Introducing a Pure Weight-Based Concept for Automatic Test Case Generators

Franco Bazzichi

Abstract—A concept for a *weight-based* test case generator for context-free grammars is introduced, where *weights* are set for each rule and used as *unique* driver while deciding which rule to use during the derivation process; we demonstrate why *static flat* weights generally cannot work but, introducing a simple corrective measure, a *dynamic* weight-based approach is able to produce random sentences while ensuring coverage and convergence; finally, analyzing the grammar from a new perspective, we describe how it is always possible to define *constraints* for the identification of *balanced weights*: balanced weights own the interesting characteristic of making it possible to drive the sentence generation, by a pure weight-based automatic test case generator, of extremely random but finite sentences able to stress the system being tested more extensively than the shortest derivation methods. An example of the application of balanced weights approach is included, about a simplified context-free grammar for a theoretical programming language, and possible future directions to strengthen the balanced approach are eventually outlined.

Index Terms— Test case generators, Test program generation, Test coverage, Random derivation, Weight-based grammars, Weight-based rules, Weight-based derivation, Random derivation, Neural network

----- ◆ -----

1 INTRODUCTION

TEST case generators for context-free grammars have been developed to offer an automatic way to obtain sentences to exercise systems during the test phase; automatic test case generators usually are more effective than manual process, and several tools have been successfully developed and applied in the industry for decades.

One of the required characteristics of test case generators is the *coverage* of all the syntax elements to be tested, meaning the usage of all corresponding rules of the underlying context-free grammar; options to exclude some syntax elements as well as to introduce controlled wrong sequences is welcome and in general test case generators offer options to configure these features.

Main concern while approaching automatic test case generation is represented by the *convergence* of the process, and methods to ensure the process will finish in a finite time producing the expected results have to be adopted; a milestone in such a direction is represented by the process described in [1] and refined in more recent papers [2], [3], [4]: other than the convergence, such a

process also owns the interesting feature of generating sentences having the minimal length.

Another relevant characteristic of automatic test case generators (and very powerful for effective tests) is the capacity to produce random but syntactically valid sentences, where huge combinations of primitive syntax elements, which would probably not be thought by humans, are generated; the random feature, able to deeply and widely exercise the system being tested, is the primary goal addressed by this paper, while it is not the main characteristic of [1] which privileges the shortness of generated sentences and where the produced output is always the same anytime the process is repeated.

Where automatic test case generators differ from each other is the logic designed to select, between all rules expanding the same nonterminal, the rule to be used to substitute it. Within this paper all the presented approaches share the same logic, and this is simply and uniquely based on *weights*, introduced in Section 3.

A weight-based approach emphasizes the *random* feature (while releasing the goal to obtain shortest sentences) and represents an alternative against the most widely tools designed so far.

Beginning with a theoretical flat approach, where all productions have the same chance to be selected, we will add a *penalty* concept to make a weight-based approach a working solution; then, starting from a simple example and thereafter moving to more complex grammars, we will see how formal conditions which have to be satisfied by *static balanced weights* can be designed: definition of such conditions, making the penalty concept unnecessary, can be obtained by analyzing the embedded structure of productions and corresponding relationships between

• F. Bazzichi is at Via Francesco Petrarca, 26 – I-22071 Cadorago (Como) - Italy. E-mail: franco.bazzichi@gmail.com.

nonterminals from a new perspective.

The Balanced approach, described in Section 6, represents the final step in the weight-based proposals introduced in this paper, as it is able to indefinitely generate full random but always finite sentences, ensuring coverage and combined with a very simple logic in selecting the rule to be used during the generation cycles.

2 DEFINITIONS

According to [5] a context free grammar G is a Type-2 grammar defined as a 4-tuple $G = (T_G, N_G, P_G, S_G)$, where:

1. T_G is a finite set; elements of T_G are called the *terminals* of G ; within this paper, the typeface used for terminals is **bold**;
2. N_G is a finite set, disjoint from T_G ; elements of N_G are called the *nonterminals* (or *symbols*) of G ; within this paper, the typeface used for nonterminals is *italic*;
3. P_G is a finite relation from N_G to $(T_G \cup N_G)^*$; elements of P_G are called the *productions* (or *rules*) of G , and identified as p_i ; a production p_i is expressed here in the form $A \rightarrow \alpha$, where A is an element of N_G , $\alpha \in (T_G \cup N_G)^*$ and i is named *index* of the rule;
4. S_G is an element of N_G and is known as the *start symbol* of G : no productions $A \rightarrow \alpha$ exist where $S_G \in \alpha$.

We define the function $\mathcal{R}: A \rightarrow \{i \mid p_i: A \rightarrow \alpha\}$; in other terms, $\mathcal{R}(A)$ corresponds to the set containing the indexes of all productions expanding the nonterminal A .

Moreover, we also define the function $\mathcal{S}: A \rightarrow \{A_j \in N_G \mid \exists p_i: A \rightarrow \alpha A_j \beta\} \cup \{\mathcal{S}(A_k) \mid A_k \in \mathcal{S}(A)\}$; in other terms, $\mathcal{S}(A)$ represents the *successors* of A , meaning the set of symbols contained in all the derivation trees which can be generated starting from nonterminal A .

Then, we define as *recursive* a grammar G where $\exists A \in N_G \mid A \in \mathcal{S}(A)$.

The relevance of such a definition is that non-recursive grammars generate *finite* languages, while recursive grammars generate *infinite* languages, and this has an impact on the application scope of the different weight-based approaches.

