Uniform Random Sampling of Traces in Very Large Models
Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lasseigne, Sylvain Peyronnet

To cite this version:
Uniform Random Sampling of Traces in Very Large Models

The RaST group

More informations in section 5

ABSTRACT

This paper presents some first results on how to perform uniform random walks (where every trace has the same probability to occur) in very large models. The models considered here are described in a succinct way as a set of communicating reactive modules. The method relies upon techniques for counting and drawing uniformly at random words in regular languages. Each module is considered as an automaton defining such a language. It is shown how it is possible to combine local uniform drawings of traces, and to obtain some global uniform random sampling, without construction of the global model.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification; D.2.5 [Software Engineering]: Testing and Debugging

Keywords

model-based testing, random walk, modular models, model checking, randomised approximation scheme, uniform generation

1. INTRODUCTION

Model based testing has received a lot of attention for years and is now a well established discipline (see for instance [27, 11]). Most approaches have focused on the deterministic derivation from a finite model of some so-called checking sequence, or of some complete/exhaustive set of test sequences, that ensure conformance of the implementation under test (IUT) with respect to the model. However, in very large models, such approaches are not practicable and some selection strategy must be applied to obtain tests of reasonable size. A popular selection criterion is transition coverage. Other selection methods rely upon the statement of some test purpose.

With the emergence of model checking, several sophisticated techniques for the representation and the treatment of models and formulas have been proposed and used for developing powerful verification tools for large models. Among them, one can cite: symbolic model checking, partial-order reduction methods, reactive modules, symmetry reduction, hash compaction and bounded model checking.

In this area, several authors have recently suggested the use of random walks in the state space of very large models in order to get good approximate checks in cases where exhaustive check is too expensive ([31, 18, 16, 30]). This is in the line of testing methods developed earlier in the area of communication protocols ([29, 28, 26, 11]).

A random walk in the state space of a model is a sequence of states $s_0, s_1, \ldots, s_n$ such that $s_i$ is a state that is chosen uniformly at random among the successors of the state $s_{i-1}$, for $i = 1, \ldots, n$. It is easy to implement and only requires local knowledge of the model. In [33], West reported experiments where random walk methods had good and stable error detection power. In [28], Mihail and Papadimitriou identified some class of models that can be efficiently tested by random walk exploration: the random walk converges to the uniform distribution over the state space in polynomial time with respect to the size of the model. These were first evidence of the interest of such approaches for dealing with special classes of large models.

However, as noted by Sivaraj and Gopalakrishnan in [31], random walk methods have some drawbacks. In case of irregular topology of the underlying transition graph, uniform choice of the next state is far from being optimal from a coverage point of view (see Figure 1). Moreover, for the same reason, it is generally not possible to get any estimation of the test coverage obtained after one or several random walks: it would require some complex global analysis of the topology of the model. One way to overcome these problems has been proposed by Gouraud et al. for program testing in [14] and implemented in [27]. It relies upon techniques for counting and drawing uniformly at random combinatorial structures. Two major approaches have been developed for dealing with these problems: The Markov Chain Monte-Carlo approach (see e.g. the survey by Jerrum and Sinclair [22]) and the so-called recursive method, as described by Flajolet et al. in [14] and implemented in [27]. Although the former is more general in its applications, we chose to work with the latter because it is particularly efficient for generating the kind of random walks we deal with. The idea in [14, 27] is to give up the uniform choice of the next state and to bias this choice according to the number of elements (traces, or

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

RT’06, July 20, 2006, Portland, ME, USA
Copyright 2006 ACM 1-59593-457-X/06/0007 ...$5.00.
We propose some way, inspired from [11], for uniformly random sampling traces in systems described by reactive modules. This method opens interesting perspectives for random model based testing, namely transition systems where transitions are labelled by atomic actions of a given language Act.

Our approach is based on a rather classical kind of model in testing, namely transition systems where transitions are labelled by atomic actions of a given language Act.

**Definition 1.** An action-labelled transition system (ALTS) is a structure $\mathcal{M} = (S, T, s_0, Act)$ where $S$ is a set of states, $s_0$ the initial state, $T \subseteq S \times Act \times S$ a transition relation and $Act$ a set of actions.

In this paper we consider finite ALTS. Note that, with this definition, ALTS may be non deterministic: the transition relation may associate several target states to a given state and a given action.

### 2.1 Reactive Modules

In this paper, we use the Reactive Modules language [3] for describing ALTS. This language is used in the probabilistic model checking community for modeling programs and protocols as transition systems. Two model checkers are using a subset of it as input language: PRISM [29, 24] and APMC [5].

In this language, transition systems are represented by modules that can interact together. Each module is composed of local variables and guarded commands. The global state of the system is given by the local states (i.e. the values of the local variables) of the modules. More precisely, at any moment the global state of the system is represented by a vector containing the values of all the variables of the system. A guarded command is a description of an atomic transition. It is written as

$$\text{[sync]} \text{ guard} \rightarrow \text{act}_1 + \ldots + \text{act}_k ;$$

where guard is a propositional formula over the variables of the system and where each action $\text{act}_1, \ldots, \text{act}_k$ defines a new assignment of some local variables. The choice of the action to be activated is done non deterministically among those with a valid guard.

