A random testing approach using pushdown automata

Aloïs Dreyfus, Pierre-Cyrille Héam, Olga Kouchnarenko, Catherine Masson

A Random Testing Approach Using Pushdown Automata

Aloïs Dreyfus Pierre-Cyrille Héam Olga Kouchnarenko
Catherine Masson

Sept. 2014

FEMTO-ST, DISC, CNRS UMR 6174, INRIA
Unversité de Franche-Comté – 16 route de Gray
25000 Besancon, FRANCE

Abstract

Since finite automata are in general strong abstractions of systems, many test cases which are automata traces generated uniformly at random, may be un-concretizable. This paper proposes a method extending the abovementioned testing approach to pushdown systems providing finer abstractions. Using combinatorial techniques guarantees the uniformity of generated traces. In addition, to improve the quality of the test suites, the combination of coverage criteria with random testing is investigated. The method is illustrated within both structural and model-based testing contexts.

1 Introduction

1.1 General Overview

Producing secure, safe and bug-free programs is one of the most challenging problems of modern computer science. In this context, two complementary approaches addressing this problem are verification and testing. On the one hand, verification techniques mathematically prove that a code or a model of an application is safe. However, complexity bound makes verification difficult to apply to large systems. On the other hand, testing techniques do not provide any proof but are relevant in practice for developing high quality software. Over the past decade, many works have been done in order to upgrade hand-made (or experience-based) testing techniques to formal methods based frameworks.

Since exploring all the configurations of a software is time consuming and error prone, one of the key problems for a validation engineer is to choose a relevant test suite while controlling the number of tests. The crucial question raised is then: "What does *relevant* mean?". A frequent answer, in the literature

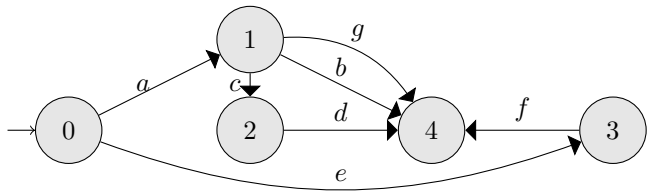


Figure 1: Example of a finite graph

and in practice, is to consider a test suite as relevant if it fulfils some well-known coverage criteria; for instance, a code coverage criterion, which is satisfied if all the lines of the code are executed at least once when running the tests. It is important to point out that coverage criteria can be applied to the code (white box or structural testing) or to a model of the implementation (black box or functional testing [1]). Since there are many ways to fulfil coverage criteria [2], other criteria can be taken into account, for example based either on computing minimal/maximal length test suites, or on selecting boundary or random values for the test data.

Recently, a method for drawing paths in finite graphs uniformly has been proposed [3], and the authors have shown how to use these techniques for a control flow graph based testing of C programs. The idea is to define probabilistic coverage criteria for testing. The method consists in applying a combinatorial based algorithm to solve the following probabilistic problem.

Random Generation of a Path in a Finite Graph
Input: A finite labelled graph $\mathcal{G}$, a vertex v_0, a positive integer n
Output: Randomly generate a path of length n in $\mathcal{G}$, starting from v_0. The generation has to be uniform relatively to all paths of length n in $\mathcal{G}$.

The technique by A.Denis et al. [4] proposes an efficient way to decide the above problem, even for very large graphs. Nevertheless, a finite graph often represents a strong abstraction of the system under test, and many abstract tests generated by the approach may be impossible to play on the implementation. This paper addresses this problem when using pushdown graphs as abstractions. More precisely, the present article makes the following contributions. The first contribution in Sec. 3 consists in improving the random approach proposed by A.Denis et al. [4] by extending it to pushdown graphs, and thus providing finer abstractions for systems, particularly useful to encode the call stack of a program. The second contribution in Sec. 3 consists in providing a way to efficiently solve the *Random Generation of a Path in a Normalised Deterministic Pushdown Automaton* problem described below. The third contribution in Sec. 4 concerns the combination of coverage criteria with random testing, in order to take benefit of both approaches for evaluating the quality of the test suites. The fourth contribution in Sec. 5 and 6 consists in illustrating the application of the

```
int power(float x, int n){
   int res;
   if (n==0) {
      return 1;
   } else {
      res = power(x,n/2);
      if (n%2==0) {
         return res*res ;
      } else {
         return res*res*x ;
      }
   }
}
```

Figure 2: C program computing x^n

proposed techniques within both structural and model-based testing contexts.

Random Generation of a Path in a Normalised Deterministic Pushdown Automaton
Input: A normalised deterministic pushdown automaton $\mathcal{A}$, the initial state v_0, a positive integer n
Output: Randomly generate a NPDA-trace in $\mathcal{A}$ of length n, starting from v_0. The generation has to be uniform relatively to all NPDA-traces of length n in $\mathcal{A}$.

In the paper NPDA stands for *Normalised Deterministic Pushdown Automaton*, and DFA for *Deterministic Finite Automaton*. For the before mentioned problem, informally a NPDA-trace is a path in the underlying finite automaton consistent with the stack operations. Its precise definition is given in Sec. 2.

1.2 Motivation

The random generation of test cases from a graph based model constitutes the context of the paper. This section explains why classical random walks are unfair for testing purposes, and why pushdown graphs are better than basic graphs for the concretization step.

1.2.1 Random Paths Generation vs. Random Walks

The first intuitive way to generate random paths is to perform a Markov-like random generation: starting from v_0, at each step the next vertex is randomly and uniformly picked in the neighbourhood of the current vertices. This technique leads to an unknown distribution on paths of length n and is very sensitive to the topology of the graph. Consider for instance the labelled graph depicted in Fig. 1. Using a Markovian approach to generate a path of length 2 starting from 0: path ef occurs with probability 1/2, and paths ab, ac and ag both with probability 1/6. The generation is not uniform. For a graph with a complex topology, severe disparities may be observed on the occurring probabilities of paths of same length.

The Markovian approach may not ensure a well-balanced coverage of the graph. The technique developed by A.Denise et al. [4] to handle the *Random Generation of a Path in a Graph problem* requires two steps: In a first step

the number of paths of length $i \leq n$ between each pair of vertices is computed recursively and using combinatorial techniques. It is easily computed using a recursive schema. Denoting by $s_i(p,q)$ the number of paths of length i from p to q and by $\alpha(p,q)$ the number of edges from p to q, one has $s_1(p,q) = \alpha(p,q)$. Moreover $s_{i+1}(p,q) = \sum \alpha(p,r)s_i(r,q)$, where the sum is taken for all vertices r. In a second step, the random path is recursively generated using the computed probabilities: In order to generate a path of length n from p to q, the probability that the second vertex visited by the path (after p) is r is $\alpha(p,r)\frac{s_{n-1}(r,q)}{s_n(p,q)}$. The first edge is picked uniformly from the edges from p to r (if there are many). A new path from r to q of length $n-1$ is randomly generated using the same technique. For instance, for the labelled graph depicted in Fig. 1, to generate a path of length 2 from 0 to 4, the probability to choose 3 as the second vertex will be $\frac{s_1(3,4)}{s_2(0,4)} = \frac{1}{3}$. The probability to choose 1 as the second vertex will be $\frac{s_1(1,4)}{s_2(0,4)} = \frac{2}{3}$. The probability to choose 2 or 4 or 0 is null. Next, if 3 is chosen, the generated path is $(0,e,3)(3,f,4)$. It happens with probability 1/3. If 1 is chosen as second state, the last edge is equiprobably chosen between $(1,b,4)$ and $(1,g,4)$. Paths$(0,a,1)(1,b,4)$ and $(0,a,1)(1,g,4)$ are both generated with probability 1/3. Each path occurs equiprobably.

Remark that it is also possible to design a graph for which the Markovian approach provides a more balanced coverage of the states/transitions than the uniform approach. However, the advantage of the uniform path generation approach is that it is possible to bias the distribution in order to optimise the coverage of the graph, as it is explained in Sec. 4.

1.2.2 Finite Automata vs. Pushdown Automata

Let us consider the recursive C program described in Fig. 2 and computing x^n: the control flow graph of this program is depicted in Fig. 3. The control flow graph on the left disregards the recursive calls to the `power` function. On the right, the invocations of the *power* function are represented by dashed arrows labelled either by *call(power)* for the `power` function invocation, or by *return(power)* for the the `power` function return.
Since in the left graph recursive calls to `power` are ignored, it is impossible to compute arbitrarily long paths. On the contrary, it is possible on the right graph. For instance, the run of `power(3,2)` corresponds to the path

$$0 \to 1 \to 5 \dashrightarrow 0 \to 1 \to 5 \dashrightarrow 0 \to 1 \to 2 \to 4 \dashrightarrow 6 \to 9 \to 10 \dashrightarrow 6 \to 7 \to 8$$

The first occurrence of $5 \dashrightarrow 0$ corresponds to the call to `power(3,1)`. The second occurrence $5 \dashrightarrow 0$ corresponds to the call to `power(3,0)`. The transition $4 \dashrightarrow 6$ corresponds to the return of `power(3,0)`, and the transition $10 \dashrightarrow 6$ corresponds to the return of `power(3,1)`.

Now let a stand for `int res;`, b for `n==0`, c for `n!=0`, d for *call(power)*, e for `return 1;`, f for *return(power)*, g for `n%2==0`, h for `n%2!=0`, i for the instruction `return res*res;`, and j for `return res*res*x;`. The words accepted by the automaton depicted on the right of Fig. 3 (see also Fig. 5), are those in the

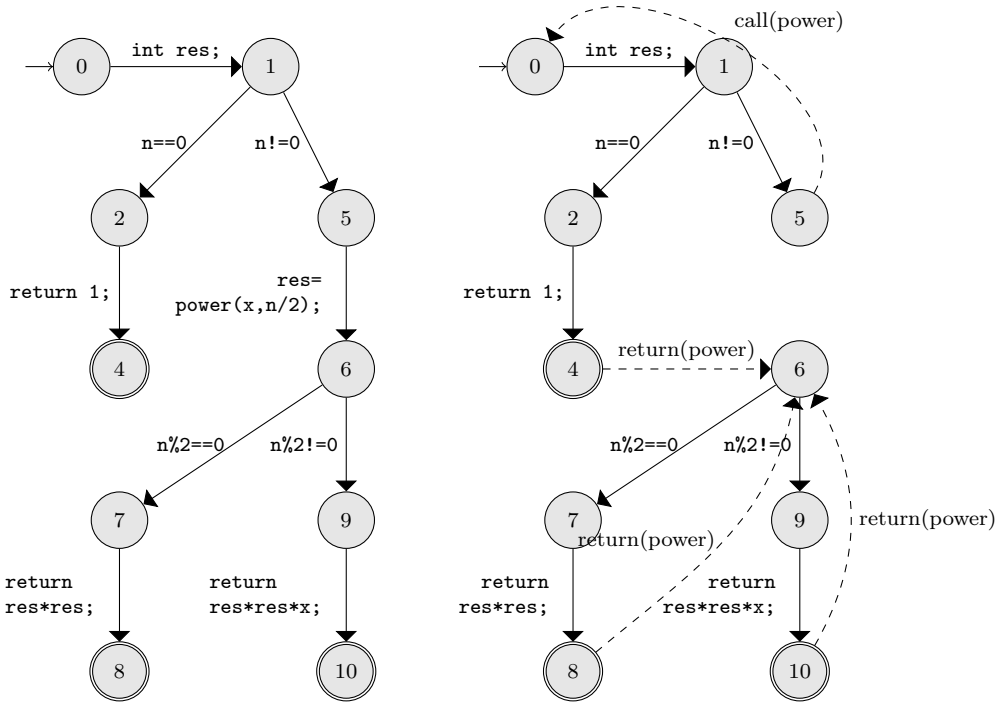


Figure 3: Examples of a control flow graph

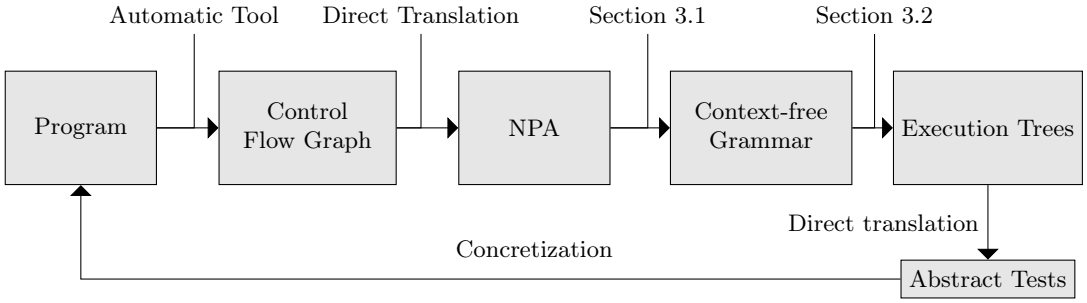


Figure 4: Random generation process

language of $[(acd)^*abef(gif + hjf)^*(gi + hj)] + (acd)^*abe$ (if considered paths are from 0 to a final state). Moreover, the accepted words corresponding to the correct occurrences[1] of *call(power)*'s and *return(power)*'s are those described by $[(acd)^k abef(gif + hjf)^{k-1}(gi + hj)] + abe$, where $k \geq 1$. For instance, path $0 \to 1 \to 5 \dashrightarrow 0 \to 1 \to 2 \to 4$, corresponding to *acdabe* is a successful path (also called a DFA-trace) in the automaton but cannot be concretized since the call to the `power` function (letter d) is never returned. Therefore, the probability that a path of length n in the automaton corresponds to a consistent path w.r.t. calls/returns of the `power` function is in $o(\frac{1}{2^{n/6}})$ and converges to 0 while n grows to $+\infty$. It implies that the approach consisting in generating paths uniformly at random in the finite automaton leads to un-concretizable tests most of the time. Moreover, it rules out the possibility of using a rejection approach consisting in generating DFA-traces until they correspond to NPDA-traces when generating long test cases.