About recursion, we can have two different situations: 1) *direct* recursion; and 2) *indirect* recursion.

A symbol A is *direct* recursive if a rule $A \rightarrow \alpha A \beta$ exists; direct recursion is easy to discover by just looking at productions, one at a time.

To detect *indirect* recursion we need to look at more productions to discover all the successors: an indirect recursion exists when we can identify a *cycle* starting from a

nonterminal and returning to the same, passing through distinct nonterminals; the length m of the cycle is the number of distinct nonterminals involved.

The following is an example having $m=3$ (involved nonterminals are A_0, A_1 and A_2):

$$\begin{aligned} A_0 &\rightarrow \alpha_0 A_1 \beta_0 \\ A_1 &\rightarrow \alpha_1 A_2 \beta_1 \\ A_2 &\rightarrow \alpha_2 A_0 \beta_2 \end{aligned}$$

Finally, we define symbols A_1 and A_2 as part of the same *cluster* if $A_1 \in \mathcal{S}(A_2)$ and $A_2 \in \mathcal{S}(A_1)$.

3 INTRODUCING WEIGHTS

In this paper, all approaches apply the same logic to identify which production has to be used at the time when a nonterminal needs to be replaced, and that logic is simply and uniquely based on *weights*.

A *weight-based context-free grammar* could be defined as a 5-tuple $G = (T_G, N_G, P_G, S_G, W_G)$, where W_G has a finite relation 1:1 with P_G (w_i identifies the weight set for production p_i).

Weights, having values $0 \leq w_i \leq 1$, define the *chance* of each p_i to be selected: in case p_i and p_j are candidate productions to expand the same nonterminal at some time of the generation cycle, having $w_i > w_j$ means that p_i has a greater chance to be selected than p_j (but this does not mean p_i will surely be the actual selected one, being the effective choice determined by the random selection mechanism); if $w_i = 0$, p_i will not be selected.

Moreover, considering all p_i expanding the same symbol, within a weight-based approach the sum of corresponding w_i must always be 1; using the function $\mathcal{R}$ introduced in Section 2, this can be expressed as $\forall A \in N_G: \sum_{i \in \mathcal{R}(A)} w_i = 1$.

An implementation of a random selection mechanism to identify which rule to apply for a weight-based context-free grammar can be designed using a function (e.g. *rnd()*), generating random real numbers r ($0 \leq r < 1$) with uniform distribution; assuming that all the n rules expanding the same nonterminal are $p_k, \dots, p_{k+n-1}$, a possible algorithm is the following, which at the end identifies p_i as rule to be applied:

```

r = rnd()
i = k
sum = w_i
while sum < r do
    i = i + 1
    sum = sum + w_i
end while

```

The definition of weights is completed by specifying their *type*, which can be *static* or *dynamic*: 1) *static*: W_G are kept with their initial values during the overall genera-

tion cycles; and 2) dynamic: W_G could change while progressing with the generation cycle, depending on some conditions.

As additional note, programming languages usually own a function to set an initial *seed* in order to exactly reproduce the same sequence of random numbers being generated: so, in case it would be interesting, setting a seed at the beginning of the generation cycles allows implementing a repeatable process, where the output is always the same, even with a weight-based generator.

4 FLAT

With **Flat** we mean a weight-based approach where all productions expanding the same nonterminal always have the same chance to be selected; this means that the Flat approach uses *static* weights, set to $1/n$ to all n rules expanding the same symbol.

The Flat approach is always effective in case of non-recursive grammars (so, for *finite* languages), as in the following example, introduced in [6]:

```

S → NP VP .
NP → capitalized_article subject
VP → verb article object
article → a | the
capitalized_article → A | The
subject → customer | system administrator | programmer
object → system | compiler | program
verb → used | installed | developed

```

Starting from the nonterminal S , rules can be actually selected until no more nonterminals exist; the generation cycle is repeated from the beginning until all rules are used at least once. Assuming that the random selection mechanism is designed correctly, the overall process always terminates in a finite time.

Moving towards *recursive* grammars, we have situations where a Flat approach is safely applicable too; let us consider the following grammar, which is recursive due to p_3 :

```

1. S → id
2. id → letter
3. id → letter id
4. letter → a | b | ... | z

```

By definition, we have $w_2=w_3=0.50$ and the process converges as soon as to expand nonterminal *id* the production p_2 is eventually selected instead of p_3 (and this will happen, sooner or later, as $w_2 > 0$).

But, in general, the Flat approach does not work as it could lead to a never-ending process; this because convergence during sentence generations is threatened by recursive rules.

As example, we can empirically assess that the Flat approach is not safe for the following recursive grammar G_0 :

```

1. S → exp
2. exp → term + exp
3. exp → term
4. term → id
5. term → ( exp )

```

According to the definition, $w_2=w_3=w_4=w_5=0.50$; based on that and assuming that the random selection mechanism always identifies p_2 and p_3 alternatively (the same for p_4 and p_5), we see that an infinite generation $S \Rightarrow exp \Rightarrow term + exp \Rightarrow id + exp \Rightarrow id + term \Rightarrow id + (exp) \Rightarrow \dots$ is started, using in sequence rules 1-2-4-3-5: at that point, we are back to the initial situation where nonterminal *exp* needs to be expanded, meaning that the process is definitely not able to converge.

Section 6 will include a formal proposal to calculate the *convergence index*, measuring the capacity of a weight-based context-free grammar to have finite generation cycles, and we will demonstrate that the Flat approach cannot work for the above grammar.