Basically, to compute an execution of the whole system, the algorithm is the following (when there is no synchronization):

1. Choose non deterministically one of the modules.
2. Check all the guards of the module, keep a list of the valid guards.
3. If there is no valid guards, no action can be executed, then the execution is stopped (to avoid livelock situation).
4. Choose non deterministically among the valid guards, execute non deterministically one of the corresponding actions.
5. Modify the local state, thus inducing a modification of the global state.
6. Go to step 1.

Moreover, one can see that there is a specific field in the guarded command: [sync]. This field is used to synchronize modules. By putting a synchronization between guards of different modules, we force the actions associated to the states, or transitions) reachable via each successor. Considering the number of traces makes it possible to ensure a uniform probability on traces. Considering elements, such as states or transitions, makes it possible to maximise the uniform probability on traces. Considering elements, such as states or transitions) reachable via each successor. Con-
guards to be done together (this is a way to describe succinctly a complex behaviour). Basically, we have to maintain, together with the valid guards, the corresponding synchronisations. At the step 2 of the computation, a guard \( g \) synchronised by \( s \) in a module \( m \) is considered valid if and only if the guard is true and if there exists, in each module, at least one guard which is true and synchronised by \( s \). If \( g \) is picked at the step 4, then in each module one of the actions corresponding to one (chosen non deterministically) of the synchronised valid guard is executed together with the one of actions of \( g \).

In the following, we give an example of a simple Reactive Modules system composed of three modules. All the modules act together via synchronization. The figure 2 summarizes the example.

### Module `timer`

\[
\begin{align*}
t &: [0..1] \text{ init } 0; \\
[tic] \quad & t=0 \rightarrow t'=1; \\
[tac] \quad & t=1 \rightarrow t'=0;
\end{align*}
\]

endmodule

```
module on_tic

state1 : [0..1000] \text{ init } 0;
[tic] state1<1000 \rightarrow state1'=(state1+2);
[tic] state1>1000 \rightarrow state1'=0;
```

endmodule

```
module on_tac

state2 : [1..1001] \text{ init } 1;
[tac] state2<1001 \rightarrow state2'=(state2+2);
[tac] state2>1001 \rightarrow state2'=1;
```

endmodule

We now explain quickly the short example. To compute executions of the model, one has to first pick one of the modules, for instance module `on_tic`. Then the algorithm checks the valid guards. At the beginning, the variable `state1` is lower than 1000, so only the first guard is valid. We have to activate the first guard, but one can see that there is a synchronisation on it: `tic`. So we have to made each module acting with the two others via a guard synchronised with `tic`. It means that the only valid execution is to activate the first guard of the timer and the module on `tic`. It synchronizes the example succinctly.

The Reactive Modules notation, makes it possible to describe huge transitions systems via synchronised product (\(\bigotimes\)). In practice, this notation allows to manipulate large systems without being subject to the exponential blowup of the state space (for instance systems with more than \(10^{30}\) states, see \([35]\)).

Most of the very large models come from the product of several times the same module. This is the case with classical distributed algorithms (\([17]\)), real systems/protocols (\([14,11,21]\)).

### 2.2 Classical random walks

An execution path, or a trace in an \(\text{ALTS}\), is a finite or infinite sequence \(\sigma = (s_i,a_i,s_{i+1})\) of transitions satisfying: for all \( i \geq 0 \), there exists \( a_i \in \text{Act} \) such that \((s_i,a_i,s_{i+1}) \in T\). To perform a random walk in an \(\text{ALTS}\) \(M\), it is sufficient to have a succinct representation of it, that we call \(\text{diagram}_M\), that allows to generate algorithmically, for any state \( s \), the set of successors of \( s \). An example of such a diagram is a set of reactive modules defining a large model \( M \) (as seen above). But OBDD or other representations of \(\text{LTS}\) satisfy this requirement.

The size of such a diagram can be substantially lower than the size of the corresponding \(\text{ALTS}\). Typically, for Reactive Modules, the size of \(\text{diagram}_M\) is poly-logarithmic in the size of \( M \).

The following function `Random Walk`\(^1\) uses such a succinct representation to generate a random path of length \( k \) and to check if this path leads to the detection of some conformance error. We make the simplifying assumption that there is a reliable verdict that detects an error when a fault is reached during the execution of the random walk by the implementation under test (IUT).

\(^1\)This classical algorithm actually defines a so-called “preset” random walk. For the distinction between preset and adaptive checking sequences see \([15]\). We give some hints on how to cope with adaptive random walks in the conclusion.
2.3 Randomised approximation scheme

Many enumeration and counting problems are known to be strongly intractable. For example, counting the number of elementary paths between two given nodes in the graph of a transition system is \#P-complete. We recall that \#P is the complexity class of functions associated with counting the numbers of solutions of \textit{NP} decision problems. A classical method to break this complexity barrier is to approximate counting problems.

We show that we can approximate the error detection probability with a simple randomised algorithm. A probability problem is defined by giving as input a succinct representation of a transition system, a property \textit{A} and a real number \(\varepsilon\) such that for any \(x\), \(\varepsilon > 0\), and \(\delta > 0\):

\[
\Pr[|A(x, \varepsilon, \delta) - \mu(x)| < \varepsilon] \geq 1 - \delta.
\]

If the running time of \(A\) is polynomial in \(|x|\), \(\frac{1}{\varepsilon}\) and \(\log \left( \frac{1}{\delta} \right)\), \(A\) is said to be fully polynomial.