1.3 Layout of the Testing Approach

In a *structural testing context, i.e., when the test campaign is performed from the code of a program,* to generate abstract tests from a program, our approach consists in (1) generating its control flow graph, (2) translating its graph flow into a normalised pushdown automaton, (3) transforming this NPDA into an equivalent non-ambiguous context-free grammar, (4) uniformly generating execution trees of the grammar and (5) translating back these trees into paths in the program. This approach is depicted in Fig. 4. Note that automatic tools exist for Step (1) for a variety of used programming languages [5]. Step (2) only consists in transforming function calls into stack operations, and can easily be automatized. Step (3) is very basic [6, Section 2.2]. Abstract test cases at Step (4) are then PDA-traces directly obtained from execution trees of the grammar using the classical word generation/recognition by a context-free grammar. The GenRgenS tool [7] or the CS package of Mupad [8] can be used for this purpose. Afterwards, the concretization (5) consists in finding out some input values of the program which make the execution of the program correspond to a given NPDA-trace. Note that the concretization step requires the use of specific tools, for instance constraint solvers, and is not investigated in the paper.

For *model-based testing*, i.e., when test cases are generated from an abstract model of the system, the approach is similar when omitting the generation of the control flow graph step. Indeed, the program, or the system under test, is provided conjointly with its model, in this case a push-down automaton. Let us emphasise the fact that in this context the concretization step may be much harder, depending of the level of abstraction. However, this is a general issue for all model-based testing approaches whose solutions depend on how the model and the system are linked. This problem is out of the scope of this paper focusing on how to randomly generate the tests.

[1] It means that the word is consistent with the calls to `power` function, like a well braced expression.

The present paper addresses the problem of computing at random NPDA-traces from the NPDA (Steps (3) and (4)). The translation of a program into a pushdown automaton and the trace concretization steps (Steps (1), (2) and (5)) are out of the scope of the paper, and are only illustrated with several examples. Concretizing abstract test cases is a theoretically undecidable problem in both model-based testing or structural testing frameworks, and it is a central issue in the software testing context [9, 10, 11, 12].

1.4 Related Work

Graph-based Testing. Testing systems from finite state machines representing models of the system [13, 1] (model-based testing) consists in describing the system by a labelled transition system on which different algorithms may be used to extract the test cases. This is, for instance, the principle of SpecExplorer [14] or TGV [15]. Testing from control-flow graphs (structural testing) is one of the major testing approaches developed in hundreds of articles. The interested reader is referred to the reference paper [9] or to the reference book by P.Amman and J.Offut [16] for more information.
Grammar-based Testing. Grammar-based testing was used to test parsers [17] and re-factoring engines (program transformation software) [18]. A systematic generation for grammar based inputs is proposed by D.Coppiet and J.Lian [19]. However because of the explosion of tests, symbolic approaches are frequently preferred [20, 21, 22]. Recently, a generic tool for test generation from grammars has been developed [23]. This tool does not provide any random feature but is based on rule coverage approaches/algorithms, as in many other works [17, 24, 25, 26].
Random Testing. Random-based approaches for testing were initially proposed by J.W.Duran et al. [27] and R.Hamlet [28]. Random testing can be employed for generating test data, such as in DART [11] or to generate complete test sequences, as in the Jartege tool [29]. A recent work [4] provides an approach combining random testing and model-checking and is discussed deeper in this paper. Other approaches [30, 31, 32, 33] propose random and grammar-based techniques for testing: in this context, grammar are used to specify input data-structures. Note that such algorithms were used in a work by P.Godefroid et al. [22] for testing in a white-box fuzzing context. An approach combining model-based testing and randomness is presented in a paper by F.Dadeau et al. [34].

Uniform Random Generation of Derivation Trees. Combinatorial techniques to generate uniformly at random execution trees from context-free grammars were developed by Flajolet et al. [35, 36]. Several existing tools such as GenRgenS [7] or the CS package of Mupad [8] can be used for this purpose. The experiments presented in this paper make use of GenRgenS—a software dedicated to random generation of sequences that supports several classes of models, including Markov chains, context-free grammars, etc.

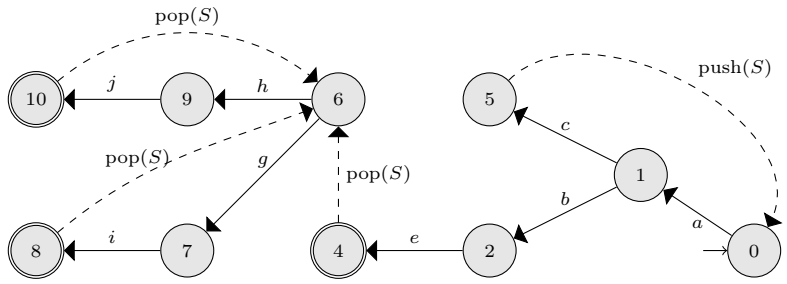


Figure 5: $\mathcal{A}_{\text{power}}$, example of a NPDA

2 Formal Background

If X is a finite set, X^* denotes respectively the set of finite words over X. The empty word (on every alphabet) is denoted ε. The set $X^* \setminus \{\varepsilon\}$ is denoted X^+ .

2.1 Pushdown Automata

A *deterministic finite automaton* is a tuple $(Q, \Sigma, \delta, q_{\text{init}}, F)$ where Q is a finite set of *states*, Σ is a finite alphabet, $q_{\text{init}} \in Q$ is the initial state, F is the set of final states and δ is a partial function from $Q \times \Sigma$ into Q. A *successful path* or a *DFA-trace* in a finite automaton is a (possibly empty) finite sequence of elements of $Q \times \Sigma \times Q$ of the form $(p_1, a_1, q_1) \dots (p_n, a_n, q_n)$ such that $p_1 = q_{\text{init}}$, $q_n \in F$ and for each i, $q_i = p_{i+1}$ and $\delta(p_i, a) = q_i$. The integer n is the length of the path and $a_1 \dots a_n$ is its label.

A *pushdown automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ where Q is a finite set of *states*, Σ and Γ are disjoint finite alphabets – Σ is the alphabet of the actions and Γ is the stack alphabet – satisfying $\varepsilon \notin \Sigma$ and $\perp \in \Gamma$, $q_{\text{init}} \in Q$ is the initial state, F is the set of final states and δ is a partial function from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ into $Q \times \Gamma^*$ such that for every $q \in Q$, for every $X \in \Gamma$, $a \in \Sigma \cup \{\varepsilon\}$, (1) if $\delta(q, a, X) = (p, w)$ then $w \in (\Gamma \setminus \{\perp\})^*$, and (2) if $\delta(q, a, \perp) = (p, w)$, then the first letter of w is $\perp$. Letter $\perp$ is called the *empty stack letter*. A *configuration* of a pushdown automaton is an element of $Q \times \{\perp\}(\Gamma \setminus \{\perp\})^*$ that is a pair whose first element is in Q and the second is a word starting by $\perp$ and whose others letters are not $\perp$. Informally, the second part encodes the current value of the stack. The *initial configuration* is $(q_{\text{init}}, \perp)$. Two configurations $(q, \perp u)$ and $(p, \perp v)$ are *a-consecutive*, with $a \in \Sigma \cup \{\varepsilon\}$, either if $u = \varepsilon$ and $\delta(q, a, \perp) = (p, \perp v)$, or if u is of the form $u_0 X$ with $X \in \Gamma \setminus \{\perp\}$ and there exists $w \in \Gamma^*$ such that $\delta(q, a, X) = (p, w)$ and $v = u_0 w$.

A *NPDA-trace* of length n in a pushdown automaton is a sequence $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ where the C_i's are configurations, the a_i's are in $\Sigma \cup \{\varepsilon\}$ and such that C_1 is the initial configuration, for each i, C_i and C_{i+1} are a_i-consecutive, and C_{n+1} is of the form $(p, \perp)$ with $p \in F$. A *normalised pushdown automaton* (NPDA)

is a pushdown automaton such that for every state q, every $a \in \Sigma \cup \{\varepsilon\}$, every $X \in \Gamma$, if $\delta(q, a, X) = (p, w)$ then one of the following cases arises:

(i) $a = \varepsilon$ and $w = \varepsilon$, or

(ii) $a = \varepsilon$ and w is of the form $w = XY$ with $Y \in \Gamma$, or

(iii) $a \neq \varepsilon$ and $w = X$.

It is also required that if $\delta(q, \varepsilon, X) = (p, XY)$, then for every $Z \in \Gamma$, $\delta(q, \varepsilon, Z) = (p, ZY)$: transition pushing an element on the top of the stack can be fired independently of the stack[2].

Intuitively, case (i) corresponds to *pop* on the stack, case (ii) to *push* and case (iii) to an action that does not modify the stack. The *underlying finite automaton* of an NPDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ is the finite automaton $(Q, \Sigma \cup \{\text{pop(X)}, \text{push(X)} \mid X \in \Gamma\}, \mu, q_{\text{init}}, F)$ where μ is defined by

(i) $\delta(q, \varepsilon, X) = (p, \varepsilon)$ iff $\mu(q, \text{pop(X)}) = p$,

(ii) $\delta(q, \varepsilon, X) = (p, XY)$ iff $\mu(q, \text{push(Y)}) = p$,

(iii) $\delta(q, a, X) = (p, X)$ iff $\mu(q, a) = p$.

The above definitions can be illustrated with the NPDA depicted in Fig. 5, corresponding to the graph on the right of Fig. 3. For this automaton $Q = \{0, \ldots, 10\}$, $\Sigma = \{a, b, c, e, g, h, i, j\}$, $\Gamma = \{\perp, S\}$, $q_{\text{init}} = 0$, $F = \{4, 8, 10\}$ and δ is defined by the arrows: if there is an edge of the form (q, a, p) with $a \in \Sigma$, then one has $\delta(q, a, \perp) = (p, \perp)$ and $\delta(a, \perp) = (p, S)$; if there is an edge of the form $(q, \text{pop}(S), p)$ then $\delta(q, \varepsilon, S) = (p, \varepsilon)$; if there is an edge of the form $(q, \text{push}(S), p)$ then $\delta(q, \varepsilon, S) = (p, SS)$ and $\delta(q, \varepsilon, \perp) = (p, \perp S)$. The sequence

$$(0, \perp)a(1, \perp)c(5, \perp)\varepsilon(0, \perp S)a(1, \perp S)c(5, \perp)\varepsilon(0, \perp SS)a(1, \perp SS)b(2, \perp SS)$$
$$e(4, \perp SS)\varepsilon(6, \perp S)h(9, \perp S)j(10, \perp S)\varepsilon(6, \perp)g(7, \perp)i(8, \perp)$$

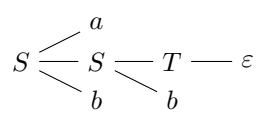
is a NPDA-trace in the NPDA. This trace corresponds to the execution of `power(3,2)`, already pointed out in Sec. 1.2.2. Note that each pair (q, w) encodes that the system is in state q and that the call stack is w. For instance, $(2, \perp SS)$ means that system is in state 2 and that there are two no-closed calls to the function `power`.

Each NPDA-trace of an NPDA is associated with a DFA-trace in the underlying automaton using the natural projection: the NPDA-trace $C_1 a_1 \ldots a_n C_n$ is associated with the path $(q_1, a_1, q_2) \ldots (q_{n-1}, a_n, q_n)$, where the C_i's are of the form (q_i, w_i). For instance, the DFA-trace associated with the above NPDA-trace is $(0, a, 1)(1, c, 5)(5, \varepsilon, 0) \ldots (6, g, 7)(7, i, 8)$. This projection is denoted proj and is injective. Its image forms a subset of DFA-traces, called *consistent DFA-traces* of the underlying automaton, which is in bijection with the set of NPDA-traces.

[2]Actually this is not a restriction for the expressiveness since it is possible to encode a transition of the form $\delta(q, \varepsilon, X) = (p, XY)$, requiring X on the top of the stack to be fired into $\delta(q, \varepsilon, X) = (q_{\text{new1}}, \varepsilon)$ and for every $Z \in \Gamma$, $\delta(q_{\text{new1}}, \varepsilon, Z) = (q_{\text{new2}}, ZX)$ and $\delta(q_{\text{new2}}, \varepsilon, Z) = (p, ZY)$

2.2 Context-free Grammars

A *context-free grammar* is a tuple $G = (\Sigma, \Gamma, S_0, R)$, where Σ and Γ are disjoint finite alphabets, $S_0 \in \Gamma$ is the initial symbol, and R is a finite subset of $\Gamma \times (\Sigma \cup \Gamma)^*$. Elements of Γ are called *non-terminal symbols*. An element of R is called *a rule* of the grammar. A word $w \in (\Sigma \cup \Gamma)^*$ is a *successor* of $v \in (\Sigma \cup \Gamma)^*$ for the grammar G if there exist $v_0 \in \Sigma^*$, $v_1, v_2 \in (\Sigma \cup \Gamma)^*$, $S \in \Gamma$ such that $v = v_0 S v_1$ and $w = v_0 v_2 v_1$ and $(S, v_2) \in R$. A *complete derivation* of the grammar G is a finite sequence $x_0, \ldots, x_n$ of words of $(\Sigma \cup \Gamma)^*$ such that $x_0 = S_0$, $x_n \in \Sigma^*$ and for every i, x_{i+1} is a successor of x_i. A *derivation tree* of G is a finite tree whose internal nodes are labelled by letters of Γ, whose leaves are labelled by elements of $\Sigma \cup \{\varepsilon\}$, whose root is labelled by S_0 and satisfying: if a node is labelled by $X \in \Gamma$ and if its children are labelled by $\alpha_1, \ldots, \alpha_k$ (in this order), then $(X, \alpha_1 \ldots \alpha_k) \in R$ and either $\alpha_1 = \varepsilon$ and $n = 1$, or all the α_i's are in $\Gamma \cup \Sigma$. Consider for instance the grammar $G = (\{a, b\}, \{S, T\}, S, R)$, with $R = \{(S, Tb), (S, aSb), (T, \varepsilon)\})$. The sequence $S, aSb, aTbb, abb$ is a complete derivation of the grammar. The associated derivation tree is



Note that there is a natural bijection between the set of complete derivations of a grammar and the set of derivation trees of this grammar.