5 FLAT WITH PENALTIES

A mitigation mechanism to address the weakness of the Flat approach is necessary to ensure the convergence for any context-free grammar using a weight-based approach: with **Flat with Penalties** we introduce the possibility to dynamically change initial weights by *penalties*, while generating sentences: the idea is to downgrade weights (i.e. the chance to be selected) of not *smart* rules as soon as they are used.

For each nonterminal, a rule is identified as *smart* if it has the shortest derivation length; the process to identify smart rules is very similar to the computation of *SLEN* as introduced in [1], with the only difference (useful here to address coverage) being for each nonterminal we could have *multiple* rules identified as smart¹.

Flat with Penalties uses *dynamic* weights, initially set by using the same principle of the Flat approach² and, in case that at the end of a generation cycle not all rules have been used (or used less than the minimum number of wished occurrences), it resumes after all weights have been reset to their initial flat value.

¹According to the official definition, *SLEN* is a single-dimension array: always *one* production only is identified as shortest, even if multiple productions with same length could exist; nevertheless, the additional data structures and the logic adopted to select rules make this irrelevant within [1]

²Choosing adequate initial weights could produce a more balanced output (without deep nesting primarily concentrated at the beginning of generated sentences), but this option is not addressed in this paper

The penalty mechanism works through a configuration parameter m ($0 < m < 1$): every time a non-smart production p_i is used, weight w_i is reduced by $k=\min(w_i, m)$; at the same time, to respect the weight definition, weight of all n smart rules expanding the same nonterminal is increased by k/n ; no weights are changed if the selected production is a smart one.

Due to reductions, weight w_i for a non-smart rule at some time could be 0, meaning rule p_i will not be selected anymore during the current generation cycle.

No specific criteria are defined to identify the value for the configuration parameter m , but it is quite clear that the convergence (end of a generation cycle) is faster for higher values of m .

The same mechanism to select a rule for a weight-based context-free grammar introduced in Section 3 is used even with Flat with Penalties (all rules act as candidates, smart as well as non-smart), and the unique difference with Flat is because of penalties.

Considering the grammar G_0 introduced in the previous section, the Flat with Penalties starts setting the same flat weight 0.50 to rules 2/3 and 4/5; about *smart*, Table I contains the corresponding values for each nonterminal:

TABLE I
SMART FOR G_0

Nonterminal	Smart
S	1
exp	3
$term$	4

Then, assuming that the configuration parameter m is set to 0.05, once the rules 1-2-4-3-5 have been used, the weights will have values as in Table II:

TABLE II
WEIGHTS VALUES FOR G_0 AFTER 5 STEPS

Rule	Initial Weight	Current Weight
1	1.00	1.00
2	0.50	0.45
3	0.50	0.55
4	0.50	0.55
5	0.50	0.45

In case the random selection mechanism will identify again rules 2 and 5 to expand exp and $term$ (at this stage the difference between w_2 and w_3 , as well as between w_4 and w_5 , is not so high), w_2 and w_5 will decrease again while w_3 and w_4 will rise up correspondingly; so, the probability of using a smart rule is increased every time a non-smart rule is selected, ensuring the convergence of the generation process.

So, just by using one internal data structure (*smart*) and a very simple process logic, the Flat with Penalties approach is effective and always produces random sentences in a finite time (a higher number of sentences might be required in or-

der to cover all syntax elements compared with [1]).

6 BALANCED

As it was addressed in Section 4, the Flat approach safely works just for *some* recursive grammars, as *static flat* weights (same weight for all rules expanding the same nonterminal) cannot generally ensure the convergence. To resolve such an issue, the Flat with Penalties uses *dynamic* weights adopting a penalty mechanism.

The **Balanced** concept represents a step forward, as it defines *static* weights which *always* ensure convergence, while eliminating the need to use the penalty mechanism; given any consistent context-free grammar, *balanced* weights can always be identified and a pure weight-based test case generator can be safely used to obtain full random finite sentences.

The challenge here is *how* to find such balanced weights: the idea is to analyze the grammar from a new perspective in order to capture the *contribution* of each rule to change the total number of occurrences of nonterminals in the sentence being generated, and consequently to identify which conditions the weights have to fulfill to be balanced.

6.1 Delta

Before outlining how balanced weights can be found, we firstly introduce *delta* (Δ), a table where the element $\delta_{i,A}$ contains the resulting difference of occurrences of nonterminal A in the sentence being generated in case $p_i: A_j \rightarrow \alpha$ would be used; if $A \notin \{A_j\} \cup \alpha$, $\delta_{i,A}$ is not defined.

Values of Δ are identified by just observing each rule of P_G independently of each other: for rule p_i , $\delta_{i,A}$ (when-ever is defined) is set to the total number of occurrences of nonterminal A within α , reduced by 1 in case $i \in \mathcal{R}(A)$.

As example, referring to the grammar G_0 introduced in Section 3, Table III shows how Δ looks like ($\delta_{i,A} = '-'$ where $\delta_{i,A}$ is not defined):

TABLE III
DELTA (Δ) FOR G_0

Rule	S	exp	$term$
1	-1	+1	-
2	-	0	+1
3	-	-1	+1
4	-	-	-1
5	-	+1	-1

Then, using Δ , the contribution of rule i to change the number of total occurrences of symbol A in the sentence being generated can be calculated as $w_i \delta_{i,A}$.

6.2 Constraints, Basics and Advanced

Next step is the definition of *constraints*, conditions the weights have to satisfy to represent *balanced* weights; any set of weights which satisfies all the defined constraints

represents a valid set of balanced weights able to ensure that the generation process always ends in a finite time.