Let \(\text{Paths}(s_0)\) be the set of execution paths of origin \(s_0\) and of depth \(k\). We generate random paths in the associated probabilistic space and compute a random variable \(A\) which approximates the error detection probability on the paths of depth \(k\), \(\text{Prob}_k[\text{error}]\). Consider now the random sampling algorithm \(\text{GAA}\) designed for the approximate computation of \(\text{Prob}_k[\text{error}]\):

**Theorem 1.** (see [24]). The generic approximation algorithm \(\text{GAA}\) is a fully polynomial randomised approximation scheme for computing \(p = \text{Prob}_k[\text{error}]\) whenever \(p \in [0, 1]\).

The property of existence of conformance error detection is monotone: if it is true for a finite path \(\pi\), then it is also true for every infinite extension of this path. Let \(\text{Prob}[\text{error}]\) be the error detection probability in the probabilistic space associated to the set \(\text{Paths}(s_0)\) of infinite execution paths of origin \(s_0\). Then the sequence \((\text{Prob}_k[\text{error}])_k\) converges to the limit \(\text{Prob}[\text{error}]\).

We can obtain a randomized approximation of \(\text{Prob}[\text{error}]\) by increasing \(k\).

**Corollary 1.** The fixed point algorithm defined by iterating the approximation algorithm \(\text{GAA}\) is a randomised approximation scheme for the probability problem \(p = \text{Prob}[\psi]\) whenever \(p \in [0, 1]\).

The main interest of this randomised approximation scheme is that it allows some quantification of the error detection power of a random walk without construction and analysis of the global system.

3. IMPROVING RANDOM WALK COVERAGE

In this section we study how to improve random walk by changing the random choice of the successors in such a way that traces are uniformly distributed. After some preliminaries, we first address the case of systems described by a set of concurrent, non synchronised reactive modules, and then we consider the case where there is some synchronisation. In both cases, we analyse the (intractable) complexity of explicitly building the product \(\prod\) of the models corresponding to the modules. Then we propose a much more efficient alternative, based on the representation of the modules, hence without explicitly constructing the whole system.

3.1 From reactive modules to automata

We briefly recall that a finite state automaton \(A\) is denoted as a 5-tuple \(A = \langle X, Q, q_0, F, \Delta \rangle\) where \(X\) is the alphabet, \(Q\) is the finite set of states, \(q_0\) is the initial state, \(F\) is the set of final states and \(\Delta : Q \times X \rightarrow Q\) is the state transition relation. A finite state automaton \(A\) defines a regular language \(L\) on the alphabet \(X\).

Let \(M_1, M_2, \ldots, M_r\) be a set of reactive modules, each one standing for an ALTS. Each of the \(M_i\)'s can be represented in a straightforward way by a finite-state automaton \(A_i = \langle X_i, Q_i, q_i^0, F_i, \Delta_i \rangle\) where

- each state of \(Q_i\) corresponds to a state of \(M_i\),
- any two different transitions are labelled by two different letters of \(X_i\) (hence the cardinality of \(X_i\) equals the numbers of transitions in \(A_i\)),
- all states are final states (hence \(F_i = Q_i\)),
- the \(X_i\)'s are pairwise disjoint.

Consequently, each of the \(A_i\)'s defines a regular language \(L_i\) where each word is in one-to-one correspondence with a trace in the reactive module.

\(\text{This is just a way to identify transitions in order to use their numbers in the following developments. This has no consequence on the kind of model considered, deterministic or not.}\)
3.2 Combinatorial and algorithmic preliminaries

3.2.1 Automata and word counting

Let $L$ be a regular language and let $\ell(n)$ be the number of words of $L$ of length $n$. According to a well known result (see e.g. [13, Theorem 8.1]), there exist an integer $N_1$, a finite set of complex numbers $\omega_1, \omega_2, \ldots, \omega_k$ and a finite set of polynomials $R_1(n), R_2(n), \ldots, R_k(n)$ such that

$$n \geq N_1 \implies \ell(n) = \sum_{j=1}^{k} R_j(n)\omega_j^n.$$  

The number $N_1$, as well as the $\omega_j$'s and the $R_j$'s, can be computed from an automaton of $L$, with an algorithm of polynomial complexity according to the size of the automaton. Technical details are given in Appendix 1.

If the automaton of $L$ satisfies certain conditions (see below), then there is an unique solution $\omega = \omega_1$ to the following formula, asymptotically:

$$\ell(n) \sim C_n \omega^n.$$  

This gives a very good estimation of $\ell(n)$ even for rather small $n$ since, according to Formulas (2) and (3), $C_n \omega^n/\ell(n)$ converges to 1 at an exponential rate.

A simple sufficient condition for Formula (2) to hold is: the automaton is aperiodic and strongly connected. An automaton is aperiodic if, for any sufficiently large $n$, $\ell(n) \neq 0$. Now, as stated in Section 3.1, all the states of any automaton which represents a reactive module are final states. Thus any automaton which represents a reactive module is aperiodic. Concerning strong connectivity, it is satisfied as soon as there is a reset. Moreover, it is a sufficient yet not mandatory condition. For instance, for satisfying Formula (2), in fact it suffices to have some unique biggest strongly-connected component in the automaton. Hence, most “natural” automata are such that this formula is satisfied. Note that in the sequel we use Formula (2) for the automata corresponding to the component modules.

