Random Grammar-Based Testing for Covering All Non-terminals
Aloís Dreyfus, Pierre-Cyrille Héam, Olga Kouchnarenko

To cite this version:

HAL Id: hal-00940429
https://hal.archives-ouvertes.fr/hal-00940429
Submitted on 31 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Random Grammar-based Testing for Covering All Non-Terminals

Aloïs Dreyfus, Pierre-Cyrille Héam and Olga Kouchnarenko
FEMTO-ST - CNRS UMR6174 - Université de Franche-Comté - INRIA CASSIS
16 route de Gray - 25030 Besançon, France
Email: firstname.lastname@femto-st.fr

Abstract—In the context of software testing, generating complex data inputs is frequently performed using a grammar-based specification. For combinatorial reasons, an exhaustive generation of the data – of a given size – is practically impossible, and most approaches are either based on random techniques or on coverage criteria. In this paper, we show how to combine these two techniques by biasing the random generation in order to optimise the probability of satisfying a coverage criterion.

Keywords—Random testing, Grammar-based testing.

I. INTRODUCTION

A. Motivation

Producing trusted software is a central issue in software engineering. Testing remains an inescapable step to ensure software quality. In reaction to the limitations of manual testing, recent years have seen a rise in the research interest for systematic testing frameworks grounded in theory. Random testing is a natural approach, empirically known to detect many kinds of bugs. However, by definition, low-probability behaviours cannot be adequately tested in that way. Conversely, non-random testing tends to focus on a few edge cases of particular interest to the tester, at the expense of all others. Indeed, it can cover various behaviours, but their choice depends on tester’s priorities and, in general, each behaviour is tested in a unique way.

In [1], it is explained how to bias a uniform random testing approach using constraints given by a coverage criterion, in order to optimise the probability of satisfying this criterion. The technique is developed for path generation in a graph. The contribution of the present paper consists in enriching this approach with a coverage criterion on non-terminal symbols of the grammar, allowing the user to apply it to grammar-based testing.

B. Related Work

Grammar-based testing is frequently used for generating structured inputs, as in [2] for parser testing or in [3] to test refactoring engines (program transformation software). Systematic combinatorial approaches [4] lead to a huge number of sequences, and symbolic approaches are frequently preferred [5], [6], [7]. In [8], a generic tool for generating test data from grammars has been proposed. This tool does not provide any random feature but is based on rule coverage algorithms and techniques, as defined in [2], [9], [10], [11].

Random test generation techniques – initially proposed in [12], [13] – are frequently used for practical reasons, as in [14], [15], [1]. Combining random generation and grammar-based testing is explored in [16], [17], [18], [19], [20], [21], without exploiting any coverage criteria, or using an isotropic random walk as in [22].

C. Layout

Section II presents the notions and notations used in this paper. Section III explains how to optimise random testing to satisfy a given coverage criterion. The theoretical contributions are provided in Section IV, which shows how to use this technique to optimise the coverage of non-terminal symbols in a grammar-based testing context. An illustrating example is developed in Section V. Finally, Section VI concludes.

II. FORMAL BACKGROUND

A. Context-free Grammars and Random Generation

In this paper, the cardinality of a finite set S is denoted |S|.

a) Context-free Grammars: A context-free grammar is a tuple G = (Σ, Γ, S₀, R), where Σ and Γ are disjoint finite alphabets, S₀ ∈ Γ is the initial symbol, and R is a finite subset of Γ × (Σ ∪ Γ)∗. The elements of Σ are called terminal symbols, and the elements of Γ are called non-terminal symbols. An element (X, u) of R is called a rule of the grammar and is frequently denoted X → u. A word w ∈ (Σ ∪ Γ)∗ is a successor of v ∈ (Σ ∪ Γ)∗ for the grammar G if there exist v₀ ∈ Σ∗, v₁, v₂ ∈ (Σ ∪ Γ)∗, S ∈ Γ such that v = v₀Sv₁ and w = v₀v₂v₁ and S → v₂ ∈ R. A complete derivation1 of the grammar G is a finite sequence x₀, . . . , xₖ of words of (Σ ∪ Γ)∗ such that x₀ = S₀, xₖ ∈ Σ∗ and for every i, xᵢ₊₁ is a successor of xᵢ. A derivation tree of G is a fine tree whose internal nodes are labelled by letters of Γ, whose leaves are labelled by elements of Σ ∪ {ε}, whose root is labelled by S₀ and satisfying: if a node is labelled by X ∈ Γ and its children are labelled by α₁, . . . , αₖ (in this order), then either α₁ = ε and k = 1, or all the αₛ are in Γ ∪ Σ and (X, α₁ . . . αₖ) ∈ R. The size of a derivation tree is given by the number of tree nodes.

