π_0, the probability to get s_0 at the first step, is greater or equal than some ε.

	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}.

Let us illustrate this method with the example. Given the coverage criterion "all-transitions" and given $n = 10$, Table 4 presents the coefficients $\alpha_{i,j}$, where i and j 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 the initial state to the final state which cross both transitions c and f in the automaton of Figure 2.

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 maximised. The second term is one of the p_i's, computed according to Formula 20. For example, the first line means that p_{min} must be lower or equal to p_{a}, the probability of reaching transition 'a' with a random path. By maximising p_{min}, one maximises the lowest p_i, i.e. the quality of test. The last line ensures that the probabilities π_i 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 $\pi_{\mathrm{b}} = \pi_{\mathrm{k}} = \frac{5}{16} = 0.3125$ and $\pi_{\mathrm{e}} = \frac{6}{16} = 0.375$. This gives $p_{min} = \frac{1}{2} = 0.5$, therefore the quality of test is $1 - \frac{1}{2^N}$, according to Formula 18. But with this distribution, the paths *acfhj* and *acfhicfhj* have a null probability to be drawn. To ensure the coverage of all paths, one adds the following constraints $\pi_x \geq \varepsilon$ for all $x \in E_C(\mathcal{A})$. In the example, with $\varepsilon = 0.0001$, the new distribution is $\pi_{\mathrm{a}} = \pi_{\mathrm{b}} = \pi_{\mathrm{c}} = \pi_{\mathrm{e}} = \pi_{\mathrm{f}} = \pi_{\mathrm{h}} = \pi_{\mathrm{i}} = \pi_{\mathrm{j}} = \frac{1}{1000}$, while $\pi_{\mathrm{d}} = \frac{35729}{115200} \approx 0.3101$, $\pi_{\mathrm{g}} = \frac{11867}{32000} \approx 0.3708$, and $\pi_{\mathrm{k}} = \frac{179141}{576000} \approx 0.311$. This gives $p_{min} = \frac{58893}{120000} \approx 0.4908$

The new p_{min} is very close to the previous one.

4.3 Conditions on paths and operations on automata

As seen above, our approach involves counting and randomly generating paths subject to constraints that depends on the kind of coverage that is considered. Here we show how to deal with the automaton in order to take into account such constraints.

At first, let us focus on a simple example: we want to construct, given the automaton A of Figure 2 and the transition labeled e, an automaton B whose set of paths is equal to the set of those paths of A which cross the transition labeled e. This can be done by using the following procedure:

1. Create a copy A' of A, in which the transitions are labeled exactly as the ones of A, and in which any state label s in A becomes s' in A'.
2. As the transition labeled e joins state 3 to state 4 in A, delete it and replace it with a new transition labeled e between state 3 (in A) and state $4'$ (in A').
3. Set F' as the set of final states, instead of F (and s_0 remains the initial state.)
4. Delete all the states (and their adjacent transitions) to which no path from s_0 exists.
5. Delete all the states (and their adjacent transitions) from which no path to a state of F' exists.

This concludes the construction of B. Figure 3 shows the result of the procedure. This transformation can be done in linear time and linear memory requirement with respect to the size of the initial automaton. Note that steps 4 and 5 are not mandatory: they are used only to "clean" the final automaton by deleting useless elements.

The process we just illustrated can be widely generalised, by considering classical operations on formal languages. As seen in Section 2, the set of paths from s_0 to F in A can be seen as a regular language $L(A)$. And any constraint on the set of paths can be expressed as the intersection of $L(A)$ and another formal language L'. For example, the above condition "cross the transition e" can be represented by the language $L' = X^*eX^*$, that is the language of

$$
\begin{array}{rcccccccccccc}
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.

