Empir Software Eng (2017) 22:928–961
DOI 10.1007/s10664-015-9422-4

Generating valid grammar-based test inputs by means of genetic programming and annotated grammars

Fitsum Meshesha Kifetew[1] · **Roberto Tiella**[1] · **Paolo Tonella**[1]

Published online: 14 January 2016

Abstract Automated generation of system level tests for grammar based systems requires the generation of complex and highly structured inputs, which must typically satisfy some formal grammar. In our previous work, we showed that genetic programming combined with probabilities learned from corpora gives significantly better results over the baseline (random) strategy. In this work, we extend our previous work by introducing *grammar annotations* as an alternative to learned probabilities, to be used when finding and preparing the corpus required for learning is not affordable. Experimental results carried out on six grammar based systems of varying levels of complexity show that grammar annotations produce a higher number of valid sentences and achieve similar levels of coverage and fault detection as learned probabilities.

Keywords Grammar based testing · Genetic programming · Grammar annotations

1 Introduction

Programs whose inputs exhibit complex structures, which are often governed by a specification, such as a grammar, pose special challenges to automated test data generation. An example of such systems, which we refer to as *grammar based systems*, is `Rhino`, a compiler/interpreter for the JavaScript language. Test cases for this system are JavaScript

Communicated by: Claire Le Goues and Shin Yoo

✉ Fitsum Meshesha Kifetew
kifetew@fbk.eu

Roberto Tiella
tiella@fbk.eu

Paolo Tonella
tonella@fbk.eu

[1] Fondazione Bruno Kessler–IRST, Trento, Italy

programs that must respect the rules of the underlying JavaScript grammar specification. The challenge in generating test cases for grammar based systems lies in choosing a set of input sentences, out of those that can be potentially derived from the given grammar, in such a way that the desired test adequacy criterion is met. In practice, the grammars that govern the structure of the input are far from trivial. For instance, the JavaScript grammar, which defines the structure of the input for `Rhino`, contains 331 rules and many of these rules are deeply nested and recursive. Hence, an appropriate mechanism for generating (deriving) sentences from such grammars is needed. Despite some efforts made in recent years in this direction (Beyene and Andrews 2012; Poulding et al. 2013; Majumdar and Xu 2007), there is still a need for a solution that is effective at achieving the desired level of coverage adequacy and is able to scale up to reasonably large/complex grammars.

In our previous work (Kifetew et al. 2014), we applied a search based testing approach (McMinn 2004; Pargas et al. 1999; Fraser and Arcuri 2013) which combined Genetic Programming (GP) (McKay et al. 2010) and stochastic grammars to achieve high system-level branch coverage. In particular, the results obtained were significantly better when the stochastic grammars were learned from a corpus of manually written sentences. This is because the learned probabilities allow the generation of structurally well formed sentences while at the same time reducing the risk of infinite or unbounded recursion during sentence derivation. Moreover, GP combined with a suitably defined fitness function has been found to be very effective in evolving the stochastically generated sentences while maintaining their structural validity. Together, GP and stochastic grammars with learned probabilities effectively explore the space of sentences that could be generated from the given grammar, under the guidance of a fitness function targeting system-level branch coverage.

While learned stochastic grammars work well in combination with GP during sentence derivation, *learning* such stochastic grammars comes with added effort, for: (1) transforming the grammar into the format required by the (inside-outside) learning algorithm; and, (2) acquiring and preparing a fairly representative corpus of sentences accepted by the grammar. While for some languages it may be easy to find corpora (e.g., JavaScript programs), for others this might be quite difficult or even impossible (e.g., a custom language defined for a particular purpose). Hence, in cases where learning stochastic grammars is not possible or affordable for a given grammar based system, another mechanism for guiding the stochastic generation of valid sentences would be beneficial, so as to eventually maximize coverage. To this end, we propose a *grammar annotation* scheme which allows the developer to annotate the grammar in such a way that the generated sentences respect semantic constraints of the language. Annotations are specified in the form of *semantic rules* associated with grammar elements (productions and terminal/non terminal symbols).

Experimental results show that annotations outperform learning in terms of number of semantically valid sentences generated from the stochastic grammar, while both are equally effective in terms of achieved coverage and mutation score. Hence, annotations can be practically adopted as an alternative to learning in order to provide GP with an effective stochastic grammar, to be used for sentence derivation.

The key novel contributions of this paper as compared to the previous conference version (Kifetew et al. 2014) are:

1. a novel annotation scheme aimed at increasing the proportion of valid sentences derived from a stochastic grammar
2. three additional subjects added to the empirical study, which double the total number of subjects considered

3. a direct comparison between annotations and learning, which provides practical indications for developers about the choice between these two alternative mechanisms for the improvement of stochastic sentence derivation

The remainder of this paper is organized as follows: in Section 2 we discuss closely related works. In Section 3 we present basic background on stochastic grammars and evolutionary test case generation. The proposed annotation scheme and its implementation is presented in Section 4, while experimental results are described in Section 5. Finally Section 6 concludes the paper and outlines future works.