A first set of constraints, named *basics*, descends by the definition of weights, as introduced in Section 3: 1) $\forall A \in N_G: \sum_{i \in \mathcal{R}(A)} w_i = 1$; and 2) $\forall i \in \mathbb{N} \mid 1 \leq i \leq |P_G|: w_i > 0^3$.

Then, observing the source grammar from a particular perspective, we also identify *advanced* constraints which, for each symbol, establish *relationships* between weights and Δ : advanced constraints are the most significant components which qualify the Balanced approach.

At this point, let us see how advanced constraints can be defined, starting from a simple grammar G_1 :

1. $S \rightarrow A$
2. $A \rightarrow \mathbf{a}$
3. $A \rightarrow \mathbf{b} A$
4. $A \rightarrow \mathbf{c} A A$
5. $A \rightarrow \mathbf{d} A A A$

Corresponding values Δ_1 are shown in Table IV (column for non-recursive symbols, as by definition the start symbol S , can be omitted from Δ as irrelevant):

Rule	A
1	+1
2	-1
3	0
4	+1
5	+2

Graphically, G_1 can be represented as in Fig. 1 (characteristics of such representation are described later):

Fig. 1. Graphic representation of G_1

To define the advanced constraint for symbol A , we have to consider how it should be possible to make all of its occurrences, as introduced by direct recursive rules, *disappear* from the generation sentence: this, for grammars not having indirect recursive rules like G_1 , is simply obtained when the overall contribution of rules *removing* A is higher than the overall contribution of rules *adding* A .

So, in this simple scenario the advanced constraint is represented by condition (1):

$$\sum_{i \in \mathcal{R}(A)} w_i \delta_{i,A} < 0. \quad (1)$$

For simplicity, we identify $\sum_{i \in \mathcal{R}(A)} w_i \delta_{i,A}$ as σ_A ; therefore, (1) can also be rewritten as follows:

$$\sigma_A < 0. \quad (2)$$

By applying (2) and using the defined $\delta_{i,A}$ as for Δ_1 , the condition which defines the advanced constraint for G_1 becomes

$$w_2 \delta_{2,A} + w_3 \delta_{3,A} + w_4 \delta_{4,A} + w_5 \delta_{5,A} < 0 \\ \Rightarrow -w_2 + w_4 + 2w_5 < 0.$$

The fact that w_3 is not part of the resulting advanced constraint means that productions having $\delta_{i,A}=0$ (i.e. direct recursive rules having just 1 occurrence of expanding symbol on the right side, like p_3) do not influence convergence at all.

Therefore, the entire set of constraints for this grammar is composed by the following conditions:

$$\begin{cases} \forall i \in \{1, \dots, 5\}: w_i > 0 \\ \sum_{i \in \mathcal{R}(A)} w_i = 1 \\ -w_2 + w_4 + 2w_5 < 0. \end{cases}$$

Being the grammar of this example very simple, even the constraints are simple too: this offers the possibility to *empirically* find valid balanced weights that satisfy *all* defined constraints, both basics and advanced ones⁴.

Some examples of valid balanced weights are represented by the following sets, for which the corresponding σ_A are enclosed in brackets: 1) $w_2=0.50, w_3=0.25, w_4=0.10, w_5=0.15$ (-0.10); 2) $w_2=0.45, w_3=0.20, w_4=0.30, w_5=0.05$ (-0.05); and 3) $w_2=0.45, w_3=0.40, w_4=0.10, w_5=0.05$ (-0.25).

Differences between σ_A of different sets of balanced weights for symbol A express different *speed* of convergence, hence the ability to be faster in eliminating all occurrences of A ; to introduce a measure of such an ability, we define the *convergence index* z_A , where higher values mean greater convergence.

Writing (1) as $x < k$, the convergence index expresses the measure of the *strength* of the disequation, that is how much x is less than k ; it can be defined by $k-x$ and, in this simple scenario, we consequently have $z_A = -\sigma_A$.

Convergence is safely ensured where $z > 0$; nevertheless, having a higher convergence index may not lead to better results: balanced weights having higher convergence index

³In case, for some reasons, we are not interested in coverage or wish to avoid using specific rules, basic constraints of type 2) could be $w_i \geq 0$ or $w_i = 0$, respectively; in the paper we always assume $w_i > 0$

⁴As advanced constraints are disequations, multiple solutions could exist

usually require more generation cycles in order to also satisfy coverage.

Referring uniquely to sets 1), 2) and 3), the corresponding convergence indexes z_A are, respectively, 0.10, 0.05 and 0.25. That means with set 3) the convergence will be the fastest (generally, as it is depending on the random mechanism), while using set 2) it will be the slowest.

Vice versa, convergence is not ensured, for instance, for set $w_2=0.55$, $w_3=0.05$, $w_4=0.10$, $w_5=0.30$ (0.15), set $w_2=0.40$, $w_3=0.30$, $w_4=0.20$, $w_5=0.10$ (0) and set $w_2=0.35$, $w_3=0.10$, $w_4=0.30$, $w_5=0.25$ (0.45) because, even if all basic constraints are satisfied, the advanced one is not, meaning they cannot represent valid balanced weights.

They compute z_A respectively, as -0.15, 0 and -0.45, meaning that, over time, with the first set the occurrences of symbol A in the sentence being generated will increase slightly, while with the last set they will increase greatly; cases where $\sigma_A=z_A=0$ represent limit situations where the generation cycle never terminates, not being able statistically to definitely remove all the occurrences of symbol A in the generation string.