3 Uniform Random Generation of Consistent DFA-Traces

Given a NPDA, let us remind that the main goal of the paper is to uniformly generate successful consistent traces of a given length in its underlying finite automaton. In Sec. 3.1 a well-known connection between NPDA and context-free grammars is recalled. A uniform random generation of consistent DFA-traces is then explained in Sec. 3.2.

Note that in this paper, test cases are consistent DFA-traces or, equivalently (up to a trivial bijection), NPDA-traces.

3.1 From NPDA to Context-free Grammars

Transforming a NPDA into a context-free grammar can be done using classical algorithms on pushdown automata [6]. The following result is a direct combination of well-known results on pushdown automata.

Theorem 1 *Let $\mathcal{A}$ be a pushdown automaton. One can compute in polynomial time a grammar G satisfying the following assertions:*

$$
\begin{aligned}
\Sigma_G =& \{(0,a,1),(1,c,5),(5,\mathrm{push}(S),0),(1,b,2),(2,e,4),\\
&(4,\mathrm{pop}(S),6),(6,g,7),(7,i,8),(8,\mathrm{pop}(S),6),(6,h,9),(9,j,10),\\
&(10,\mathrm{pop}(S),6)\},\text{and}\\
R_G =& \{(T,LRD(6,g,7)(7,i,8)),(T,LRD(6,h,9)(9,j,10)),\\
&(T,(0,a,1)(1,b,2)(2,e,4)),(R,LRD)\\
&(R,(0,a,1)(1,c,5)(5,\mathrm{push}(S),0)(0,a,1)(1,b,2)(2,e,4)(4,\mathrm{pop}(S),6)),\\
&(L,(0,a,1)(1,c,5)(5,\mathrm{push}(S),0)),\\
&(D,(6,g,7),(7,i,8)(8,\mathrm{pop}(S),6)),\\
&(D,(6,h,9)(9,j,10)),(10,\mathrm{pop}(S),6))\}.
\end{aligned}
$$

Figure 6: Example of a grammar generating consistent DFA-traces

- *The size of G is at most quadratic in the size of $\mathcal{A}$, and there is no rule of the form (X,Y) in G, where X and Y are stack symbols.*
- *There exists a bijection φ from the set of complete derivations of G and the set of NPDA-traces of $\mathcal{A}$.*
- *Given a complete derivation of G, its image by φ can be computed in polynomial time.*

Note that the precise complexity of algorithms depends on the chosen data-structures, but all of them can be implemented in a very efficient way.

Consider for instance the underlying finite automaton for the NPDA depicted in Fig. 5. Consistent DFA-traces of $\mathcal{A}$ are generated by the grammar $(\Sigma_G, \Gamma_G, T, R_G)$ in Fig. 6, where $\Gamma_G = \{T, L, R, D\}$.

3.2 Random Generation of Consistent DFA-Traces

The random generation of consistent DFA-traces of a NPDA is performed using Theorem 1. First, the related grammar G is computed. Next, derivation trees are randomly generated: successful traces are computed using φ. Since φ is bijective, if the random generation of derivation trees is uniform, so is the random generation of consistent DFA-traces (using the fact that proj is bijective). The general scheme of the generation process is sketched in Fig. 7.

The random generation of derivation trees is performed using classical combinatorial techniques [37] that are sketched below. Let $G = (\Sigma, \Gamma, X, R)$ be a context-free grammar satisfying the conditions of Theorem 1. For each symbol $S \in \Gamma$, the sequence of positive integers $s(1), \ldots, s(k), \ldots$ is introduced, where $s(i)$ is the number of size i derivation trees of (Σ, Γ, S, R). The recursive computation of these $s(i)$'s is as follows. For each strictly positive integer k and

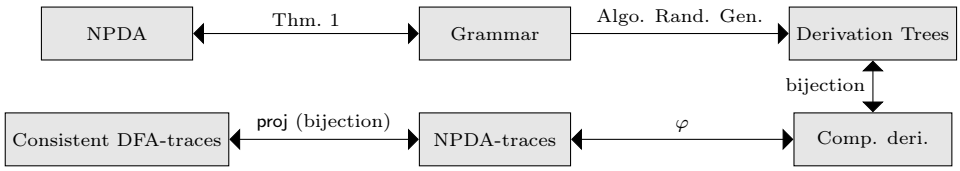


Figure 7: Random generation of consistent DFA-traces

each rule $r = (S, w_1 S_1 \dots w_n S_n w_{n+1}) \in R$, with $w_j \in \Sigma^*$ and $S_i \in \Gamma$, let

$$\begin{cases} \beta_r = 1 + \sum_{i=1}^{n+1} |w_i| \\ \alpha_r(k) = \sum_{i_1+i_2+\dots+i_n=k-\beta_r} \prod_{j=1}^{j=n} s_j(i_j) \quad \text{if } n \neq 0 \\ \alpha_r(k) = 0 \quad \text{if } n = 0 \text{ and } k \neq \beta_r \\ \alpha_r(\beta_r) = 1 \quad \text{if } n = 0. \end{cases}$$

It is known [37, Theorem I.1] that $s(k) = \sum_{r \in R \cap (S \times (\Sigma \cup \Gamma)^*)} \alpha_r(k)$.

Since, by hypothesis, there is no rule of the form (S, T) in R, with $S, T \in \Gamma$, all i_j's involved in the definition of β_r are strictly less than k. This way, the $s(i)$'s can be recursively computed. Consider, for instance, the grammar $(\{a, b\}, \{X\}, X, \{r_1, r_2, r_3\})$ with $r_1 = (X, XX)$ $r_2 = (X, a)$ and $r_3 = (X, b)$. One has $\beta_{r_1} = 1 + 0 = 1$, $\beta_{r_2} = 1 + 1 = 2$, $\beta_{r_3} = 1 + 1 = 2$. Therefore, $x(k) = \sum_{i+j=k-1} x(i)x(j)$ if $k \neq 2$ and $x(2) = 1 + 1 + \sum_{i+j=2-1} x(i)x(j) = 2$ otherwise. It follows that $x(1) = 0$, $x(2) = 2$, $x(3) = x(1)x(1) = 0$, $x(4) = x(1)x(2) + x(2)x(1) = 0$, $x(5) = x(2)x(2) = 4$, etc. The two derivation trees of size 2 are $\overset{X}{\underset{a}{|}}$ and $\overset{X}{\underset{b}{|}}$. The four derivation trees of size 5 are the trees of the form $\overset{X}{\underset{Z_1\ Z_2}{/\backslash}}$ where both Z_1 and Z_2 are derivation trees of size 2.

Consider for instance the grammar described in Fig. 6. The computation provides that $T(3) = 1$, $T(9) = 2$, $T(15) = 4$, $T(21) = 8$, and $T(i) = 0$ for all other values of i less or equal to 21. The result $T(9) = 2$ points out that there are two DFA-traces of length 9: $(0, a, 1)(1, c, 5)(5, \text{push}(S), 0)(0, a, 1)(1, b, 2)(2, e, 4)(4, \text{pop}, 6)(6, g, 7)(7, i, 8)$ and $(0, a, 1)(1, c, 5)(5, \text{push}(S), 0)(0, a, 1)(1, b, 2)(2, e, 4)(4, \text{pop}, 6)(6, h, 9)(9, j, 10)$.

In order to generate derivation trees of size n, all $s(i)'s$, for $S \in \Gamma$ and $i \leq n$, have to be computed with the above method. This can be performed in polynomial time. Afterwards, the random generation is done recursively using the Random Generation algorithm in Fig. 8.

It is known [37] that this algorithm provides a uniform generation of derivation trees of size n, i.e., each derivation tree occurs with the same probability. Note that an exception is returned at Step 2 if there is no element of the given size. For the example presented before, there is no element of size 3, then it is impossible to generate a derivation tree of size 3. Running the algorithm on this example with $n = 2$, one consider at Step 1 the set $\{r_1, r_2, r_3\}$ since all these rules have X as left element. Since $\alpha_{r_1}(2) = 0$, $\alpha_{r_2}(2) = 1$, $\alpha_{r_3}(2) = 1$, at

Random Generation
Input: $G = (\Sigma, \Gamma, X, R)$ a context-free grammar, n a strictly positive integer.
Output: a derivation tree t of G size n.

Algorithm:

1. Let $\{r_1, r_2, \ldots, r_\ell\}$ be set of the elements of R whose first element is X.
2. **If** $\sum_{j=1}^{j=\ell} \alpha_{r_j}(n) = 0$, **then return** "Exception".
3. **Pick** $i \in \{1, \ldots, \ell\}$ with probability $Prob(i = j) = \frac{\alpha_{r_i}(n)}{\sum_{j=1}^{j=\ell} \alpha_{r_j}(n)}$.
4. Let $r_i = (X, Z_1 \ldots Z_k)$, with $Z_j \in \Sigma \cup \Gamma$.
5. Root symbol of t is X.
6. Children of t are $Z_1, \ldots, Z_k$ in this order.
7. Let $\{i_1, \ldots, i_m\} = \{j \mid Z_j \in \Gamma\}$.
8. **Pick** $(x_1, \ldots, x_m) \in \mathbb{N}^m$ such that $x_1 + \ldots + x_m = n - \beta_{r_i}$ with probability

$$Prob(x_1 = \ell_1, \ldots, x_m = \ell_m) = \frac{\prod_{j=1}^{j=m} z_{i_j}(\ell_j)}{\alpha_{r_i}(n)}.$$

9. For each i_j, the i_j-th sub-tree of T is obtained by running the **Random Generation** algorithm on $(\Sigma, \Gamma, Z_{i_j}, R)$ and ℓ_j.
10. **Return** t.

Figure 8: Random Generation algorithm

Step 3 the probability that $i = 1$ is null, the probability that $i = 2$ is $1/2$ and the probability that $i = 3$ is $1/2$. If $i = 2$ is picked, the generated tree has X as root symbol and a as unique child. Running the algorithm on this example with $n = 3$ stops at Step 2 since there is no tree of size 3. When running the algorithm on this example with $n = 5$, the set $\{r_1, r_2, r_3\}$ is considered at Step 1. Since $\alpha_{r_1}(5) = 4$, $\alpha_{r_2}(5) = 0$, $\alpha_{r_3}(5) = 0$, $i = 1$ is picked with probability 1. Therefore, the tree has X as root symbol, and its two children are both labelled by X. Therefore, at Step 7, the considered set is $\{1, 2\}$. At Step 8, one has $n - \beta_{r_1} = 5 - 1 = 4$. The probability that $i_1 = 1$ and $i_2 = 3$ is null since $x(1) = 0$. Similarly, the probability that $i_1 = 3$ and $i_2 = 1$ is null too. Now the probability that $i_1 = 2$ and $i_2 = 2$ is 1. Afterwards the algorithm is recursively executed on each child with $n = 2$: each of the 4 trees is chosen with probability $1/4$.

From a NPDA with n states, the generation of k NPDA-traces of length n can be performed in time $O(n^6 + n^3 k \log(n))$. Notice that computing a grammar from a pushdown automaton requires, in the worst case, $\Omega(n^3)$ operations, leading to the given complexity. In practice, as many computed grammar rules are useless, i.e. they cannot be used in a successful execution, it is frequently possible to drastically reduce the size of the grammar to be much more efficient for the random generation. Using more advanced random generation techniques, as Botlzmann samples, may also improve both theoretical and practical complexity.

4 Quality of the Test Suite and Coverage Criteria

Most of testing techniques are based on coverage criteria to ensure a kind of sufficiency of the test suites. In this section the combination of coverage criteria with random testing is investigated, in order to take benefit of both approaches.

Given a pushdown automaton, a coverage criterion is a set C defined using this pushdown automaton or its underlying automaton. In general C is either the set of states, or the set of transitions, or the the set of all paths with a loop restriction. Following [**?**] and [4], one can define the quality of a randomised testing technique as the minimal probability $q_{C,N}$ for covering any element of C when drawing N random tests (of length n). Since tests are independent, one has $q_{C,N} = 1-(1-q_{C,1})^N$. Therefore, computing or estimating $q_{C,1}$ is a central issue to determine a priori the coverage power of the approach. For a dual approach consisting in generating tests until all elements of C were covered, the average number of required tests is bounded by $\frac{|C|}{q_{C,1}}$. Since probabilities of covering different elements by generating a trace are not independent, the exact computation of the expected number of required tests cannot be performed in a general case.

4.1 Criterion *All states*

The criterion *All states*, AS for short, is defined by the set of states of the pushdown automaton. In this context $q_{AS,1}$ is the minimal probability of visiting a state by generating a derivation of length n. For each state q_0 of the pushdown automaton, let us denote by $pr(q_0)$ the probability that an execution trace of length n corresponds to a path visiting q_0. Clearly $q_{AS,1} = \min_q\{pr(q)\}$. Since there are finitely many states, $q_{AS,1}$ can be deduced by computing all $pr(q)$'s. Since $pr(q_0)$ is the number of NPDA-traces of length n visiting q_0 divided by the number of NPDA-traces of length n[3], it suffices to compute these two numbers. The second number corresponds to the value $s(n)$ defined at the beginning of Sec. 3.2. It remains to compute the number of NPDA-traces of length n visiting q_0, which is done using the construction described bellow (which is an adaptation of the classical product of automata).