2 Related Works

The idea of exploiting formal specifications, such as grammars, for test data generation has been the subject of research for several decades now. In the 70s Purdom proposed an algorithm for the generation of short programs from a Context Free Grammar (CFG) making sure that each grammar rule is used at least once (Purdom 1972). The algorithm ensures a high level of coverage of the grammar rules. However, rule coverage does not necessarily imply code coverage nor fault exposure (Hennessy and Power 2005).

Maurer (1990) reports on using extended CFGs for test cases generation. In his work, he introduces constructs that improve uniform random sentence generation by specifying rule-selection probabilities. No indication is given on how to choose them. In the proposed notation, terminal and non-terminal symbols can be replaced by *actions* and *variables* in production rules, to specify dynamic rule-selection strategies: during sentence generation, selection rules can change adaptively. While this is an extremely powerful mechanism, only few simple usage examples are presented in this work. No attempt is made to foresee how complex it could be writing such actions to constrain, for example, the generation of samples from a statically typed language such as Pascal.

In a recent work by Poulding et al. (2013), the authors propose to automatically optimize the distribution of weights for production rules in stochastic CFGs using a metaheuristic technique. Weights and dependencies are optimized by a local search algorithm with the objective of finding a weight distribution that ensures a certain level of branch coverage. In our experience, weight optimization is only part of the problem, which we address by means of learning. To increase coverage, dynamic recombination of sentences, as supported by GP, proves to be another essential ingredient.

Guo and Qiu (2014) propose an approach to generate sentences from a Stochastic CFG (SCFG), where initially uniform probabilities are updated during the generation process. The approach avoids the possible exponential explosion, and the corresponding non-termination, that can be experienced in uniform random based generation algorithms. Their approach generates structurally different test cases whose size increases as the generation algorithm proceeds. The approach produces test suites that contain very different tests starting from the simplest sentences the grammar can generate up to samples that are more and more complex. While the approach seems very promising from the point of view of termination and exponential explosion avoidance, coverage of the System Under Test (SUT) is not a direct goal of the approach.

Symbolic Execution (SE) has been applied to the generation of grammar based data by Godefroid et al. (2008) and Majumdar and Xu (2007). Both approaches reason on symbolic tokens and manipulate them via SE. The work of Godefroid et al. focuses on grammar based fuzzing to find well formed, but erroneous, inputs that exercise the system under test

with the intention of exposing security bugs. The work of Majumdar et al. focuses on string generation via concolic execution with the intention of maximizing path exploration. As both works employ SE, they are affected by its inherent limitations, for instance scalability. Furthermore, the success of these approaches depends on the accuracy of the symbolic tokens that summarize several input sequences into one.

Boltzmann samplers, presented by Duchon et al. (2004), are alternative means for generating instances of structured data, e.g., trees, graphs, and sentences. The work focuses on the problem of the uniform generation of given-size instances and on the efficiency of the generation algorithm, in terms of time and space complexity, when instances big in size are drawn. While in principle, Boltzmann generators can be used as an alternative generation technique to the ones used in our work, it is not clear how this technique scales when applied to rather big grammars, such as the one describing JavaScript programs, that are by far bigger, in terms of number of terminals, non-terminals and rules, than the ones used in their work.

In the present work, we define an annotation language to extend CFGs for imposing (context-sensitive) constraints that cannot otherwise be expressed using CFGs alone. Other approaches, such as QuickCheck presented by Claessen and Hughes (2011) and GödelTest devised by Feldt and Poulding (2013), employ full-fledged programming languages (Haskell in QuickCheck and Ruby in GödelTest) enriched with non-deterministic constructs, to develop test data generators. Of course, both approaches, due to the Turing-completeness of the employed programming languages, can sample at least the same context-sensitive languages our method is able to. However, to follow these approaches, testers have to manually develop generators from scratch, a rather demanding and error-prone activity when huge grammars are involved. Moreover, both works do not discuss the efficacy of the proposed generation methods in terms of code coverage or bug revealing ability.

In the context of generating test data from grammars for code coverage, a recent work closely related to ours is that of Beyene and Andrews (2012). Their approach involves generating Java classes from the symbols (terminals and non terminals) in the grammar. The invocation of a sequence of methods on instances of these classes results in the generation of strings compliant with the grammar. In their work, they apply various strategies for generating method sequences, including metaheuristic algorithms and deterministic approaches, such as depth-first search, with the ultimate objective of finding a test suite that maximizes statement coverage of the system under test.

Our approach differs from the aforementioned works in that we deal with the optimization of probabilities by introducing either a predefined heuristic (80/20) or learning. However, we further use GP to evolve individual syntax trees via mutation/crossover operators guided by a fitness function (sum of branch distances) measured from the execution of the SUT. This way the search gets a more fine-grained and responsive guidance, because the sum of the branch distances is used instead of the overall coverage value.