Example 1 – Context-free grammar. Let us consider the grammar G = (\{a, b\}, \{S, T\}, S, R), with R = \{S →

1As v₀ ∈ Σ∗, this derivation is obviously a left-most derivation.
It is known [23, Theorem I.1] that
\[ \sum \] of \( n \) recursively computed. Consider for instance the grammar
covers an element \( X \) of \( \Gamma \) if at least one of its nodes is labelled
by \( X \). For instance, for the tree in Example 1, the elements \( S \) and \( T \) are covered since they appear in the derivation tree.

b) Uniform Random Generation: The present issue is,
given a positive integer and a context-free grammar, to compute randomly with a uniform distribution a derivation tree of size \( n \) of this grammar. We will briefly explain here how to tackle this problem by using well-known counting techniques [23]. Notice that more advanced techniques allow a faster computation, like in [24].

As usual, the non-terminals symbols are denoted by capital letters. Given a context-free grammar \( G = (\Sigma, \Gamma, S_0, R) \), a non-terminal symbol \( X \) in \( \Gamma \), and a positive integer \( i \), the number of derivation trees of size \( i \) generated by \( (\Sigma, \Gamma, X, R) \) is denoted by \( x(i) \), i.e., using the corresponding lowercase letter.

Given a positive integer \( n \), for each symbol \( S \in \Gamma \), the sequence of positive integers \( s(1), \ldots, s(k), \ldots \) is introduced. The recursive computation of these \( s(i) \)'s is as follows. For each strictly positive integer \( k \) and each rule \( r = (S, w_1S_1 \ldots w_nS_nw_{n+1}) \in R \), with \( w_j \in \Sigma^* \) and \( S_i \in \Gamma \), let us set
\[ \begin{align*}
\beta_r &= 1 + \sum_{i=1}^{\infty} |w_i| \\
\alpha_r(k) &= \sum_{i_1 + \ldots + i_n = k - \beta_r} \prod_{j=1}^{n} s_j(i_j) \quad \text{if } n \neq 0 \\
\alpha_r(k) &= 0 \quad \text{if } n = 0 \text{ and } k \neq \beta_r \\
\alpha_r(\beta_r) &= 1 \quad \text{if } n = 0.
\end{align*} \]

It is known [23, Theorem I.1] that \( s(k) = \sum_{r \in R^*(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 \( \beta_r \) are strictly less than \( k \). This way, the \( s(i) \)'s can be recursively computed. Consider for instance the grammar \((\{a, b\}, \{X, Y\}, \{r_1, r_2, r_3\}) \) 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 \( x(2) = 1 + 1 + \sum_{i+j-2} x(i)x(j) = 2 \), 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 \( S \) and \( T \). The four derivation trees of size 5 are the trees of the form \( Z_1Z_2 \) where both \( Z_1 \) and \( Z_2 \) are derivation trees of size 2.

In order to generate derivation trees of size \( n \), all \( s(i) \)'s, for \( S \in \Gamma \) and \( i < n \), have to be computed with the

---

**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 \( \sum_{j=1}^{\ell} \alpha_{r_j}(n) = 0 \), then return “Exception”.
3. Pick \( i \in \{1, \ldots, \ell \} \) with probability \( Prob(i = j) = \frac{\alpha_{r_j}(n)}{\sum_{j=1}^{\ell} \alpha_{r_j}(n)} \).
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
   \[ Prob(x_1 = \ell_1, \ldots, x_m = \ell_m) = \frac{\prod_{j=1}^{m} x_j (\ell_j)}{\alpha_{r_i}(n)} \]
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 \( \ell_j \).
10. Return \( t \).