3.2.2 Automata and word shuffling

The shuffle of two words $w, w' \in \mathcal{X}^*$, denoted $w \shuffle w'$ is the set $w \shuffle w' = \{w_1w_2 \ldots w_mw_{m+1}w_{m+2} \ldots w_n | w \in \mathcal{X}^*, w' \in \mathcal{X}^*, w = w_1 \ldots w_m, w' = w_{m+1} \ldots w_n\}$. For example, $ab\shuffle cdef = \{abcde, acbde, acdeb, cdabe, cdeab\}$. The shuffle operation is associative and commutative. It naturally generalises for languages: the shuffle of two languages $L_1$ and $L_2$ is the set

$$L_1 \shuffle L_2 = \bigcup_{w_1 \in L_1, w_2 \in L_2} w_1 \shuffle w_2.$$  

This easily generalises to any finite number $r$ of languages. And the following property holds: the shuffle of a set of regular languages is a regular language. Indeed, let $r > 0$ and let $L_1, L_2, \ldots, L_r$ be $r$ regular languages. Let $A = \langle X, Q, q_0, F, \Delta \rangle$ be an automaton of $L_1$, for any $1 \leq i \leq r$. Then the following finite state automaton recognises $L$: $A = \langle X, Q, q_0, F, \Delta \rangle$, where

- $X = X_1 \cup X_2 \cup \ldots \cup X_r$;
- $Q = Q_1 \times Q_2 \times \ldots \times Q_r$;
- $q_0 = (q_0^1, q_0^2, \ldots, q_0^r)$;
- $F = F_1 \times F_2 \times \ldots \times F_r$;
- $\Delta((q_1^1, q_2^1, \ldots, q_r^1), x) = (\Delta_1(q_1^1, x), \Delta_2(q_2^1, x), \ldots, \Delta_r(q_r^1, x))$ if $x \in X_1$;
- $\Delta((q_1^1, q_2^1, \ldots, q_r^1), x) = (\Delta_1(q_2^1, x), \ldots, \Delta_r(q_r^1, x))$ if $x \in X_r$.

We call this automaton a shuffling automaton of $L_1, L_2, \ldots, L_r$.

First discussed by Hickey and Cohen[19], the method for generating words of regular languages has been improved and widely generalized by Flamotol et al [14]. The principle of the generation process is simple: Starting from state $q_0$, one draws a word step by step; at each step, the process consists in choosing a successor of the current state and going to it.

The problem is to proceed in such a way that only (and all) words of length $n$ can be generated, and that they are equiprobably distributed. This is done by choosing successors with suitable probabilities. Given any state $s$ of the automaton, let $g_m(s)$ denote the number of words of length $m$ which connect $s$ to any final state $f \in F$. Suppose that, at any 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 $m > 0$ transition remain to be done in order to get a word of length $n$. Then the condition for uniformity is that the probability of choosing state $s_i$ ($1 \leq i \leq k$) equals $g_{m-1}(s_i) / g_m(s)$. In other words, the probability to go to any successor of $s$ must be proportional to the number of words of suitable length from this successor to any $f$.

So there is a need to compute the numbers $g_i(s)$ for any $0 \leq i \leq n$ and any state $s$ of the automaton. This can be done by using the following recurrence relations:

$$g_0(s) = \begin{cases} 1 & \text{if } s \in F \\ 0 & \text{otherwise} \end{cases}$$

$$g_i(s) = \sum_{s \rightarrow s'} g_{i-1}(s') \text{ for } i > 0$$

where $s \rightarrow s'$ means that there exists an letter $x \in X$ such as $(s, x, s') \in \Delta$.

Now the generation scheme is as follows:
• Preprocessing stage: Compute a table of the \( g_i(s) \)’s for all \( 0 \leq i \leq n \) and all states.

• Generation stage: Draw the word according to the scheme seen above.

Note that the preprocessing stage must be done only once, whatever the number of words to be generated. Easy computations show that the memory space requirement is \( n \times Q \) integer numbers, where \( Q \) stands for the number of states in the automaton. The number of arithmetical operations needed for the preprocessing stage, as well as for the generation stage, is linear in \( n \).

### 3.3 Generating traces of a system of modules without synchronisation.

Here we focus on the problem of uniformly (that is equiprobably) generating traces of a given length \( n \) in a system of \( r \) reactive modules. In a first step, we consider that there is no synchronisation between the modules.

Each one is represented by a finite state automaton \( A_i = (X_i, Q_i, q_i^0, F_i, \Delta_i) \). As stated in Section 3.1, each of the \( A_i \)'s defines a regular language \( L_i \), whose words correspond to the traces within the corresponding module. Since there is no synchronisation in the system, clearly there is a one-to-one correspondence between the set of traces of the system and the words of \( L = L_1 \cup L_2 \cup \cdots \cup L_r \). 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.