Nevertheless, due to random mechanism, it could happen that the generation cycle sometimes terminates by using unbalanced weights too, depending on the effective sequence the rules are selected.

Obviously grammars are in general not so simple like G_1 , so now we move a step forward by analyzing G_2 , which includes both direct and indirect recursions.

1. $S \rightarrow E$
2. $E \rightarrow E + T$
3. $E \rightarrow T$
4. $T \rightarrow T * F$
5. $T \rightarrow F$
6. $F \rightarrow \text{id}$
7. $F \rightarrow E$

Corresponding values Δ_2 are shown in Table V:

TABLE V
DELTA (Δ) FOR G_2

Rule	E	T	F
1	+1	-	-
2	0	+1	-
3	-1	+1	-
4	-	0	+1
5	-	-1	+1
6	-	-	-1
7	+1	-	-1

Graphically, we can describe it as in Fig. 2: path from symbol A_1 to A_2 exists if a rule $A_1 \rightarrow \alpha A_2 \beta$ exists, while paths to black circles represent a rule $A \rightarrow \alpha$ where $\alpha \in T_G$; a graphical representation helps in understanding the logic behind the advanced constraints, as it easily allows to follow the flow from a nonterminal to the next one.

Focusing on symbol E of G_2 , dotted lines represent the path which can reintroduce it (through symbol F , when p_7 is used).

Fig. 2. Graphic representation of G_2 (focus on E)

In such a similar situation, having (2) as constraint is not enough, as it does not take into account indirect recursive rules either, with the possibility to reintroduce the symbol E : to take care of such indirect contributions, we need to estimate *how many* symbols E are statistically reintroduced by walking through the recursive path $E \Rightarrow T \Rightarrow F \Rightarrow E$.

To evaluate such indirect contributions, the *rally* concept is introduced (for symbol A , denoted as λ_A): the rally specifies the average number of steps which are necessary to statistically exit from expanding an occurrence of a symbol, applying the weight-based selection mechanism.

The rally λ is function of σ , and defined as follows:

$$\lambda = f(\sigma) = \begin{cases} -\frac{1}{\sigma}, & \sigma < 0 \\ +\infty, & \sigma \geq 0. \end{cases}$$

For G_2 we have $\lambda_E = -1 / (w_2\delta_{2,E} + w_3\delta_{3,E}) \Rightarrow \lambda_E = 1 / w_3$; as example, 4 steps are necessary (on average) to exit from expanding E just by using p_2 and p_3 in case $w_3=0.25$.

The condition defining the advanced constraint for symbol E (part of an indirect recursive path) has to ensure that the capacity to make the symbol E disappear, by applying rules expanding E only, is enough to counteract the impact of the number of occurrences of symbol E introduced for effect of indirect recursive path $E \Rightarrow T \Rightarrow F \Rightarrow E$.

Using the rallies λ_E , λ_T and λ_F , the advanced constraint for E of G_2 can be expressed by

$$\lambda_E \sigma_E + \lambda_E (w_2\delta_{2,T} + w_3\delta_{3,T}) \cdot \lambda_T (w_4\delta_{4,F} + w_5\delta_{5,F}) \cdot \lambda_F (w_7\delta_{7,E}) < 0.$$

The first term of the left side of the constraint, $\lambda_E \sigma_E$, analyzes the effect of rules expanding E , including the direct recursive too.

The remaining terms of the left side, always greater than 0, analyze the indirect recursive path (defined as *cycle*) of G_2 , $E \Rightarrow T \Rightarrow F \Rightarrow E$, and estimate the combined effect of counting:

1. How many occurrences of symbol T are introduced in the string being generated during the λ_E steps, computed as $\lambda_E (w_2\delta_{2,T} + w_3\delta_{3,T})$;
2. For each T introduced by 1, how many occurrences

of symbol F are introduced in the string being generated during the λ_T steps, computed as $\lambda_T(w_4\delta_{4,F} + w_5\delta_{5,F})$;

3. For each F introduced by 2, how many occurrences of symbol E are reintroduced, due to the λ_F steps, computed as $\lambda_F(w_7\delta_{7,E})$.

In case an indirect recursive path should not exist, these additional terms do not exist either, and the resulting advanced constraint simplifies into $\lambda_E\sigma_E < 0$, which represents a valid generalization of previous $\sigma_E < 0$: indeed, $\lambda_E\sigma_E$ is less than 0 if and only if $\sigma_E < 0$ (λ_E cannot be negative).

From the definition of rally and assuming $\sigma_E < 0$ (if not, we have no convergence, even if ignoring the negative effect of indirect recursive rules), the first term is always equal to -1; so, the condition for the advanced constraint can be rewritten as follows:

$$\lambda_E(w_2\delta_{2,T} + w_3\delta_{3,T}) \cdot \lambda_T(w_4\delta_{4,F} + w_5\delta_{5,F}) \cdot \lambda_F(w_7\delta_{7,E}) < 1.$$

If we wanted now to shift the focus on symbol T of G_2 (as well as on F), by applying the same logic we would obtain the same condition: indeed, considering a cycle, the advanced constraint is *always the same* for *all* symbols, which are part of the same cycle, no matter what position they have in the cycle.

The left side of the condition can easily be found by looking at the graphical representation, being the advanced constraint for a cycle simply obtained by the product of elements part of the cycle itself (ignoring direct recursive paths as already embedded in the corresponding λ), where:

1. Name of each nonterminal A is replaced by corresponding λ_A ;
2. Indexes over a path ending with nonterminal A are replaced by sum of corresponding $w_i\delta_{i,A}$.