Fig. 1. Random Generation algorithm
each of the 4 trees is chosen with probability 1/4.

III. MIXING RANDOM TESTING AND COVERAGE CRITERIA

In a context of functional testing, the strength of random testing is to quickly provide many different test data, for each behaviour of the system. Moreover, these test data are independent of the choices of the test designer, and consequently they can catch problems (s)he did not anticipate. For instance, fuzz testing is particularly relevant for testing security requirements [7]. However, random testing can miss an important behaviour occurring with a very small probability. To exploit the advantages of both random testing and deterministic testing, a solution is to combine random generation and coverage criteria.

The general schema for this combination, as described in [1], is the following: considering a random generation algorithm of test data of size n and a coverage criterion C (each element of C is or is not covered by each possible test), the goal is to use the generation algorithm N times in order to optimise the probability of covering all elements of C. For each element e ∈ C, we denote by pne,n the probability that a generated test of size n covers e. One can easily check that generating N test data independently of C provides a probability of covering C of 1−(1−pmin)N, where pmin = minn∈C {pne,n}. This probability is the way to measure the quality of the testing approach, relatively to C. A better way is to repeat N times the following procedure:

1) Pick at random an element e ∈ C with a probability pπ,e, and
2) Generate uniformly a test of size n covering e.

This procedure requires to know how to uniformly generate a test of size n covering a given element, and to choose the probabilities pπ,e’s to optimise the probability of covering all elements of C.

Following [1], the optimisation requires solving the following constraint system: maximise p satisfying

\[
\begin{align*}
  \left\{ \begin{array}{l}
    p \leq \sum_{e \in C} \pi_e p_{e,n} \quad \text{for all } f \in C \\
    \sum_{e \in C} \pi_e = 1
  \end{array} \right.
\]

where pπ,e,f,n is the probability that a randomly generated test of size n covers both e and f. This linear programming problem can be solved in an efficient way, using simplex-like approaches.

In summary, in order to combine random testing and a coverage criterion, it is required to solve a constraint system and to know 1) how to randomly generate a test of a given size covering a given element, 2) how to compute the pπ,e,n’s; and 3) how to compute the pπ,e,f,n’s.

The rest of the paper is dedicated to the problem of the random generation of execution trees of a grammar, with the coverage criterion All non-terminal symbols. More precisely, given a grammar G = (Γ, Γ, S0, R) the coverage criterion being Γ, a test of size n being a derivation tree of G of size n, we say that X ∈ Γ is covered by a test if the derivation tree covers X.

IV. COMPUTING pX,n AND pX,Y,n

In this section G = (Γ, Γ, S0, R) is a context-free grammar. We denote by En(G) the set of derivation trees of size n of G. We respectively denote by Ex,n(G) and Ex,Y,n(G) the set of derivation trees of size n of G covering X, and covering both X and Y.

Let px,n be the probability of a randomly generated derivation tree of size n to cover X. Clearly, if En(G) is empty then px,n = 0 [resp. px,Y,n = 0]. Otherwise, px,n = |Ex,n(G)| / |En(G)| [resp. px,Y,n = |Ex,Y,n(G)| / |En(G)|].

Therefore, computing the probability defined in Section III − needed to solve the linear constraint program − reduces to the computation of the cardinality of sets Ex,n(G) and Ex,Y,n(G).

A. Computing |Ex,n(G)| and |Ex,Y,n(G)|

To compute |Ex,n(G)|, we build a grammar GX such that En(GX) and Ex,n(G) are in bijection (and therefore have the same number of elements).

For every w ∈ (Γ ∪ Σ)*, [w]0 is recursively defined by: [ε]0 = ε, [Z]0 = (Z, 0) ∈ Γ and [aw]0 = a[w]0 (with a ∈ Σ). Intuitively, [w]0 is obtained from w by changing each letter of w in Γ by the corresponding pair with 0 as second element. For instance, with the grammar of Example 1, one has [aSbbT]0 = a(S, 0)bb(T, 0). For every w ∈ (Γ ∪ Σ)*, [w]2 is defined exactly in the same way, changing all 0’s by 2’s.