3 Background

In our proposed approaches for grammar based sentence generation, we exploit the potential of stochastic grammars to generate non-trivial sentences from a given grammar, avoiding potential problems due to recursive grammar definitions. Furthermore, we rely on evolutionary algorithms for generating a set of sentences geared towards maximizing system level branch coverage. Consequently, this section provides the necessary background on these

$$T = \{n, (,), +\}$$
$$N = \{E\}$$
$$s = E$$
$$\pi_1 : E \rightarrow E + E$$
$$\pi_2 : E \rightarrow (E)$$
$$\pi_3 : E \rightarrow n$$

$$E \Rightarrow_{\pi_1}$$
$$E + E \Rightarrow_{\pi_2}$$
$$(E) + E \Rightarrow_{\pi_3}$$
$$(n) + E \Rightarrow_{\pi_3}$$
$$(n) + n$$

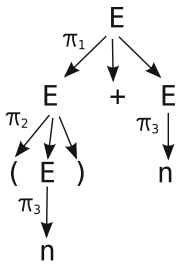


Fig. 1 A simple grammar, a derivation for the string "(n)+n" and its syntax tree

two topics, i.e. *stochastic grammars* and *evolutionary algorithms*, which are extensively used throughout this paper.

3.1 Stochastic Grammars

3.1.1 Notation and Definitions

Figure 1 shows a simple context free grammar $G = (T, N, P, s)$, with four terminal symbols (contained in set T), one non-terminal symbol (set N), three production rules (π_1, π_2, π_3) and start symbol s. A derivation for the sentence "(n)+n" and the associated parse tree are also shown in Fig. 1. $L(G)$ indicates the language generated by the grammar, namely the set of words that can be derived from the start symbol, $L(G) = \{w | s \Rightarrow^* w\}$. A grammar G is *ambiguous* if there exists a word $w \in L(G)$ that admits more than one derivation. For example, there are two distinct derivations for the sentence "n+n+n" in the grammar shown in Fig. 1, namely $\pi_1\pi_3\pi_1\pi_3\pi_3$ and $\pi_1\pi_1\pi_3\pi_3\pi_3$.

Algorithm 1 Generation of a string using a CFG

$S \leftarrow s$
$k = 1$
while $k <$ MAX_ITER and S has the form $\alpha \cdot u \cdot \beta$, where $\alpha \in T^*$ and $u \in N$ **do**
  $\pi \leftarrow$ **choose**(P_u)
  $S \leftarrow \alpha \cdot \pi(u) \cdot \beta$
  $k = k +$
end while
if $k <$ MAX_ITER **then**
  return S
else
  return TIMEOUT
end if

A CFG can be used as a tool to randomly generate strings that belong to the language $L(G)$, expressed by grammar G, by means of the process described in Algorithm 1. The algorithm applies a production rule, randomly chosen from the subset of applicable rules P_u (by means of the function `choose`), to the left-most non terminal u of the working sentential form S, so obtaining a new sentential form that is assigned to S. The algorithm iterates until there are no more non-terminal symbols to substitute (i.e., $S \in T^*$, since it does not have the form $\alpha \cdot u \cdot \beta$ with $u \in N$) or a maximum number of iterations is reached. The

behavior of Algorithm 1 can be analyzed by resorting to the notion of *Stochastic Context-free Grammars* (Booth and Thompson 1973).

Definition 1 (Stochastic Context-free Grammar) A *Stochastic Context-free Grammar* S is defined by a pair (G, p) where G is a CFG, called the *core* CFG of S, and p is a function from the set of rules P to the interval $[0, 1] \subseteq \mathbb{R}$, namely $p : P \to [0, 1]$, satisfying the following condition:

$$\sum_{u \to \beta \in P_u} p(u \to \beta) = 1, \text{ for all } u \in N \tag{1}$$

Condition (1) ensures that p is a (discrete) probability distribution on each subset $P_u \subseteq P$ of rules that have the same non-terminal u as left hand side.

An invocation of Algorithm 1 can be seen as realizing a derivation in a stochastic grammar based on G where probabilities are defined by the function `choose`. The number of iterations that Algorithm 1 requires to produce a sentence depends on the structure of the grammar G and on the probabilities assigned to rules. As a matter of fact, interesting grammars contain (mutually) recursive rules. If recursive rules have a high selection probability p, the number of iterations needed to derive a sentence from the grammar using Algorithm 1 can be very large, in some cases even infinite, and quite likely beyond the timeout limit MAX_ITER.

Let us consider the grammar in Fig. 1, with $p(\pi_3) = q$, $p(\pi_2) = 0$ and $p(\pi_1) = 1-q$. The probability that the generation algorithm terminates (assuming MAX_ITER $= \infty$) depends on q. If $q < 1/2$ the probability that the algorithm terminates is less than 1 and it decreases at lower values of q, reaching 0 when $q = 0$.

This example shows that when Algorithm 1 is used in practice, with a finite value of MAX_ITER, the timeout could be reached frequently with some choices of probabilities p, resulting in a waste of computational resources and in a small number of sentences being generated. A method to control how often recursive rules are applied is hence needed. We discuss two methods widely adopted in practice: the 80/20 rule and grammar learning.

3.1.2 The 80/20 Rule

Given a CFG $G = (T, N, P, s)$, for every non-terminal $u \in N$, P_u is split into two disjoint subsets P_u^r and P_u^n, where P_u^r (respectively P_u^n) is the subset of rules in P_u which are (mutually) recursive (respectively non-recursive). Probabilities of rules are then defined as follows:

$$p(\alpha \to \beta) = \begin{cases} q/|P_u^n|, \text{ if } \alpha \to \beta \in P_u^n \\ (1-q)/|P_u^r|, \text{ if } \alpha \to \beta \in P_u^r \end{cases}$$

so as to assign a total probability q to the non-recursive rules and $1 - q$ to the recursive ones. A commonly used rule of thumb consists of assigning 80 % probability to the non-recursive rules ($q = 0.80$) and 20 % to the recursive rules. In practice, with these values the sentence derivation process has been shown empirically to generate non-trivial sentences in most cases, while keeping the number of times the timeout limit is reached reasonably low.

