# Automatic Grammar-Based Test Generation

Hai-Feng Guo[1] and Zongyan Qiu[2]

[1] Department of Computer Science, University of Nebraska at Omaha, USA
haifengguo@unomaha.edu
[2] Department of Informatics, Peking University, Beijing 100871, P.R. China
qzy@math.pku.edu.cn

**Abstract.** In this paper, we present an automatic grammar-based test generation approach which takes a symbolic grammar as input, requires zero control input from users, and produces well-distributed test cases. Our approach utilizes a novel dynamic stochastic model where each variable is associated with a tuple of probability distributions, which are dynamically adjusted along the derivation. The adjustment is based on a tabling strategy to keep track of the recursion of each grammar variable. We further present a test generation coverage tree illustrating the distribution of generated test cases and their detailed derivations, more importantly, it provides various implicit balance control mechanisms. We implemented this approach in a Java-based system, named *Gena*. Experimental results demonstrate the effectiveness of our test generation approach and show the balanced distribution of generated test cases over grammatical structures.

## 1 Introduction

Grammar-based test generation is especially useful on testing applications which require structured data as inputs, such as data conversion tools and compilers [7,3], and those which response to well-ordered external events, such as reactive systems [18], VLSI circuit simulator [14], and software product line [1]. One common setting of these applications is using a context-free grammar (CFG) to describe the input structures for the systems. However, even though grammar-based test generation has been introduced since early 1970s [5] and has played important roles in software development and testing [9,3], it is well known that without extra control mechanisms, naive grammar-based test generation has never become practical due to the facts that exhaustive test production is often explosive and its testing coverage is often quite unbalanced.

Prior work on grammar-based test generation mainly fall into the following two approaches: *stochastic* or *annotating*. The stochastic approach [13,15,19] randomly select production rules for derivation based on their pre-assigned probabilities. Practically, a test case may easily blow out – becoming infinitely long – even if it is suggested that the probabilities of non-recursive production rules should be much higher than those of recursive rules [13] to avoid an infinite recursion [15]. Hence, other constraints (e.g., length control), heuristics, or hints are often required to make sure the termination of generating test cases. The *lava* tool [19] takes a production grammar as well as a seed, which consists of a high-level description that guides the production process, to generate effective test suites for Java virtual machine.

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 17–32, 2013.

The annotation approaches [10,11,7] become much popular recently. *Geno* [11], a C#-based test data generator, takes a hybrid between EBNF and algebraic signatures, where combinatorial control parameters are specified, to approximately achieve expected full combinatorial coverage. *YouGen* [7] supports many extra-grammatical annotations which guide effective test generation, and uses a generalized tag form of pairwise testing [20]. However, embedding tags into a grammar to control its production is not only a burden on users, but may be still difficult to get expected test cases.

In this paper, we present an automatic grammar-based test generation approach which takes a symbolic CFG [12] as an input, requires zero annotation, and produces well-distributed test cases for testing. Symbolic terminals are adopted to hide the complexity of different terminal inputs which share syntactic as well as expected testing behavior similarities. Our approach utilizes a novel dynamic stochastic model where each variable is associated with a tuple of probability distributions, which are dynamically adjusted along with the derivation. The more a production rule has contributed to self-loops while generating a test case, the *significantly* less probability the same rule will be applied in the future. To achieve the dynamic adjustment, we use a tabling strategy [4] to keep track of re-occurrences of grammar variables. The tuple associated with a grammar variable records the degrees of recursion caused by each of its rules. These tuples eventually determine the probability distribution of selecting a next rule for further derivation. We further use a test generation coverage tree, where each path from the root to a leaf node corresponds to a generated test case. Not only does the tree show the distribution of test cases and how each of them has been generated, but it also contains implicit balance control mechanism based on local probability distribution on each node.

We implemented the proposed test generation algorithm in a Java-based tool *Gena*, which takes a symbolic grammar and a number, the total number of test cases to request, as inputs, and automatically produces a set of test cases with test requirements and a test generation coverage tree. Experimental results demonstrate the effectiveness of our test generation approach, and indicate the balanced distribution of generated test cases over grammatical structures.

The rest of the paper is organized as follows. Section 2 introduces the grammar-based test generation. Section 3 presents our dynamic stochastic approach as well as a tabling strategy to keep track of recursion. Section 3.2 illustrate how a dynamically growing test generation coverage tree maintains its local probability tuples on each node to balance test generation distribution, followed by detailed algorithms and termination properties. Section 4 and Section 5 present a Java-based implementation and our experimental results, respectively. Conclusions and discussions are given in Section 6.

## 2 Grammar-Based Test Generation

A CFG is represented as a four-tuple $G = (V, T, S, P)$, where $V$ is a set of variables (or non-terminals), $T$ is a set of terminals, $S$ is the start variable, and $P$ is a set of production rules in the form of $A ::= x$, where $A \in V$ and $x \in (V \cup T)^*$. Given a CFG $G$, automatic test generation is typically done by simulating the leftmost derivation from its start variable $S$.

### 2.1 Symbolic Terminal

We first introduce a notation of symbolic terminals [12], which are adopted to hide the complexity of different terminal inputs which share syntactic similarities as well as similar expected testing behaviors. A symbolic terminal, highlighted by a pair of square brackets, is an abstract notation for a finite domain, which is represented as an ordered sequence of individual atoms or a bound form $Lower..Upper$, where $Lower$ is smaller than or equal to $Upper$ in their lexicographic order. We treat a symbolic terminal as a regular terminal except that it returns a random element within the defined domain whenever a symbolic terminal is seen during derivation.