For every w ∈ (Γ ∪ Σ)*, {w}1,2 is defined as the set of words w’ ∈ (Σ ∪ {1, 2})* obtained from w by replacing occurrence of each letter Z of Γ either by (Z, 1) or by (Z, 2), with the restriction that at least one is replaced by (Z, 1). The letters in Σ remain unchanged. For instance, if w = aSbbT, then {w}1,2 = {a(S, 1)b(T, 1), a(S, 2)b(T, 1), a(S, 1)b(T, 2)}. Notice that if w ∈ Σ* then {w}1,2 = 0 since the constraint is not satisfied.

Let GX = (Γ, Γ × {0, 1, 2}, (S0, 1), RX) where RX = R0 ∪ R1 ∪ R2 with:

- R0 = {(Z, 0) → [w]0 | Z → w ∈ R},
- R1 = {((Z, 1) → w’ | Z ≠ X and ∃Z → w ∈ R such that w’ ∈ {w}1,2),
- R1 = {(X, 1) → [w]0 | X → w ∈ R},
- R2 = {(Z, 2) → [w]2 | Z → w ∈ R and Z ≠ X}.

Intuitively, adding the value 0 to a symbol in Γ means that if this rule is used, there exists an occurrence of X at an upper position in the derivation tree. Adding the value 1 to a symbol in Γ means that there is no occurrence of X at an upper position, but there exists an occurrence of X at this or a lower position in the derivation tree. The value 2 means there is no occurrence of X appearing in the tree at an upper or lower position.

Example 2 − G X. Consider the grammar G = (\{a, b\}, \{S, T, X\}, S, R) with R = \{S → SS, S → aT, S → b, T → aa, X → TX, X → b\}. The grammar GX has the set of rules as follows:

\{S, 0\} → \{(S, 0)(S, 0), (S, 0) → a(T, 0), (S, 0) → ...\}.
(X, 0)b, (T, 0) → aa,(X, 0) → b, (X, 0) → (T, 0)(X, 0) \cup
\{(S, 1) \rightarrow (S, 1)(S, 1), (S, 1) \rightarrow (S, 1)(S, 2), (S, 1) \rightarrow
(S, 2)(S, 1), (S, 1) \rightarrow a(T, 1), (S, 1) \rightarrow (X, 1)b \cup
\{(X, 1) \rightarrow b, (X, 1) \rightarrow (T, 0)(X, 0) \cup
\{(S, 2) \rightarrow (S, 2)(S, 2), (S, 2) \rightarrow
a(T, 2), (S, 2) \rightarrow (X, 2)b, (T, 2) \rightarrow aa\}.

**Proposition 1 – Bijection.** There exists a bijection between
\( E_n(G_X) \) and \( E_{X,n}(G) \).

Example 3 illustrates several elements of the following proof.

**Proof:** Let \( \varphi \) be the function from \( (\Gamma \times \{0, 1, 2\} \cup \Sigma)^* \)
to \( (\Gamma \cup \Sigma)^* \) inductively defined by: \( \varphi(\varepsilon) = \varepsilon \) and \( \varphi(a\omega) = a\varphi(\omega) \) if \( a \in \Sigma \cup \Gamma \) and \( \varphi(Z\alpha\omega) = Z\varphi(\omega) \) if \( Z \in \Gamma \) and \( \alpha \in \{0, 1, 2\} \).

Intuitively, \( \varphi \) is a projection deleting the components in \( \{0, 1, 2\} \).