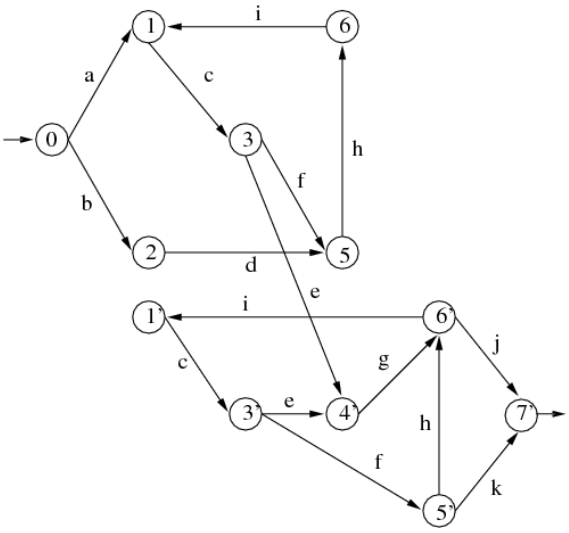

Fig. 3: An automaton that contains only the paths of the automaton of Figure 2 which cross transition labeled 'e'.

words that contains the letter e at least once. And, obviously, $L(A) \cap L'$ equals the language that corresponds to the paths from s_0 to F in A that cross transition e. In the case where L' is a regular language, there are classical algorithms to construct its automaton and the automaton of the intersection of the two languages (see e.g. [21]). The complexity of the intersection algorithm, and the number of states of the corresponding automaton, are proportional to the product of the numbers of states of the two automata. A number of constraints can be modelled by a regular language, notably:

- to cross one transition (or one state), or more generally a set of transitions (or states), in a given order or in an arbitrary order;
- not to cross a given transition (or state) or a given set of transitions (or states);
- to cross one or several transitions, or one or several sequences of transitions, a fixed number of times;
- any combination of the above constraints;
- etc.

Meanwhile, there are constraints that cannot be modelled by a regular language. For example, "cross transition a not a fixed number of times, but exactly as many times than transition b" is a constraint that typically needs a more general class of languages in Chomsky's hierarchy [21]. In this case, it can be modelled by a *context-free* language. The method described here for regular languages can be generalised to context-free languages, by using context-free grammars in place of automata. But this is beyond the scope of the present paper. On the other hand, there exist constraints that need an even more general class in Chomsky's hierarchy. They cannot latter be handled by the methods described here in general. An example of such a constraint is "the paths to be generated must not cross twice the same state". This defines the set of self-avoiding paths in an automaton. Problems on self-avoiding paths are known to be extremely difficult.

5 Experimental results

In this section we present the experiments we did to prove the effectiveness of our approaches.

More precisely, in Section 5.1 we give numerical results for the uniform generation of random paths in both non-synchronised and synchronised systems while in Section 5.2 we present experiments of our method for testing programs under different coverage criteria.

5.1 Uniform path generation

We present experimental results that prove the feasibility of the method we described in Section 3. We first describe the implementation of our approach for the uniform generation of paths of a given length from components of a transition system described in the BCG format (Binary Coded Graphs bcg-format). We then present our methodology for the experiments together with our experimental framework. Last we give tables summarising the numerical results together with a discussion, both in the non-synchronised and synchronised cases. Moreover, we compare our approach to the one that build the whole system (e.g., computing the product of several components) before generating paths.

5.1.1 Implementation and methodology

Random paths are uniformly generated according to the following process: First, the graph underlying the transition system described in the BCG format is used to generate a MuPAD (see [3]) script. This script allows to compute the values of ω and C (see Section 3.2.1). Then, we use ad-hoc tools written in C++ in two different manners:

1. Non-synchronised case (see Section 3.2.1): we first compute the so-called counting table, which is the only preprocessing step done for this case. Then we compute the length of paths we will pick in each component and generate these paths. Last we shuffle these paths to obtain a path in the whole system.
2. Synchronised case (see Section 3.2.2): it is almost the same case as previously described, except that in the preprocessing phase we need to compute binomial coefficients and we generate a counting table for each sublanguage (B, C, T, and E) of each module. Before computing the length of paths we will pick in each module, we have to choose the number of synchronised transitions taken and where they occur in the global path.

For our implementation, we use several tools that we mention here: the BCG library of the CADP toolbox [12], the GMP (Gnu Multiple Precision) library [16] and the random function of the BOOST library [25] in order to generate random numbers.

We did all our experiments on a dedicated server whose hardware is composed of an Intel Xeon 2.80GHz processor with 1GB memory. Each BCG graph used for our experiments comes from the VLTS (Very Large Transition Systems [11]) benchmark suite. These models correspond to real industrial systems. Each model name is of the form *vasy_X_Y*, where X is the number of states divided by 1000, and Y is the number of transitions divided by 1000. Measurements were done 10 times and we give the mean value and standard deviation of these measurements.

