Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

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

February 17, 2011

Abstract. This paper presents several randomised algorithms for generating paths in large models according to a given coverage criterion. Using methods for counting combinatorial structures, these algorithms can efficiently explore very large models, based on a graphical representation by an automaton or by a product of several automata. This new approach can be applied to random exploration in order to optimise path coverage and can be generalised to take into account other coverage criteria, via the definition of a notion of randomised coverage satisfaction.

Our main contributions are: a method for drawing paths uniformly at random in composed models, i.e. models that are given as products of automata, first without and then with synchronisation; a new efficient approach to draw paths at random taking into account some coverage criterion.

Experimental results show promising agreement with theoretical predictions and significant improvement over previous randomised approaches. This work opens new perspectives for future studies of statistical testing and model checking, mainly to fight the combinatorial explosion problem.

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.

Random methods can be classified into three categories : those based on the input domain of the system under test, those based on some knowledge of its environment, and those based on some model of its behaviour.

In the first case, classical random testing (as studied by Duran and Ntafos in [13,14]) consists in selecting test data uniformly at random from the input domain of the program. In some variants, some knowledge on the input domain is exploited, for instance to focus on the boundary or limit conditions of the software being tested [39].

In the second case, the selection is based on an operational profile (an estimate of the relative frequency of use of various inputs to the program). Such testing methods are called statistical testing. They can serve as a statistical sampling method to collect failure data for reliability estimation (for a survey see [35], and [38] for more recent results).

In the third case, some graphical description of the behaviour of the system under test is used. Random walks [2] are performed on the set of paths of the description. Then inputs that trigger the executions of these paths are selected. This is in the line of testing methods

developed early in the area of communication protocols [47,34]. More recently, random walks have been used for model checking [26]. Classical random walk methods, sometimes called isotropic, progresses from one state by drawing among the successors uniformly at random. A serious drawback is that in case of irregular topology of the underlying graph, uniform choice of the next state is far from being optimal from a coverage point of view. We come back to this point in Section 2.

As a matter of fact, when a graphical model of the system under test is available (either as the functional level or at the code level), most testing or checking methods are based on coverage criteria. A coverage criterion defines a set of elements of the graph that must be covered by the execution of at least one test. In practical testing context, coverage criteria are often used to assess the quality of a given test set. Such criteria provide information about how well the system under test has been exercised and may lead testers to design additional test cases.

In this paper, we study the combination of coverage criteria with the third kind of random methods mentioned above. Namely, we develop some methods for selecting paths at random in a model. The selection is biased toward a coverage criterion.

We use methods for counting and generating combinatorial structures. They make it possible to efficiently draw paths uniformly at random in large graphs. This lead to random exploration methods that optimise path coverage. Then, we show how to combine other coverage criteria and randomness, introducing a notion of coverage-guided random exploration.

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.

The paper is organised as follows. After this introduction, we develop and explain the motivations of our work in Section 2. Then, Section 3 recalls briefly some classical definitions, namely labelled transition systems, control flow graphs, and automata.

In Section 4, we first present a method for drawing paths uniformly at random in a single automaton (Section 4.1). Then we address the new problem of the uniform exploration of composed models (Section 4.2), i.e. models that are given as products of automata, first without and then with synchronisation. We present algorithms for solving this problem.

In Section 5 we study the problem of drawing paths at random taking into account some coverage criterion: after some preliminaries, we define in Section 5.2 a notion of random path generation biased towards a coverage criterion. We develop a new approach to obtain a given approximation of the coverage and we show how to implement it.

In Section 6 we report several experimental results, first on random generation of paths in Labelled Transition Systems (in Section 6.1), second on statistical testing of some C programs (in Section 6.2).

Section 7 discusses some related works, and Section 8 gives some conclusions and perspectives.

2 Motivations

As said above, in this paper we study how to draw paths, with a coverage criterion as a guide. However, in this section we first come back to some existing methods where randomness is used for input selection, and coverage criteria are taken into account, either *a posteriori* or *a priori*. Then we come to random path selection, which naturally emerges when coverage is considered. We develop the need for improvement of classical isotropic random walk.

2.1 Random input selection and coverage

A basic way to combine random testing and coverage criteria is to proceed to random input generation and then assess the achieved coverage a posteriori. Of course, the choice of the distribution on the input domain is an important issue. Drawing uniformly at random from the input domain seems attractive because it is easy to implement in most cases, or more exactly to implement good approximations. However, classical results on its use for program testing, first by Duran and Ntafos [13,14], and then others, showed that its detection power is very variable and related to the degree of achieved code coverage. Since these first experiments, it has been generally observed that this type of testing suffers from the so-called "diminishing return phenomenon": some faults are discovered

by the first experiments, but then the detection power decreases, and the method becomes ineffective. This is due to the existence in most programs of special cases that correspond to few inputs and have a low probability to be triggered by this method.

Another way is to consider a partition of the input domain, or a decomposition into subdomains[1], possibly induced by a coverage criterion[2], and then to draw at random some inputs from each of them. As said in [27], it is a way to force diversification in the choice of test inputs. And when a coverage criterion is the basis of the subdomains, it forces diversification of the parts of the program that are traversed since at least one test input must be selected from each subdomain. The big issue with partition testing is the non homogeneity of the subdomains with respect to failures: homogeneous subdomains would be such that either all inputs yield a failure, or none. It is unrealistic to imagine that they could be discovered before testing, at the stage of the definition of the testing strategy. An attempt to partially deal with this issue is to select several inputs from each subdomains, often by drawing them uniformly at random.