Let $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ be a normalised pushdown automaton and $q_0 \in Q$. The automaton $\mathcal{A}^{q_0}$ is set as $(Q \times \{0,1\}, \Sigma, \Gamma, \delta', (q_{\text{init}}, 0), F')$, with $F' = F \times \{1\}$ if $q_0 \notin F$ and $F' = F \times \{1\} \cup \{(q_0, 0)\}$ if $q_0 \in F$. Moreover, $\delta'(q, a, X)$ is defined as follows:

- If $q \neq q_0$ and if $\delta(q, a, X) = (p, w)$, then $\delta'((q, 0), a, X) = ((p, 0), w)$ and $\delta'((q, 1), a, X) = ((p, 1), w)$.
- If $q = q_0$ and if $\delta(q, a, X) = (p, w)$, then $\delta'((q_0, 0), a, X) = ((p, 1), w)$ and $\delta'((q_0, 1), a, X) = ((p, 1), w)$.

[3] Assuming that there are NPDA-traces of length n; otherwise random testing is not possible.

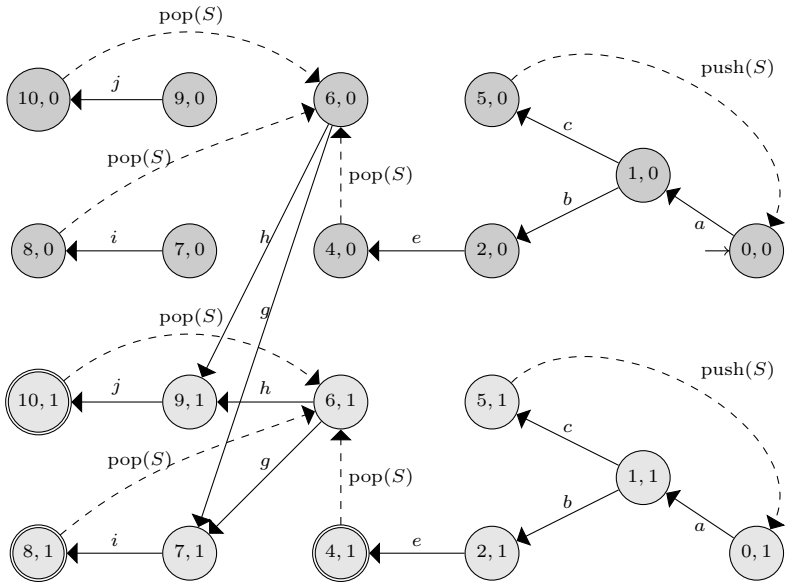


Figure 9: Underlying automaton of $\mathcal{A}^6_{exe}$

Intuitively, the boolean added to the states tells whether or not the state q_0 has been visited before. For instance, consider the automaton $\mathcal{A}_{exe}$ whose underlying automaton is depicted in Fig. 5. The automaton $\mathcal{A}^6_{exe}$ has an underlying automaton depicted in Fig. 9. Note that this automaton can be trimmed and simplified by deleting useless states $(9,0)$, $(10,0)$, $(8,0)$, $(7,0)$, $(0,1)$, $(1,1)$, $(2,1)$ and $(5,1)$.

Let θ be the function from the set of configurations of $\mathcal{A}^{q_0}$ into the set of configurations of $\mathcal{A}$. The following result is a direct consequence of the definition of $\mathcal{A}^{q_0}$.

Proposition 2 *For any NPDA-trace $C_1a_1C_2a_2\dots C_na_nC_{n+1}$ of $\mathcal{A}^{q_0}$, the sequence $\theta(C_1)a_1\theta(C_2)a_2\dots\theta(C_n)a_n\theta(C_{n+1})$ is a NPDA-trace of $\mathcal{A}$ visiting q_0.*

Proof. One has $C_1 = ((q_{\text{init}},0),\bot)$, therefore $\theta(C_1) = ((q_{\text{init}},\bot)$ is the initial configuration of $\mathcal{A}$. Similarly C_{n+1} is of the form $((q_f,1),\bot)$ with $q_f \in F$. Therefore $\theta(C_{n+1}) = ((q_f,\bot)$ is in a final state with the empty stack.

By construction of δ' and since C_i and C_{i+1} are a_i-consecutive, $\theta(C_i)$ and $\theta(C_{i+1})$ are a_i-consecutive too. It follows, by a direct induction, that $\theta(C_1)a_1\theta(C_2)a_2\dots\theta(C_n)a_n\theta(C_{n+1})$ is a NPDA-trace of $\mathcal{A}$.

It remains to prove that $\theta(C_1)a_1\theta(C_2)a_2\dots\theta(C_n)a_n\theta(C_{n+1})$ visits q_0. If C_{n+1} is of the form $((q_f,1),\bot)$, then, since $C_1 = ((q_{\text{init}},0),\bot)$, there exists i such that C_i is of the form $((q_i,0),\bot u_i)$ and C_{i+1} is of the form $((q_i,1),\bot u_{i+1})$. By construction of δ', $q_i = q_0$. If $q_0 \in F$ and if C_{n+1} is of the form $((q_0,0),\bot)$, then $\theta(C_{n+1}) = (q_0,\bot)$, which concludes the proof.

□

Conversely, for each NPDA-trace of $\mathcal{A}$ visiting q_0, there exists a unique execution in $\mathcal{A}^{q_0}$ corresponding to it.

Proposition 3 *For any NPDA-trace $C'_1 a_1 C'_2 a_2 \dots C'_n a_n C'_{n+1}$ of $\mathcal{A}$ visiting q_0, there exists a unique NPDA-trace $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ of $\mathcal{A}^{q_0}$, such that for each i, $\theta(C_i) = C'_i$.*

Proof. Let $C'_i = (q_i, \perp u_i)$ for every i. Assuming that $q_{n+1} \neq q_0$, then, since $C'_1 a_1 C'_2 a_2 \dots C'_n a_n C'_{n+1}$ visits q_0, there exists $i_0 = \min\{i \mid q_i = q_0\}$. Set $C_i = ((q_i, 0), \perp u_i)$ if $i \leq i_0$ and $C_i = ((q_i, 1), \perp u_i)$ otherwise. By construction of δ', one can easily check that $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ is a NPDA-trace of $\mathcal{A}^{q_0}$ and that for each i, $\theta(C_i) = C'_i$. Now if $q_0 \in F$ and $C_{n+1} = (q_0, 0)$, then for every i set $C_i = ((q_i, 0), \perp u_i)$. Again, $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ is a NPDA-trace of $\mathcal{A}^{q_0}$.

Assume now that $D_1 a_1 D_2 a_2 \dots D_n a_n D_{n+1}$ is a NPDA-trace of $\mathcal{A}^{q_0}$ such that for each i, $\theta(D_i) = C'_i$. By definition of θ, D_i is of the form $((q_i, b_i), \perp u_i)$. First, if $C_{n+1} = (q_0, 0)$, then, by definition of δ', for $i \leq i_O$, $b_i = 1$, and for $i \geq i_0$, $b_i = 0$. It follows that for every i, $D_i = C_i$. Similarly, if $q_0 \in F$ and $C_{n+1} = (q_0, 0)$, then, by definition of θ, D_i is of the form $((q_i, 0), \perp u_i)$, proving the uniqueness and concluding the proof.

□

It follows that θ induces a bijection between NPDA-traces of length n of $\mathcal{A}^{q_0}$ and NPDA-traces of length n of $\mathcal{A}$ visiting q_0. Using the approach described in Sec. 3.2 on $\mathcal{A}^{q_0}$, one can compute the number of NPDA-traces in $\mathcal{A}$ visiting q_0, and $pr(q_0)$ in the same time.

4.2 Criterion *All transitions*

The criterion *All transitions*, AT for short, is defined by the set of transitions of the underlying automaton: $q_{AT,1}$ is the minimal probability of using a transition by generating a derivation of length n. For each transition (q_0, a_0, p_0) of the underlying automaton, let us denote by $pr((q_0, a_0, p_0))$ the probability that an execution trace of length n corresponds to a path using (q_0, a_0, p_0). As for the *All states* criterion, $q_{AT,1} = \min_{(q,a,q')}\{pr((q, a, q'))\}$, which can be computed when all $pr((q, a, q'))$ are computed. Moreover, $pr((q_0, a_0, p_0))$ is the number of NPDA-traces of length n using (q_0, a_0, p_0) divided by the number of NPDA-traces ($s(n)$, which is supposed to be strictly positive).

Let $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ be a normalised pushdown automaton, and (q_0, a_0, q_1) be a transition of the underlying automaton. The automaton $\mathcal{A}^{(q_0,a_0,q_1)}$ is set as $(Q \times \{0, 1\}, \Sigma, \Gamma, \delta^{\square}, (q_{\text{init}}, 0), F \times \{1\})$, where $\delta^{\square}(q, a, X)$ is defined as follows:

- If $a_0 = \text{push(Y)}$ and if $\delta(q, a, X) = (p, w)$, then
  - if $q \neq q_0$ or if $\delta(q, \varepsilon, X) \neq (p, XY)$, then $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ and $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - if $q = q_0$ and if $\delta(q, \varepsilon, X) = (p, XY)$, then $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), XY)$ and $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 1), XY)$;
- If $a_0 = \text{pop(Y)}$ and if $\delta(q, a, X) = (p, w)$, then
  - if $q \neq q_0$ or if $\delta(q, \varepsilon, X) \neq (p, XY)$, then $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ and $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - if $q = q_0$ and if $\delta(q, \varepsilon, X) = (p, \varepsilon)$, then $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), \varepsilon)$ and $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 1), \varepsilon)$;
- If $a_0 \in \Sigma$ and if $\delta(q, a, X) = (p, w)$, then
  - if $q \neq q_0$ or if $a_0 \neq a$ or if $\delta(q, a, X) \neq (p, \varepsilon)$, then $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ and $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - if $q = q_0$ and if $\delta(q, a_0, X) = (p, \varepsilon)$, then $\delta^{\square}((q, 1), a_0, X) = ((p, 1), \varepsilon)$ and $\delta^{\square}((q, 0), a_0, X) = ((p, 1), \varepsilon)$;

Intuitively, the boolean added to the states tells whether or not the transition (q_0, a_0, p_0) has been used before: it is switched from 0 to 1 when the transition (q_0, a_0, p_0) is fired. For instance, let us consider again the automaton $\mathcal{A}_{exe}$. The underlying automaton of $\mathcal{A}_{exe}^{(6,h,9)}$ is depicted in Fig. 10. Again, several useless states may be removed.

In a way very close to the one used for the *All states* criterion, one can check that there is a bijection between NPDA-traces of length n in $\mathcal{A}$ using (p_0, a_0, q_0) and the NPDA-traces of length n in $\mathcal{A}^{(p_0,a_0,q_0)}$, allowing to compute $pr((p_0, a_0, q_0))$.

4.3 Combining Random Testing and a Coverage Criterion

This section discusses the combination of the random testing and the coverage criteria. The discussion concerns the *All states* criterion and exploits the results of Sec. 4.1. However, the *All transitions* criterion can be handled in exactly the same way using the results of Sec. 4.2.

In order to combine the random testing and the *All states* coverage criterion on a pushdown automaton $\mathcal{A}$, a natural approach could be as follows.

1. Compute all the $pr(q)$'s.
2. If some of them are equal to 0, then it is not possible to cover all states by generating executions of length n.
3. Otherwise, pick at random a state q that has not been concerned yet with previously generated tests (if any).

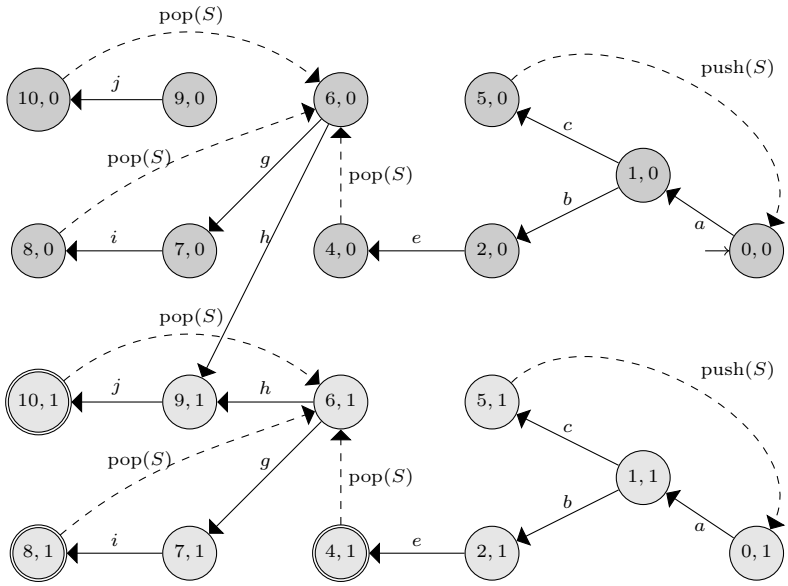


Figure 10: Underlying automaton of $\mathcal{A}_{exe}^{(6,h,9)}$

4. Using $\mathcal{A}^q$, generate a test visiting q.
5. If there is a state that has not been visited yet by the test suites, goto step 3.