5.1.2 Generation in single models

Here, we give results for the uniform generation of paths in a transition system described by a single component. Table 6 shows the time measurements for the uniform generation of paths in models of various sizes.

The main drawback of the brute force approach is its memory consumption. When this approach is feasible, it is efficient. However, we can see that it is not possible to deal with systems of size more than 10^4 states for reasonable path lengths. In the next section, we show that generating paths from composed systems allows to handle systems up to 10^{27} states and maybe more.

5.1.3 Generation in composed systems without synchronisation

We present here the results for the uniform generation of paths in a system succinctly described as the composition of vasy_0_1 with itself several times (from 2 to 12 times). If r is the number of such modules used for the composition, then the number of states of the whole system is 289^r since there is 289 states in the component and there is no synchronisation here.

name	length # states	200	1000	2000	3000	5000	8000
vasy_0_1	289	0.0s	0.9s	2.9s	6.3s	15.9s	40.1s
vasy_1_4	1183	0.1s	1.0s	3.2s	6.7s	18.2s	✘
vasy_5_9	5486	0.0s	0.9s	2.4s	5.2s	✘	✘
vasy_8_24	8879	0.2s	0.8s	2.4s	✘	✘	✘
vasy_10_56	10^4	0.0s	1.3s	✘	✘	✘	✘
vasy_12323_27667	10^7	✘	✘	✘	✘	✘	✘

Table 6. Uniform generation of paths in models of various sizes: size versus time. ✘ means there is not enough memory to build the counting table.

MuPAD needs 8.23s, measured using the Unix time function, to compute the value of ω for each component.

Table 7 and Table 8 give respectively the time needed in order to build the counting table for vasy_0_1 (the table that gives the number of paths leaving each state of the module) and the time required for the generation of 100 paths. The time measurements are here done with the timer function of the BOOST library.

We observe that 289^{12} is of the same order of magnitude as 10^{27}. It means that this method is tractable and still efficient for very large models.

We also did experiments with systems composed of different components. For instance, Table 9 (resp. Table 10) shows the time needed to generate 100 paths in a system composed of two (resp. three) different components. As soon as it is possible to build the counting table of each component, we can draw paths in the whole system.

5.1.4 Generation in composed systems with synchronisation

Now, we experiment the generation of paths in composed systems when the composition is synchronised. Two case-studies show the advantages and drawbacks of a *brute force* approach and the *on-line* method. We first use a real case-study to draw paths in a communication protocol, based on a *brute force* approach (see 3.2.2). Then, we analyse an artificial case-study for the *on-line* generation of paths in systems in which components have a single synchronised transition.

Brute force approach. We studied a well-known communication protocol: the INRES protocol [20]. In this three-model protocol, the *Initiator* sends data to the *Responder* through a *Medium* that offers an unreliable data-transfer service. We used a LOTOS description of this protocol[2].

The method is as follows:

1. we build an automaton that represents the whole system from the LOTOS description,
2. we uniformly draw paths in this system.

The step 1 starts from a LOTOS description of each component. Component sizes are summarised in Table 11. We build an automaton that represents the composed system by using CADP tools: First, we translate LOTOS abstract data types into a concrete implementation in C with the program *caesar.adt*. Then, we translate the system described in LOTOS into a finite state automaton in the BCG format with the program *caesar*.

This process takes less than one second to build the BCG automaton. Table 12 describes its sizes.

Once the whole system is built, the step 2 consists in drawing uniformly at random paths as in the single-model case. Table 13 and Table 14 show the time needed to build the counting table and to generate 100 paths in the whole system, respectively.

This case-study reveals that the *brute force* approach is feasible and provides good results in specific composed systems in which there are few non-synchronised transitions. Actually, in such systems, the size of the product automaton is not too large according to the size of the biggest component. So we can build the whole system and then draw paths in this system. However, if there are many non-synchronised transitions, the size of the whole system will be

[2] ftp://ftp.inrialpes.fr/pub/vasy/demos/demo_09

length	200	500	1000	2000	4000	8000
	0.12s	0.34s	0.77s	2.06s	6.01s	20.70s

Table 7. Preprocessing: time for the construction of the counting table of vasy_0_1, according to the path length.