Various works have been devoted to the comparison of variants of such methods with what we call pure random testing (i.e. uniform drawing from the whole input domain). For instance, in [27], Gutjahr has compared the fault detection capability of pure random testing with partition testing when one input is drawn uniformly at random from each subdomain. He introduces random variables for modelling the failure rates of each partition, and shows that in the special case where the failures rates are sort of balanced (independent with equal expected values) partition testing is better or the same as pure random testing. In [48], Weyuker and Jeng have shown that if all subdomains have equal size, if an equal number of inputs is selected from each of them, subdomain testing has a better or equal probability of failure detection. In [7], Chen and Yu have introduced the idea of "proportional partition testing", where the number of selected inputs in a subdomain is proportional to its size. These results and several other ones have been collected and analysed by Ntafos in [36]. His moderate conclusion is that the subdomain testing strategies mentioned above are likely to perform better than pure random testing, with respect to fault detection. However, citing him, "the real issue has always be cost-effectiveness". Generally, subdomain testing is more effort-consuming than pure random testing. It means that using random testing enriched by a small number of additional cases to cover low probability subdomains may be preferable to subdomain testing. It also means that providing an efficient way of discovering partitions or subdomains and implementing the corresponding testing strategies would be of great interest.

An interesting attempt to take into account coverage criteria without making explicit the induced subdomains was proposed in [44, 45], where Thévenod-Fosse and Waeselynck developed what they called a statistical testing method: the input distribution considers a coverage criterion to avoid the existence of low probability cases. The authors have reported several experiments, which led to the conclusion that their approach has a better fault detection power than pure random testing and than 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 in the model. For large programs, or in presence of loops, the construction is empirical, based on preliminary observations of the behaviour of the program [45].

To avoid this explicit construction of some input distribution, we have developed another approach where the random space is no more the input domain but the paths of a graphical model of the system under test. It makes it easier to combine coverage criteria and random testing. Moreover, as the coverage criterion is taken into account in the definition of the distribution, the achieved coverage can be estimated a priori for a given number of tests.

2.2 Random selection of paths

2.2.1 Uniformity

The most demanding coverage criterion is to go through all paths, where the underlying fault assumption is that any path is likely to produce

[1] The difference between partitions and subdomains is that the first one are required to be disjoint; this point is not relevant in the present discussion and we use both terms indifferently.

[2] Each element to be covered defines a sub-domain that is the set of inputs that cause its execution

a failure. However, this criterion is most of the time impracticable because of the exponential number of paths. Thus this criterion is a natural candidate for combination with randomness. When deciding on a distribution on paths, according to the above assumption, one must take into account the fact that every path must have a non-null probability. In a testing or a checking context, where the aim is fault detection, one must maximise the probability to cover every path. This leads to the choice of a uniform distribution on paths and to the introduction of a (high) bound on their lengths when there are infinite paths, as it is unavoidable when testing.

In the case of other coverage criteria, the choice of a distribution is more involved as it is developed in Section 5.

2.2.2 Isotropic random walk and coverage issues

A classical way to explore at random a graph is to use isotropic random walks. If we know every state of the system and their successors, and we are able to assign a probability to each successor of a state such that it sums to 1, then we can explore such systems by using random walks. In the case of isotropic random walks, every successor has the same probability. Algorithm 1 draws paths of length less than or equal to n.

Algorithm 1 Isotropic random walk

Require: a graph $\mathcal{G}$, an initial state s_0 and a length n
Ensure: a path σ such that $|\sigma| \leq n$
  $i \leftarrow 0$
  $s \leftarrow s_0$
  while $i \neq n$ and $succ(s) \neq \emptyset$ **do**
    Choose a state s' uniformly at random among the successors of s ($succ(s)$)
    $\sigma \leftarrow \sigma \cup t$ with t the arc $s \rightarrow s'$
    $i \leftarrow i + 1$
    $s \leftarrow s'$
  end while
  return σ

Since it doesn't need any knowledge but the current state and its successors, an isotropic random walk could be the ideal candidate to explore very large models at random. Unfortunately, the underlying probability distribution of paths induced by isotropic random walks is hard to know as it depends on the graph topology. Sometimes, an isotropic random walk is totally inefficient, like in the following example.

In Figure 1, the expected number N of isotropic random walks to execute before getting n different paths (of length n) is:

$$\begin{aligned} E(N) &= E(N_1) + E(N_2) + \cdots + E(N_n) \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= 1 + 2 + \cdots + 2^{n-1} \\ &= 2^n - 1 \end{aligned} \tag{1}$$

where $E(N_i)$ (resp. p_i) is the expected value (resp. probability) to get a new path after $i-1$ different isotropic random walks. In other words, Formula 1 show that isotropic random walks need an exponential amount of drawings to cover paths of the graph in Figure 1.

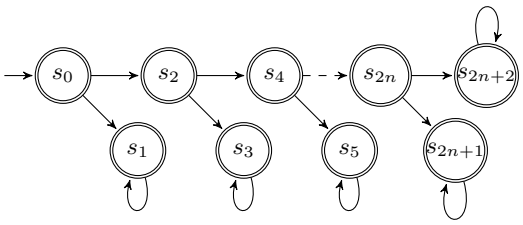

Fig. 1: A pathological graph for isotropic random walk

This exponential amount of drawings comes from the choice done by the random walk at each state. It has to choose between a state that leads to a single path and a state that leads to an exponential number of paths. But, in the case of an isotropic random walk, those two states have the same probability, hence a lot of paths have a low probability to occur in favour of a single path.

If the number of paths that start from each state is known, the random walk can be guided to balance the probability to draw every path so as to get a uniform distribution.

2.3 Contributions

In this paper, we first show how to efficiently compute numbers of paths starting from states in large models. Then we show how to use these numbers to draw paths in a way guided by coverage criteria. When the coverage criterion is

about paths, we use an exploration method with a uniform probability distribution of the set of paths to be covered, as explained in Section 4. When the coverage criterion is not about paths, but, for instance, about states or transitions, it is clear that uniform path drawing is not optimal. Then we show how to adjust it in order to get a good approximation of the required coverage, as explained in Section 5.

3 Background

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.

3.1 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 $\sigma = (s_0, a_0, s_1, \dots, s_i, a_i, s_{i+1}, \dots)$ of transitions satisfying:

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

3.2 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.