*Example 1.* Consider the following CFG:

$$E ::= [N] \mid E + E \mid E - E \qquad [N] ::= 1..100$$

where $[N]$ is a symbolic terminal. An example of test generation based on leftmost derivation would be as follows:

$$E \Rightarrow E+E \Rightarrow E-E+E \Rightarrow [N]-E+E \Rightarrow [N]-[N]+E \Rightarrow [N]-[N]+[N].$$

### 2.2 A Penalty Maze

In fact, generating a terminal string acts like getting out a penalty maze, which is not only a confusing intricate network of passages, but also a network with many self-loops; and those self-loop passages, once taken, would *magically* make the maze bigger and harder to find an exit.

Consider Example 1 again, where the variable $E$ has two double-recursive rules. As shown in Table 1, the start variable $E$ is similar to the beginning of a maze; the leftmost derivation, like navigating a penalty maze, is actually a procedure finding each variable in the current derived string a terminal, like a segmented exit in a penalty maze. The more variables in a current derived string, the more challenges – each variable needs to become terminal – for a leftmost derivation to generate a terminal string.

| Derived String | Probability of becoming terminal |
|---|---|
| $E$ | $1/3$ |
| $\Rightarrow E + E$ | $1/3 * 1/3 = 1/9$ |
| $\Rightarrow E - E + E$ | $1/3 * 1/3 * 1/3 = 1/27$ |
| $\cdots$ | $\cdots$ |

**Fig. 1.** A penalty maze

Figure 1 shows how fast the probability, for a derived string to become terminal instantly, could drop as a leftmost derivation moves on. Each occurrence of $E$ has a probability, $1/3$, to become a terminal instantly since $E$ has only one terminal rule out of three rules. However, as the derivation moves on, the number of $E$'s expands much faster than they become terminal. It is in nearly two-third probability that a naive grammar-based test generation will blow out generating a single terminal string of $E$ due to non-termination.

*Example 2.* Consider a grammar for a subset of arithmetic expressions as follows:

$$E ::= F \mid E + F \mid E - F \qquad F ::= T \mid F * T \mid F/T$$
$$T ::= [N] \mid (E) \qquad [N] ::= 1..1000$$

The grammar has only one terminal exit, $E \Rightarrow F \Rightarrow T \Rightarrow [N]$, but the rest are full of recursive rules. Moreover, direct recursions (e.g, $E \Rightarrow E + F$) are even entangled with indirect one (e.g., $E \Rightarrow T \Rightarrow (E)$), which makes static analysis difficult. For example, without runtime information, it is difficult to tell whether the production rule $E ::= F$ is going to be recursive or not due to the possibility of indirect recursion.

It is a common phenomenon getting into infinite recursion during grammar-based test generation due to the recursive natures of a CFG. Hence, it is really challenge getting out of a penalty maze, so is generating a terminal string.

## 3 A Dynamic Stochastic Approach

The essential problem in grammar-based test generation is how to generate a terminal string without getting lost in a "penalty maze". Our approach utilizes a novel dynamic stochastic model where each variable is associated with a tuple of probability distributions, which are dynamically adjusted along the derivation. When a variable is encountered during left-most derivation, it applies one of its defined production rules stochastically, based on the tuple of probability distribution among those rules. Then the key problem is how to dynamically adjust the probability distribution so that a test generator is able to avoid keeping getting into loops with potential explosive "penalties". The general principles, for a dynamic stochastic model to satisfy, are:

- initially, the probability distribution allows a variable to have an equal chance to apply different defined production rule;
- as a derivation moves on, since the probability of generating a terminal string could become low in a dramatic speed (see the example in Figure 1), it has to be effectively fast pushing derivations to applying non-recursive rules;
- the generated terminal tests are evenly distributed over the grammatical structures of the given CFG; in other words, every terminal test may have a good chance to be generated as long as the total requested number of test cases is sufficiently big.

### 3.1 A Tabling Strategy

We present a tabling strategy to detect derivation loops and eventually achieve dynamic probability distribution. Tabling has been extensively used in logic programming [21,4], where it successfully resolves lots of termination issues by detecting re-occurrences of recursively defined predicates at runtime in an automatic way.

In grammar-based test generation, we introduce a global data structure **table**, where each grammar variable has a tuple entry, initially all 1's, recording the degrees of recursion caused by each of its defined production rules at runtime, and the size of a tuple is determined by the number of its defined production rules.

**Definition 1.** *Given a CFG $G = (V, T, S, P)$, where $E$ is a variable in $V$ and $\mathcal{R}$ is a production rule of $E$, we say that $\mathcal{R}$* **causes a recursion** *of $E$, if there exists a leftmost derivation of $E$ in a form of*

$$E \stackrel{\mathcal{R}}{\Longrightarrow} \omega \stackrel{*}{\Longrightarrow} \alpha E \beta,$$

*where* $\stackrel{\mathcal{R}}{\Longrightarrow}$ *is a single derivation applying the rule* $R$*,* $\omega \in (V \cup T)^*$*,* $\alpha \in T^*$*,* $\beta \in (V \cup T)^*$*, and there is no other leftmost occurrence of a variable* $E$ *during* $\omega \stackrel{*}{\Longrightarrow} \alpha E \beta$*.*

Such a caused recursion is dynamically detected by maintaining a derivation stack that tracks whether a variable leads to a self-loop during its leftmost derivation. Once a recursion of $E$, caused by a defined rule $\mathcal{R}$, is detected, the degree tuple of $E$ will be adjusted by *doubling* the degree of recursion associated with the contributing grammar rule $\mathcal{R}$. We say such an adjustment a **double strategy**.