length / # components	200	500	1000	2000	4000	8000
2	0.58s (0.04)	0.91s (0.08)	1.83s (0.08)	4.71s (0.08)	14.93s (0.54)	37.39s (0.40)
4	0.91s (0.07)	1.25s (0.10)	2.42s (0.11)	5.09s (0.25)	14.57s (0.12)	36.05s (1.04)
6	1.37s (0.11)	1.73s (0.08)	3.00s (0.10)	6.39s (0.27)	18.31s (0.13)	42.21s (0.69)
8	1.78s (0.17)	2.20s (0.07)	3.70s (0.08)	7.91s (0.18)	22.61s (0.32)	49.88s (0.72)
10	2.16s (0.12)	2.76s (0.16)	4.57s (0.13)	9.30s (0.25)	26.82s (0.15)	58.61s (0.69)
12	2.65s (0.22)	3.41s (0.15)	5.31s (0.13)	11.23s (0.17)	31.36s (0.18)	68.73s (1.23)

Table 8. Generation: average time (and standard deviation) for the generation of 100 paths in composed models without synchronisation (*vasy_0_1* is composed with itself).

length	200	1000	2000	3000	5000	8000
	0.3s	1.0s	2.4s	4.1s	9.2s	✘

Table 9. Average time for the generation of 100 paths in a system composed of vasy_0_1 and vasy_1_4. ✘ means there is not enough memory to build the counting table of each components.

length	200	1000	2000	3000	5000	8000
	0.2s	1.6s	2.3s	2.9s	✘	✘

Table 10. Average time for the generation of 100 paths in a system composed of vasy_0_1, vasy_1_4, and vasy_5_9. ✘ means there is not enough memory to build the counting table of each components.

model	# states	# transitions	
		synchronised	internal
Initiator	34	111	4
Responder	26	81	2
Medium	65	294	0

Table 11. Description of the three models (Initiator, Responder and Medium) that compose the INRES protocol.

model	# states	# transitions	
		synchronised	internal
Global system	981	2290	262

Table 12. Description of the BCG automaton built from a LOTOS description of the INRES protocol.

length	200	500	1000	2000	4000	8000
	0.21s	0.74s	1.76s	4.53s	12.40s	✘

Table 13. Preprocessing: time to build the counting table from the INRES system according to the path length. ✘ means there is not enough memory to build the counting table.

length	200	500	1000	2000	4000	8000
	$0.08s$	$0.23s$	$0.86s$	$2.54s$	$8.54s$	✘

Table 14. Generation: average time to generate 100 paths in the INRES system. ✘ means there is not enough memory to build the counting table.

too large for the *brute force* approach. Therefore, we use the *on-line* method instead of the *brute force* approach.

On-line generation. Here we give the numerical results for the uniform generation of paths in system with synchronisation when using the method that avoids building the whole system. For these experiments, we used the *vasy_0_1* model in which we picked at random a transition and labelled it with the synchronised label α. This modified *vasy_0_1* component was composed several times (2 to 12) with itself.

MuPAD needs $17.18s$ to compute the value of the constant ω and C for the component. The time is again measured using the Unix time function.

Table 15 summarises the time needed for the second step of the preprocessing phase : the construction of the counting tables for the languages B, C, E and T of vasy_0_1.

Table 16 contains time measurements for the computation of the numbers $l(n, m)$ and $l(n, m, i_0, i_m)$ as defined in Formula 11 and Formula 17, respectively.

Last, Table 17 presents the time used to generate 100 paths with some synchronisations.

Again, the method is feasible and efficient for very large models. However, there are some limitations: with $1GB$ of memory, it is not possible to generate paths of lengths 600+.

5.2 Uniform generation of paths and testing programs

Here we present some experiments in the area of program testing. The objectives of this campaign were the following : first evaluate the fault detection power of the approach, second study the stability of this detection power w.r.t. randomness.

We have developed a prototype, AuGuSTe, which has been used for testing some C functions extracted from an industrial application. This test suite is a part of the one used in [32]. In that paper Thevenod-Fosse and Waeselynck presented an experimental evaluation of their statistical structural testing method [31]. We used the same sets of mutants (see below), and the same set of experiments as in [32] was replayed with some minor differences due to the evolution of the C compiler and of the operating system in the last years.

Below we briefly recall the principles of mutation testing. Then, we present the prototype and the context of the experiments: the programs under test and their mutants, the considered coverage criteria, and the number of performed tests.