By construction of \( G_X \), if \( (Z, \alpha) \rightarrow w \) is a rule of \( G_X \)
then \( \varphi((Z, \alpha)) \rightarrow \varphi(w) \) is a rule of \( G \). Therefore, if \( x_0, \ldots, x_k \)
is complete derivation of \( G_X \), then \( \varphi(x_0), \ldots, \varphi(x_k) \) is a
complete derivation of \( G \). Moreover, the initial symbol of \( G_X \) is \( (S, 1) \)
and all rules of \( R_X \) with a left hand side in \( (\Gamma \setminus \{X\}) \times \{1\} \) have a right hand side where an element of \( \Gamma \setminus \{X\} \) occurs. Therefore, since \( x_n \in \Sigma^* \), the only way to
destroy the component 1 is to use a rule with the left hand
side \( (X, 1) \). It follows that the derivation tree associated to
\( \varphi(x_0), \ldots, \varphi(x_k) \) covers \( X \).

Consequently, \( \varphi \) induces a function from \( E_n(G_X) \) into
\( E_{X,n}(G) \). Let \( x_0, \ldots, x_k \) and \( x'_0, \ldots, x'_k \) be complete derivations of
\( G_X \), such that \( \varphi(x_0), \ldots, \varphi(x_k) = \varphi(x'_0), \ldots, \varphi(x'_k) \).
Assuming that \( x_0, \ldots, x_k \neq x'_0, \ldots, x'_k \), there exists a minimal
index \( t_0 \) such that \( x_{t_0} \neq x'_t \) since \( x_{t_0} = \varepsilon \) and \( t_0 \geq 0 \).
Therefore \( \varphi(x_{t_0}) = \varphi(x'_{t_0}) = v_0 \) and \( Z'_{t_0} = v_0 \), with
\( Z' \in \Gamma \) and \( \alpha \in \{0, 1, 2\} \). One of the following cases arises:

- If \( \alpha = 0 \), then there exist \( Z \rightarrow w \) and \( Z' \rightarrow w' \) in \( R \)
such that \( x_{t_0} = v_0[w]v_1 \) and \( x'_{t_0} = v_0[w']v_1 \). Since
\( \varphi(x_{t_0}) = \varphi(x'_{t_0}) \), it follows that \( \varphi(w) = \varphi(w') \).

It follows that \( \varphi(w) = w \) and \( \varphi(w') = w' \), proving that \( x_{t_0} = x'_{t_0} \),
a contradiction.

- If \( \alpha = 2 \), then the same proof holds, replacing 0 by 2. \[ \]

- If \( \alpha = 1 \) and \( Z = X \), then again, the same proof holds.

- If \( \alpha = 1 \) and \( Z \neq X \), then there exist \( Z \rightarrow w \) and \( Z' \rightarrow w' \) in \( R \)
such that \( x_{t_0} = v_0[w]v_1 \) and \( x'_{t_0} = v_0[w']v_1 \), with
(continues on next page)
the uniform random computation of execution trees of $G$ of a given size and covering $X$.

Since $E_{X,Y,n}(G) = E_{X,n}(G)$, computing $|E_{X,Y,n}(G)|$ is a direct application of the above techniques. Computing $|E_{X,Y,n}(G)|$, with $Y \neq X$, can almost be done by a similar construction: the difference is that the construction of the rules of the grammar $G_{XY}$, from the grammar $G_X$, must take into account that both $X$ and $Y$ have to appear in the derivation. Let $G_{XY} = (\Sigma, \Gamma \times \{0,1,2\} \times \{0,1,2\}, \{(S_0, 1, 1), R_{XY}\})$ where $R_{XY} = R_0 \cup R_1 \cup R_2$ with:

- $R_0 = \{(Z,i),0 \rightarrow \{w\}_0 | (Z,i) \rightarrow w \in R_X\}$,
- $R_1 = \{(Z,i),1 \rightarrow w' | Z \neq Y \text{ and } \exists(Z,i) \rightarrow w \in R_X \text{ such that } w' \in \{w\}_{1,2}\}$,
- $R_2 = \{(Y,i),1 \rightarrow \{w\}_1 | \{Y,i\} \rightarrow w \in R_X\}$,
- $R_2 = \{(Z,i),2 \rightarrow \{w\}_2 | (Z,i) \rightarrow w \in R_X \text{ and } Z \neq Y\}$.

A proof similar to the one of Proposition 1 allows showing that there is a computable bijection between $E_n(G_{XY})$ and $E_{X,Y,n}(G)$. Note that the size of $G_{XY}$ is approximatively $4^\ell$ times greater than the size of $G$.