The main purpose of maintaining a table of degree tuples is determining a dynamic probability distribution for selecting a next production rule. Given a degree tuple $(d_1, d_2, \cdots, d_n)$, where $n \geq 1$ is the number of rules for a variable, its corresponding probability distribution $(p_1, p_2, \cdots, p_n)$ is determined as follows:

$$p_i = \frac{w_i}{T}, \text{ where } w_i = \frac{d_1}{d_i} \text{ and } T = \sum_{i=1}^{n} w_i.$$

| vars | deg. tuple | probabilities |
|---|---|---|
| $E$ | $(1,\ 1,\ 1)$ | $(.33,\ .33,\ .33)$ |
| $F$ | $(1,\ 1,\ 1)$ | $(.33,\ .33,\ .33)$ |
| $T$ | $(1,\ 1)$ | $(.5,\ .5)$ |

**Fig. 2.** Initial degree/prob

We introduce a probability weight $w_i$, which is a ratio showing the relative degrees of the first rule over the $i$-th production rule. Hence, the probability weight $w_1$ is always 1, while other weight $w_i$, $i > 1$, may drop below 1 if the $i$-th rule causes more recursions than the first rule does; otherwise, $w_i \geq 1$. Thus, the initial table for the grammar in Example 2 is given in Figure 2.

Dynamic probability distribution is shown in Figure 3 as the double strategy pushes the derivation to choose the exit rule of $E$ at runtime. Note that when the leftmost variable $T$ is derived to $(E)$, the *tabling strategy* will detect this indirect recursion caused by the rule $E ::= F$, and then apply the double strategy to adjust its degree of recursion from 1 to 2; at this point, the degree tuple of $T$ remains $(1, 1)$. However, as the derivation continues, only when another $T$ is seen as a leftmost variable, the degree of tuple of $T$ will then be updated to $(1, 2)$ due to its indirect recursion, leaning the derivation of $T$ toward its exit rule.

| derivation | tuple of $E$ | probabilities |
|---|---|---|
| $E$ | $(1,\ 1,\ 1)$ | $(.33,\ .33,\ .33)$ |
| $\Rightarrow E + F$ | $(1,\ 2,\ 1)$ | $(.4,\ .2,\ .4)$ |
| $\Rightarrow E - F + F$ | $(1,\ 2,\ 2)$ | $(.5,\ .25,\ .25)$ |
| $\Rightarrow E + F - F + F$ | $(1,\ 4,\ 2)$ | $(.57,\ .14,\ .29)$ |
| $\Rightarrow T + F - F + F$ | $(1,\ 4,\ 2)$ | $(.57,\ .14,\ .29)$ |
| $\Rightarrow (E) + F - F + F$ | $(2,\ 4,\ 2)$ | $(.4,\ .2,\ .4)$ |
| $\Rightarrow (E + F) + F - F + F$ | $(2,\ 8,\ 2)$ | $(.44,\ .11,\ .44)$ |
| $\cdots$ | $\cdots$ | $\cdots$ |

**Fig. 3.** Dynamic Probability Distribution

The more a production rule has contributed to self-loops while generating a test case, the *significantly* less probability the same rule will be selected in the future due to the double strategy. Note that our tabling strategy detects a self-loop by actually seeing a same variable during its own derivation, instead of by selecting a potential recursive rule; therefore, a non-recursive rule, no matter how many times it has been applied, its corresponding degree of recursion will not be doubled. Our tabling strategy, incorporated with the double adjusting strategy, provides an effective approach to solving the "penalty maze" challenges.

### 3.2 A Coverage Tree

Our approach ensures that the test case generator will generate a terminal string. Once a terminal string has been successfully generated, all the degree tuples in the global table will be reset to 1's for next test case generation. The next problem is that how to generate test cases in a balanced coverage distribution on given CFG structures.

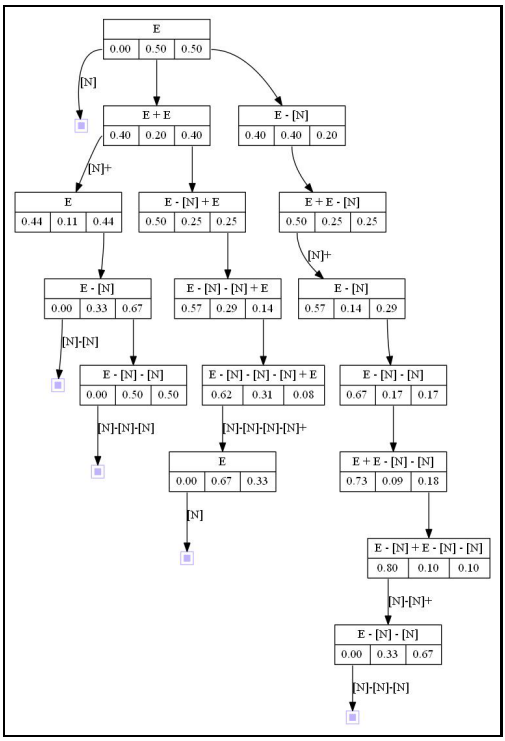


**Fig. 4.** A Coverage Tree on Example 1

We present a test generation coverage tree, a *coverage tree* in short, to show the distribution of generated test cases and how each of them are generated. Figure 4 shows a coverage tree, generating five different test cases based on the CFG in Example 1. Each node in a coverage tree contains an intermediate derived string, starting from a leftmost variable, and a *local probability tuple* for the leftmost variable. Each label along the transition from a parent node to a child node, if any, shows a leftmost terminal substring derived from the parent node, where the rest derived substring, beginning with a leftmost variable, is shown in the child node. Thus, each path from the root to a leaf node corresponds to a complete leftmost derivation generating a test case, where each transition corresponds to a derivation step, and a leaf node, represented by a little gray box, denotes the completion of a leftmost derivation. A coverage tree always starts with a root node containing the start variable of a CFG.