When using this approach, the number of tests is bounded by the number of states. Unfortunately, the state space can be very large and thus cause trouble when tests are hard to execute, for instance when physical manipulations are required for playing the tests. In this case the above procedure can be stopped after a fixed number N of generated tests. Following the work by A.Denise et al. [4, Section 5.2], it is possible to optimise the expected quality of the test suites by choosing the state q at step 3 with a non uniform distribution. This method is not described here in details, the interested reader is referred to original paper [4] for explanations. The method consists in choosing, at step 3 the state q with a probability π_q defined by solving the following linear programming problem: maximise $p_{\min}$ satisfying

$$\begin{cases} p_{\min} \leq \sum_{p \in Q} \pi_p \frac{pr(p,q)}{pr(p)} \text{ for all } q \in Q \\ \sum_{q \in Q} \pi_q = 1 \end{cases}$$

where $pr(p,q)$ is the probability that a NPDA-trace of length n visits both p and q. The construction of $\mathcal{A}^q$ can be adapted for this purpose: the new set of states becomes $Q \times \{0,1\} \times \{0,1\}$, the first boolean encoding whether q has been visited, and the second one whether p has been visited. Final states are

$F \times \{1\} \times \{1\}$, ensuring that both states have been visited. Moreover, note that the above linear problem has to be solved with real numbers, and it can be done efficiently. Therefore, it is possible to compute the π_q's to have a biased generation that optimises the probabilistic quality relatively to the AS criterion. The same approach can be developed for the AT criterion.

5 Experiments

The main purpose of this section is to provide some quantitative experimental information on the approach proposed in Section 4. In Section 5.2, it is illustrated on an example of two mutually recursive functions. Then, another illustrative example is provided in Section 5.3 where an XPath query is translated into a pushdown automaton. Finally, Section 5.4 reports on time and space requirements for the examples of the present paper and from S.Schwoon PhD [38]. Note that an additional qualitative study of the uniform random testing algorithm is presented in Section 6.

5.1 Technical Information

All the experiments described in this section have been obtained on an Intel-Core2 Duo 2GHz personal computer with 4GB of RAM, running on Ubuntu 10.04. In order to validate the approach and to obtain an order of magnitude of the running times and of the sizes of the involved grammars, the algorithms have been implemented within a Python 2.6 prototype. Python is a script/prototyping programming language whose main advantage consists in allowing fast code developments. However, it is worth mentioning that it is frequently admitted that C/C++ implementations are 10 to 100 times faster than Python implementations, particularly for programs managing large data structures. Consequently, the time estimations presented in this section have to be considered in this context.

Python is an interpreted language using several internal mechanisms consuming space. The space bottleneck of the approach is the size of the involved grammars. Therefore, in order to estimate the space consumption, it seems more relevant to provide the sizes of the involved grammars rather than the used memory. Notice also that the experiments point out that the critical resource for the approach is time, not space.

5.2 Illustrating Example

This section illustrates the proposed random testing approach on the program of Fig. 11, using two mutually recursive functions `S` and `M`.

The `M(x,n)` function computes $x\%n$, and the `S(x,y,n)` function computes $(x * y)\%n$. Our objective is to test the function `S` using the random approach developed in this paper. The NPDA $\mathcal{A}_{\text{modulo}}$ associated with this pair of functions is depicted in Fig. 12. The top part of the NPDA describes the `S` function,

```
int S(int x, int y, int n){
    int z;
    if (y == 1){                int M(int x, int n){
        z = M(x,n);                 if (x < 1){
        return z;                       return x;
    } else {                        } else {
        z = x + S(x,y-1,n);             return M(x-n,n);
        z = M(z,n);                 }
        return z;               }
    }
}
```

Figure 11: Mutually recursive functions

whereas its bottom part describes the `M` function. The recursive calls to `M` in `S` are encoded by the stack symbols M_1 (for the case `y==1`) and M_2 (for the case `y!=1`). The stack symbol M_3 encodes the call to `M` in `M`.

NPDA vs. Finite Automata. Using the NPDA $\mathcal{A}_{\text{modulo}}$, a related grammar satisfying the properties of Theorem 1 can be computed. The computed grammar has 45 stack symbols and 50 rules. To compare our approach with the results of A.Denis et al. [4], the related random generation approach has been implemented. Table 1 presents the obtained results. The first column contains the length of generated traces/paths in the underlying automaton in Fig. 12. For each length, 10 traces are generated. The second column reports on the generated NPDA-traces: the sequence of states is only given, and the number in brackets indicates the number of times this trace has been randomly generated. The content of the last column is similar but given for DFA-traces.

One can first observe that there is a unique NPDA-trace of length 8: 1-2-4-I-II-III-5-6-7, which is obviously generated at each time. Note this NPDA-trace corresponds to the execution of `S(3,1,2)`. Conversely, there are two DFA-traces of length 8: the first one corresponds to the NPDA-trace, whereas the second trace 1-2-4-I-II-III-10-11-12 is not consistent with stack calls. Therefore, this not consistent DFA-trace cannot be concretized, (i.e., there is no input value for S whose execution would correspond to this trace). The same situation occurs for generation of traces of size 10: there is no consistent DFA-trace but four DFA-traces.

These experimental results show that for traces of length 8, 40% of DFA-traces generated by the approach developed in the paper by A.Denise et al. [4] would be not consistent, as well as 100% for DFA-traces of length 11. Table 2 reports on the number of NPDA-traces—or, equivalently, of consistent DFA-traces, and on the number of DFA-traces for $\mathcal{A}_{\text{modulo}}$. It shows that the probability of obtaining a consistent DFA-trace by generating a DFA-trace becomes negligible when the trace length grows. For instance, the probability for a DFA-trace of length 80 to be consistent is $5.9\ 10^{-8}$. Moreover, for several lengths, as 21, 29 or 37, there is no NPDA-traces of this length: all generated

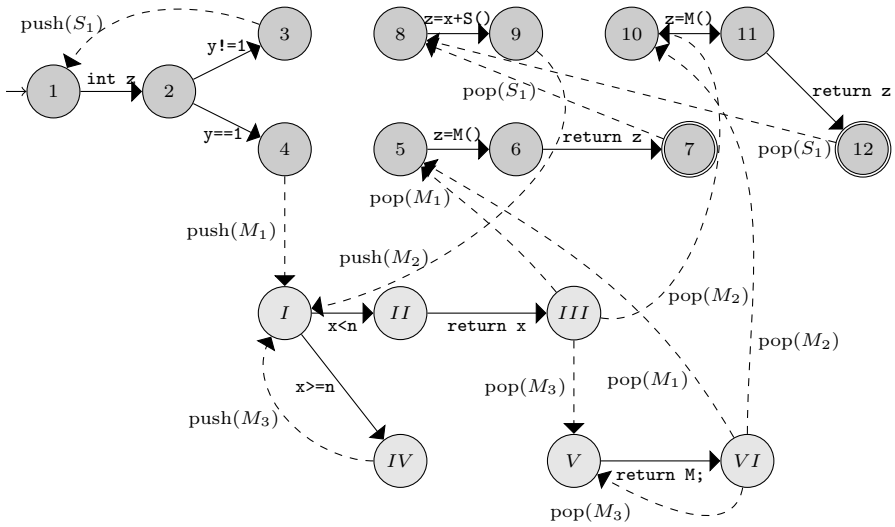


Figure 12: $\mathcal{A}_{\text{modulo}}$: NPDA of the running example

Size of the traces	Our approach	Approach of [4]
from 1 to 7	no NPDA-traces of this length	no DFA-traces of this length
8	1-2-4-I-II-III-5-6-7 (10)	1-2-4-I-II-III-10-11-12 (6) 1-2-4-I-II-III-5-6-7 (4)
9	no NPDA-trace of this length	no DFA-trace of this length
10	no NPDA-trace of this length	1-2-4-I-IV-I-II-III-5-6-7 (3) 1-2-4-I-IV-I-II-III-10-11-12 (2) 1-2-4-I-II-III-V-VI-10-11-12 (3) 1-2-4-I-II-III-V-VI-5-6-7 (2)
11	no NPDA-trace of this length	A-2-3-1-2-4-I-II-III-10-11-12 (6) 1-2-31-2-4-I-II-III-5-6-7 (4)

Table 1: Qualitative experimental results

n	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
NPDA	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0
DFA	6	4	10	6	18	10	34	18	64	34	114	64	200	114	356

n	27	28	29	30	31	32	33	34	35	36	37	80
NPDA	3	1	0	1	4	1	0	3	5	1	0	142
DFA	200	640	356	1152	640	2066	1152	3692	2066	6598	3692	$2.4\ 10^9$

Table 2: NPDA-traces vs. DFA-traces

DFA-traces would be not consistent.

In regards to test cases generation of length less or equal to 60, the pre-computation can be performed in less than 1 second. Next, 100 traces can be generated in less than 0.34 second. More information on the computation time for different lengths is available in Section 5.4.

Coverage Criteria. The approach developed in Section 4.1 has been applied to the example of $\mathcal{A}_{\text{modulo}}$. To deal with the *All states* criterion, for each state q of $\mathcal{A}_{\text{modulo}}$, the grammar generating NPDA-traces of $\mathcal{A}^q_{\text{modulo}}$—that is a grammar generating traces visiting q—has been computed. Then a cleaning step has been performed. It consists in removing useless non-terminal symbols, i.e., when there is no execution tree of the grammar using these non-terminal symbols. For each state q, Table 3 gives the size of the corresponding grammar: the number of non-terminal symbols, and the number of rules. Notice that the left-hand side of each rule contains at most 3 symbols. For each state q, Table 3 also provides the time (in seconds) required to generate the grammar and to clean it, and the time (in seconds) to perform the pre-computation step for the random generation. This step makes it possible both the computation of the probability $pr(q)$ and the random generation of NPDA-traces of $\mathcal{A}_{\text{modulo}}$ visiting q. The two given times are given for respectively the generation of the cleaned grammar, and for the pre-computation step for the random generation (for traces of length up to 60). For instance, for the state 8, the cleaned grammar generating traces of $\mathcal{A}^8_{\text{modulo}}$ has 81 non-terminal symbols and 90 rules. It has been computed in 250 seconds. The pre-computation step for the random generation has been performed in 1.28 seconds.

Computing the $pr(p, q)$'s is quite more expensive. Before performing the cleaning step, the related grammars are generated in about 1 second but every one has about 9000 non-terminal symbols and 70000 rules. For example, the greatest one with 10581 non-terminal symbols and 91572 rules has been generated for the states $\{7, 24\}$. After the cleaning step, the number of non-terminal symbols and rules falls in general to about 100. For the states $\{7, 24\}$ mentioned above, the corresponding cleaned grammar has 110 non-terminal symbols and 124 rules. For each grammar, the cleaning step has been performed in about 420 seconds. At the end, the computation of all the probabilities involved in the linear programming system presented in Section 4.3 has been done in about 34

state	1	2	3	4	5	6	7	8	9	10	11	12
grammar size	80+ 88	80+ 89	80+ 87	81+ 90	82+ 91	83+ 92	84+ 93	81+ 90	82+ 91	83+ 92	84+ 93	85+ 92
time	248 1.19	250 1.18	249 1.19	250 1.23	249 1.23	249 1.23	248 1.24	250 1.28	251 1.35	250 +1.35	249 1.32	251 1.72

state	I	II	III	IV	V	VI
grammar size	83+ 92	83+ 92	83+ 92	92+ 101	92+ 104	92+ 104
time	249 1.29	250 1.29	250 1.29	250 1.31	250 1.29	251 1.32

Table 3: Computing the probabilities $pr(q)$'s for $n \leq 60$ (in seconds).

hours. Then, solving this system requires less than one second thanks to the recent Simplex-based tools, as `lp_solve` [39]. Finally, as explained in Section 5.4, the random generation of traces is done in few seconds for hundreds of traces.

In regards to the criterion *All transitions*, the time to compute probabilities is very close. However, since there are more transitions, the total time required to compute the linear system is estimated to 216 hours. Section 5.4 explains how this computation time may be significantly reduced to an expected time of few hours.

5.3 Illustrating Example for Coverage Criteria

In this section the approach developed in Section 4 is experimented on an example of the NPDA $\mathcal{A}_{\text{xpath}}$ in Fig. 13 corresponding to an XPath query. Let us remind that for stream processing purposes, XPath queries can be automatically translated into a subclass of pushdown automata, called *Visibly pushdown automata* [40]. The example in Fig. 13 is inspired by the example in [41, Fig. 5].

Following the proposals in Section 4, for a given length n of traces, three following algorithms have been compared. Algorithm 1 proceeds with a uniform generation of NPDA-traces of length n until all states are covered. Algorithm 2 generates a NPDA-trace of length n uniformly at random, and then chooses at random (uniformly) a non visited state and generates a trace visiting it. This second step is repeated until all states are visited. Algorithm 3 consists in solving the linear programming system presented in Section 4.3 and in generating an NPDA-trace with the computed probabilities, until all states are covered. Algorithms 4, 5 and 6 correspond respectively to Algorithms 1, 2 and 3 but they aim to cover all transitions.

The grammars needed to generate the traces covering a given state have been computed in 38 seconds in an automatic way. The linear system described in Section 4.3 is computed in 250 seconds, and its resolution indicates that the optimised solution consists in always generating a trace visiting 4. Indeed, a

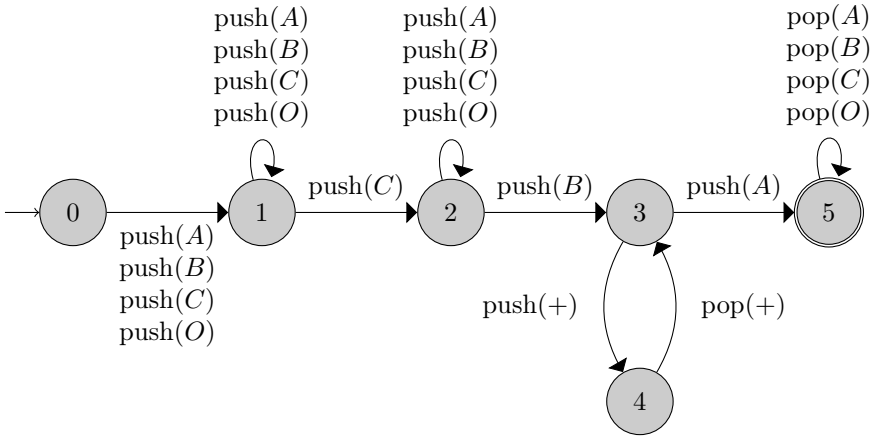


Figure 13: Automaton $\mathcal{A}_{\text{xpath}}$

n	12	14	16	18	20	22	24	26	28	30	32
Algorithm 1	7.51	6.1	5.0	4.77	4.74	4.24	4.43	4.37	4.34	4.34	4.33
Algorithm 2	1.87	1.84	1.83	1.77	1.8	1.78	1.79	1.77	1.77	1.76	1.77
Algorithm 3	1	1	1	1	1	1	1	1	1	1	1
Algorithm 4	14.1	11.03	10.08	9.27	8.83	9.06	8.64	9.36	9.45	8.59	9.43
Algorithm 5	6.9	6.07	5.57	5.08	4.92	4.68	4.66	4.71	4.51	4.52	4.51
Algorithm 6	14.1	-	-	9.1	-	-	-	-	8.8	-	-

Table 4: Average number of generated traces to cover all states/transitions

quick look at the automaton shows that all NPDA-traces cover all states but 4; therefore such a trace visits all the states. The results on the average number of generated traces for the six above-mentioned algorithms are given in Table 4. These average values have been obtained by repeating each experiment 100 times. Algorithm 6 has been experimented for sizes 12, 18 and 28. Actually, for Algorithm 6, the approach requires several, manual at this point, transformations which can be automated.