### V. Experiments

The approach has been evaluated on a simplified version of the grammar of JSON\(^2\) (for JavaScript Object Notation) – a language independent common format for declaring objects. Formally, let us consider the grammar $G = (\Sigma, \Gamma, \text{Object}, R)$ with $\Sigma$ having the eight following elements $\Sigma = \{, \{, : \}, \text{letter, digit, [\}}, \text{.}$. The set $\Gamma$ of non-terminal symbols\(^3\) is composed of the elements "Object", "Members", "Pair", "Array", "Elements" and "Value". Finally, the set $R$ contains the following rules:

- $\text{Object} \rightarrow \{} | \{\text{Members}\}$
- $\text{Members} \rightarrow \text{Pair} | \text{Array, Members}$
- $\text{Pair} \rightarrow \text{letter, Value}$
- $\text{Array} \rightarrow [ ] | \{\text{Elements}\}$
- $\text{Elements} \rightarrow \text{Value} | \text{Value, Elements}$
- $\text{Value} \rightarrow \text{letter} | \text{Object} | \text{digit} | \text{Array}$

\(^2\)http://www.json.org/
\(^3\)To provide a more readable specification, the convention consisting in using capital letters for non-terminal symbols is not entirely respected here.

In order to optimise the coverage criterion, we have to solve the following system while maximising $p$ satisfying

$$\begin{align*}
 p &\leq \pi_{\text{Object}} + \pi_{\text{Members}} + \pi_{\text{Pair}} + \pi_{\text{Array}} + \pi_{\text{Elements}} + \pi_{\text{Value}} = 1 \\
 p &\leq \pi_{\text{Object}} + \pi_{\text{Members}} + \pi_{\text{Pair}} + \pi_{\text{Array}} + \pi_{\text{Elements}} + \pi_{\text{Value}} = 1
\end{align*}$$

Using a slightly modified version of the Hoa tool ([25]), the computation of the probabilities $p_{X,n}$ and $p_{X,Y,n}$ for all $X, Y \in \Gamma$ and $n = 20$ has been performed efficiently (a few seconds). The system becomes as below, and we have then to solve it while maximising $p$ satisfying

$$\begin{align*}
 p &\leq \pi_{\text{Object}} + \pi_{\text{Members}} + \pi_{\text{Pair}} + \pi_{\text{Array}} + \pi_{\text{Elements}} + \pi_{\text{Value}} = 1 \\
 p &\leq \pi_{\text{Object}} + \pi_{\text{Members}} + \pi_{\text{Pair}} + \pi_{\text{Array}} + \pi_{\text{Elements}} + \pi_{\text{Value}} = 1
\end{align*}$$
This linear programming problem can be solved in an efficient way, using simplex-like approaches. We have used the tool lp_solve\(^4\) to solve it, and the result is that \( p = 1 \) if \( \pi_{\text{Object}} = 0, \pi_{\text{Members}} = 0, \pi_{\text{Pair}} = 0, \pi_{\text{Array}} = 0, \pi_{\text{Elements}} = 1, \) and \( \pi_{\text{Value}} = 0 \). It means that, for this simple example, the optimised approach to cover all the non-terminals symbols, consists in generating derivation trees covering \textit{Elements}. Indeed, in this grammar, the generation of a derivation tree covering the non-terminal symbol \textit{Elements} provides a tree covering all the other non-terminal symbols.

VI. CONCLUSION

In this paper, we have presented a method for exploiting a coverage criterion together with random testing in the context of grammar-based testing. This automatic method lies in building a grammar and then in resolving a linear constraint system, which can be done by adapted tools, even for large values. In the future, we plan to extend the approach to other coverage criteria such as rules coverage, and also to handle attribute grammars with constraints formalising the semantics of context-free languages.

REFERENCES


\[25\] I. Enderlin, “Hoa project, a set of php libraries.” \texttt{URL: http://hoa-project.net}.

\[4\] \texttt{http://lp.solve.sourceforge.net/}