Not only does a coverage tree show the distribution of generated test cases and each detailed derivation step, but it also contains implicit balance control mechanism based on the local probability tuple on each tree node. Consider the coverage tree in Figure 4. When a new node is created with an intermediate string whose leftmost variable is $E$, its local probability tuple is calculated based on the *current* degree tuple of $E$ stored in the global table. For example, when the root node $E$ in Figure 4 was initially created, its local probability tuple is $(0.33, 0.33, 0.33)$, which tells that at this point, each of the three possible derivation branches has equal probability. The test generator will take one branch stochastically based on the local probability, to continue a test generation.

Once a derivation branch has been fully explored for test generation, its local probability tuple will be adjusted dynamically for future test generation. See the present status of the root node $E$ in Figure 4, its local probability tuple has been updated to

$(0.00, 0.50, 0, 50)$; that is because the first branch, corresponding to the rule $E ::= [N]$, has been fully explored. Therefore, the probability of the first branch will be set to 0, and the remaining probabilities on the same node will be adjusting accordingly.

In Figure 4, even though there is only a single variable $E$, and all the local probability tuples are used to direct the derivation of $E$'s, probability tuples in different nodes are quite different. That is mainly because each local probability tuple is like a snapshot of the degree tuple of $E$ in the global table when the hosting node is created, while the degree tuple dynamically changes during a test generation.

*Example 3.* Consider the coverage tree with 10 generated test cases in Figure 5, given the following symbolic grammar:

$$E ::= [N] \mid F - F \mid E + F \qquad F ::= [N] * [N] \mid [N] \qquad [N] ::= 1 \,..\, 100$$

Example 3 gives a better idea how complete branches will shift probabilities to incomplete branches, thus pushing future test generation to other unexplored parts. As a result, our test generation algorithm, based on a dynamic stochastic approach as well as a test generation coverage tree for implicit balance control, guarantees that every generated test case is *structurally different* as long as the given grammar is unambiguous. The coverage tree expands as more test cases are generated.

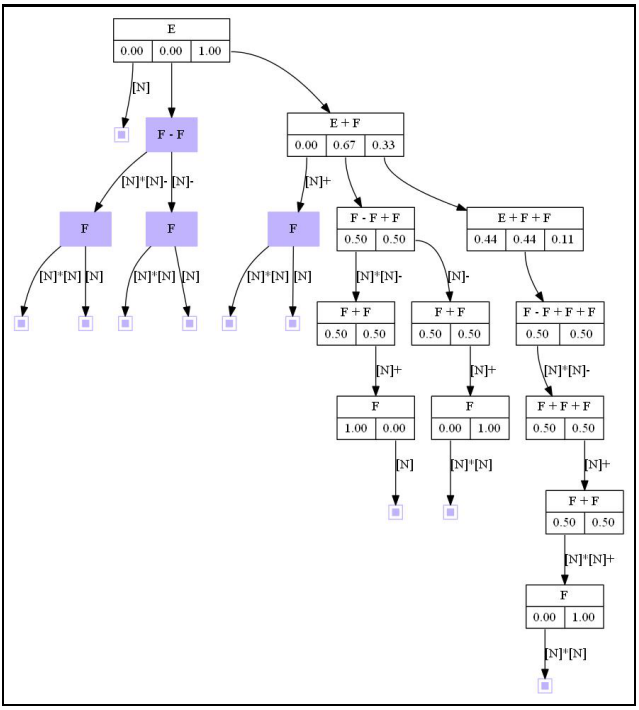


**Fig. 5.** A Coverage Tree on Example 3

Note that the starting position of each transition tells which production rule to apply. If a transition starts from a second probability number, it is implied applying the second corresponding production rule for derivation. We use a gray box to denote a *complete node*, all of whose branches have been completely explored.

### 3.3 Algorithms

This subsection presents a complete algorithm for our dynamic stochastic model, including the tabling strategy and a coverage tree construction. To support the tabling strategy, we use a **derivation stack**, which dynamically maintains an ancestor relation among the grammar variables to detect self-loops during derivation. Next, we outline the basic supported functionalities for a derivation stack, a global table supporting the tabling strategy, and a coverage tree node, respectively.

**Definition 2.** *Following the same convention as Definition 1, let $E_1$ and $E_2$ be two different occurrences of a variable $E$, we say $E_1$ is an* **ancestor** *of $E_2$ if there exists a leftmost derivation as follows:*

$$E_1 \overset{\mathcal{R}_i}{\Longrightarrow} \omega \overset{*}{\Longrightarrow} \alpha E_2 \beta,$$

*where $\mathcal{R}_i$, the $i$-th production rule of $E$, is the cause of recursion. If there is no other leftmost occurrence of a variable $E$ during $\omega \overset{*}{\Longrightarrow} \alpha E_2 \beta$, we say $E_1$ is* **the least ancestor** *of $E_2$.*

The derivation stack contains entries in a form of pair, $(E, i)$, which corresponds to a derivation step of a variable $E$ by applying its $i$-th production rule. Note that the index $i$ begins from 1. Following methods for the stack are provided:

- *void push(E, i)*: push a pair $(E, i)$ into the derivation stack;
- *void pop()*: pop out a pair $(E, i)$;
- *int leastAncestor(E)*: given a variable $E$, *leastAncestor(E)* returns an integer $i$; if $i \geq 1$, it means that there exists a pair entry $(E, i)$ in the stack, that is, a self-loop detected; otherwise, $i = 0$, which tells no occurrences of $E$ in the current stack.