#### 3.3.1 Brute force method

This first approach consists in constructing the shuffling automaton that has been defined in Section 3.2.2. Then the classical algorithms for randomly generating words of a regular language can be processed, as described in Section 3.2.3. Let \( C_1 = \sum_{0 \leq i < r} \text{Card}(X_i) \) and \( C_2 = \prod_{0 \leq i < r} \text{Card}(Q_i) \). 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 \( C_1C_2 \) in time and space requirements.

2. Generating traces: Using classical algorithms, generating one word requires \( nC_1 \) time requirement, after a preprocessing stage having worst-case complexity \( nC_1C_2 \) in time and space. This preprocessing stage is performed once, whatever the number of traces to be generated.

Hence the worst case complexity for generating \( m \) traces of length \( n \) is \( O(nC_1C_2 + mnC_1C_2) \) in time and \( O(nC_1C_2) \) in space. This is linear in \( n \), in \( m \), in the total size of the alphabets. Since \( C_2 = \prod_{0 \leq i < r} \text{Card}(Q_i) \), the complexity is exponential according to the number of modules. Thus the algorithm will be efficient only for a small number of modules.

#### 3.3.2 “On line” shuffling method

Here we describe an alternative method which avoids constructing the above automaton. We recall that \( \ell(k) \) is the number of words of length \( k \) belonging to the language \( L_i \), and \( \ell(n) \) is the number of words of length \( n \) belonging to the language \( L \). The method consists first in choosing at random, with a suitable probability, the length \( n_i \) of each word \( w_i \) of \( L_i \) which will contribute to the word \( w \) of \( L \) to be generated. Then each \( w_i \) is generated independently. Finally, the shuffle operation is processed. We detail the method just below.

1. Choose at random a \( r \)-uple \((n_1, \ldots, n_r)\) with probability \( \Pr(n_1, \ldots, n_r) \) such that
   \[
   \Pr(n_1, \ldots, n_r) = \frac{n!}{n_1! \cdots n_r!} \ell(n_1) \cdots \ell(n_r)
   \]
   (6)

2. For each \( 0 \leq i \leq r \), 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 \( n_1, \ldots, n_r \)
| Output: word \( w \) of length \( n = \sum_i n_i \) and drawn uniformly among the set of shuffles of \( w_1, \ldots, w_r \).

\[
\begin{align*}
&\text{\textbf{Input:}} \ r \text{ words } w_1, \ldots, w_r, \text{ of length } n_1, \ldots, n_r \\
&\text{\textbf{Output:}} \text{ word } w \text{ of length } n = \sum_i n_i \text{ and drawn uniformly among the set of shuffles of } w_1, \ldots, w_r. \\
&w \leftarrow \varnothing \\
&n \leftarrow \sum_i n_i \\
&\text{while } n > 0 \text{ do} \\
&\quad \text{choose an integer } i \text{ between 1 and } r \text{ with probability } \frac{n_i}{n} \\
&\quad \text{add the first letter of } w_i \text{ at the end of } w \\
&\quad n_i \leftarrow n_i - 1 \\
&\text{if } n_i = 0 \text{ then remove the letter } \epsilon \text{ of } w \\
&n \leftarrow n - 1
\end{align*}
\]

The word \( w \) has been generated equiprobably among all the words of \( L \) of length \( n \). Regarding complexity issues, clearly the complexity of step 3 is linear in \( n \). The complexity of step 2 is linear in \( n_i \), in the maximum of \( \text{Card}(X_i) \) and in the maximum of \( \text{Card}(Q_i) \), 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 (6). The space requirement is \( O(1) \) but the number of terms in \( \ell(n) \) is exponential in \( n \). However, if the \( L_i \)'s satisfy the hypothesis of Formula (1), then, by Formula (1):

\[
\Pr(n_1, \ldots, n_r) \sim \left( \frac{n}{n_1 \cdots n_r} \omega^{n_1} \cdots \omega^{n_r} \right)^n. 
\]

(7)

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

Well, one could argue that Formula (1) only provides an asymptotic approximation of \( \Pr(n_1, \ldots, n_r) \) as \( n \) tends to infinity. However, as noticed in Section 3.1, the rate of convergence is exponential, so Formula (1) is precise enough even for rather small \( n \). And for really small \( n \) (at least when \( n < N_1 \) in Formula (1)), \( \Pr(n_1, \ldots, n_r) \) can be computed exactly by Formulas (6) and (7).

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 \( n \), to the maximum of \( \text{Card}(X_i) \) and to the maximum of \( \text{Card}(Q_i) \), in time as well as in space requirements.
3.4 Generating traces in presence of synchronisation.