For this example, Algorithm 3 is unsurprisingly the best one since a unique NPDA-trace suffices to visit all states.

For Algorithm 1 the obtained results are conform to the theoretical expectations. Indeed, for this algorithm NPDA-traces are generated until a trace visiting state 4 is picked. Theoretically this requires an expected number of $1/p$ generations, where p is the probability that a path vising 4 is generated. For instance, the theoretical probability that a NPDA-trace of length 14 visits 4 is about 0.16. Therefore, the theoretically expected number of NPDA-traces to be generated in order to pick one visiting 4 is close to $1/0.16 \simeq 6.25$. This explains

the experimental result of 6.1 obtained for Algorithm 1. Similarly, there are 6372 NPDA-traces of length 16 and 1252 of them visit state 4. Therefore the theoretical probability that a trace of length 16 visits 4 is $\frac{1252}{6372} \simeq 0.196$. Thus the theoretically expected number of generated traces required to visit 4, and consequently all states, is $\frac{6372}{1252} \simeq 5.1$, which is close to the value 5.0 obtained experimentally.

For Algorithm 2, one iteration suffices if the first generated path visits 4, with the probability of 0.16. Otherwise, there are two iterations. For instance, for NPDA-traces of length 14, the theoretically expected number of generated traces to cover all states with this algorithm is $0.16 * 1 + (1 - 0.16) * 2 = 1.84$ which is exactly the observed value.

Similar remarks and comparison with theoretical results can be done for Algorithms 4 and 5. Moreover, Algorithm 5 works unsurprisingly much better than Algorithm 4.

For the *All transitions* criterion, Algorithm 6 does not work better than Algorithm 5, contrary to the *All states* criterion. In fact, the induced optimisation of the minimal probability to visit a transition is light. For instance for traces of length 18, the minimal probability to visit a transition is 0.2 by generating uniformly a NPDA-trace. It becomes 0.25 using the optimisation of Algorithm 5. For traces of length 28, the minimal probability grows from 0.22 to 0.25. On this example, there is no good benefit from Algorithm 5. In comparison, for the *All states* criterion, the minimal probability to visit a state grows from 0.22 (Algorithm 1) to 1 (Algorithm 3).

5.4 Computation Time on More Examples

This section is dedicated to the study of the experimental time and space required by the algorithms proposed in this paper. The NPDA used for experiments are the $\mathcal{A}_{\text{modulo}}$ automaton in Fig. 12, the $\mathcal{A}_{\text{xpath}}$ automaton in Fig. 13, the $\mathcal{A}_{\text{power}}$ automaton in Fig. 5, the automaton $\mathcal{A}_{\text{SY}}$ encoding the *Shunting Yard algorithm* described in Section 6, and the automaton $\mathcal{A}_{\text{plotter}}$ from S.Schwoon PhD [38] describing an algorithm over graphs. Note that NPDA can be automatically computed from C or Java source code using either the JimpleToPDSolver[4] tool [42] or the PuMoc[5] tool [5]. The NPDA used in this section have sizes comparable with the sizes of pushdown systems corresponding to Windows drivers given in a paper by F.Song and T.Touili [43].

Uniform Random Testing. The experimental results on the uniform random testing algorithm from Section 3 are presented in Table 5. In this table, the second column gives the size (number of states + number of transitions) of the considered NPDA. Here the number in brackets indicates the number of transitions of the NPDA considered as a PDA; for instance for the transition

[4] A mu-calculus checker over pushdown systems and pushdown parity games: `http://www.cs.ox.ac.uk/matthew.hague/pdsolver.html`

[5] A CTL model-checker for pushdown systems and sequential programs: `http://www.liafa.univ-paris-diderot.fr/~song/PuMoC/`

$(1, \texttt{intz}, 2)$ in Fig.12 there are in the pushdown automaton as many transitions as stack symbols X, of the form $\delta(1, \texttt{intz}, X) = (2, X)$. The third column of the table indicates the size of the generated grammar (number of non-terminal symbols + number of rules) before cleaning, whereas the fourth column gives the computation time for this generation. The two next columns respectively give the size of the grammar after cleaning, and the time required for this cleaning. Let us recall that the cleaning step consists in removing useless non-terminal symbols and rules. The last-but-one column gives the details on the pre-computation time for the random generation corresponding to the $s(i)'s$ computation as defined in Section 3.2: the first number in brackets is the maximal size used for stopping the pre-computation step, and the second one is the computation time. For instance, for the $\mathcal{A}_{\text{xpath}}$ automaton, (30) 0.57 means that the pre-computation time to generate traces of size at most 30 is 0.57 second. The last column gives the time to generate 100 traces of the size indicated by the number in brackets. For example, for the first line of $\mathcal{A}_{\text{modulo}}$, (8) 0.07 points out that 0.07 second are required to generate 100 traces of length 8. Notice that the time is given for the size 8 since there is no NPDA-traces of size 10. It can be noticed that the grammar cleaning is an expensive step. The pre-computation step becomes significant for very long traces. However, once the pre-computation is done, the random generation of NPDA-traces is non expensive, as well as the rough grammar generation.

Random Biased Testing. For generating NPDA-traces of a given length, the first step consists in computing for each state q, a grammar generating the consistent DFA-traces (equivalently NPDA-traces) visiting q. It allows both to compute the probability $pr(q)$ that a NPDA-trace visits q, and to uniformly generate such traces. Table 6 gives for each example the time required to compute all the $pr(q)$'s. It also provides the average sizes of the related grammars (as usual, number of non-terminal symbols + number of rules). Notice that for the Shunting-Yard algorithm, the generated grammars—computed in about 25 seconds—have about 17000 non-terminal symbols, and 400000 rules. The time for $\mathcal{A}_{\text{plotter}}$ is not pointed out since this NPDA has only one state. These results show that the approach proposed at the beginning of Section 4.3 can be easily applied to our examples. The last line of Table 6 also gives the time for computing all the probabilities associated with transitions.

The first line of Table 7 gives the experimental average time to compute one probability $pr(p, q)$ (for states). For a pair of transitions, there is no special line in Table 7 since the computed average time is very close to this of states, the related automata being of similar size and structure. This table also provides the total time to compute the linear programming system of Section 4.3. The notation (*) is used to indicate that the reported value is an estimated time computed from a sample of computed probabilities. For $\mathcal{A}_{\text{plotter}}$, the probabilities for states are irrelevant since this automaton has only one state. Two lines at the bottom of Table 7 provide estimated time to compute the linear system of Section 4.3 using some simplifications. For instance, for $\mathcal{A}_{\text{power}}$ depicted

	NPDA Size	Grammar Size	Grammar Gen. Time	Cleaned Grammar	Grammar Cleaning Time	Pre. Time	100 Traces Gen. Time Gen. Time
$\mathcal{A}_{\text{xpath}}$	6+21 (102)	217+ 3463	0.03	27+87	0.38	(10) 0.07 (20) 0.22 (30) 0.57 (100) 17.21 (200) 141	(10) 0.09 (20) 0.17 (30) 0.63 (100) 1.1 (200) 2.83
$\mathcal{A}_{\text{power}}$	10+12 (24)	201+369	0.02	27+31	0.05	(100) 0.98 (200) 5.63 (500) 78.96	(100) 0.41 (196) 0.94 (496) 3.94
$\mathcal{A}_{\text{modulo}}$	18+24 (90)	1621+ 7572	0.07	45+50	6.07	(10) 0.03 (20) 0.08 (50) 0.41 (100) 2.29 (200) 14.86	(8) 0.07 (21) 0.14 (49) 0.32 (99) 0.28 (199) 1.4
$\mathcal{A}_{\text{SY}}$	23+ 36 (102)	1937+ 16735	0.13	143+ 261	16.61	(10) 0.16 (20) 0.58 (30) 1.46 (100) 39.21 (200) 326.28	(10) 0.13 (20) 0.24 (30) 0.34 (100) 1.14 (200) 2.35
$\mathcal{A}_{\text{plotter}}$	1+22	21+24	0.01	21+24	0.01	(20) 0.05 (30) 0.11 (100) 2.21 (200)15.23 (300) 52.37 (500) 255.18	(17) 0.07 (30) 0.11 (100) 0.44 (200) 1.01 (300) 1.7 (500) 3.41

Table 5: Experiments for the Uniform Random Testing Algorithm (time in seconds)

	$\mathcal{A}_{\text{xpath}}$	$\mathcal{A}_{\text{power}}$	$\mathcal{A}_{\text{modulo}}$	$\mathcal{A}_{\text{SY}}$	$\mathcal{A}_{\text{plotter}}$
av. time (state)	38s	5.25s	22min	136min	-
av. grammar size	31.3+98.8	32.2+36.8	65.2+72.9	342.7+692.9	26+31
av. time (transition)	91s	6.3s	33min	219min	15.3s
av. grammar sizes	30.5+94.8	30.7+34.3	66.2+73.7	98.3+205.6	23.8+27.3

Table 6: Computation time of the $pr(q)$'s (traces of size 60)

	$\mathcal{A}_{\text{xpath}}$	$\mathcal{A}_{\text{power}}$	$\mathcal{A}_{\text{modulo}}$	$\mathcal{A}_{\text{SY}}$	$\mathcal{A}_{\text{plotter}}$
av. time (one pair of states)	13.4s	0.9s	408s	110min	1.16s
total (state)	7min19s	79s	34h12min	847h (*)	-
total (transition)	25min	81s	60h39min	1200h (*)	9min
total (simplified, state)	1min	-	9h (*)	120h (*)	-
total (simplified, transition)	8min (*)	-	18h (*)	350h (*)	-

Table 7: Computation time of the $pr(p, q)$'s (traces of size 60) and of the linear systems

in Fig. 5, all NPDA-traces visiting 9 also visit 10, and conversely. Therefore $pr(9, q) = pr(10, q)$ for every state q. Moreover, all NPDA-traces visit 1; it follows that $pr(1, q) = pr(q)$ for every q. With this kind of observations, the number of probabilities $pr(p, q)$ to be computed falls down. Similar remarks can be done for the transitions. With these observations, Table 7 shows that the (estimated) computation time is significantly better. For $\mathcal{A}_{\text{plotter}}$ and $\mathcal{A}_{\text{power}}$, the time is not provided since even the brute force computation time is already small. Before finishing with Table 7, let us notice that computing the linear programming system for $\mathcal{A}_{\text{SY}}$ for both criteria seems to be intractable with the implemented prototype. Moreover, the possible simplifications to compute the probabilities (for different examples) have not been implemented in the prototype; however several simple dependencies could be detected automatically. The estimated probabilities have been obtained by counting the number of $pr(p, q)$'s that are enough to obtain the system using these simplification; and by multiplying this number by the average time to compute one $pr(p, q)$. The indicated values give therefore the order of magnitude of an expected time.

Analysis of the Experimental Results. Firstly, it has to be noticed that the examples used for the experiments are of the size comparable to the size of pushdown systems modelling recursive programs, as it is reported in a paper by F.Song and T.Touili [43] on verification of Windows drivers: involved pushdown automata have in general less than 10 states and about few dozens of transitions.

Secondly, as mentioned before, it is frequently admitted that C/C++ implementations are 10 to 100 times faster than Python implementations, particularly for programs managing large data structures. Therefore, having a C/C++-based tool can make the approach very useful for practical applications, with tractable computation times.

Thirdly, the experimental results analysis shows that the computation of the cleaned grammars for the random biased approaches is an expensive step. However, each (simplified) grammar can be computed independently. Therefore, the approach can be trivially distributed on several computers, for instance one for each computed probability. For example, for $\mathcal{A}_{\text{xpath}}$ and for the criterion *All transitions*, there are about 200 probabilities $pr(t_1, t_2)$ to compute (let us remind

that $pr(t_1, t_2) = pr(t_2, t_1)$). Using 10 computers in parallel can divide the computation time by 10: experiments show that each probability is computed in a similar time, making the parallelisation very efficient.

To conclude, experimental results show that the uniform random testing approach (corresponding to Algorithm 1 and 4 of Section 5.3) is easily applicable, and practical applications are tractable. The random biased testing based on either already visited states or transitions (corresponding to Algorithms 2 and 5 of Section 5.3) can be easily performed too, even with a Python prototype (see Table 6). Optimising the quality of the testing approach (corresponding to Algorithms 3 and 6 of Section 5.3) requires more resources. Excepted for $\mathcal{A}_{\mathrm{SY}}$, testing with the criterion *All states* is tractable but, for the criterion *All transitions*, several optimisations—additional computation simplifications, C++, distributed computation—have to be investigated to obtain reasonable computation times. Notice that for $\mathcal{A}_{\mathrm{SY}}$, both computation simplifications and a C++ implementation would provide an expected reasonable time of few hours.