The global table is used for storing and retrieving the dynamic degree tuples for each variable, which supports the following methods:

- *void reset()*: set for each variable a degree tuple with all 1's.
- *Degree-Tuple get(E)*: return the degree tuple of $E$ in the table;
- *void doubleDegree(E, i)*: double the $i$th-degree entry of $E$.

A coverage tree is gradually constructed during a test generation, and further expanded as more test cases are generated. A coverage tree node, containing a derived substring, which always starts with a variable if non-empty, and a probability tuple. The tree supports the following methods:

- *Node newChild(i, str)*: create the $i$-th child node, set *str* as the child's derived substring, and return the child node; if *str* is an empty string, the child node will be automatically a compete node.

- *Node childAt(i)*: return the $i$-th child node if already exists; otherwise, *null* is returned.
- *void setProbFromDTuple(t)*: transform a degree tuple *t* into a probability tuple, and then set the probability tuple into the node;
- *void setZeroProbability(i)*: set the $i$-th probability 0, and adjust the rest of probabilities accordingly to make sure that the sum of all probabilities equals to 1, except that all probabilities have been set 0, that is, the node has been fully explored.
- *bool isComplete()*: check whether the node has been fully explored or has an empty derived substring.
- *int chooseByProbability()*: return a random index of production rule based on the probability distribution in the node.

**Algorithm 1.** Generating A Test Case

```
1: Global: symbolic grammar G = (V, T, S, P), table TABLE , and derivation stack DS
2: Input: a dummy node root for the coverage tree
3: Output: a test case
4: function TESTGENERATION(root)
5:     DS ← an empty stack                          ▷ initialize the derivation stack
6:     TABLE.reset()                                ▷ initialize the global TABLE
7:     return DERIVATION(root, 1, "S", "")
8: end function
```

Algorithm 1 shows a main procedure, TESTGENERATION, on how to generate a new test case, given a dummy root node for a coverage tree. We give a dummy root node outside the test case generation procedure, so that it can use a same coverage tree for generating as many test cases as users expect. The procedure starts with creating an empty derivation stack and initializing the global TABLE for simulating a new leftmost derivation, which is implemented in a recursive function, named DERIVATION.

Algorithm 2 shows how a leftmost derivation has been implemented by applying a dynamic stochastic model. Assuming that a string, *str*, has been derived from a parent node, *pNode*, in the coverage tree by applying $i$-th production rule of the leftmost variable in *pNode*, the function DERIVATION takes the four-tuple as an input, prepares to create a child node holding *str*, and then continue the derivation from the child node recursively until *str* is empty (lines 5-7).

If the derived string, *str*, starts with a variable $E$ (line 8), we first checks whether there exists a least ancestor of $E$ in the derivation stack; if that is the case (lines 10-12), we apply the *double strategy* to increase its associated degree of recursion. Secondly, we create the $i$-th child node *cNode*, if not existing yet, to hold *str* and a corresponding probability tuple based on its latest degree tuple of $E$ (line 13-17). We then choose a production rule (e.g., a $j$-th rule $E ::= \alpha$) of $E$ randomly based on the local probability distribution of *cNode*, and push a pair $(E, j)$ into the derivation stack for future self-loop detection (line 18-19). To find out when the substring $\alpha$ will become completely terminal, we insert a special symbol $\updownarrow$ as an indicator right after the substring $\alpha$ in the derived string (lines 20-22). The pair $(E, j)$ will be popped only after the derived substring $\alpha$ has become completely terminal (lines 29-31), that is, when the indicator " $\updownarrow$ " becomes the leftmost symbol of $str$ (line 29). After a recursive call, in line 22,

**Algorithm 2.** DERIVATION: Core Algorithm

```
1: Global: symbolic grammar G = (V, T, S, P); table TABLE, and derivation stack DS
2: Input: a coverage tree parent node, pNode; the index of a next rule to apply, i; a derived
   substring, str; and a label on the transition from the parent node to a child node, label
3: Output: a partial or complete test case
4: function DERIVATION(pNode, i, str, label)
5:     if (str is an empty string) then                          ▷ end of a derivation
6:         pNode.newChild(i, "")                                 ▷ a complete node
7:         return label
8:     else if (str is in form of Eβ) then                       ▷ E ∈ V, β ∈ (V ∪ T)*
9:         int k ← DS.leastAncestor(E)
10:        if (k ≥ 1) then                                       ▷ self-loop detected
11:            TABLE.doubleDegree(E, k)                          ▷ Double Strategy
12:        end if
13:        Node cNode ← pNode.childAt(i)
14:        if (cNode is null) then                               ▷ expanding the tree
15:            cNode ← pNode.newChild(i, Eβ)
16:            cNode.setProbFromDTuple(TABLE.get(E))
17:        end if
18:        int j ← cNode.chooseByProbability()
19:        DS.push(E, j)                                         ▷ critical to track self-loop
20:        Let E ::= α be the j-th production rule of E
21:        str ← α + " ↕ " + β          ▷ special symbol ↕ is an indicator to pop (E, j)
22:        String rLabel ← DERIVATION(cNode, j, str, "")
23:        if (cNode.isComplete()) then
24:            pNode.setZeroProbability(j)
25:        end if
26:        return label + rLabel                                 ▷ + is concatenation
27:    else if (str is in form of aβ) then                       ▷ a ∈ T, β ∈ (V ∪ T)*
28:        return DERIVATION(pNode, i, β, label + "a")
29:    else if (str is in form of " ↕ "β) then                   ▷ β ∈ (V ∪ T)*, str leads with ↕
30:        DS.pop()                                              ▷ paired with push operations
31:        return DERIVATION(pNode, i, β, label)
32:    end if
33: end function
```

processing further derivation from *cNode*, lines 23-25 check whether a child node has been completely explored, if so, the information will be propagated to its parent node by adjusting its local probability distribution. A generated test case is the concatenation of all labels from the dummy root node to a leaf node where *str* becomes empty.