5.2.1 Mutation testing

Classically, mutation testing is used as a selection method [4], but it can be also used to evaluate the efficiency of dynamic testing generation methods. The idea is to create clones of the program under test where one elementary error is introduced. These clones are called mutants.

A test data set, and by extension the method which created this set, is evaluated by measuring the proportion of mutants which are killed. A mutant is said to be killed when the program and the mutant have different outputs. This proportion of killed mutants is called the mutation score [4]. It is a number between 0 and 1: a high mutation score indicates that the test data set has been very good at detecting the faults in the mutants.

5.2.2 The AuGuSTe prototype

The prototype has been developed for experimenting the method described in Section 4. Its modular architecture allows for an easy switch of the programming language of the programs to test, the constraint solver and the distribution on the elements to be used.

AuGuSTe takes four input data: a program under test P, a coverage criterion C, a number of tests to be generated N, and a maximal length of paths n. Currently, the program P is written in a simple imperative language inspired from C and Pascal. The basic construc-

length	100	200	300	400	500	600
	0.24s	0.48s	0.73s	0.98s	1.27s	1.51s

Table 15. Preprocessing phase : time to build the 2 counting tables for the number of paths leaving each state of the module vasy_0_1 with a unique synchronised transition.

length	100	200	300	400	500	600	700
	0.91s	7.88s	28.28s	66.93s	134.77s	239.71s	✘

Table 16. Preprocessing : time to compute values of $l(n, m, i_0, i_m)$ and $l(n, m)$.

length # components	100	200	300	400	500	600
2	0.57s (0.60)	1.17s (0.10)	2.05s (0.05)	4.97s (0.03)	7.16s (0.08)	9.50s (0.29)
4	1.09s (0.09)	1.71s (0.17)	3.96s (0.12)	6.09s (0.23)	13.37s (0.68)	18.38s (0.18)
6	1.53s (0.17)	2.44s (0.23)	5.87s (0.24)	9.15s (0.14)	13.99s (0.09)	27.01s (0.95)
8	1.93s (0.09)	3.22s (0.12)	5.64s (0.07)	11.92s (0.22)	18.05s (0.10)	25.53s (0.28)
10	2.38s (0.08)	4.01s (0.09)	6.85s (0.16)	10.47s (0.22)	23.04s (0.10)	32.18s (0.09)
12	3.01s (0.05)	4.80s (0.14)	8.17s (0.17)	12.54s (0.14)	27.29s (0.12)	38.38s (0.39)

Table 17. Generation : average time (and standard deviation) to generate 100 paths in composed systems with synchronisation (the modified version of *vasy_0_1* is composed with itself).

tions are sequential composition, *If...Then...Else* construction (*Else* is optional), *While* loop and *For* loop. The data types we consider are booleans, integers, arrays of booleans and arrays of integers. The criterion C is chosen among "all paths of length $\leq n$", "all branches" and "all statements". Then, AuGuSTe draws the paths and computes the corresponding input data.

AuGuSTe proceeds in three main steps: the analysis, the paths generation and the resolution.

The analysis step builds the control graph G of the program P, and the necessary automata required by the chosen criterion. If C is "all statements" (resp. "all branches") then the distribution on the nodes (resp. edges) which is a linear programming system is built and solved by an optimisation function using a simplex algorithm of MuPAD.

The path generation step is performed in one or two steps as described in Section 4.

Finally, the resolution step builds the predicates corresponding to each path and then tries to solve them. Each path predicate, which is a conjunction of boolean expressions, is translated into logical constraints and a constraint solver package is used to compute a solution of the resulting constraint system. This package is borrowed from the GATeL tool [24]. This solver uses randomised resolution i.e. variables are randomly instantiated [15]. This kind of resolution has two advantages. First, when a path is generated several times, the solver very lilely yields different input data to execute this path, something really important in software testing. Second, whenever the resolution of a predicate does not succeed, if this predicate actually has a solution, it is more likely that this solution will be obtained if the same path is generated again.

When an unfeasible path is detected (or suspected) by the constraint solver, it is rejected and another path is drawn. This so-called rejection strategy does not affect the uniform distribution on paths: feasible paths are still drawn with uniform probability.

5.2.3 Programs under test

The experiments were performed on four C functions that are part of an industrial software. We used 2914 mutants of these functions created by the SESAME mutation tool[32].