6 Case Study: the Shunting Yard Algorithm

In this section, the application of the random generation techniques is presented within the model-based testing context. For this purpose a web implementation[6] of the shunting yard algorithm is analysed.

6.1 Description

This section describes the shunting yard algorithm [**?**] proposed by Dijkstra for converting mathematical expressions from the usual infix notation into the reverse Polish notation. Its name is due to its operation reminding this of a railroad shunting yard. The reverse Polish notation is a stack-based notation, close to the syntax tree notation, and used for instance by HP pocket calculators. For example, the expressions $3+4$ or $3+4*(2-1)$ become $3\ 4\ +$ and $3\ 4\ 2\ 1\ -\ *\ +$ in the reverse Polish notation.

Since the shunting yard algorithm is stack-based, it can fruitfully be modelled by a pushdown automaton. The conversion uses two text variables (strings), for the input and the output, and a stack containing operators that have not been handled yet. To convert, the algorithm reads a symbol from the input, and then handles it depending on the read symbol and on the last operator on the top of the stack. Figure 14 displays a textual description of the shunting yard algorithm simplified by removing the function tokens and the ˆ operator.

6.2 Underlying Automaton for the Shunting Yard Algorithm

As explained before, using pushdown automata as a model fits well with the needs of the shunting yard algorithm exploiting a stack. Starting from the

[6] `http://en.wikipedia.org/wiki/Shunting-yard_algorithm`

Shunting yard
Input: a list e of characters which is an arithmetic expression in the usual form. Each element of e is either a number in $\{0, \ldots, 9\}$, or a parenthesis symbol, or an arithmetical symbol.
Output: a list t encoding the same expression in the reverse Polish notation.

Algorithm:

```
t is the empty list.
s is a local empty stack.
While e is non empty do
    Let c be the first element of e.
    Remove the first element of e.
    If c is a number, then add it to the end of t. EndIf
    If c is an operator, then
        While the top of the stack s contains an operator o with a greater or equal precedence
        to that of c do
            pop o out of the top of s
            add o to the end of t.
        EndWhile
        Push c onto s.
    EndIf
    If c is a left parenthesis then push c into s. EndIf
    If c is a right parenthesis then
        While the top of the stack s contains an element o which is not a left parenthesis do
            pop o out of the top of s
            add o to the end of t.
        EndWhile
        pop the top of s (which is a left parenthesis)
    EndIf
EndWhile
While s is non empty do pop the top element of s onto the end of t. EndWhile
Return t.
```

Figure 14: Shunting yard algorithm

description in Fig. 14, the shunting yard algorithm is modelled by a PDA whose underlying automaton is depicted in Fig. 15. For the readability reasons, the model takes into account only the "+" and "*" operators. Notice however that using these operators suffices for illustrating and validating this paper proposals. In this automaton, the stack symbols are Z (for the empty stack), X_+, $X_($ and X_* to encode that the stack contains respectively $+$, (and $*$. The *read* transitions model what is read from the input, while the *write* transitions model what is written in the output. Notice that the x's used on the transitions between q_0 and q_d can be replaced by any digit in $\{0, 1, \ldots, 9\}$. The EOI action encodes that there is nothing more to be read on the input. Pairs of transitions of the form $(q_+, \text{pop}(Z), q_1)(q_1, \text{push}(Z), q_{+end})$ model the checking of whether the stack (in the implementation) is empty (on the model it checks that the stack contains only Z).

For instance, let us consider the input string $3 * 4 + 2$. In the underlying automaton the corresponding path is:

$$(q_{\text{init}}, \text{push}(Z), q_0)(q_0, \text{read } 3, q_d)(q_d, \text{write } 3, q_0)(q_0, \text{read } *, q_*)(q_*, \text{pop}(Z), q_4)(q_4, \text{push}(Z), q_{*end})$$
$$(q_{*end}, \text{push}(X_*), q_0)(q_0, \text{read } 4, q_d)(q_d, \text{write } 4, q_0)(q_0, \text{read } +, q_+)(q_+, \text{pop}(X_*), q_{+*})$$
$$(q_{+*}, \text{write } *, q_+)(q_+, \text{pop}(Z), q_1)(q_1, \text{push}(Z), q_{+end})(q_{+end}, \text{push}(X_+), q_0)(q_0, \text{read } 2, q_d)$$
$$(q_d, \text{write } 2, q_0)(q_0, EOI, q_8)(q_8, \text{pop}(X_+), q_6)(q_6, \text{write } +, q_8)(q_8, \text{pop}(Z), q_f)$$

6.3 Experiments

This section reports on the experiments performed on a C implementation of the shunting yard algorithm available on the web[7] and slightly simplified to handle the "+" and "*" operators. This simplified code is given in Figures 16 and 17. Firstly, the approach is evaluated by measuring the proportion of the code lines executed, and the proportion of the transitions used. Secondly, it is also evaluated using mutation operators: 10 modified pieces of code have been generated using a freely available tool, called Mutate[8]. Following the classical classification of mutants [**?**], four out of ten mutants are of the Oido type (increment/decrement), one of the Varr type (Array reference replacement), one of the SSWM type (Switch Statement Mutation), and four are of the ONLG (Logical Negation) type. Notice that the tested code implements the detection of the syntax errors in the input files, and the parts of the code handling wrong inputs are not handled by the model since our push-down automaton is designed for only valid inputs. Therefore, random test case cannot reach all the lines of the codes. This is why the mutants have been chosen to be on reachable parts.

Like in Sec. 5, for the experimental purposes the developed prototype has been used. Again, thanks to this prototype a pushdown automaton is transformed into a normalised pushdown automaton, which is used to automatically

[7] `http://en.wikipedia.org/wiki/Shunting-yard_algorithm`

[8] Its code is available, e. g., at `http://members.femto-st.fr/pierre-cyrille-heam/mutatepy`

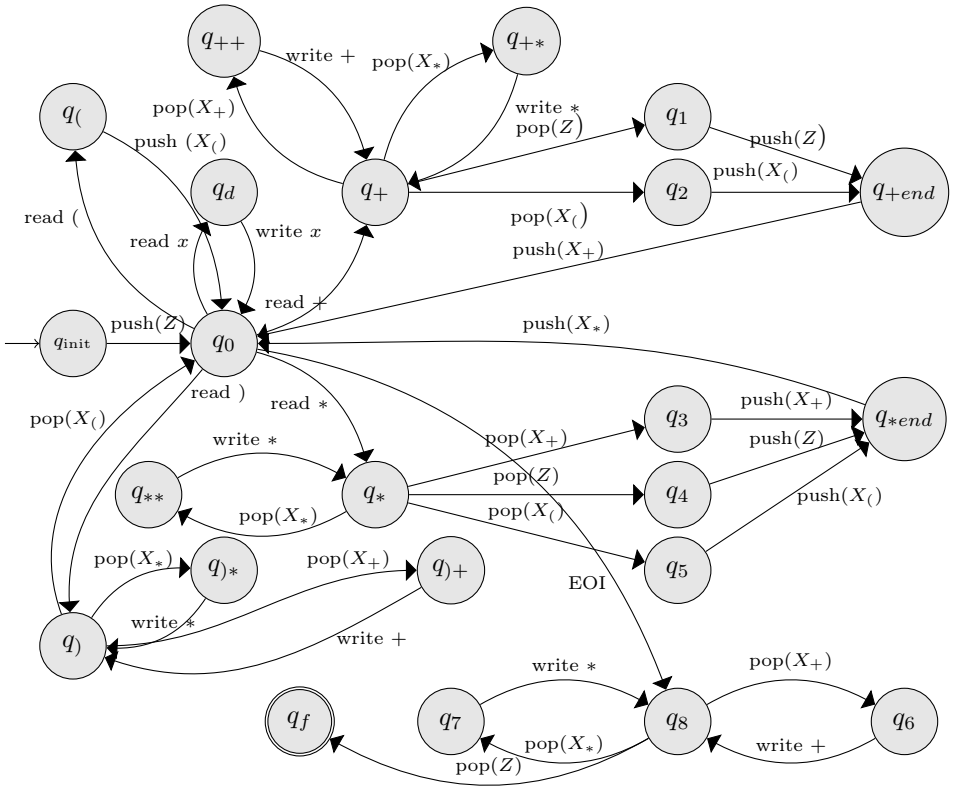


Figure 15: The underlying automaton for the shunting-yard algorithm

produce a context-free grammar. In its turn, the obtained grammar is used to automatically generate executions thanks to the GenRGenS tool.

Our prototype computes, for each execution, the corresponding input sequence for the program (which is read on the automaton) and compares the output of the automaton with the output of the program. If they are equal, the test is successful, otherwise it fails: outputs of the automaton constitute the oracle for the test campaign. The coverage of the C code source lines and of the automaton transitions have been computed too. Finally, the prototype also reports on the killed (detected) mutants.

For the first experiment, the goal is to detect all the mutants, with the following experimental protocol: for each mutant, a test pointing out the related code error has to be generated. The second experiment goal is to generate random test until all transitions are covered by the test suite. For the last experiment, the goal is to cover most of the code lines. However, as explained before, several lines handling wrong inputs cannot be reached. Therefore the goal is to cover 100% of the reachable lines, representing about 79% of the lines

```c
#include <string.h>
#include <stdio.h>
#define bool int
#define false 0
#define true 1
int op_preced(const char c){
    switch(c)    { case '*': return 3; case '+': return 2;}
    return 0;}
bool op_left_assoc(const char c){
    switch(c)    {case '*':  return true; case '+': return true;  }
    return false;
}
#define is_operator(c)  (c == '+' || c == '*')
#define is_ident(c)     (c >= '0' && c <= '9')

bool shunting_yard(const char *input, char *output){
    const char *strpos = input, *strend = input + strlen(input);
    char c, *outpos = output;
    char stack[32];       // operator stack
    unsigned int sl = 0;  // stack length
    char     sc;          // used for record stack element
    while(strpos < strend)   {
        c = *strpos;
        if(c != ' ')    {
  if(is_ident(c))  {
    *outpos = c; ++outpos;
  }
  else if(is_operator(c))  {
    while(sl > 0)    {
      sc = stack[sl - 1];

    if(is_operator(sc)&&((op_left_assoc(c)&&(op_preced(c)<=op_preced(sc)))||(op_preced(c)<op_preced(sc))){
                        // Pop op2 off the stack, onto the output queue;
                        *outpos = sc;
                        ++outpos;
                        sl--;}
                    else   {
                        break;}
                }

stack[sl] = c;
                ++sl;}

    else if(c == '(')   {
                stack[sl] = c;
                ++sl; }

    else if(c == ')')    {
                bool pe = false;
while(sl > 0)     {
                    sc = stack[sl - 1];
                    if(sc == '(')    {
                        pe = true;
                        break;}
                    else  {
                        *outpos = sc;
                        ++outpos;
                        sl--;}
                }

if(!pe)  {
                    printf("Error: parentheses mismatched\n");
                    return false;}
sl--;
if(sl > 0)   {
                    sc = stack[sl - 1];}
            }
            else  {
                printf("Unknown token %c\n", c);
                return false; // Unknown token
          }
        }
        ++strpos;
    }
    while(sl > 0)  {
        sc = stack[sl - 1];
        if(sc == '(' || sc == ')')   {
            printf("Error: parentheses mismatched\n");
            return false;}
        *outpos = sc;
        ++outpos;
        --sl;}
    *outpos = 0; // Null terminator
    return true;
}
```

Figure 16: Tested C-implementation of the shunting yard algorithm 1/2

```
int main (int argc, char *argv[],char **envp) {
    const char *input = argv[1];
    char output[128];
    if (argc != 2) {
      printf("Error:%d arguments expected",argc-1);
      if (argc > 1) {
printf(" ( ");
int i;
for(i=1;i<argc-1;i++) {
  printf("%s , ",argv[i]);   }
printf("%s ) ",argv[argc-1]);
      }
     printf("Error: a unique argument is expected\n");
     return -1;   }
    if(shunting_yard(input, output))    {
        printf("input: %s\n", input);
        printf("output: %s\n", output);    }
    return 0;
}
```

Figure 17: Tested C-implementation of the shunting yard algorithm 2/2

of the application. Each experiment has been performed 20 times. The minimal, the maximal and the average numbers of tests required to reach the goal are given in Table 8. Each result has been obtained on a usual commercial laptop in few seconds.

In conclusion, the experiments show that the proposed testing technique is very efficient with a very reasonable size of tests and a small number of tests.

	Killing mutants			Covering transitions			Covering 79% of the code		
length of the paths	min	max	aver.	min	max	aver.	min	max	aver.
10	2	16	7.8	7	48	15.3	3	17	6.9
15	1	7	3.7	4	19	8.8	2	7	4.5
20	2	11	3.2	3	10	5.4	2	6	2.9

Table 8: Experimental results for the shunting yard algorithm

7 Discussion and Conclusion

Choosing Testing Parameters. The proposed testing framework makes use of two parameters: the number and the length of the test cases. For the number of test cases, several approaches can be adopted. For instance, it is possible to generate test cases until a given coverage criterion is fulfilled as it is done with Algorithms 1 and 4 in Section 5.3. In this case, the average number of test cases is $\frac{1}{p_{\min}}$, where $p_{\min}$ is the minimal probability to cover an element of the coverage criterion under consideration. Following A.Denise et al. paper [4], the number of test cases can also be fixed *a priori*—using the formula $N \geq \frac{\log(1-p_{\text{quality}})}{\log(1-p_{\min})}$— in order to obtain a chosen probability p_{quality} to cover all elements of the coverage criterion. The same formula can be adopted for the approaches based on the resolution of the linear system. For a random biased approach close to Algorithms 4 and 6 in Section 5.3, the average number of tests to ensure

a desirable a priori defined quality is hard to compute since there are many dependences. It is however bounded by the number of tests required by the unbiased approach. Of course, the number of generated test cases may also depend on the context: for a test campaign requiring manual manipulations it can be reduced. Conversely, for massive testing, like Fuzz testing, it must be huge.