### 3.4 Termination

Our dynamic stochastic approach *almost surely* [8] guarantees the termination of a single test case generation, as long as a *proper* symbolic CFG is given, where a symbolic CFG is said to be *proper*, if it has

- no inaccessible variables: $\forall E \in V, \exists \alpha, \beta \in (V \cup \Sigma)^* : S \overset{*}{\Rightarrow} \alpha E \beta$;
- no unproductive variables: $\forall E \in V, \exists \omega \in \Sigma^* : E \overset{*}{\Rightarrow} \omega$.

Let $E$ be a variable in a given symbolic CFG, $R$ be a recursive rule of $E$, and $n$ be the number of times that $R$ has been applied to cause its own recursions. Assuming that $E$ has only two rules, $R$ and a non-recursive one, we have $(1, 1)$ as the initial degree tuple of $E$; and $(2^n, 1)$ will be degree of tuple after $n$ applications of $R$. Thus, the probability of choosing $R$ in the next derivation of $E$, $P(R, n) = \frac{1}{2^n}/(1 + \frac{1}{2^n})$, and we have

$$\lim_{n\to\infty} P(R, n) = \lim_{n\to\infty} \frac{\frac{1}{2^n}}{1 + \frac{1}{2^n}} = \lim_{n\to\infty} \frac{1}{2^n} = 0$$

If a variable $E$ contains more than two production rules, the probability drops even faster. On the other hand, the probability that a derivation will take terminal exits approximates to 1 infinitely as the derivation gets deeper and deeper. One say that an event happens **almost surely** if happens with probability one in probability theory [8].

## 4 Gena – A Java Implementation

In this section, we present *Gena*, a Java-based implementation of an automatic grammar-based test generator. Its system overview is illustrated in Figure 6.


**Fig. 6.** Gena – a system overview

Gena, requiring zero annotation from users, takes inputs a symbolic grammar and how many test cases to request, and produces well-distributed test cases as well as a coverage tree showing the distribution of test cases along with their detailed leftmost derivation. Our test generator utilizes a novel dynamic stochastic model and local control mechanisms to maintain balanced distribution of test cases; and the distribution is demonstrated in a coverage tree.

Note that the graphical representation of a coverage tree, as seen in Figure 4 and Figure 5, is automatically produced by *Gena* along with the test generation.

### 4.1 Implicit Control Mechanisms

Even though *Gena* requires zero control from users, it implicitly enables various useful control mechanisms through the implementation of tabling, double strategy, and coverage tree maintenance, while in Geno [11], those control mechanisms are defined by users. We introduce some typical control mechanisms as follows:

**Depth Control:** Due to our double strategy, Gena always puts high penalty on the rules causing recursion, which brings about a low probability distribution to the recursive rules. As a result, when generating multiple test cases, Gena tends to generate test

cases in an order, not strictly, from short to long ones; and in terms of the coverage tree, it will be explored from shallow towards deep as more test generations are requested.

**Recursion Control:** Our tabling strategy, utilizing a derivation stack, detects causes of recursion, and then applies the double strategy to put exponentially increased penalty on the recursive rules. Based on probability theory, it will almost surely push derivation to stay away from those recursive rules eventually.

**Balance Control:** Incorporated with the double strategy for recursion control and complete nodes detection for local probability adjustment, Gena does not only avoid non-terminating recursion, but also work along the derivation in an evenly distributed way. It is extremely important that if a grammar has multiple recursive rules, each recursive rule under a same variable is explored with similar probabilities. Also, due to complete nodes detection, Gena guarantees that every generated test case is **structurally different** in terms of grammar structures, as long as the grammar is unambiguous.

**Construction Control:** With such a coverage tree, it is easy to extend Gena with customized constraint controls. For example, users could specify the lengths of expected test cases, or specify the quantitative data constraints; both specifications can be easily supported by incorporating constraints as part of coverage nodes, so that the test generator will only explore those tree parts where constraints are satisfied.

### 4.2 Structural Test Case Requirements

A test case generated automatically often comes with a set of test requirements, so that when the test case is used in software testing, we know what features of a system have been tested. Test requirements are also critical to the areas of test cases minimization, selection and prioritization, serving as comparison criteria [22]. For example, in automated model-based test case generation where test cases are typically generated based on a data flow model, definition-use pairs of variables [6] in a program are popularly identified as test requirements serving as effective reduction criteria.

*Gena* generates a test case as well as an associated set of structured test requirements. It breaks each complete derivation path, from the root node to a leaf one in a coverage tree, into small basic components which represent structural patterns of the generated test case. For example, consider the Figure 4. The rightmost path in the coverage tree corresponds to a generated expression:

$$[N] + [N] - [N] + [N] - [N] - [N],$$

where each $[N]$ is automatically substituted with a random integer from its domain during the generation. Its associated set of test requirements, produced by *Gena*, is $\{E3E2E1, E3E2E3E1, E1\}$. Each test requirement is actually a structural input pattern which consists of a sequence of production rule indices, denoting a leftmost derivation sub-sequence starting from a leftmost variable $E$ until a terminal symbol is seen at the leftmost while deriving $E$.