The functions belong to a component, extracted from a *nuclear reactor safety shutdown system*, which periodically scans the position of the reactor's control rods[32]. At each operating cycle, 19 rod positions and some general

control data on the hardware device are processed. This data acquisition is performed by two functions FCT1 and FCT2. After acquisition, a filtering process is performed by an other function (FCT3) in order to detect and eliminate doubtful measures (for instance, by checking the parity bit). Finally, measures are converted by a function called FCT4 into a sequence of mechanical steps. In this paper, we will only consider the experiments on functions FCT1 and FCT4.

Table 18 gives main characteristics for each function i.e. its number of code lines, and the number of paths (∞ if there is a loop), of blocs (maximal sequence of statements without choice points), of edges and of choice points (*While*, *IfThen*, *IfThenElse*) in its control graph.

For each function, the number of mutants is different according to the code complexity and the length: there are 279 mutants of FCT1 and 605 of FCT4.

5.2.4 Coverage criteria and test quality

Since FCT1 has a finite number of paths (there is no loop), we were able to use the strongest structural criterion for it, namely "all paths", whereas for FCT4 we choose to use (as it is also done in [32] the weaker, but feasible, "all branches" criterion.

The number of tests needed for each function was calculated in order to obtain a test quality q_C of 0.9999. Table 19 summarises the number of runs we perform for each function in order to ensure the good coverage.

The program, FCT4 contains both a loop and a huge number of unfeasible paths. The coverage criterion considered for the experiments was "all branches". The maximal length of paths was 234 (in number of edges of the control graph, thus much more in number of statements). The length 234 was chosen according to the characteristics of the loop. With such a length, the predicates to be solved were rather long too: on the average they were conjunctions of 190 conditions.

In order to reduce the number of unfeasible paths, we adapted the automaton according to the characteristics of the feasible paths [14]. This manipulation dramatically decreased the proportion of infeasible paths from $\frac{1}{1000}$ to $\frac{1}{2}$. It reduced the test generation time (drawing paths and solving predicates) but increased the time of the preprocessing stage (construction of the automaton and counting the number of paths); this was largely compensated by the reduction of the number of unfeasible path rejections.

	FCT1	FCT4
♯lines	30	77
♯paths	17	∞
♯blocs	14	19
♯edges	24	41
♯choice pts	5	10

Table 18. Main characteristics of FCT1 and FCT4

	FCT1	FCT4
criterion	all paths	all branches
♯runs	1	5
♯tests N per run	170	850

Table 19. Number of tests

The mutation scores are presented in Table 20. The results are significantly better than with uniform random testing on inputs (first line). They are comparable to the ones reported in [32] for a different statistical structural testing method, which has been proposed in the 90's by Thevenod-Fosse and Waeselynck [31]. The main difference with our approach is that it is based on drawing inputs, with an explicit construction of an input distribution that takes into account the structure of the program. This construction cannot be automated in presence of loops and is performed by successive experimental refinements. Our approach is based on drawing paths and has the advantage of being fully automated.

6 Related Work

Recently, various authors have addressed the issue of improving random testing by taking into account coverage issues. Some approaches consider program coverage, some consider model coverage. In those considering programs, significant experimental results have been reported by combining dynamic and symbolic evaluations. We first report on this class of work.

Directed Automated Random Testing (DART) [13] is a method and a tool recently proposed by Godefroid et al., which combines static and dynamic program analysis for automatically testing software. It is similar to ideas proposed ear-

	FCT1	FCT4		
		min	ave	max
uniform testing[32]	1	0.8950	na	0.9150
structural statistical testing[32]	1	0.9898	0.9901	0.9915
AuGuSTe	1	0.9854	0.9854	0.9854

Table 20. Mutation scores

lier by Ferguson and Korel in [8]. A DART directed search attempts to sweep through all the feasible execution paths of a program using dynamic test generation: the program under test is first executed on some random well-formed input; symbolic constraints on inputs are gathered at conditional branches during that run; then a linear constraint solver is used to generate variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. This process is repeated until all feasible program paths of the program are executed, while detecting various type of errors using run-time checking tools.

The only form of randomness that is used by DART concerns the inputs: the test driver is initialised by random values. The directed aspect of the method is deterministic.