The choice of the length n of the generated traces (tests cases) depends on the system under test and on the context. For performance testing, several lengths may be chosen to observe the evolution of the resource consumption. For robustness testing, very important lengths can be chosen to observe the system's behaviour in extreme cases. For functional testing, the goal is to find a length n allowing to cover all states or transitions. Let us notice that several lengths may be chosen: for instance, it happens when a state can only be visited by traces of odd length, whereas another one is only covered by traces of even length. An experience-based and convenient way could be to compute the probabilities (and the related grammars) for covering each state (transition) for lengths up-to ten times the number of states of the NPDA. Another way to define the length n could be to use the following (sketched) procedure. Given for the states coverage criterion, it can be defined in a dual way for the transitions coverage criterion.

1. Compute the set Q_{reach} of states that can be visited by an NPDA-trace; some states representing for instance the dead-code may be not reachable.

2. For each state $q \in Q_{\text{reach}}$, compute n_q the length of the smallest NPDA-trace visiting q.

3. Perform the pre-computation step (of the random generation procedure) for all states and all k less or equal to $n = \max\{n_q \mid q \in Q_{\text{reach}}\}$. Choose n as maximal length of the test cases.

Step 1 can be performed in an efficient automatic way using existing works [44]. Step 2 can be done on the fly while performing Step 1, or by adapting the approach developed in a paper by S.Basu et al. [45]. Consequently, computing n can be done efficiently.

Comparison with the Random Generation of DFA-traces. In the paper by A.Denise et al. [4], the random testing approach without any coverage criterion is applied to graphs with few thousands of states and for traces of comparable lengths. Larger graphs cannot be handled by that approach directly, because of important memory resources needed to store the pre-computation table. To address this problem, the approach has been adapted to cope with large graphs implicitly defined by a synchronisation of smaller graphs. The NPDA-based approach proposed in the paper cannot handle so large models. However, as exposed in Section 5, on pushdown models, the approach of A.Denise et al. [4] ignoring the stack operations is not fruitful since it provides non consistent paths. A pushdown model can be extended to a graph model where all paths are correct wrt. the stack operation: it *suffices* to compute the graph of

the NPDA configurations as defined in Section 2.1. However, this graph is in general infinite. It can be approximated by bounding the depth of the stack. However, for $\mathcal{A}_{\text{modulo}}$, with 4 stack symbols and 18 states, and with a stack depth bounded by 5, the corresponding graph can have up to $18 * 4^5 = 18432$ nodes. Even if several of these nodes/configurations are unreachable, the technique consisting in bounding the stack—or, equivalently, for structural testing in bounding the function invocations—leads to huge graphs that cannot be directly handled in the framework of the approach by A.Denise et al.[4].

Limits and Strength of the Approaches. The experimental results exposed in Section 5 show that for the uniform random generation of test cases, the expensive step is the computation of the cleaned context-free grammar. For the $\mathcal{A}_{\text{SY}}$ automaton, the computation of a probability $pr(p, q)$ can be done in about 110 minutes. The related automaton $\mathcal{A}_{\text{SY}}^{p,q}$ has 96 states and 144 transitions. Even if this computation time can be improved thanks to a better implementation, a hundred of states and of transitions is approximately the maximal NPDA size that can be exploited for the uniform random testing approach in a short time.

For the random biased testing approach with Algorithms 2 or 5 in Section 5.3, the computation of a grammar—for each state or each transition—is required. It is possible for all the examples considered in Section 5, even if the $\mathcal{A}_{\text{SY}}$ size seems to be close to the maximal size acceptable for a treatment in a reasonable time. Notice also that this limit corresponds to the size of NPDA automatically built in recent works [43] using the PuMoX tool [5], for many industrial examples.

The approach consisting in solving a linear programming system to optimise the probability to fulfil a coverage requirement is time-consuming. Experimental results show that it is limited to the *All states* criterion and to automata with few states, as $\mathcal{A}_{\text{xpath}}$. Notice that several industrial examples studied in the literature [43] have the size of this order. Moreover, several optimisations can be foreseen in order to address this problem for larger NPDA. This way one can hope to handle NPDA of the size of $\mathcal{A}_{\text{modulo}}$, but there is still work to do.

Using random biased approaches consisting in generating a trace to visit a non already covered element—Algorithms 2 and 4 of Section 5.3—allows a significant reduction of the size of the test suites. Moreover, these two algorithms do not require an expensive computation time. Optimising the minimal probability to cover an element by solving a linear system is more expensive. When probabilities of covering elements are not well distributed (for instance for states for $\mathcal{A}_{\text{xpath}}$), this approach can be very fruitful. Conversely, when probabilities of covering elements are well distributed, the benefit is insignificant (for instance for transitions for $\mathcal{A}_{\text{xpath}}$). The efficiency of this approach deeply depends on the topology of the NPDA.

Conclusion. In this paper a random testing approach using pushdown models is developed. Although this approach is not as efficient as the approach proposed in the seminal paper by A.Denise et al. [4], it is still tractable (poly-

nomial complexity). In addition, it deeply increases the chance of computing paths corresponding to real executions. In the structural testing context, an example shows the importance of handling stack operations in order to model recursive functions invocations. In the model-based testing context, the approach is fruitful for programs or systems managing a stack, like the shunting yard algorithm. The paper provides a qualitative and quantitative study of the testing procedures, and it is explained how to use a biased random generation to combine random testing and coverage criteria in an optimised way. An important perspective consists in adapting the developed framework to randomly generate test cases under a distribution given by some statistical information on the systems, i.e., for the statistical testing purpose.

The authors would like to thank Frédéric Dadeau and Fabien Peureux for helpful discussions and advice, and the anonymous reviewers for their constructive comments and suggestions to improve the quality of the paper. This work is partially granted by the French ANR project FREC.

References

[1] Beizer B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons: New York, USA, 1995.

[2] Offutt A, Xiong Y, Liu S. Criteria for Generating Specification-Based Tests. *ICECSS'99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society: Las Vegas, NV, USA, 1999; 119–129.

[3] Groce A, Joshi R. Random testing and model checking: building a common framework for nondeterministic exploration. *WODA'08: Proceedings of the 2008 International Workshop on Dynamic Analysis*, ACM: New York, NY, USA, 2008; 22–28.

[4] Denise A, Gaudel MC, Gouraud SD, Lassaigne R, Oudinet J, Peyronnet S. Coverage-biased random exploration of large models and application to testing. *Software Tools for Technology Transfer (STTT)* 2012; **14**(1):73–93.

[5] Song F, Touili T. PuMoC: a CTL model-checker for sequential programs. *ASE'12: Proceeding of the IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2012; 346–349.

[6] Sipser M. *Introduction to the Theory of Computation*, chap. 2. PWS, 1996.

[7] Ponty Y, Termier M, Denise A. Genrgens: Software for generating random genomic sequences and structures. *Bioinformatics* 2006; **22**(12):1534–1535.

[8] Denise A, Dutour I, Zimmermann P. Cs: a mupad package for counting and randomly generating combinatorial structures. *FPSAC'98*, 1998; 195–204.

[9] Gotlieb A, Botella B, Rueher M. Automatic test data generation using constraint solving techniques. *ISSTA'98: Proceeding of the International Symposium on Software Testing and Analysis*, 1998; 53–62.

[10] Gotlieb A, Denmat T, Botella B. Constraint-based test data generation in the presence of stack-directed pointers. *ASE*, Redmiles DF, Ellman T, Zisman A (eds.), ACM, 2005; 313–316.

[11] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. *PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ACM: New York, NY, USA, 2005; 213–223.

[12] Williams N, Marre B, Mouy P, Roger M. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. *EDCC*, *Lecture Notes in Computer Science*, vol. 3463, Cin MD, Kaâniche M, Pataricza A (eds.), Springer, 2005; 281–292.

[13] Lee D, Yannakakis M. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 1996; 1090–1123.

[14] Campbell C, Grieskamp W, Nachmanson L, Schulte W, Tillmann N, Veanes M. Testing concurrent object-oriented systems with spec explorer. *FM'05: Proceeding of Formal Methods Europe*, *Lecture Notes in Computer Science*, vol. 3582, Springer: Newcastle, UK, 2005; 542–547.

[15] Jard C, Jéron T. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)* October 2004; **6**.

[16] Ammann P, Offutt J. *Introduction to Software Testing*. Cambridge University Press, 2008.

[17] Purdom P. A sentence generator for testing parsers. *BIT* 1972; **12**(3):366–375.

[18] Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. *ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press: New York, NY, USA, 2007.

[19] Coppit D, Lian J. Yagg: an easy-to-use generator for structured test inputs. *ASE'05: Proceeding og the20th IEEE/ACM International Conference on Automated Software Engineering*, 2005; 356–359.

[20] Lämmel R, Schulte W. Controllable combinatorial coverage in grammar-based testing. *TestCom'06: Proccedings of the 18th IFIP TC6/WG6.1 International Conference*, *Lecture Notes in Computer Science*, vol. 3964, 2006; 19–38.

[21] Majumdar R, Xu RG. Directed test generation using symbolic grammars. *ASE'07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2007; 134–143.

[22] Godefroid P, Kiezun A, Levin M. Grammar-based whitebox fuzzing. *PLDI'08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, ACM, 2008; 206–215.

[23] Xu Z, Zheng L, Chen H. A toolkit for generating sentences from context-free grammars. *Software Engineering and Formal Methods*, IEEE, 2010; 118–122.

[24] Lämmel R. Grammar testing. *FASE'01: Proceedings of 4th International Conference on Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, vol. 2029, Springer, 2001; 201–216.

[25] Zheng L, Wu D. A sentence generation algorithm for testing grammars. *COMPSAC*, IEEE Computer Society, 2009; 130–135.

[26] Alves T, Visser J. A case study in grammar engineering. *SLE'08: Proceedings of the First International Conference on Software Language Engineering*, *Lecture Notes in Computer Science*, vol. 5452, Springer, 2008; 285–304.

[27] Duran J, Ntafos S. A report on random testing. *ICSE '81: Proceedings of the 5th international conference on Software engineering*, IEEE Press: Piscataway, NJ, USA, 1981; 179–183.

[28] Hamlet R. Random testing. *Encyclopedia of Software Engineering*, Wiley, 1994; 970–978.

[29] Oriat C. Jartege: A tool for random generation of unit tests for java classes. *QoSA/SOQUA'05: Proceeding of the First International Conference on the Quality of Software Architectures*, *Lecture Notes in Computer Science*, vol. 3712, Springer, 2005; 242–256.

[30] McKenzie B. Generating string at random from a context-free grammar. *Technical Report TR-COSC 10/97*, Univ ersity of Canterbury 1997.

[31] Hickey TJ, Cohen J. Uniform random generation of strings in a context-free language. *SIAM Journal on Computing (SICOMP)* 1983; **12**(4):645–655.

[32] Maurer P. The design and implementation of a grammar-based data generator. *Software: Practice and Experience* 1992; **22**(3):223–244.

[33] Héam PC, Nicaud C. Seed: An easy-to-use random generator of recursive data structures for testing. *ICST'11: Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation*, IEEE Computer Society, 2011; 60–69.

[34] Dadeau F, Levrey J, Héam PC. On the use of uniform random generation of automata for testing. *Electronic Notes in Theoretical Computer Science* 2009; **253**(2):37–51.

[35] Flajolet P, Zimmermann P, Cutsem BV. A calculus for the random generation of labelled combinatorial structures. *TCS* 1994; **132**(2):1–35.

[36] Flajolet P, Sedgewick R. *Analytic Combinatorics*. Cambridge University Press, 2008.

[37] Flajolet P, Sedgewick R. *Analytic Combinatorics*. Cambridge University Press, 2008.

[38] Schwoon S. Model-checking pushdown systems. Ph.D. Thesis, Technische Universität München Jun 2002. URL `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/schwoon-phd02.pdf`.

[39] M Berkelaar KE, Notebaert P. lp_solve: a mixed interger linear program 2004. `http://lpsolve.sourceforge.net/5.5/`.

[40] Alur R, Madhusudan P. Visibly pushdown languages. *STOC'04: Proceedings of the 36th Annual ACM Symposium on Theory of Computing,*, ACM, 2004; 202–211.

[41] Clark R. Querying streaming xml using visibly pushdown automata. *Technical Report UIUCDCS-R-2008-3008*, University of Illinois at Urbana-Champaign 2008.

[42] Hague M, Ong CHL. Analysing mu-calculus properties of pushdown systems. *SPIN'10: Proceeding of the 17th International SPIN Workshop, Enschede*, *Lecture Notes in Computer Science*, vol. 6349, Springer, 2010; 187–192.

[43] Song F, Touili T. Efficient CTL model-checking for pushdown systems. *CONCUR'11: Proceedings of the 22nd International Conference on Concurrency Theory*, *Lecture Notes in Computer Science*, vol. 6901, Springer, 2011; 434–449.

[44] Bouajjani A, Esparza J, Finkel A, Maler O, Rossmanith P, Willems B, Wolper P. An efficient automata approach to some problems on context-free grammars. *Information Processing Letters (IPL)* Jun 2000; **74**(5-6):221–227.

[45] Basu S, Saha D, Lin YJ, Smolka SA. Generation of all counter-examples for push-down systems. *FORTE'03: Proccedings of the 23rd IFIP WG 6.1 International Conference*, *Lecture Notes in Computer Science*, vol. 2767, Springer, 2003; 79–94.