Now we suppose that each module contains exactly one synchronised transition, denoted $\alpha$. Thus, in the global system all modules must take $\alpha$ 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 $\alpha$, such that

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

Let $S_1, \ldots, S_r$ be the respective languages recognised by $A_1, \ldots, A_r$. Here, any trace can be represented by a word belonging to the language $S$ defined as follows: $S$ is the set of words $w \in X_1 \cup \ldots \cup X_r$ such that

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

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

3.4.1 Again the brute force approach.

Here the approach consists in constructing the synchronised product of $A_1, A_2, \ldots, A_r$, as follows. Let $X_i \alpha = X_i \setminus \{\alpha\}$. The synchronised product $\otimes$ of $A_1, A_2, \ldots, A_r$ with $X_i$ as synchronisation set is the finite automaton $A = (X, Q, q_0, F, \delta)$, where

- $X = X_1 \cup X_2 \cup \ldots \cup X_r$;
- $Q = Q_1 \times Q_2 \times \ldots \times Q_r$;
- $q_0 = (q_0^1, q_0^2, \ldots, q_0^r)$;
- $F = F_1 \times F_2 \times \ldots \times F_r$;
- $\delta$ is as follows:

$$\Delta((q_1, \ldots, q_r), x) = \begin{cases} 
\Delta((q_1, x), \ldots, q_r) & \text{if} \ x \in X_1 \alpha, \\
\Delta(q_1, \ldots, q_r, x) & \text{if} \ x \in X_2 \alpha, \\
\Delta(q_1, \ldots, q_r, x) & \text{if} \ x \in X_r \alpha, \\
\Delta((q_1, \ldots, q_r, \alpha), \alpha) = \delta_1(q_1, \alpha), \delta_2(q_2, \alpha), \ldots, \delta_r(q_r, \alpha)) & \text{otherwise}.
\end{cases}$$

This automaton accepts the language $S$ of synchronised traces. Once it has been built, the generation process is exactly as in Section 3.3.1, with the same time and space requirements.

3.4.2 “On line” generation of synchronised traces

Here we sketch an algorithm for almost uniformly generating random synchronised traces of length $n$, avoiding the construction of the synchronised product. The approach is similar to the one we described in Section 3.3.2 although we must be more careful because of the synchronisations.

Given that each automaton $A_i$ contains a unique transition labeled by $\alpha$ (the synchronised transition), let $q_{i,1}$ and $q_{i,2}$ be the states just before and just after this transition, respectively. Now let us define, for each $S_i$, 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 $\alpha$ transition, and which stop at $q_{i,1}$.
- The central language: $C_i$ is the set of words corresponding to the paths which start at $q_{i,1}$, which do not cross the $\alpha$ transition, and which stop at $q_{i,1}$.
- The ending language: $E_i$ is the set of words corresponding to the paths which start at $q_{i,2}$, which do not cross the $\alpha$ transition, and which stop anywhere.
- The non-synchronised language: $T_i$ is the set of words which start at the initial state of $A_i$, which never cross the $\alpha$ transition, and which stop anywhere.

For any $i$, the language $S_i$ can be defined according to $B_i$, $C_i$, $E_i$ and $T_i$:

$$S_i = B_i \cup C_i \cup E_i \cup T_i .$$

Thus, if we define $B = \bigcup_{i=1}^{r} B_i$, (resp. $C = \bigcup_{i=1}^{r} C_i$, $E = \bigcup_{i=1}^{r} E_i$ and $T = \bigcup_{i=1}^{r} T_i$), we have:

$$S = B \cup C \cup E \cup T .$$

Now let $s(n)$ (resp. $s(n)$, $b(n)$, $b(n)$, $c(n)$, $c(n)$, $c(n)$, $c(n)$, $t(n)$, $t(n)$) be the number of words of length $n$ in $S$ (resp. $S$, $B$, $B$, $C$, $C$, $E$, $E$, $T$, $T$). Additionally, let $a(n)$ be the number of words of $S$ of length $n$ which contain $\alpha$ exactly $m$ times. Let $w$ be one of these words. If $m = 0$, then $w$ writes $w = w_0 \alpha w_1 \alpha \ldots \alpha w_m$ where $w_0 \in B$, $w_i \in C$ for any $1 \leq i \leq m$, and $w_m \in E$. Finally, let $s(n, m, i_0, i_m)$ be the number of such words such that the length of $w_0$ equals $i_0$ and the length of $w_m$ equals $i_m$. Then we have

$$s(n) = \sum_{i=0}^{n} s(n, i),$$

where

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

and, for $m > 0$,

$$s(n, m, i_0, i_m) = b(i_0)c(i_0) \ldots c(i_{m-1})c(i_m).$$

Now suppose that all the $B_i$’s, the $C_i$’s, the $E_i$’s and the $T_i$’s satisfy Formula (2), that is:

$$b_i(k) \sim C_{b_i}, \omega^{b_i k},$$

$$c_i(k) \sim C_{c_i}, \omega^{c_i k},$$

$$e_i(k) \sim C_{e_i}, \omega^{e_i k},$$

$$t_i(k) \sim C_{t_i}, \omega^{t_i k}.$$ 

Then, similarly to Formula (3), we have:

$$b(k) \sim C_{b_1}, \ldots, C_{b_r}, (\omega^{b_1 k} + \ldots + \omega^{b_r k}),$$

$$c(k) \sim C_{c_1}, \ldots, C_{c_r}, (\omega^{c_1 k} + \ldots + \omega^{c_r k}),$$

$$e(k) \sim C_{e_1}, \ldots, C_{e_r}, (\omega^{e_1 k} + \ldots + \omega^{e_r k}),$$

$$t(k) \sim C_{t_1}, \ldots, C_{t_r}, (\omega^{t_1 k} + \ldots + \omega^{t_r k}).$$
Consequently, for $m > 0$, 

$$s(n, m, i_0, i_m) \sim \left(C_{b_1} \cdots C_{b_r} \right) \left(C_{s_1} \cdots C_{s_r} \right)^{m-1} \left(C_{c_1} \cdots C_{c_r} \right)$$