In general, considering a cycle of length m ($m \geq 2$) composed by nonterminals $A_0, A_1, \dots, A_{m-1}, A_m$ (where $A_m \equiv A_0$), the condition representing the advanced constraint for any of those symbols can be expressed as in (3):

$$\prod_{k=0}^{m-1} \lambda_{A_k} \sum_{i \in \mathcal{R}(A_k)} w_i \delta_{k,A_{k+1}} < 1.$$

According to the definition, in the scenario where symbols are part of a cycle the convergence index is calculated as $z = 1 - \prod_{k=0}^{m-1} \lambda_{A_k} \sum_{i \in \mathcal{R}(A_k)} w_i \delta_{k,A_{k+1}}$.

By applying (3) to G_2 and using values of Δ_2 , we obtain the following expressions representing λ_E , λ_T and λ_F : $\lambda_E = 1/w_3$, $\lambda_T = 1/w_5$, and $\lambda_F = 1$; then, being $(w_2 + w_3) = (w_4 + w_5) = 1$, the condition representing the advanced constraint for symbols which are part of cycle $E \Rightarrow T$

$\Rightarrow F \Rightarrow E$ becomes

$$w_7/(w_3w_5) < 1.$$

Therefore, balanced weights for this grammar have to satisfy the following set of constraints (the basics ones and the advanced ones):

$$\begin{cases} \forall i \in \{1, \dots, 7\}: w_i > 0 \\ \sum_{i \in \mathcal{R}(E)} w_i = 1 \\ \sum_{i \in \mathcal{R}(T)} w_i = 1 \\ \sum_{i \in \mathcal{R}(F)} w_i = 1 \\ w_7/(w_3w_5) < 1. \end{cases}$$

The most qualifying condition is represented by the advanced constraints and, due to its simplicity, for G_2 it is quite easy to empirically find possible solutions, like 1) $w_2=0.10$, $w_3=0.90$, $w_4=0.40$, $w_5=0.60$, $w_6=0.47$, $w_7=0.53$ ($z=0.02$); 2) $w_2=0.60$, $w_3=0.40$, $w_4=0.70$, $w_5=0.30$, $w_6=0.90$, $w_7=0.10$ ($z=0.17$); and 3) $w_2=0.90$, $w_3=0.10$, $w_4=0.10$, $w_5=0.90$, $w_6=0.95$, $w_7=0.05$ ($z=0.44$).

Back to the Flat approach, we are now able to formally demonstrate that it is not safe for G_2 ; this can be done just by setting all w_i to the flat values, and then to assess if all the constraints are satisfied: by definition, for G_2 $w_2, \dots, w_7$ are all set to 0.50, which computes $w_7/(w_3w_5)$ as 2, formally confirming that the Flat approach does not work for G_2 .

Similarly, about G_0 introduced in Section 4, we can now formally assess that the Flat approach is not applicable either: using the same method described above, the resulting qualifying condition is

$$\lambda_{exp}(w_2\delta_{2,term} + w_3\delta_{3,term}) \cdot \lambda_{term}(w_4\delta_{4,exp}) < 1.$$

Such a condition is not satisfied, as the left side is equal to 1: being convergence index $z = 0$, representing the limit condition for non-convergence (ensured where $z > 0$), it formally confirms the previous empirical deduction.

To make another up-shift in grammar complexity, let us consider a grammar having n clustered symbols (e.g. 2 clusters, composed by symbols $\{A, B, C\}$ and $\{D, E\}$, respectively), each one containing one or more distinct cycles (e.g.: $A \Rightarrow B \Rightarrow A$ and $A \Rightarrow B \Rightarrow C \Rightarrow A$ for the cluster $\{A, B, C\}$) as in Fig. 3:

Fig. 3. A grammar with 2 clusters where a cluster has 2 cycles

In this scenario we will have n advanced constraints, where the left side of each one is the sum of expressions defining the advanced constraint about each cycle, part of

the same cluster, considered separately.

As a last example, and concrete application of all the above definitions, we introduce now G_3 , a context-free grammar representing a subset of a typical programming language⁵; we will see how advanced constraints look like, we will identify valid balanced weights and, finally, we will show a sample of generated sentences.

1. $S \rightarrow \text{body}$
2. $\text{body} \rightarrow \text{begin } \text{stm_list} \text{ end}$
3. $\text{stm_list} \rightarrow \text{stm}$
4. $\text{stm_list} \rightarrow \text{stm} ; \text{stm_list}$
5. $\text{stm} \rightarrow \text{body}$
6. $\text{stm} \rightarrow \text{simple_stm}$
7. $\text{stm} \rightarrow \text{comp_stm}$
8. $\text{simple_stm} \rightarrow \text{ass_stm}$
9. $\text{ass_stm} \rightarrow \text{id} := \text{exp}$
10. $\text{comp_stm} \rightarrow \text{if } \text{cond_exp} \text{ then } \text{stm} \text{ else } \text{stm}$
11. $\text{comp_stm} \rightarrow \text{while } \text{cond_exp} \text{ do } \text{body}$
12. $\text{cond_exp} \rightarrow \text{id}$
13. $\text{cond_exp} \rightarrow (\text{exp} < \text{exp})$
14. $\text{exp} \rightarrow \text{term}$
15. $\text{exp} \rightarrow \text{term} + \text{exp}$
16. $\text{term} \rightarrow \text{factor}$
17. $\text{term} \rightarrow \text{factor} * \text{term}$
18. $\text{factor} \rightarrow \text{id}$
19. $\text{factor} \rightarrow (\text{exp})$