For example, $E3E2E1$ represents a segment of derivations starting from the root, where the leftmost system is the variable $E$. The derivation moves on by first applying the third production rule of $E$, followed by applying a second rule of $E$ and a first rule of $E$ in order, until the leftmost symbol in the derived string becomes a terminal

**Table 1.** Among 1000 generated expressions

| Operators | Total Frequencies |
|---|---|
| $+$ | 2149 |
| $-$ | 2162 |
| $*$ | 4452 |
| $/$ | 4472 |
| $()$ | 1823 |
| $[N]$ : a random integer | 14235 |

**Table 2.** Among 2000 generated program code

| Keywords | Total Frequencies |
|---|---|
| $:=$ | 3353 |
| *print* | 3554 |
| *if-then* | 2116 |
| *if-then-else* | 1693 |
| *while* | 2099 |
| *repeat* | 1829 |

$[N]$ (shown in a label "$[N]+$"). The other two terms $E3E2E3E1$ and $E1$ are similar, representing the segments of derivations following that of $E3E2E1$. Figure 7, using a standard derivation tree for the generated expression, shows that each requirement actually corresponds to a lefty tree showing nested structural information.

We believe that each test case may contain some structural patterns which could be possibly linked to potential failures of software under test. These structural test requirements could serve as testing coverage measurement criteria for test case minimization, selection and prioritization, especially in data-intensive or data-critical applications.

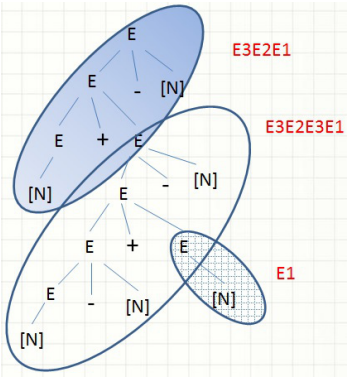


**Fig. 7.** Lefty Subtrees

## 5 Experimental Results

We have carried out some experiments to measure how balanced the test cases generated by *Gena* over grammar structures, and tried some related applications.

### 5.1 Balance Results

Table 1 shows a statistic report among the first 1000 generated arithmetic expressions by *Gena*, given a symbolic grammar in Example 2. By comparing the total frequencies among operators, we measure how balanced in overall among test case generation. (1) The total frequencies of the operators $+$ and $-$ are close, which indicates the balanced distribution between two recursive rules under the same variable $E$; similar reasons apply on the frequency observation between $*$ and $/$. (2) The total frequency of $+$, an operator at the production rule of $E$, is about half lower than the total frequency of $*$, an operator at the level of $F$. This observation makes sense because every single $+$ and $-$ involved in the derivation introduces an extra $F$ variable into a derived string, and every $F$ may possibly derive multiple $*$ or $/$ operators. Similar reasons for the observation that the frequency sum of $()$ and $[N]$ is almost double than the sum of $*$ and $/$. (3) The total frequency of $()$ is much lower than $[N]$ due to the reasons that the indirect recursion caused by $T ::= (E)$; and the majority of $T$ derivation takes the terminal rule $T ::= [N]$ due to unbalanced probability distribution caused by our double strategy.

We further tested *Gena* on a symbolic grammar for a *pascal*-like program code. The grammar, containing 34 production rules and 13 variables, is partially shown below:

$$
\begin{aligned}
&P ::= K. \qquad K ::= \text{ begin } C \text{ end} \qquad C ::= S; C \mid S \\
&S ::= I := E \mid \text{if } (B) \text{ then } S \mid \text{if } (B) \text{ then } S \text{ else } S \mid \\
&\qquad K \mid \text{while } (B) \text{ do } S \mid \text{repeat } C \text{ until } (B) \mid \text{print } E \\
&\quad \ldots
\end{aligned}
$$

Table 2 shows a statistic report among the first 2000 generated programs by *Gena*. Here we compare the total frequencies of keywords to measure how balanced in overall among generated cases in a more complicated grammar setting, where all those listed keywords are distributed in the production rules of $S$. (1) The observation, that the frequencies of keywords := and $print$ are similar, justifies the even probability distribution among terminal rules if multiple ones exist. (2) The frequencies of *if-then* and $while$ are less than that of $print$ because a single recursion of $S$ is involved in both statements. (3) The frequency of *if-then-else* is lower than that of *if-then* because *if-then-else* contains a double recursion, which results in double penalties in the dynamic probability distribution. (4) Interestingly, the frequency of $repeat$ is between those of *if-then* and *if-then-else* because the derivation of $C$ in the $repeat$ statement may result in a single recursion (if $C ::= S$ is applied), or a double recursion (if $C ::= S; C$ is applied).

### 5.2 A Grading System

We have implemented an automatic grading system for Java programs, based on our test case generator, *Gena*. Consider a Java programming assignment which takes an infix arithmetic expression as an input string, performs stack operations to convert the input into a postfix expression, and finally returns a number by calculating the postfix expression. We use a correct program to calculate the expected results for each generated expression, and compare the result with the one returned from each submission.

Table 3 shows grading results on 14 Java program submissions, by running 1000 different arithmetic expressions generated by *Gena*. The second column shows the ratios

**Table 3.** Grading Results

| | ratio | failure-inducing patterns | Possible Causes |
|---|---|---|---|
| 1 | 14% | {-+, //, /*, --, */} | right-associativity |
| 2 | 78% | {()} | parenthesis not properly handled |
| 3 | 100% | {} | |
| 4 | 2% | {+, -, /, *, ()} | not working at all |
| 5 | 9% | {*/, /-, *-, *+, /*, //, /+, -+, --} | right-associativity; operator precedence ignorance |
| 6 | 6% | {*/, /-, *-, *+, /*, //, /+, +/, --, -/} | right-associativity; operator precedence ignorance |
| 7 | 53% | {*/} | $[N] * [N]/[N]$ |
| 8 | 100% | {} | |
| 9 | 68% | {-*-, -*+, -/-, -/+} | partial operator precedence ignorance |
| 10 | 100% | {} | |
| 11 | 14% | {*/, //, -+, --, /*} | right-associativity |
| 12 | 54% | {/-, *-, *+, /+} | operator precedence ignorance |
| 13 | 4% | {/, *, -, +} | operators not supported |
| 14 | 10% | {*/, /*, //, -+, (), --} | right-associativity and parenthesis problem |

of correctness for each submission. For example, the first submission performs correctly on $14\%$ of the $1000$ test cases. The third column demonstrates common failure-inducing patterns for each submission; and the last column gives typical causes on processing arithmetic expressions.