As said above, we consider control graphs with two distinguished states named s_0 and s_f: they correspond to the entry 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 [24]). 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 language. This is a well-known issue that in control graphs some paths may be unfeasible: those with a predicate that is not satisfiable because of the occurrence of contradictory conditions when traversing them. In the case of linear constraints it is possible to decide whether a path is unfeasible, and there exist good tools for that. In more realistic cases, such as the ones addressed in Section 6.2, more powerful solvers must be used [5,10,32]. In some cases they may fail to return a decision, since the general satisfiability problem is known to be undecidable. However, thank to significant progresses both on constraint solvers [4] and on heuristics for predetection of unfeasible paths [31], these failures turn out to be more and more manageable (see also our experimental results in Section 6.2).

3.3 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

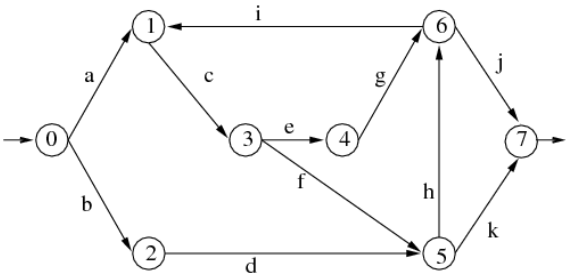

Fig. 2: A finite automaton

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 convenient when considering models of reactive systems such as LTS). Moreover, the automata we deal with have the following property : any two different transitions have distinct labels.

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.

4 Uniform path random generation

Here we are interested in drawing uniformly at random a set of paths in one or several automata that represent a model, as seen above. In Section 4.1 we suppose that there is only one automaton, and we recall a known algorithm for generating paths uniformly at random: this is a special case of a general method of generation of combinatorial structures, which has been first addressed by Wilf [49] and then generalised and systematised by Flajolet, Zimmermann and Van Cutsem [18].

Then, in Section 4.2, 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.

4.1 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 $f_s(m)$ denote 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:

$$\begin{aligned} f_s(0) &= 1 && \text{if } s \in F \\ &= 0 && \text{otherwise} \\ f_s(i) &= \textstyle\sum_{s \to s'} f_{s'}(i-1) && \text{for } i > 0 \end{aligned} \qquad (2)$$

where $s \to s'$ means that the sum is on all the states s' and all the transitions between s and

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

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

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 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 transition from s'_0 to s_0 and a loop transition from s'_0 to itself. Label both transitions with a same new letter. 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.

4.2 Composed automata

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

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

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 and uniformly generating words of a regular language can be processed, as described in Section 4.1.

Let $C_1 = \sum_{1 \leq i \leq r} |X_i|$ and $C_2 = \prod_{1 \leq i \leq r} |S_i|$. 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, randomly and uniformly generating one word requires nC_1 time requirement, after a preprocessing stage having worst-case complexity nC_1C_2 in time and in nC_2 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 $O(C_1C_2 + mnC_1)$ in time and $O(C_1C_2 + nC_2)$ in space. This is linear in n, in m, in the total size of the alphabets. However, since $C_2 = \prod_{1 \leq i \leq r} |S_i|$, the complexity is exponential according to the number of modules. Thus the algorithm will be efficient only for a small number of modules.

"On the fly" shuffling method. Here we describe an alternative method which avoids constructing the above automaton. It allows to generate random traces almost uniformly, with a time complexity linear time according to n and to r. However this method suffers from the following limitation: every word $w_i \in L_i$ that participates to the trace must be longer than a constant ν. As will be seen below, ν can be computed from the automatas of the languages L_i's.

At first 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:

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

where $\binom{n}{k_1,\ldots,k_r} = \frac{n!}{k_1!k_2!\ldots k_r}$ is a multinomial coefficient.

Now, generating a trace of length n can be done in three steps, as follows. At first, choose 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 generate each w_i independently. Finally shuffle the w_i's. That is:

1. Choose at random a *composition* of n in r parts, that is a r-tuple $(n_1, \ldots, n_r)$ such that $n_i \geq 0$ for every i and $n_1 + \ldots + n_r = n$. This compositions must be chosen with probability
$$\Pr(n_1, \ldots, n_r) = \frac{\binom{n}{n_1,\ldots,n_r} \ell_1(n_1) \cdots \ell_r(n_r)}{\ell(n)}$$
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 and in r. 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 (1) in step 1. The space requirement is $O(1)$ but, unfortunately, the number of terms that have to be computed is exponential in n. However, with additional conditions on

the n_i's and on the L_i's, there is a way of drastically simplifying the step 1, by using *asymptotic approximates*.

According to a well known result (see *e.g.* [42] or [17, Section IV.5.1]), if L is a regular language, then there exist an integer N, 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

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

The number N, 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 is *aperiodic* and *strongly connected*, then there is a 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. (An automaton is aperiodic if the greater commun divisor of the lengths of all its cycles is 1.)

Thus, if we define $\omega = \omega_i$ then we have, for any $n \geq N$,

$$\ell(n) = C\omega^n + O(\omega'^n) \qquad (4)$$

where ω' is the value of largest modulus among the ω_j's except ω. And, since $\omega > \omega'$ if follows immediately that $\ell(n) \sim C\omega^n$.

It is worth noticing that the strong connectivity condition is sufficient, yet not necessary. For instance, it suffices to have some unique biggest strongly-connected component in the automaton for Formula (4) to hold.

Now, assume that all the L_i's are such that, for all $k > N_i$,

$$\ell_i(k) = C_i\omega_i^k + O(\omega_i'^k)$$

where N_i, C_i, ω_i, and ω_i' are four constants that play the same roles, respectively, as N, C, ω, and ω' in (4). This can be restated as:

$$\begin{aligned}\ell_i(k) &= C_i\omega_i^k(1 + O(\frac{\omega_i'^k}{\omega_i^k}) \\ &= C_i\omega_i^k(1 + O(\alpha_i^k))\end{aligned}$$

where $0 \leq |\alpha_i| < 1$.