Values of Δ for this grammar are shown in Table VI below (as usual, only recursive symbols have been included as columns, because relevant for the identification of advanced constraints, opposite to non-recursive ones):

TABLE VI
DELTA (Δ) FOR G_3

Rule	body	stm_list	stm	comp_stm	exp	term	factor
1	+1	-	-	-	-	-	-
2	-1	+1	-	-	-	-	-
3	-	-1	+1	-	-	-	-
4	-	0	+1	-	-	-	-
5	+1	-	-1	-	-	-	-
6	-	-	-1	-	-	-	-
7	-	-	-1	+1	-	-	-
8	-	-	-	-	-	-	-
9	-	-	-	-	+1	-	-
10	-	-	+2	-1	-	-	-
11	+1	-	-	-1	-	-	-
12	-	-	-	-	-	-	-
13	-	-	-	-	+2	-	-
14	-	-	-	-	-1	+1	-
15	-	-	-	-	0	+1	-
16	-	-	-	-	-	-1	+1
17	-	-	-	-	-	0	+1
18	-	-	-	-	-	-	-1
19	-	-	-	-	+1	-	-1

The grammar can graphically be represented as in Fig. 4, where dotted rectangles refer to the 2 embedded distinct clusters: γ_1 , composed by symbols $\{\text{body}, \text{stm_list}, \text{stm}, \text{comp_stm}\}$, and γ_2 , composed by symbols $\{\text{exp}, \text{term}, \text{factor}\}$:

Fig. 4. Graphic representation of G_3

Cluster γ_2 contains 1 cycle ($\text{exp} \Rightarrow \text{term} \Rightarrow \text{factor} \Rightarrow \text{exp}$), while cluster γ_1 contains 3 cycles:

- a) $\text{body} \Rightarrow \text{stm_list} \Rightarrow \text{stm} \Rightarrow \text{body}$
- b) $\text{body} \Rightarrow \text{stm_list} \Rightarrow \text{stm} \Rightarrow \text{comp_stm} \Rightarrow \text{stm} \Rightarrow \text{body}$
- c) $\text{body} \Rightarrow \text{stm_list} \Rightarrow \text{stm} \Rightarrow \text{comp_stm} \Rightarrow \text{body}$

Applying the logic as previously introduced for the definition of the advanced constraints where multiple clusters and/or cycles exist, for clusters γ_1 and γ_2 we obtain the following corresponding conditions, respectively:

$$\begin{aligned}
 &\lambda_{\text{body}} w_2 \delta_{2,\text{stm_list}} \cdot \lambda_{\text{stm_list}} (w_3 \delta_{3,\text{stm}} + w_4 \delta_{4,\text{stm}}) \\
 &\quad \cdot \lambda_{\text{stm}} w_5 \delta_{5,\text{body}} + \lambda_{\text{body}} w_2 \delta_{2,\text{stm_list}} \\
 &\quad \cdot \lambda_{\text{stm_list}} (w_3 \delta_{3,\text{stm}} + w_4 \delta_{4,\text{stm}}) \\
 &\quad \cdot \lambda_{\text{stm}} w_7 \delta_{7,\text{comp_stm}} \cdot \lambda_{\text{comp_stm}} w_{10} \delta_{10,\text{stm}} \\
 &\quad \cdot \lambda_{\text{stm}} w_5 \delta_{5,\text{body}} + \lambda_{\text{body}} w_2 \delta_{2,\text{stm_list}} \\
 &\quad \cdot \lambda_{\text{stm_list}} (w_3 \delta_{3,\text{stm}} + w_4 \delta_{4,\text{stm}}) \\
 &\quad \cdot \lambda_{\text{stm}} w_7 \delta_{7,\text{comp_stm}} \cdot \lambda_{\text{comp_stm}} w_{11} \delta_{11,\text{body}} \\
 &< 1;
 \end{aligned}$$

$$\begin{aligned}
 &\lambda_{\text{exp}} (w_{14} \delta_{14,\text{term}} + w_{15} \delta_{15,\text{term}}) \cdot \lambda_{\text{term}} (w_{16} \delta_{16,\text{factor}} \\
 &\quad + w_{17} \delta_{17,\text{factor}}) \cdot \lambda_{\text{factor}} (w_{19} \delta_{19,\text{exp}}) \\
 &< 1.
 \end{aligned}$$

Then, using the values of Δ from Table VI, substituting the rallies with corresponding expressions and, moreover, taking advantages of basics constraints 1), advanced constraints for γ_1 and γ_2 can eventually be, respectively, rewritten as:

$$[w_5 + w_7 \cdot (2w_5 w_{10} + w_{11})] / w_3 < 1;$$

$$w_{19} / (w_{14} w_{16}) < 1.$$

To identify valid solutions, other than empirically (not so easy when we have complex advanced constraints), it is always possible to utilize a byproduct of the Flat with Penalties approach: indeed, at the end of generation cycles, relative percentages of usage of each rule expanding the same symbol always represent safe values to be used as balanced weights.

Anyhow, starting from a valid set of balanced weights, manual changes can be made, for instance to drive the sen-

⁵Additional rules could be added, e.g. for further statements as well as operators (like "+", "/", ">" etc), but have been omitted to keep the example simple but representative to show how the Balanced works

tence generation to increase or to reduce the nesting level caused by recursive rules: in any case, after a manual modification, the new set needs to be re-validated to assess if all the basics as well as the advanced constraints are still satisfied.