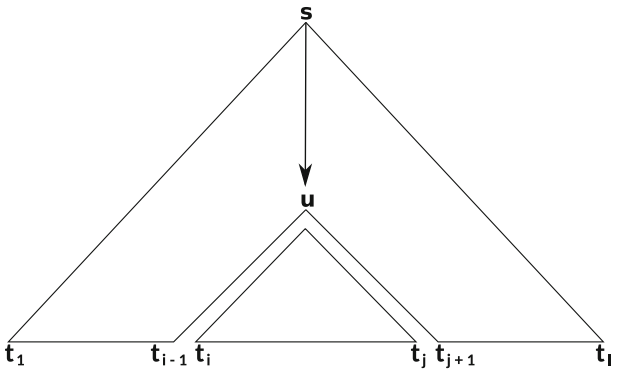


Fig. 2 Graphical representation of a derivation of w which uses non-terminal u

3.1.3 Learning Probabilities from Samples

Another approach to assign rule probabilities to a CFG consists of learning the probabilities from an available corpus. If the grammar is not ambiguous, every sentence has only one parse tree and probabilities can be easily assigned to rules by observing how many times a rule is used in the parse tree for each sentence in the corpus. In the presence of ambiguity, learning can take advantage of the *Inside-outside* algorithm (Lari and Young 1990). The inside-outside algorithm is an iterative algorithm based on expectation-maximization. Starting from randomly chosen probability values, it repeatedly refines the rule probabilities so as to maximize the corpus likelihood.

3.1.4 The Inside-Outside Algorithm

Let us assume that grammars are in Chomsky's Normal Form (CNF), i.e., they comprise only rules of the type (a) $u \rightarrow t$ or (b) $u \rightarrow ab$, where u, a, b are non-terminal symbols and t is a terminal symbol. As every CFG grammar can be put in CNF, this doesn't constitute a restriction.

Let us consider a SCFG (G, p). If G is ambiguous, a sentence w from a corpus $W \subseteq L(G)$ is the frontier of more than one derivation tree. Thus, we cannot know which rules among the given alternatives were actually used to generate w. However, by knowing the probabilities p associated to rules in G, we can derive a *probability of rule usage* in generating sentence w.

Given a derivation for the sentence $w = t_1 ... t_l$ from the start symbol s that exhibits a non-terminal u in a certain point of the derivation (the situation is depicted in Fig. 2), we first compute two probability functions, namely the *outer probability*:

$$f(u, i, j) = P(s \stackrel{*}{\Rightarrow} t_1 ... t_{i-1}\ u\ t_{j+1} ... t_l),$$

i.e., the probability that the sentential form $t_1 ... t_{i-1}\ u\ t_{j+1} ... t_l$ is generated from the start symbol s, and the *inner probability*:

$$e(u, i, j) = P(u \stackrel{*}{\Rightarrow} t_i ... t_j),$$

i.e., the probability that the subsequence $t_i ... t_j$ of w is generated by applying a specific rule to the non-terminal u.

Probability functions e and f can be recursively computed from probabilities p assigned to rules. For example, $e(u, i, j)$ is defined by the following two equations:

$$\begin{aligned} e(u, i, i) &= p(u \to t_i) \\ e(u, i, j) &= \sum_{a,b} \sum_{k=i}^{j-1} p(u \to ab) e(a, i, k) e(b, k+1, j), \text{ if } i < j \end{aligned}$$

f is defined by similar expressions (the interested reader may refer to the work by Lari and Young (1990) for an exhaustive explanation of the algorithm's details). The tool we used, `io`[1], performs this computation by means of a modified version of the CYK bottom-up parser (Grune and Jacobs 1990).

Once e and f are known, the probability $P(u \to ab|u \text{ used})$ that a rule $u \to ab$ was used if non-terminal u is produced during the derivation of w can be expressed in terms of e, f and p as follows:

$$P(u \to ab|u \text{ used}) = \frac{\sum_{i=1}^{l-1} \sum_{j=i+1}^{l} \sum_{k=i}^{j-1} p(u \to ab) e(a, i, k) e(b, k+1, j) f(u, i, j)}{\sum_{i=1}^{l} \sum_{j=i}^{l} e(u, i, j) f(u, i, j)}$$

Since $P(u \to ab|u \text{ used})$ (left hand side) is the same as $p(u \to ab)$ on the right hand side, we can apply an expectation-maximization algorithm. We initially guess such probabilities, indicated as $p^{(0)}$. Then we compute e and f using $p^{(0)}$ and we use the formula above to obtain a new, better estimate $p^{(1)}$ for the rule probabilities. Then we continue with $p^{(2)}$, ..., $p^{(k)}$, until convergence. The actual algorithm computes new estimates using the whole corpus W instead of a single sentence w, but computations are the same as the one presented here for a single sentence w. At each iteration k the likelihood $\Lambda(W)$ of the corpus W:

$$\Lambda(W) = \prod_{w \in W} P(s \stackrel{*}{\Rightarrow} w)$$

is computed. The algorithm has been proven to provide estimates for p that never decrease $\Lambda(W)$, hence converging to a (possibly local) maximum. In practice, the computation continues until no sensible increase in likelihood is observed. The complete algorithm is shown as Algorithm 2.