Now let us denote $\ell(n_1, n_2, \ldots, n_r)$ the number of words of L of size $n = n_1 + n_2 + \ldots + n_r$ having n_i letters in the word from L_i for every i. Then we have

$$\begin{aligned}&\ell(n_1, n_2, \ldots, n_r) \\ &\quad = C_1 \cdots C_r \binom{n}{n_1, \ldots, n_r} \ell_1(n_1) \cdots \ell_r(n_r) \\ &\quad = C_1 \cdots C_r \cdot \omega_1^{n_1} \cdots \omega_r^{n_r} \\ &\qquad (1 + O(\alpha_1^{n_1})) \cdots (1 + O(\alpha_r^{n_r})) \\ &\quad = C_1 \cdots C_r \omega_1^{n_1} \cdots \omega_r^{n_r} (1 + O(\alpha^\nu))^r\end{aligned}$$

where $\alpha = \max_i \alpha_i$ and $\nu = \min_i n_i$. And finally

$$\begin{aligned}&\ell(n_1, n_2, \ldots, n_r) \\ &\quad = C_1 \cdots C_r \omega_1^{n_1} \cdots \omega_r^{n_r} (1 + rO(\alpha^\nu))\end{aligned}$$

where $\alpha < 1$ and $\nu \leq \min_i n_i$, provided that $n_i > N_i$ for all i.

In other words, we just stated that

$$\ell(n_1, n_2, \ldots, n_r) \approx C_1 \cdots C_r \omega_1^{n_1} \cdots \omega_r^{n_r}$$

and the relative error tends to zero at an exponential rate according to ν, the minimum of the n_i's. In practice, taking for example $\nu = 2 \max_i N_i$ will suffice to get a very good approximation of $\ell(n_1, n_2, \ldots, n_r)$ in most cases.

Now, this approximation gives us a fast way to perform the step 1 of our random generation algorithm. What we have to do is to generate the composition of n, namely $(n_1, n_2, \ldots, n_r)$, with probability

$$\Pr(n_1, \ldots, n_r) \;\sim\; \frac{\binom{n}{n_1, \ldots, n_r} \omega_1^{n_1} \omega_2^{n_2} \ldots \omega_r^{n_r}}{(\omega_1 + \omega_2 + \ldots + \omega_r)^n} \qquad (5)$$

since

$$\begin{aligned}&\textstyle\sum_{k_1 + \cdots + k_r = n} C_1 \cdots C_r \binom{n}{k_1, \ldots, k_r} \omega_1^{k_1} \cdots \omega_r^{k_r} \\ &\quad = C_1 \cdots C_r (\omega_1 + \cdots + \omega_r)^n.\end{aligned} \qquad (6)$$

There is an easy algorithm for doing it without computing the formula: 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.

Now, recall that we must ensure that $n_i > \nu$ for all i. This could be done by rejecting all r-uples that do not satisfy this requirement. But it would lead to a huge number of rejects. Alternatively, a slight change in the algorithm allows to satisfy the requirement without any reject. It is based on the fact that the distribution of compositions of n into r parts larger than ν is equal to the distribution of compositions of $n - r\nu$ into r parts. Thus the algorithm becomes: take

the set of integers $\{1, \ldots, r\}$ and draw a random sequence by picking independently $n - r\nu$ 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, plus ν.

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 and r, polynomial according to the maximum of $|X_i|$ and to the maximum of $|S_i|$, in time as well as in space requirements.

4.2.2 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 every 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 $X_{i,\alpha} = X_i \backslash \{\alpha\}$. The synchronised product [3] 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 represents the language L of synchronised traces. Once it has been built, the generation process is exactly as in Section 4.2.1. 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 Section 4.2.1). However, in presence of many synchronisations the brute force method turns out to be exploitable (see Section 6.1.4) since the reachable state space of the synchronised product remains of reasonable size.

"On the fly" generation of synchronised traces. Here we present an algorithm for, given n and m, uniformly generating random synchronised traces of length n with m synchronisations, avoiding the construction of the synchronised product.

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

$$L = B.(\alpha.C)^*.\alpha.E \ \cup \ D\,. \tag{7}$$

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 $w = w_0\alpha w_1\alpha\ldots\alpha w_m$ where $w_0 \in B$, $w_i \in C$ for any $1 \leq i < m$, and $w_m \in E$. Finally, let $\ell(i_0, i_1, \ldots, i_m)$ be the number of such words such that the length of each w_j equals i_j, for all $0 \leq j \leq m$. Then we have

$$\ell(n,m) = \begin{cases} d(n) & \text{if } m = 0, \\ \displaystyle\sum_{\substack{i_0+\cdots+i_m= \\ n-m}} \ell(i_0,\ldots,i_m) & \text{else,} \end{cases} \tag{8}$$

where

$$\ell(i_0,\ldots,i_m) = b(i_0)c(i_1)\cdots c(i_{m-1})e(i_m). \tag{9}$$

Now we can present the algorithm. At first let us remark that if $m = 0$ we are in the case where there is no synchronisation, as in the previous section. Thus we can suppose that $m > 0$.

1. Compute the $b(i)$'s, $c(i)$'s, and $e(i)$'s for all $0 \leq i \leq n-m$. As each of the languages B, C, and E is a shuffle of r languages without synchronisation, these coefficients can be computed by using one of the algorithms of Section 4.2.1, knowing that using the *on the fly* algorithm requires additional hypotheses on the involved automata and path lengths.
2. Using Formula 9, compute all the $\ell(k_0,\ldots,k_m)$ for all $(m+1)$-tuples $(k_0,\ldots,k_m)$ such that $k_0+\cdots+k_m = n-m$. In the same time, compute and keep in memory the sum

$$\ell(n,m) = \sum_{k_0+\cdots+k_m=n-m} \ell(k_0,\ldots,k_m)$$

and all the partial sums

$$S(i_0,\ldots,i_s) = \sum_{\substack{k_{s+1}+\cdots+k_m= \\ n-m-i_0-\cdots-i_s}} \ell(i_0,\ldots,i_s,k_{s+1},\ldots,k_m)$$