A valid solution is represented by the following set of weights, simply obtained by running the Flat with Penalties approach with parameter $m=0.01$ over G_3 : $w_3=0.65$, $w_4=0.35$, $w_5=0.29$, $w_6=0.42$, $w_7=0.29$, $w_{10}=0.29$, $w_{11}=0.71$, $w_{12}=0.43$, $w_{13}=0.57$, $w_{14}=0.65$, $w_{15}=0.35$, $w_{16}=0.66$, $w_{17}=0.34$, $w_{18}=0.72$, $w_{19}=0.28$.

The above weights satisfy all the conditions being the expressions on the left side representing advanced constraints for clusters γ_1 and γ_2 evaluate, respectively, to 0.84 and 0.65, both less than 1 (with corresponding convergence index of 0.16 and 0.35): this means that they are balanced and we can safely run an automatic pure weight-based test case generator to always obtain finite sentences.

Below, a real output of a pure weight-based test case generator applied to G_3 with the mentioned balanced weights, where we can see its power in generating extensively random (but always finite) sentences; moreover, this example also shows how with the Balanced approach *all* the several rules of a grammar can be used within a *single* sentence.

```
begin
  begin
    if ((id))*id<id)
    then if (id<id)
    then id:=(id)*id+id
    else id:=id*(id*id+id)
    else begin
      begin
        id:=id*(((id)+(id))))*(id)*id*id;
        while (((id+(id*id)*id*id*(id))))*id do
          begin
            begin
              begin
                id:=id*id+id;
                if (id)
                then id:=id
                else id:=(id+id+(id*id+id*id))
              end
            end;
            while (id<id) do
              begin
                id:=id+id+id+id+id*id;
                id:=(id*(id*((id))))*id*((id)*(id)+id)*id+id;
                id:=id*(id*(id))*id*(id);
                while (id<id*id) do
                  begin
                    if (((id+id*id+id*id+id))<id)
                    then begin
                      id:=id
                    end
                    else if (id<id)
                    then id:=id+id+id+id
                    else id:=id*id*id+id+id+(id)*id*(id+id)*id*id
                  end
                end;
                id:=id;
                id:=(id+id*id+id*((id+id+(id))+id)+id+id)
              end
            end
          end
        end;
        id:=id
      end
    end.
  end.
```

7 CONCLUSIONS

A demonstration about effective applicability of an automatic test case generator for context-free grammar where the unique criterion to select a rule is based on weights has been provided. The weight-based approaches differentiate against well-known methods as they easily generate sentences in a very random way while not requiring internal data structures as well as complex logic to run.

Convergence of the generation cycles for weight-based methods can be safely ensured by using a penalty mechanism or using balanced weights: how to identify constraints defining balanced weights depending on the typical recursive patterns of a grammar has been presented.

An example of generated sentences produced by using the Balanced method has been provided for a context-free grammar representing a reduced portion for a theoretical programming language; the example provides a clear understanding on how the method is capable to generate large but finite full random sentences, while ensuring coverage and convergence: this feature represents a step forward in automatic test case generation as, with a very simple generation logic, it offers a greater chance to extensively exercise the internal data structures of system being tested compared with existing well-known methods.

Possible further investigation is the development of an automatic engine to define the advanced constraints in order to eliminate the manual analysis of the grammar structure.

Moreover, although we have seen how the Flat with Penalties approach can be exploited to identify a valid set of balanced weights, another interesting area of investigation could be represented by the usage of neural network models, in order to iterate over the defined constraints to eventually obtain, starting from flat weights, valid balanced solutions (weights satisfying all the constraints): in this respect, we notice an interesting isomorphism between the elements of a neural network, the input, the parameters, the activation function and the loss function with delta, the weights, the constraints and the convergence index.

ACKNOWLEDGMENT

The author wishes to thank his family for the patience during the writing and his cats, which have always been sitting nearby throughout the period.

REFERENCES

- [1] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, pp. 366-375, 1972
- [2] B.A. Malloy and J.F. Power, "An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases", *1st Annual International Conference on Computer and Information Science*, Orlando, FL, 2001
- [3] A.M. Paracha and F. Franek, "Testing grammars for top-down parsers", *Innovations and Advances in Computer Sciences and Engineering*,

Springer Netherlands, pp. 451-456, 2010

- [4] M.H. Esterhuizen, “Test Case Generation for Context Free Grammars”, PhD thesis, Faculty of Science, Stellenbosch University, Mar. 2018
- [5] N. Chomsky, “Three models for the description of language”, *IRE Transaction on Information Theory*, Vol. 2, pp. 113-124, 1956
- [6] D. Baturin, “BNFGen”, <https://baturin.org/tools/bnfgen>

Franco Bazzichi was born in Forte dei Marmi (Lucca), Italy, on March 24, 1955. He received the degree in Computational Science, University of Pisa, Pisa (Italy), 1980.

From 1980 to 1981, consultant for Olivetti and Italtel-SIT; from 1982 to 1984 software developer at Italtel-SIT, from 1984 to 1997 IT training center coordinator and software engineer at Ciba-Geigy, Italy, from 1997 to 2020 he worked for Novartis Farma SpA, Italy, in different areas of Information Technologies, finishing as Country Service Delivery Manager for IT Service Desk and IT Client Platform. During the time he has always kept great interest in software engineering and, recently, in Artificial Intelligence.

Dr. Bazzichi is author, with I. Spadafora, of the paper “An Automatic Generator for Compiler Testing”, *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Volume: SE-8, Issue: 4, July 1982.