$$\left(\omega_{b_1} + \cdots + \omega_{b_r}\right)^n$$

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

1. Using Formula (14), compute $s(n, m, i_0, i_m)$ for all $m$ such that $1 \leq m \leq n$ and for all pairs $(i_0, i_m)$ such that $0 \leq i_0 + i_m \leq n - m$. This requires $O(n^2 \times m) = O(rn^4)$ arithmetic operations. Then compute $s(n, m)$ for all $m$ such that $1 \leq m \leq n$, using Formula (14) and, additionally, Formula (15) when $m = 0$. Finally compute $s(n)$ by Formula (16). It is worth noticing that this preliminary stage has to be done only once, whatever the number of traces of length $n$ to be generated. Its overall arithmetic complexity is $O(rn^4)$.

2. Choose $m$, the number of synchronisations, with probability

$$\Pr(m) = \frac{s(n, m)}{s(n)}.$$  

Computing these probabilities requires $O(n)$ arithmetic operations in the worst case.

3. If $m = 0$, then generate uniformly at random a word of length $n$ in $T$, with the same algorithm as in Section 3.3.2.

4. If $m > 0$, then:

   (a) Choose the length of $w_0$ and the length of $w_m$ by picking at random a pair $(i_0, i_m)$ with probability

$$\Pr(i_0, i_m) = \frac{s(n, m, i_0, i_m)}{\sum_{i_0 + i_m = m} s(n, m, k_0, k_m)}.$$  

Computing these probabilities requires $O(n^2)$ arithmetic operations in the worst case.

   (b) Choose the lengths of $w_1, w_2, \ldots, w_{m-1}$ by picking at random a $(m-1)$-uple $(i_1, i_2, \ldots, i_{m-1})$ with probability

$$\Pr(i_1, \ldots, i_{m-1}) = \frac{c(i_1)c(i_2) \cdots c(i_{m-1})}{\sum_P c(k_1)c(k_2) \cdots c(k_{m-1})}.$$  

where $P$ stands for:

$$k_1 + k_2 + \cdots + k_{m-1} = n - m - i_0 - i_m.$$  

Using Formula (14), this reduces to

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

and, similarly to Section 3.3.2 there is a simple algorithm for picking $(i_1, i_2, \ldots, i_{m-1})$ at random with this probability. This algorithm is linear according to $n$ and $m$. The algorithm and the proof of Formula (13) are given in Appendix 2.

(c) Now we have got the whole sequence $(i_0, i_1, \ldots, i_m)$ with a suitable probability. It remains to generate the words $w_0 \in B$, $w_1, w_2, \ldots, w_{m-1} \in C$ and $w_m \in E$, each $w_k$ having length $i_k$. Each of these words is simply a shuffle of the $r$ languages $(B_i)_{i=1}^r$ if $k = 0$, $(C_i)_{i=1}^r$ if $1 \leq k < m$, $(E_i)_{i=1}^r$ if $k = m$. For each of the $w_k$’s, the shuffling algorithm given in Section 3.3.2 can be used.

As remarked above, the first step of the algorithm, in $O(rn^4)$ operations, has to be done only once. After that, the overall complexity of generating any random trace of length $n$ is quadratic according to $n$. And, as in Section 3.3.2, it is linear according to the maximum of $\text{Card}(X_i)$ and to the maximum of $\text{Card}(Q_i)$, in time as well as in space requirements. Thus we have defined an efficient way for approximating the uniform coverage in presence of one synchronisation for any sufficiently large $n$. The case where there are several synchronisations labelled by different symbols is more complex but we think it can be addressed with similar techniques and simplifications. This is the subject of some ongoing work.

4. CONCLUSION AND PERSPECTIVES

One of the main interest of classical random walk is that it can be performed on large models with a local knowledge only. However, it presents some drawbacks, mainly related to the difficulty to estimate, without analysing the global topology, the test coverage for a given number of random walk of some given lengths. In Section 2, we have shown how it is possible to approximate it via a randomised approximation scheme.

In the rest of the paper we have described how to perform globally uniform random walks in very large models described as sets of concurrent, smaller, models. By globally uniform random walk, we mean that the choice of the successor at every step is biased in such a way that all traces of the global model have equal probability to be traversed.

A brute force approach is to count the number of paths of the desired length starting from each successor and to adjust its probability accordingly. This is feasible via techniques for counting and drawing uniformly random combinatorial structures. However, the complexity of this approach is linear in the number of states of the considered model. This makes it feasible for moderately-sized models only.

Then, we have shown how to use local uniform drawings to build globally uniform random walks, with a complexity that is linear in the size of the biggest component model. We use an estimation of the number of words, but as soon as the length of the random walks is sufficient, it is a very good approximation as seen in 3.2 (formulas (1) and (2)).

This method can be used for random testing, model checking or simulation of protocols that involve many distributed entities, as it is often the case in practice. It ensures a balanced coverage of all behaviours, even if the topology of the underlying model is irregular.