[1] `io` was written by Mark Johnson, see http://web.science.mq.edu.au/~mjohnson/Software.htm

Algorithm 2 The inside-outside algorithm

Require: A CFG G, a corpus of sentences W, a small real number δ
Ensure: p are probabilities associated to rules of G that (locally) maximize the likelihood of the corpus W
  Randomly assign probabilities p to rules of G and compute the likelihood L' of W
  repeat
    $L \leftarrow L'$
    parse each sentence $w \in W$ using CYK
    compute inside probability $e(u, i, j)$ and outside probability $f(u, i, j)$ using available values for p
    compute a new estimate for p using $e(u, i, j)$, $f(u, i, j)$ and available values for p
    compute the new likelihood L' of W
  until $L' - L < \delta$

3.2 Evolutionary Algorithms

Evolutionary algorithms search for approximate solutions to optimization problems, whose exact solutions cannot be obtained at acceptable computational cost, by evolving a population of candidate solutions that are evaluated through a fitness function. Genetic algorithms (GAs) have been successfully used to generate test cases for both procedural (Pargas et al. 1999) and object-oriented software (Fraser and Arcuri 2013). GAs evolve a population of test cases trying to maximize a fitness function that measures the distance of each individual test case from a yet uncovered target. The genetic operators used for evolutionary test case generation include test case mutation operators (e.g., mutate primitive value) and crossover between test cases (e.g., swap of the tails of two input sequences) (McMinn 2004).

3.2.1 Whole Test Suite Generation

Whole test suite generation (Fraser and Arcuri 2013) is a recent development in the area of evolutionary testing, where a population of *test suites* is evolved towards satisfying *all coverage targets at once*. Since in practice the infeasible (unreachable) targets for a system under test (SUT) are not generally known a priori, generating test data considering one coverage target at a time is potentially inefficient as it may waste a substantial amount of search budget trying to find a solution for infeasible targets. Whole test suite generation is not affected by this problem as it does not try to cover one target at a time. Rather, the fitness of each test suite is measured with respect to *all coverage targets*. That is, when a test suite is executed for fitness evaluation, its performance is measured with respect to all test targets.

3.2.2 Genetic Programming

Genetic programming (McKay et al. 2010) follows a similar process as GAs. However, the individuals manipulated by the search algorithm are tree-structured data (*programs*, in the GP terminology) rather than encodings of solution instances. While there are a number of variants of GP in the literature, in this work we focus on Grammar Guided GP

(GGGP) (McKay et al. 2010). In GGGP, individuals are sentences generated according to the formal rules prescribed by a CFG. Specifically, initial sentences are generated from a SCFG and new individuals produced by the GP search operators (crossover and mutation) are constrained to be valid under the associated CFG.

An individual (a sentence from the grammar) in the population is represented by its parse tree. Evolutionary operators (crossover and mutation) play a crucial role in the GP search process. Subtree crossover and subtree mutation are commonly used operators in GP. The instances of these operators that we use in our approach are described in detail in Section 4.

4 Approach

We first introduce our language for grammar annotation (Section 4.1), followed by a description of our evolutionary algorithm for sentence derivation from (possibly annotated) grammars (Section 4.2).

4.1 Annotated Grammars

Annotations allow the developer to specify constraints and relations among grammar elements which are very difficult, if not impossible, to express with CFGs. Such annotations ensure that semantically invalid sentences are never or seldom generated from the annotated grammar. We introduce an annotation scheme that allows the developer to *annotate* grammar rules with *semantic rules* in such a way that they can be processed automatically during sentence generation. Consequently, annotations ensure that the generated sentences, aside from being structurally well-formed, respect also the *semantic* constraints of the language defined by the CFG.

4.1.1 Types

Semantic rules are defined based on *types*. *Types* are tuples of the form $(t_1, t_2, \ldots ; t'_1, t'_2, \ldots ; t''_1, t''_2, \ldots)$ where t_i are either *base types* or * (which indicates the absence of any type restriction – i.e., any type is allowed). Types are grouped and type groups are separated by semicolons. Different type groups can be introduced for different aspects of the program semantics. For instance, the first group may specify *data types* (types such as number, string, etc.). Such types will be applied only to constructs for which a data type restriction makes sense (e.g., expressions). The following groups in the type tuple might be used for *structural* type constraints (e.g., `break` constructs can only appear nested inside loop or switch statements), or different kinds of data type constraints (e.g., applicable to specific constructs).