for each $0 \leq s < m$ and for each $(s+1)$-tuple $(i_0,\ldots,i_s)$. This can be done in $\mathcal{O}((n-m)^m)$ time and space requirements. It is worth noticing that these two first steps have to be done only once, whatever the number of traces to be generated afterwards.
3. Choose a $(m+1)$-tuple $(i_0,\ldots,i_m)$. For ensuring the uniformity of the generation of the final trace, this probability must be equal to

$$Pr(i_0,i_1,\ldots,i_m) = \frac{\ell(i_0,i_1,\ldots,i_m)}{\ell(n,m)}.$$

This can be done in an incremental way, by using the partial sums that have been computed during the previous step: choose i_0 with probability

$$Pr(i_0) = \frac{S(i_0)}{\ell(n,m)},$$

then i_1 with probability

$$Pr(i_1/i_0) = \frac{S(i_0,i_1)}{S(i_0)},$$

and follow by choosing each i_s, for $2 \leq s \leq m$, with probability

$$Pr(i_s/i_0,i_1,\ldots,i_{s-1}) = \frac{S(i_0,i_1,\ldots,i_s)}{S(i_0,i_1,\ldots,i_{s-1})}.$$

Each choice can be done in $\mathcal{O}(n-m)$ operations. Since there are m choices, the complexity of the whole step is $\mathcal{O}(m(n-m))$.
4. Now we have got the whole sequence $(i_0,\ldots,i_m)$ with a suitable probability. It remains to generate the words $w_0 \in B$, $w_1,\ldots,w_{m-1} \in C$ and $w_m \in E$, each w_k having length i_k. Each of these words is simply a shuffle of the r languages $(B_i)_{i=1\ldots r}$ if $k = 0$, $(C_i)_{i=1\ldots r}$ if $1 \leq k < m$, $(E_i)_{i=1\ldots r}$ if $k = m$. For each of the w_k's, the shuffling algorithm given in Section 4.2.1 can be used.

5 Randomised coverage criteria

5.1 Coverage criteria and randomness

As seen in Section 2, the idea of combining coverage criteria and random testing aims at overcoming some drawbacks of both approaches.

One main drawback of the classical use of coverage criteria (i.e selecting in a deterministic way one input for each element to be covered) is that the induced sub-domains are generally not homogeneous: 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, as seen in Section 2, in its pure random version it induces a bad coverage of cases corresponding to small input sub-domains.

In this section we study what it means for a random testing method to ensure a given coverage. This leads to a notion of randomised coverage satisfaction. A similar notion, called test quality for statistical testing, was defined by Thévenod-Fosse and Waeselynck in [43]. We reformulate it in our context.

Let A be an automaton describing a system under test. On the basis of this automaton, it is possible to define the usual 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 (this set is 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 criterion is a probabilistic notion. When drawing N tests, what can be said on the probability to cover $E_C(A)$? Let us note $q_{C,N}(A)$ the minimal probability of covering any element of $E_C(A)$ when drawing N tests. The value $q_{C,N}(A)$ is called the test quality of the random 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 4.1, 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:

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

q	0.9	0.99	0.999	0.9999
N	32	63	94	125

Table 2. Number of tests N required for a given test quality q w.r.t. the "all paths of length ≤ 10" criterion, for Figure 2

In the example, choosing $n = 10$ allows the coverage of all elementary paths (see Figure 2). Using Table 1, one can compute the number of paths of length less or equal to 10. there are 14 of them. Thus we have:

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

Table 2 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

$$q_{C,N}(A) = 1 - (1 - p_{min})^N \qquad (10)$$

where $p_{min} = min\{p_i | i \in (1..m)\}$. p_{min} is called the minimum reachability property by some authors [1]. Consequently, the number N of tests required to reach a given quality $q_C(A)$ is

$$N \geq \frac{log(1 - q_C(A))}{log(1 - p_{min})} \qquad (11)$$

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}.

However, maximising p_{min} must not lead to give up the randomness of the method. This may be the case when there exists one path traversing all the elements of $E_C(A)$: one can maximise p_{min} by giving a probability 1 to this path, going back to a deterministic testing method. When one wants to keep randomness of the testing method, another requirement must be combined to the maximisation 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 $E_C(\mathcal{A})$

If the main requirement of the testing campaign is to cover the elements of $E_C(\mathcal{A})$ whatever paths traversing them, then maximising p_{min} is the solution. But it may lead to give null probability to some paths traversing these elements. Conversely, if the main requirement is to explore those paths traversing some element of $E_C(\mathcal{A})$, then the best method is to draw uniformly at random among these paths. But it may lead to some weak probability for some elements, namely those reachable by few paths, thus to a weak probability to satisfy the coverage criterion. There is a bunch of solutions between those two extremes.

In the next section, we present one of these solutions that favours (item 1) and weakens (item 2) into: "any path traversing an element of $E_C(\mathcal{A})$ must have a non-null probability."

5.2 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 $\leq n$.

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 4.1 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 that will be seen in Section 5.3.

Note that with such criteria, the paths to be drawn are independent in the sense that there is a one-to-one correspondence between these paths and the elements to be covered.

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 loops, 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 *et al.* [12,24], a practical solution in two steps is proposed, namely:

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

$$p_i = \sum_{j=1}^{m} \pi_j \frac{\alpha_{i,j}}{\alpha_j} \qquad (12)$$

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

Note that with such criteria, the paths to be drawn may be no more independent in the sense that a path covering e_i may cover another element e_j. This is reflected by the occurrence of the $\alpha_{i,j}$ in the computations above.

We will see in the following section 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 12. Standard methods lead to a solution in time polynomial according to m.

However, some paths traversing an element to be covered may 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[3] ε.

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 11).

[3] 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 3. Table of the α_{ij}.

Let us illustrate this method with the example. Given the coverage criterion "all-transitions" and given $n = 10$, Table 3 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 4. Each line, but the last one, is an inequation which corresponds to a row in Table 3. 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 12. 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 10. 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 $x \in$

$$
\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 4. The linear program.