The basic idea of DART, i.e. the combination of dynamic testing and symbolic evaluation, has been at the origin of different variants and extensions in order to palliate the main drawback of this approach, that is the systematic execution of all feasible program paths that leads to some explosion of the number of tests, or even non termination of DART when there are some loops. All these variants make use, at different levels, of random selection of some inputs, but there is no random generation of paths.

In the area of random walk in concurrent systems, K. Sen [29] recently proposed a new way for effective random testing of concurrent programs, based on partial order reduction methods. Such methods exploit the fact that among the traces of a concurrent system, a number of interleavings are equivalent to each other because they correspond to the different executions orders of various independent instructions from concurrent threads. K. Sen has designed a novel algorithm (RAPOS) which aims at sampling partial orders more uniformly than the classical random selection of traces where successors are drawn uniformly at random. In [29], some experiments are reported. But, as the author says, it is not clear how to mathematically show that this method samples partial order more uniformly.

Object-oriented programs and models call for new notions of coverage and randomness. It has been recognised for a while that method interactions and dependences are an adequate level for defining test criteria. In [27], C. Pacheco et al. present a technique that improves random test generation of sequences of method calls. Sequences are built in an incremental way, alternating phases of random generation and test executions as follows: Feedback from previous test sequences execution is used for producing new test sequences: A method is drawn at random and appended to some previous test sequences that have shown to be extensible, i.e. able to lead to new and legal object states. Such objects are used as inputs for the new method. This method is a kind of random walk among feasible sequences of method calls. There is no coverage consideration. It is implemented by a random tester for object-oriented programs (RANDOOP).

In the area of model checking a few studies have been led to explore the introduction of random explorations in model-checkers.

Monte-Carlo Model Checking, presented by Grosu and Smolka [17] is an approximate method for model checking inspired from the work of Herault et al[18]. It uses path generation by a classical random walk on the transition graph. The main advantage of this approach is the following fact: the randomized algorithm takes also as input an approximation parameter and a confidence parameter which measure the quality of error detection. However, the drawback is that the random path generation is not uniform, as mentioned in the introduction for classical random walks.

Another approach, reported by Dwyer and al. in [7], is based on concurrent randomised

depth-first searches on models extracted from Java programs. The goal is rather different from ours, since it aims at speeding up the first error finding: as soon as one of the concurrent search reaches a counter-example, the other ones are stopped. The implementation is based on Java Path Finder [33] and makes use of the JPF's RandomOrderScheduler. The experimental results show a significant gain in time to reach the first error. Other work for DFS improvements based on various heuristics are reported by Rungta and Mercer in [28].

A more similar approach to our work is the "hit or jump" test generation method [2]. It addresses the problem of testing an embedded component in a system described as a set of communicating extended finite state machines. It is based on a kind of randomised DFS biased in order to cover all the transitions of the component under test.

Namely, the test generation is based on successive depth-bounded DFSs: when some transition of interest is hit during such a DFS, a new DFS is started from its target state; if it is not the case, one of the leaves of the DFS is chosen uniformly at random to start the next DFS. Experimental results show that this kind of random exploration avoids to be trapped in zones where there are no transitions of interest. However, its mathematical analysis remains to be done.

7 Conclusion

In this paper we have introduced several new ideas on the use of uniform random paths generation in model exploration and software testing, and we give some first experimental results.

We have presented and implemented uniform paths generation in single models, and on this basis we have developed a "on-line generation" method which makes it possible to cope with large composed models, avoiding the construction of the global model.

In case of asynchronous composition, we have handled models up to 10^{27} states and it would be possible to do more. However, for us, the main interest of asynchronous composition was its use for dealing with synchronous compositions. Then, we have started to study how to deal with such compositions of models, using either a brute force approach, where the synchronous product is built, or a similar "on-line" approach to what was done for asynchronous products, avoiding the construction of the synchronous product. From our first experiments, it turns out that the brute force approach is feasible for composed systems where there are few non-synchronised transitions. Concerning the "on-line" method, we have presented its generalisation to the case where there is one synchronised transition. The generalisation to several synchronisations is sketched in [26] and is the topic of an on-going Ph.D. thesis. Preliminary results on complexity let think that it will be practicable in the case of a small number of synchronisations only. Thus it is very likely that brute-force and on-line methods need to be combined in a way depending on architecture patterns of the global systems. The determination of these patterns (including pathological ones that may be out of the scope of the method) is part of this on-going work.