The *Gena*-based grading system frees users from constructing test cases manually and worrying about their coverage. The grading system is able to find out significant number of failing test cases on those Java programs, which lays important foundation for further program fault localization. The failure-inducing patterns, extracted from those test requirements associated with failing test cases, contain clear clues for understanding root causes of software testing failure. Due to space limitation, we will have to leave details of our *Gena*-based fault localization in our future work.

## 6 Conclusions

We presented an automatic grammar-based test generation algorithm requiring zero control inputs. Our algorithm utilizes a novel dynamic stochastic model where each grammar variable is associated with a probability tuple, which is dynamically adjusted via a tabling strategy. Our dynamic stochastic approach almost surely guarantees the termination of test case generation, as long as a *proper* CFG is given. We further presented a test generation coverage tree which does not only illustrate how each test case is generated and their distribution, more importantly, it provides various implicit control mechanisms for maintaining balanced coverage. Our grammar-based test generation guarantees that every generated test case is *structurally different*.

We have presented, *Gena*, a Java-based system based on our grammar-based test generation algorithm. *Gena*, requiring zero annotation from users, takes input a symbolic grammar and how many test cases to request, and produces well-distributed test cases as well as a graphical test case coverage tree showing the distribution of test cases along with their respective derivations. Experimental results justify the effectiveness of our test generation approach and show the balanced distribution of test cases.

Grammar-based test generation can be thought of a branch of model-based test generation since a grammar actually describes the input data model. Model-based testing [2,16,17] has been extensively researched and become increasingly popular in practical software testing. Model-based test generation typically derives test cases from a given system model (e.g. UML chart, or a control flow graph), representing the desired system behaviors. The major issues involved in grammar-based test generations, such as depth control, recursion control, and balance control, etc., occur as well in model-based test generation when dealing with a complicated control flow graph with loops and branches structures. Hence, our test generation algorithm based on a dynamic stochastic model can also be valuable in implementing other model-based test generations.

## References

1. Bagheri, E., Ensan, F., Gasevic, D.: Grammar-based test generation for software product line feature models. In: Proceedings of the 2012 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2012). IBM (2012)

2. Belli, F., Endo, A.T., Linschulte, M., da Silva Simo, A.: Model-based testing of web service compositions. In: IEEE 6th International Symposium on Service Oriented System Engineering, pp. 181–192 (2011)
3. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. ACM Sigplan Notices 43(6), 206–215 (2008)
4. Guo, H.F., Gupta, G.: Simplifying dynamic programming via mode-directed tabling. Software Practice & Experience 38(1), 75–94 (2008)
5. Hanford, K.: Automatic generation of test cases. IBM Systems Journal 9(4), 242–257 (1970)
6. Harrold, M.J., Koltit, P.C.: A compiler-based data flow testing system. In: Pacific Northwest Quality Assurance, pp. 311–323 (1992)
7. Hoffman, D.M., Ly-Gagnon, D., Strooper, P., Wang, H.Y.: Grammar-based test generation with yougen. Software Practice and Experience 41(4), 427–447 (2011)
8. Jacod, J., Protter, P.: Probability Essentials, p. 37. Springer (2003)
9. Kosindrdecha, N., Daengdej, J.: A test case generation process and technique. Journal of Software Engineering 4(4), 265–287 (2010)
10. Lämmel, R.: Grammar testing. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 201–216. Springer, Heidelberg (2001)
11. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 19–38. Springer, Heidelberg (2006)
12. Majumdar, R., Xu, R.G.: Directed test generation using symbolic grammars. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 134–143. ACM (2007)
13. Maurer, P.M.: Generating test data with enhanced context-free grammars. IEEE Software 7(4), 50–55 (1990)
14. Maurer, P.M.: The design and implementation of a grammar-based data generator. Softw. Practice and Experience 22(3), 223–244 (1992)
15. McKeeman, W.: Differential testing for software. Digital Technical Journal of Digital Equipment Corporation 10(1), 100–107 (1998)
16. Offutt, J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state based specifications. The Journal of Software Testing, Verification and Reliability 13, 25–53 (2003)
17. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One evaluation of model-based testing and its automation. In: Proceedings of the 27th International Conference on Software Engineering, pp. 392–401. ACM (2005)
18. Raymond, P., Nicollin, X., Halbwachs, N., Weber, D.: Automatic testing of reactive systems. In: 32nd IEEE Real-Time Systems Symposium, pp. 200–209. IEEE Computer Society (1998)
19. Sirer, E.G., Bershad, B.N.: Using production grammars in software testing. In: Proceedings of the 2nd Conference on Domain-Specific Languages, pp. 1–13. ACM (1999)
20. Tai, K.C., Lei, Y.: A test generation strategy for pairwise testing. IEEE Transactions on Software Engineering 28(1), 109–111 (2002)
21. Warren, D.S.: Memoing for logic programs. Communications of the ACM 35(3), 93–111 (1992)
22. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2), 67–120 (2012)