$E_C(\mathcal{A})$. In the example, with $\varepsilon = 0.0001$, the new distribution is $\pi_a = \pi_b = \pi_c = \pi_e = \pi_f = \pi_h = \pi_i = \pi_j = \frac{1}{1000}$, while $\pi_d = \frac{35729}{115200} \approx 0.3101$, $\pi_g = \frac{11867}{32000} \approx 0.3708$, and $\pi_k = \frac{179141}{576000} \approx 0.311$. This gives $p_{min} = \frac{58893}{120000} \approx 0.4908$. The new p_{min} is very close to the previous one.

The value p_{min} is computed taking into account all the paths traversing some element of $E_C(\mathcal{A})$. As mentioned above, when testing programs, there may be unfeasible paths in the control flow graph. The presence of unfeasible paths may induce some inacurracy in the maximisation of p_{min}. When exploring LTS, this problem does not occur.

5.3 Conditions on paths and operations on automata

As seen above, coverage criteria can be related to constitutive elements of the automaton, as all-states or all-transitions, or may express some constraints, as all-paths of a certain kind. For example, one can define coverage criteria such as:

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

Here we show how the theory of formal languages provides procedures for dealing with

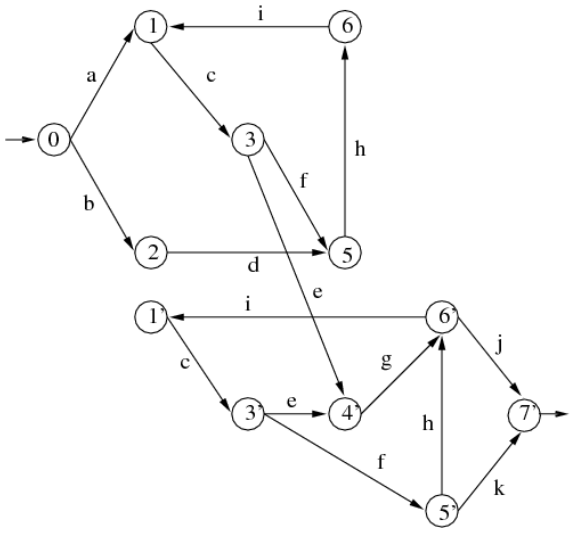

Fig. 3: An automaton that contains only the paths of the automaton of Figure 2 which cross transition labeled 'e'.

these kinds of constraints. The solution consists, given the automaton A that represents the system, in building a new automaton A' whose total set of paths is equal to the set of paths in A that satisfy the desired constraint.

All kinds of constraints enumerated above are said to be regular, because they can be expressed as a regular language L'. For example, "cross at least once the transition e" can be represented by the language $L' = X^*eX^*$, that is the language of words that contain the letter e at least once. And, obviously, $L(A) \cap L'$ equals the language that corresponds to the paths from s_0 to a final state in A that cross transition e. There are classical algorithms to construct the

automaton of the intersection of two regular languages (see e.g. [30]). Figure 3 shows the result of such an algorithm, for the automaton of Figure 2 and the constraint "cross at least once the transition e". It is known [30] that 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. Once A' has been built, the problem of generating paths subject to the given regular constraint in A simply reduces to generating paths in A'.

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 [30], the class of *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 be handled by the methods described here. An example of such a constraint is "the paths to be generated must not cross twice the same state".

6 Experimental results

In this section we present the experiments we did to prove the effectiveness of our approaches. More precisely, in Section 6.1 we give numerical results for the uniform generation of random paths in both non-synchronised and synchronised systems while in Section 6.2 we present experiments of our method for testing programs under different coverage criteria.

6.1 Uniform path generation

We present experimental results that prove the feasibility of the method we described in Section 4. 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 builds the whole system (e.g., computing the product of several components) before generating paths.

6.1.1 Implementation and methodology

For our implementation, we use several tools that we mention here: the BCG library of the CADP toolbox [20], the MuPAD computer algebra system [8], the GMP (Gnu Multiple Precision) library [25] and the random function of the BOOST library [33] 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 [19]) 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. A detail description of the models is available in Table 5. Table 6 gives the number of paths for each specified length ; those large numbers justify why we are interested in random methods.

6.1.2 Generation in single models

Here, we give results for the uniform generation of paths in a transition system described by a single component. Table 7 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.

6.1.3 Generation in composed systems without synchronisation

We present here the results for the uniform generation of paths in a system succinctly described

name	# states	# transitions	# labels	branching factor avg [min - max]
vasy_0_1	289	1224	2	4.24 [4 - 8]
vasy_1_4	1183	4464	6	3.77 [2 - 5]
vasy_5_9	5486	9676	31	1.76 [0 - 6]
vasy_8_24	8879	24411	11	2.75 [1 - 5]
vasy_10_56	10849	56156	12	5.18 [4 - 6]
vasy_12323_27667	12323703	27667803	119	2.25 [0 - 13]

Table 5. Detail description of the VLTS benchmark suite

	length					
name	200	1000	2000	3000	5000	8000
vasy_0_1	10^{121}	10^{602}	10^{1204}	10^{1806}	10^{3010}	10^{4817}
vasy_1_4	10^{97}	10^{479}	10^{957}	10^{1435}	10^{2392}	10^{3826}
vasy_5_9	10^{53}	10^{265}	10^{531}	10^{797}	10^{1328}	10^{2125}
vasy_8_24	10^{59}	10^{295}	10^{590}	10^{885}	10^{1475}	10^{2360}
vasy_10_56	10^{140}	10^{699}	10^{1399}	10^{2098}	10^{3496}	10^{5593}
vasy_12323_27667	10^{69}	10^{337}	10^{673}	10^{1009}	10^{1681}	10^{2689}

Table 6. Number of paths

length / name	200	1000	2000	3000	5000	8000
vasy_0_1	0.0s	0.9s	2.9s	6.3s	15.9s	40.1s
vasy_1_4	0.1s	1.0s	3.2s	6.7s	18.2s	✘
vasy_5_9	0.0s	0.9s	2.4s	5.2s	✘	✘
vasy_8_24	0.2s	0.8s	2.4s	✘	✘	✘
vasy_10_56	0.0s	1.3s	✘	✘	✘	✘
vasy_12323_27667	✘	✘	✘	✘	✘	✘

Table 7. Uniform generation of paths in models of various sizes: size versus time. ✘ means there is not enough memory to build the counting table

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.