We saw that, up to now, the random generation algorithm fails for large component size $|S|$ and large path length n, by lack of memory. Indeed, the memory requirement is $n \times |S|$ integer numbers. However, there are at least two approaches for solving this problem. On one hand, the counting table of the preprocessing stage could be computed *on line*, needing a memory requirement in $O(|S|)$ only [19]. On the other hand, under certain conditions, the Boltzmann generation method [6] allows to generate uniform random paths without computing a counting table. We are currently investigating in both directions.

In another part of the paper we have studied how to take into account weaker coverage criteria when drawing paths randomly, introducing a notion of randomised coverage satisfaction. We have proposed a two-step drawing strategy: first, one element to be covered is drawn at random with a suitable probability distribution; second, a path that goes through this element is drawn uniformly at random. We have shown how to choose the probability distribution of the first step in order to both maximise the minimum probability to reach an element to be covered, and ensure that any path going trough such an element has a non-null probability. This method has been implemented and applied to a C program that had the advantage to be provided with a set of mutants: this made it possible to assess the de-

tection power of the method and to favorably compare it to other random testing methods.

This paper present several first contributions on the application of the corpus of knowledge that has been developed on counting and generating combinatorial structures. They open numerous perspectives in the area of random testing, model checking, or simulation of protocols and systems that involve many distributed entities, as it is often the case in practice.

Acknowledgements We warmly thank Frédéric Magniez and Michel de Rougemont for fruitful discussions on the notion of randomised coverage satisfaction.

References

1. A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.
2. A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaïdi. Hit-or-Jump: An algorithm for Embedded Testing with Applications to In Services. In Jianping Wu and al., editors, *Proceeding of IFIP International conference FORTE/PSTV'99*, pages 41–56, Beijing, China, October 1999.
3. C. Creutzig and W. Oevel. *MuPAD Tutorial*. Springer, second edition, 2004.
4. R.A. DeMillo. Mutation analysis as a tool for software quality assurance. In *Proceedings COMPSAC'80*, pages 390–393, 1980.
5. A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34. IEEE Computer Society, 2004.
6. Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability & Computing*, 13(4-5):577–625, 2004.
7. M.B. Dwyer, S.G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 3–12, 2007.
8. Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
9. P. Flajolet and R. Sedgewick. *Analytic combinatorics*. Cambridge University Press, 2008. Currently available on the web.
10. P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of Labelled Combinatorial Structures. *Theoretical Computer Science*, 132:1–35, 1994.
11. H. Garavel and N. Descoubes. Very large transition systems. http://tinyurl.com/yuroxx, July 2003.
12. H. Garavel, F. Lang, and R. Mateescu. An overview of cadp 2001. Technical Report 0254, INRIA, 2001.
13. 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.
14. S.-D. Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique*. PhD thesis, Université Paris-Sud 11, LRI, june 2004.
15. S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *ASE*, pages 5–12. IEEE Computer Society, 2001.
16. Torbjörn Granlund. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org/manual/.
17. R. Grosu and S.A. Smolka. Monte Carlo model checking. In N. Halbwachs and L.D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.
18. T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In B. Steffen and G. Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2004.
19. T.J. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4):645–655, 1983.
20. D. Hogrefe. OSI Formal Specification Case Study: The Inres Protocol and Service (revised). Technical Report IAM-91-012, Institut fur Informatik, Universitat Bern, May 1991.
21. J.E. Hopcroft, R. Motwani, and J.D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.
22. R. M. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989.
23. D E. Knuth L. Graham and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1994. Second edition.
24. 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.
25. J. Maurer, D. Abrahams, B. Dawes, and R. Rivera. Boost random number li-

brary. http://www.boost.org/libs/random/, June 2000.
26. Johan Oudinet. Uniform random walks in concurrent models. Master's thesis, Université Paris-Sud 11, LRI, September 2007.
27. C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 75–84, 2007.
28. Neha Rungta and Eric G. Mercer. Generating counter-examples through randomized guided search. In *SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007.
29. K. Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, New York, NY, USA, 2007. ACM.
30. 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. Rapport L.A.A.S. No89043.
31. 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.
32. P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. Software statistical testing. In B.Randell, JC.Laprie, H.Kopetz, and B.Littlewood, editors, *Predictably dependable computing systems*, ISBN 3-540-59334-9, pages 253–272. Springer, 1995.
33. Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
34. H.S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.