This work is a first step only. First, we plan a campaign of experiments of the method and of some variants of it. For instance, instead of uniform coverage of traces, it is possible to consider uniform coverage of states, or of transitions as it is done in [11] for testing C programs.
Moreover, results on counting and generating combinatorial structures are not limited to words of regular languages. They open numerous perspectives in the area of random testing. A possibility that is worth to explore is the test of non deterministic systems via uniform generation of tree-like behaviours, i.e. some notion of adaptive random walk inspired from the classical notion of adaptive checking sequences [27]. It would be also interesting to study how the approach presented here for descriptions by reactive modules could be transposed to other succinct representations of large models such as OBDD, symmetry reduction, etc.

Acknowledgement. We thank Radu Grosu for interesting discussions that have motivated this work.

5. ADDITIONAL AUTHORS

The RaST group (Random Software Testing) is composed of:

- Alain Denise - alain.denise@lri.fr
  LRI, Université Paris-Sud, UMR CNRS 8623.
- Marie-Claude Gaudel - mcg@lri.fr
  LRI, Université Paris-Sud, UMR CNRS 8623.
- Sandrine-Dominique Gouraud - gouraud@lri.fr
  LRI, Université Paris-Sud, UMR CNRS 8623.
- Richard Lassaigne - lassaign@logique.jussieu.fr
  Equipe de Logique Mathématique, Université Paris VII, UMR CNRS 7056.
- Sylvain Peyronnet - syp@lrde.epita.fr
  LRDE/EPITA and Equipe de Logique Mathématique, UMR CNRS 7056, Université Paris VII.

6. REFERENCES

[22] W. A. Trotter and N. Madras. Monte Carlo
algorithms for enumeration and reliability problems.  


[29] PRISM Website: http://prismmodelchecker.com/  


http://mupad-combinat.sourceforge.net/  

### Appendix 1: Counting words of rational languages

Let $L$ be a language on an alphabet $X$, and, for $n \geq 0$, let $\ell(n)$ be the number of words of $L$ of length $n$. The generating series of $L$ is defined as:

$$f(z) = \sum_{n \geq 0} \ell(n)z^n.$$  

This is a formal power series of one variable $z$ where the coefficient of $z^n$ equals the number of words of length $n$ in $L$. According to well-known results (see e.g. [3]), if $L$ is a regular language, then its generating series can be expressed as a rational function

$$f(z) = \frac{N(z)}{D(z)},$$  

where $N$ and $D$ are two polynomials with integer coefficients. This function is a solution of a system of $m$ linear equations, where $m$ is the number of states of a deterministic automaton which recognizes $L$.

The number of words of size $n$ mainly depends on the poles of $f(z)$, that is on the roots of its denominator $D(z)$ (see e.g. [3], Theorem 8.1]). Precisely, let $\alpha_1, \alpha_2, \ldots, \alpha_k$ the poles of $f(z)$ and let $\omega_i = 1/\alpha_i$ for any $i$. Then there exist an integer $N_1$, and $k$ polynomials $R_1(n), R_2(n), \ldots, R_k(n)$ such that

$$n \geq N_1 \implies \ell(n) = \sum_{j=1}^{k} R_j(n)\omega_j^n.$$  

(18)

where the degree of any $R_j$ equals the multiplicity of its corresponding pole $\alpha_j$, minus 1.

As a corollary of the Perron-Frobenius Theorem [3], Theorem 8.5 and Corollary 8.1], if the automaton of $L$ satisfies some conditions (see below), then its generating series has an unique dominant pole, that is there exists $i$ such that $|\alpha_i| < |\alpha_j|$ for any $j \neq i$, and this pole has multiplicity 1. Hence $R_j(n)$ has degree zero, say $R_j(n) = C$ where $C$ is a constant. Thus we have, asymptotically,

$$\ell(n) \sim C\omega_i^n.$$  

(19)

A sufficient condition for the above formula to hold is: the automaton is strongly connected and aperiodic. However, as noticed in Section 3.2.1, there are a number of weaker conditions which imply it.

### Appendix 2: Proof of Formula (17) and related algorithm

We have

$$\Pr(i_1, \ldots, i_{m-1}) = \frac{c(i_1)c(i_2) \ldots c(i_{m-1})}{\sum_P c(k_1)c(k_2) \ldots c(k_{m-1})}$$

where $P$ stands for:

$$k_1 + k_2 + \cdots + k_{m-1} = n - m - i_0 - i_m.$$  

By Formula (13) this leads to

$$\Pr(i_1, \ldots, i_{m-1}) \sim \frac{\sum_P (\omega_{i_1} + \ldots + \omega_{i_{m-1}})^{n-m-i_0-i_m}}{\sum_P 1}.$$  

The denominator equals the number of distinct ways to choose $(k_1, k_2, \ldots, k_{m-1})$ in such a way that they sum to $n-m-i_0-i_m$. This means that the sequence $(i_1, i_2, \ldots, i_{m-1})$ is to be picked uniformly among all sequences such that $k_1 + k_2 + \cdots + k_{m-1} = n - m - i_0 - i_m$.

Let $Q = n - m - i_0 - i_m$ and $q = m - 1$. The number of ways to choose $Q$ numbers greater or equal to zero that sum to $Q$ equals $\binom{Q+q}{q-1}$, for any positive integers $Q$ and $q$. Hence

$$\Pr(i_1, \ldots, i_{m-1}) \sim \frac{1}{\binom{Q+q}{q-1}}.$$  

This proves Formula (13).

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