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

Table 8 and Table 9 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 is measured with the timer function of the Boost library. Measurements were made 10 times each and we give the mean and standard deviation of these measures.

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 10 (resp. Table 11) 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.

6.1.4 Generation in composed systems with synchronisation

Now, we experiment the generation of paths in composed systems when the composition is synchronised. We use a real case-study to draw

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

Table 8. 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 9. 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 10. 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

paths in a communication protocol, based on a *brute force* approach (see Section 4.2.2).

We studied a well-known communication protocol: the INRES protocol [29]. 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[4].

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 12. 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 13 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 14 and Table 15 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 too large for the *brute force* approach.

6.2 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, third, study the impact of unfeasible paths on the method.

We have developed a prototype, AuGuSTe, which has been used for testing some C functions extracted from an industrial application.

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

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

Table 11. 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 12. 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 13. Description of the BCG automaton built from a LOTOS description of the INRES protocol

This test suite is a part of the one used in [45]. In that paper Thevenod-Fosse and Waeselynck presented an experimental evaluation of their statistical structural testing method [44]. We used the same sets of mutants (see below), and the same set of experiments as in [45] 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.

6.2.1 Mutation testing

Classically, mutation testing is used as a selection method [11], 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 [11]. 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.

6.2.2 The AuGuSTe prototype

The prototype has been developed for experimenting the method described in Section 5 and a detailed description of the AuGuSTe tool can be found in [23]. 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 constructions 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

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

Table 14. 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 15. Generation: average time to generate 100 paths in the INRES system. ✘ means there is not enough memory to build the counting table

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 5.

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 [32]. This solver uses randomised resolution i.e. variables are randomly instantiated [24]. 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.

6.2.3 Programs under test

We report here experiments that were performed on two out of four C functions that are part of an industrial software. We used 2914 mutants of these functions created by the mutation operators of the SESAME tool (details can be found in [9]).

The four functions belong to a component, extracted from a *nuclear reactor safety shutdown system*, which periodically scans the position of the reactor's control rods [45]. 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 consider the experiments on functions FCT1 and FCT4 only. Indeed the two other functions do not bring any useful information: FCT2 is similar to FCT1 while FCT3 is highly dependent of the environment. On the contrary, FCT4 is very interesting since it contains a loop (thus an infinite number of paths). A complete report on the experiments with the four functions is given in [23].

Table 16 gives main characteristics for each function i.e. its number of code lines, and the number of paths (∞ if there is a loop), of blocks (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.

6.2.4 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 [45]) 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 17 summarises the number of runs we perform for each function in order to ensure the good coverage. This means that for FCT1 one run with 170 tests allows to obtain the target test quality and guarantees a good coverage while 5 runs with 850 tests each are needed for FCT4.

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.

The example of FCT4 shows that the method is likely to scale up to large and realistic programs: the research space is a set of around 10^{20} paths and associated predicates contains on average 190 conjunctions.

In order to reduce the number of unfeasible paths, we adapted the automaton according to the characteristics of the feasible paths using simple data flow considerations [22]. 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.

The mutation scores are presented in Table 18. The results are significantly better than with uniform random testing on inputs (first line). It should be noted that the average value for FCT4 in the case of uniform testing is not available in [45]. However, this value is less than 0.915, thus it remains less effective than our methods. They are comparable to the ones reported in [45] for a different statistical structural testing method, which has been proposed in the 90's by Thevenod-Fosse and Waeselynck [44]. 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. However, it requires the use of some constraint solver for getting the test inputs. As we have noted above, the technology of constraint solvers is progressing fast. During the experiments reported here, we were faced with very few non decisions of the GATeL randomised constraint solver (i.e. leaving a solving attempt because of a time-out). It is very likely that with recent, more powerful SMT solvers [10,5] the situation would be even better.

	FCT1	FCT4
♯lines	30	77
♯paths	17	∞
♯blocks	14	19
♯edges	24	41
♯choice pts	5	10
♯var: bool	11	5
♯var: int	6	10
♯var: array of bool	6	3
♯var: array of int	0	3

Table 16. Main characteristics of FCT1 and FCT4

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

Table 17. Number of tests

7 Related Work

In Section 2 we have examined several classical approaches for improving the detection power of subdomain testing by combining it with some random input selection. Various authors have addressed the different problem 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) [21] is a method and a tool proposed by Godefroid et al., which combines static and dynamic program analysis for automatically testing software. It is similar to ideas proposed earlier by

	FCT1	FCT4		
		min	ave	max
uniform testing[45]	1	0.8950	na	0.9150
structural statistical testing[45]	1	0.9898	0.9901	0.9915
AuGuSTe	1	0.9854	0.9854	0.9854

Table 18. Mutation scores

Ferguson and Korel in [16]. 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, which is very demanding, while detecting various type of errors using run-time checking tools.

The only form of randomness that is used by DART concerns the first inputs: the test driver is initialised by random values. The directed aspect of the method is deterministic. Some limitation on the form of the programs is introduced from the fact that a linear constraint solver is used.

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: it 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, Sen [41] has 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. 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 [41], 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 dependencies are an adequate level for defining test criteria. In [37], 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: 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. Feedback from previous test sequences execution is used for producing new test sequences. This method can be seen as random walks 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 [26] is an approximate method for model checking inspired from the work of Herault et al [28]. It uses path generation by a classical isotropic 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. Thus some paths may have a very low probability.

Another approach, reported by Dwyer and al. in [15], 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 [46] 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 [40].

A more similar approach to our work is the "hit or jump" test generation method [6]. 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.