The types used to annotate a given grammar are determined by the developer, depending on the nature of the language represented by the grammar. The annotation scheme does not prescribe any predefined types (except for * which stands for "any type"). The base types determined by the developer are specified at the beginning of the grammar file, so that the sentence generator can process and use them when enforcing the annotation rules. The specification may include initial default values, as well as whether a matching declaration for a construct having such type is mandatory in the language or not (e.g., a variable declaration for languages where declaration of variables is mandatory). When declarations are mandatory for some types, a 'template' for the syntax of declarations is to be provided. Base

types consisting of simple labels, without initial default values and matching declarations, are specified in a simplified syntax. Example:

```
#begintypes
#int,0,false
#bool,false,false
#date,2015.01.01,false
#qstring,'',false
#fun,nil,true
#endtypes
#declarator:def @id = @init ;
#[type;inloop;infunc;inswitch]
```

The first base type in this example is a data type, consisting of 5 possible instances (`int`, `bool`, etc.), for which initial values are provided. Except for the `fun` type, all other data types do not require any matching declaration. The template for the `declarator` (required by the `fun` type) consists of the `def` keyword, followed by the identifier to be matched (specified by means of the special meta-variable `@id`). Equal signs and semicolons are part of the syntax of the declarator, while the meta-variable `@init` is compatible with any expansion of a non terminal that can appear in the meta-variable position according to the given grammar. A simplified type specification terminates this example. It introduces three type labels (`inloop`, `infunc`, `inswitch`) for a new structural type group. These type instances have no default initializers and no matching declarators.

4.1.2 Annotation Syntax

The proposed annotation scheme includes two types of annotations for: (1) selecting productions based on types; and, (2) propagating type information. The syntax of each is described below.

Production Selection Annotations for type-based selection of a production (among the list of productions in a grammar rule) are intended to match the type information propagated along the derivation process up to the current point. These annotations are indicated by preceding each production with selector types enclosed in square brackets (*[]*). Given a grammar rule of the form `LHS ::= A1 A2` (LHS stands for Left Hand Side), selection annotations indicate which productions can be selected during derivation without invalidating the type correctness of the sentence being derived. The syntax of such an annotation is as follows:

- `LHS` ::= [$\mathtt{t}_1, \mathtt{t}_2, \ldots; \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots$] `A1 A2` this production is selected if the tuple propagated for the type of LHS (t_a; t_b; t_c) matches the given annotation. That is, t_a is any of $\{t_1, t_2, \ldots\}$ or *; t_b is any of $\{t'_1, t'_2, \ldots\}$ or *; and, t_c is any of $\{t''_1, t''_2, \ldots\}$ or *.
- `LHS` ::= [$\mathtt{\$t} = \mathtt{t}_1, \mathtt{t}_2, \ldots; \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots$] `A1 A2` this production is also selected if the tuple propagated for the type of LHS matches the given annotation. In this case, the actually propagated LHS type t_a (taken from the first type group) is assigned to variable `$t` for later use in the rest of the rule.
- `LHS` ::= [$(\mathtt{t}_1, \mathtt{t}_2, \ldots); \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots$] `A1 A2` this production is selected if the tuple propagated for the type of the LHS matches the given annotation. The parts of the annotation within brackets require strict type matching, i.e., matching with the implicit

any-type * selector is not allowed and the match occurs only and exclusively if the propagated LHS type is any of t_1, t_2, etc.

To simplify the process of annotation, if only the first type group is provided in the tuple, the others are assumed to be *. Hence, the annotation `[int,float]` is equivalent to `[int,float;*;*]`. Let us consider the following annotated excerpt, taken from a grammar for expressions:

```
<expr> ::= [int] <expr> % <expr>
| [float] <expr> * <expr>
| ( <expr> )
| [int, float] <const>
```

According to these annotations, the production `<expr> ::= <expr> % <expr>` is eligible for selection during sentence derivation if the propagated type for the LHS non-terminal `<expr>` is either `int` or `*`. Similarly, the production `<expr> ::= <const>` is eligible for selection if the propagated LHS type of `<expr>` is either `int`, `float`, or `*`, while the production `<expr> ::= ( <expr> )` is always eligible for selection.

Type Propagation The default, implicit rule is that types propagate from the LHS non-terminal to the right hand side (RHS) non-terminals during sentence derivation. However, type propagation annotations can be used to direct and control the types being propagated from LHS to RHS non-terminals. In fact, the type to be propagated to a grammar symbol can be specified explicitly by putting it into *curly braces* {}. If such an annotation is not present, the default rule applies (i.e., the LHS type propagates to the RHS). Given a grammar rule of the form `LHS ::= A1 A2`, the syntax of a type propagation annotation is as follows:

`LHS` ::= $[\mathtt{t}_1, \mathtt{t}_2, \ldots; \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots]$ `A1 A2` whichever type is selected from the list, it is by default propagated to `A1` and `A2`.

`LHS` ::= $[\mathtt{t}_1, \mathtt{t}_2, \ldots; \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots]$ `A1` $\{\mathtt{t}_3; \mathtt{t}_5; *\}$ `A2` whichever type is selected from the list, it is propagated to `A1`, whereas `A2` is assigned the type tuple $(t_3; t_5; *)$.

`LHS` ::= $[\$\mathtt{t} = \mathtt{t}_1, \mathtt{t}_2, \ldots; \mathtt{t}'_1, \mathtt{t}'_2, \ldots; \mathtt{t}''_1, \mathtt{t}''_2, \ldots]\{\mathtt{t}_3\}$ `A1` $\{\$\mathtt{t}\}$ `A2 A1` is assigned the type $(t_3; *; *)$ whereas `A2` is assigned $(\$t; *; *)$ where $\$t$ holds whichever type is instantiated from the selection annotation.

Example:

```
<expr> ::= [int,float] {int} <expr> % {int} <expr>
| [(float)] <expr> * <expr>
```

In the example shown above, if the production `<expr> ::= <expr> % <expr>` is selected, type `int` is propagated to the two `<expr>`s on the right hand side. On the other hand, if the production `<expr> ::= <expr> * <expr>` is selected, the type of the left hand side `<expr>` (in this case, necessarily `float`) is propagated to the two `<expr>`s on the right hand side. Figures 3 and 4 show an example of grammar and of a possible annotation.

The example in Fig. 4 shows a useful feature provided by our annotation language: suffix string concatenation, expressed through the syntax ##. Sometimes it is useful to distinguish different instances of the same type. For example, the type of a newly declared variable as

```
<prog> ::= <stats>
<stats> ::= <stat> <stats> | <stat>
<stat> ::= <def> ; | <expr> ;
<def> ::= def <id> = <const>
  | def <id> [ <const> ] = { <expr> }
<const> ::= <int>
  | <float>
<int> ::= 1 | 2 | 3
<float> ::= 0.5 | 1.5
<expr> ::= <expr> % <expr>
  | <expr> * <expr>
  | (<expr>)
  | <id>
  | <const>
```

Fig. 3 Example grammar

compared to the type of a previously declared variable. By adding a suffix to the variable type (e.g., `N` in `$t##N`) we can then easily separate productions to be selected only for newly declared variables (e.g., `<new>` production for the LHS `<id>`) with respect to those to be selected when previously declared variables are involved (e.g., `<pool>` production). These two cases respectively correspond to the *declaration* and *use* of a variable.

The last two productions shown in Fig. 4 are special productions that are introduced for facilitating the automated processing of the annotations. The production `<new> ::= <new()> @id` triggers the generation of a new identifier of a given type, while the production `<pool> ::= <pool()> @id` triggers a use of an existing identifier from the pool of already declared identifiers.

4.1.3 Supporting Data Structures

The implementation of the type propagation mechanism described above requires the following data structures:

Identifier Pool We maintain a pool of identifiers to be used for naming variables and functions. At the beginning of each sentence derivation, the pool is initialized with a predefined set of identifiers, of predefined types. Whenever a new identifier is created in the derivation process, the pool is updated with the new identifier. Whenever a name is needed for either a variable or a function (of a certain type), the pool is consulted and an appropriate identifier is chosen.

Symbol Table We maintain a *symbol table* in which we keep track of the identifiers used so far, along with their types, in the derivation. At the end of the derivation, some of these identifiers may need to be *declared*, if their declaration is indicated as mandatory in the type specification preamble and no declaration was already derived for them. Hence, after the derivation of a sentence is completed successfully, a *repair* phase follows, in which we refer back to the symbol table and insert declarations for those identifiers which are used in the sentence without an accompanying declaration, in cases where such declaration has been specified as mandatory.

```
#begintypes
#int,0,false
#bool,false,false
#date,2015.01.01,false
#qstring,'',false
#fun,nil,true
#int[],[],false
#endtypes
#declarator:def @id = @init ;
#[type;inloop;infunc;inswitch]

<prog> ::= <stats>
<stats> ::= <stat> <stats>
 | <stat>
<stat> ::= <def>;
 | <expr>;
<def> ::= [$t=int,float,int[]] def {$t##N} <id> = <const>
  | [$t=int[],float[]]def {$t##N} <id> [{basetype($t)} <const>] = {<expr>}
<const> ::= [int] <int>
  | [float] <float>
  | [date] <date>
  | [bool] <bool>
  | [qstring] <string>
<int> ::= 1 | 2 | 3 ...
<float> ::= 0.5 | ...
<date> ::= 2015.01.01 | ...
<bool> ::= true | false
<string> ::= ''
<expr> ::= [int] <expr> % <expr>
  | [float] <expr> * <expr>
  | ( <expr> )
  | [int,float,int[],float[],bool,date,qstring] <id>
  | [int,float,bool,date,qstring] <const>
  | [int,float] {int[]} <expr>  [ {int} <expr>  ]
<id> ::= [intN,floatN,int[]N,float[]N,boolN,dateN,qstringN] <new>
  |[int,float,int[],float[],bool,date,qstring] <pool>
<new> ::= <{new()}> @id        // generate a new identifier name
<pool> ::= <{pool()}> @id        // choose an identifier name
                                 // already declared so far
```

Fig. 4 A possible annotation for the grammar in Fig. 3

4.1.4 Annotation Example

To illustrate better the annotation scheme, we apply it to an excerpt from the JavaScript grammar. The annotated grammar is shown in Fig. 5. Missing portions of the grammar are represented by ellipses (...). The excerpt represents part of the grammar that defines the structures of statements in JavaScript. With the annotation scheme, we would like to ensure that the sentences generated from the grammar respect certain (structural and semantic) rules of the language. For instance, expressions used as conditions for *while* and *if* statements are required to be of *boolean* type. Hence, in the annotated grammar we specify this constraint by annotating the `<Expression>` non-terminals with `{bool}` so that during sentence derivation this type is propagated in the syntax tree, ultimately constraining the derivation to the generation of boolean expressions.