8 Conclusion

In this paper, we have introduced two main ideas: the first one is to use uniform path generation for very large model exploration and software testing; the second one is to combine coverage criteria and random exploration. These methods are based for the first part on algorithms for counting combinatorial structures and for the second part on a careful analysis of the new notion of randomised coverage criterion.

In a first part of the paper, we have presented, and efficiently implemented, uniform paths generation in single models, and then, on this basis we have developed a "on the fly generation" method which makes it possible to cope with large composed models, avoiding the construction of global models.

In case of non synchronised composition, we have handled models up to 10^{27} states and it would be possible to do more.

Then, we have studied how to deal with synchronised compositions of models, first using a brute force approach, where the synchronised product is built explicitly.

The brute force approach is feasible for composed systems where there are many synchronised transitions. On the contrary, when there are few synchronised transitions, the size of the global model explodes, suggesting the use of some "on the fly" method. However, the full transposition of the "on the fly" method from the non synchronised product to the synchronised product is made problematic. The approximations used in the non synchronised case are not directly applicable. Thus it is very likely that brute-force and "on the fly" 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 approach) is part of on-going work.

In another part of the paper we have studied how to take into account weaker coverage criteria when drawing paths randomly, giving up global uniformity on paths, and introducing a notion of randomised coverage satisfaction of elements of the graph such as states, transitions, etc. The definition of the corresponding testing methods requires some compromise between maximising the minimum reachability property for the elements to be covered, and uniformly covering those paths that traverse one such element. The first point favors classical coverage satisfaction and may lead to absence of randomness in path selection. The second point favors random exploration of all the paths that may contribute to coverage. Among the possible trade-offs, we have proposed and implemented 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 with the help of a powerful constraint solver and applied to C programs.

This paper presents several original applications 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.

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

References

1. N. Abed, S. Tripakis, and J.-M. Vincent. Resource-aware verification using randomized exploration of large state spaces. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *Lecture Notes in Computer Science*, pages 214–231, 2008.
2. D. Aldous. An introduction to covering problems for random walks on graphs. *J. Theoret Probab.*, 4:197–211, 1991.
3. A. Arnold. *Finite Transition Systems*. Prentice-Hall, 1994.
4. C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.
5. F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *SMT '08/BPR '08: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 1–5, New York, NY, USA, 2008. ACM.
6. 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.
7. T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Trans. Softw. Eng.*, 20(12):977–980, 1994.
8. C. Creutzig and W. Oevel. *MuPAD Tutorial*. Springer, second edition, 2004.
9. Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell. The SESAME experience: from assembly languages to declarative models. In *MUTATION '06: Proceedings of the Second Workshop on Mutation Analysis*, page 7, Washington, DC, USA, 2006. IEEE Computer Society.
10. L. Mendonça de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, 2008.
11. R.A. DeMillo. Mutation analysis as a tool for software quality assurance. In *Proceedings COMPSAC'80*, pages 390–393, 1980.
12. A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE*, pages 25–34. IEEE Computer Society, 2004.
13. J.W. Duran and S.C. Ntafos. A report on random testing. *5th IEEE International Conference on Software Engineering*, pages 179–183, 1981.
14. J.W. Duran and S.C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10:438–444, 1984.
15. 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.
16. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
17. P. Flajolet and R. Sedgewick. *Analytic combinatorics*. Cambridge University Press, 2008. Currently available on the web.
18. 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.
19. H. Garavel and N. Descoubes. Very large transition systems. http://tinyurl.com/yuroxx, July 2003.
20. H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. Technical Report 0254, INRIA, 2001.
21. 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.
22. S.-D. Gouraud. *Utilisation des Structures Combinatoires pour le Test Statistique*. PhD thesis, Université Paris-Sud 11, LRI, june 2004.
23. S.-D. Gouraud. Auguste: a tool for statistical testing. *LRI research report RR-1400, Université de Paris-Sud 11*, 2005.
24. 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.

25. Torbjörn Granlund. GNU MP: The GNU Multiple Precision Arithmetic Library. http://gmplib.org/manual/.
26. R. Grosu and S.A. Smolka. Monte Carlo model checking. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.
27. W. J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Software Eng.*, 25(5):661–674, 1999.
28. 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.
29. 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.
30. 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.
31. A. S. Kalaji, R. M. Hierons, and S. Swift. Generating feasible transition paths for testing from an extended finite state machine (EFSM). In *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 230–239, Washington, DC, USA, 2009. IEEE Computer Society.
32. B. Marre and B. Blanc. Test selection strategies for Lustre descriptions in GATeL. In *Model-Based Testing Workshop*, volume 111, 1 of *Electronic Notes in Theoretical Computer Science*, pages 93–111, 2005.
33. J. Maurer, D. Abrahams, B. Dawes, and R. Rivera. Boost random number library. http://www.boost.org/libs/random/, June 2000.
34. M. Mihail and C.H. Papadimitriou. On the random walk method for protocol testing. In D.L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 132–141. Springer, 1994.
35. J. Musa, G. Fuoco, N. Irving, , D. Krofl, and B. Juhli. The operational profile. In M. R. Lyu, editor, *Handbook on Software Reliability Engineering*, pages 167–218. IEEE Computer Society Press, McGraw-Hill, 1996.
36. S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.
37. 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.
38. S. Prowell and J. Poore. Computing system reliability using Markov chain usage models. *Journal of Systems and Software*, 73(2):215 – 225, October 2004.
39. S. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *IEEE METRICS conference*, pages 64–73, 1997.
40. N. Rungta and E. 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.
41. 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.
42. E. Seneta. *Non-negative Matrices and Markov Chains*. Springer Verlag, 2006. Second edition (numerical). The first edition appeared in 1981.
43. 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.
44. P. Thévenod-Fosse and H. Waeselynck. An investigation of software statistical testing. *The Journal of Software Testing, Verification and Reliability*, 1(2):5–26, 1991.
45. 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.
46. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.
47. C.H. West. Protocol validation in complex systems. In *SIGCOMM*, pages 303–312, 1989.
48. E.J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.
49. H.S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.