Similarly, there are structural rules that constrain the placement of certain types of language constructs in the program. For example, statements such as *break* and *continue*

```
#begintypes
#int,0,false
#bool,false,false
...
#endtypes
#declarator:var @id = @init ;
#[type;inloop;infunc;inswitch]

<Program> ::= <Sourceelement_star>
 ...

<Statementtail> ::= <Withstatement>
 | <Iterationstatement>
 | <Ifstatement>
 | [*;(inloop)] <Continuestatement>
 | [*;*;*;(inswitch)] <Breakstatement>
 | ...

...

<Whilestatement> ::= while ( {bool} <Expression> ) {*;inloop} <Statement>

<Forstatement> ::= for ( <Forcontrol> ) {*;inloop} <Statement>

<Dostatement> ::= do {*;inloop} <Statement> while ( {bool} <Expression> ) ;

<Ifstatement> ::= if ( {bool} <Expression> ) <Statement> <Alt_248_opt>
 | if ( {bool} <Expression> ) <Statement>

 ...
```

Fig. 5 Annotated grammar extracted from the JavaScript grammar

are allowed only in certain contexts (e.g. loops). Consequently, in the annotated grammar we guard the usage of such constructs by introducing type tuples that limit the generation of these types of statements in the appropriate contexts. As shown in Fig. 5, `<Continuestatement>` is annotated with the type tuple `[*;(inloop)]`, which ensures that the production is eligible for selection during derivation only if the type tuple propagated to it down from the containing statement matches the specified type tuple. In order for this production to be selected during sentence derivation, the type tuple associated with `<Statementtail>` must have the type `inloop` (brackets enforce strict match) in second position in the type tuple, while it could have any type in first position in the tuple. Correspondingly, the productions for the looping constructs (`<Whilestatement>`, `<Forstatement>`, and `<Dostatement>`) are annotated in such a way that they propagate down a type tuple permissive of statements allowed within loops. This is specified by annotating the body of the loops (represented by the non-terminal `<Statement>` in the grammar) with a type tuple {`*;inloop`}, which is propagated down the syntax tree during sentence derivation.

4.2 Evolutionary Sentence Generation

Our evolutionary approach to sentence generation combines stochastic grammars with genetic programming, using a suitable fitness function, so as to evolve *test suites* for system-level branch coverage of the SUT. Since we perform whole-test suite optimization (Fraser and Arcuri 2013), which is well suited for system level testing, we evolve both test suites as

well as the test cases inside the test suites. For test suite evolution we use GA, while for test case evolution we use grammar guided GP (see Section 3). Furthermore, we take advantage of grammar annotations (when available) in order to promote the generation of structurally and semantically well-formed sentences from a given grammar.

4.2.1 Representation of Individuals

Individuals manipulated by the GA are *test suites*. Each test suite is composed of individual *test cases*. A test case is a single input to the SUT. In other words, a test case is a well-formed sentence derived from the grammar of the SUT. Hence, a test suite in the GA is a set of sentences, represented by their parse trees. Furthermore, when grammar annotation is activated, each individual additionally stores the annotation types associated with the grammar symbols in the derivation tree. Such information is later used by genetic operators when evolving the individual.

The initial population of test suites is obtained by generating input sentences according to the stochastic process described in Algorithm 1 and by grouping them randomly into test suites. Stochastic sentence generation uses either heuristically fixed or learned probabilities, as discussed in Section 3.

When grammar annotation is activated, the initialization of the population follows a similar process as the one described in Algorithm 1. However, with grammar annotation, the function **choose** in Algorithm 1 is applied considering not only the probabilities of the productions for the current non-terminal, but also the types specified in the grammar annotations. Hence, when choosing a production to expand the current non-terminal, we first collect all the eligible productions for the non-terminal according to the matching between propagated type and selection types, as described in Section 4.1. This gives the set of eligible productions $P_{eligible}$. Then, function **choose** is applied as in Algorithm 1 on the productions in $P_{eligible}$. The weights of the productions in $P_{eligible}$ are re-computed so as to redistribute the probabilities of the non-eligible productions to the eligible ones. This is achieved by re-normalizing to 1 the probabilities in $P_{eligible}$.

4.2.2 Genetic Operators

In our approach, genetic operators work at two levels: at the upper level, GA operators are used to evolve test suites (TS); at the lower level, GP operators are used to evolve the parse trees that represent the input sentences of the test cases contained in a test suite. Evolution at the lower level is regarded as a special kind of mutation (namely, parse tree mutation) at the upper level. Hence, GP operators are activated according to the probability of parse tree mutation set in the upper GA level. In particular, the GP operator *subtree mutation* is applied to a test case that belongs to test suite T with probability $1/|T|$. The GP operator *subtree crossover* is applied with probability α.

GA Operators

TS Mutation 1: insert new test cases: with a small probability β, a new test case is added to T; additional test cases are added with (exponentially) decreasing probability. The new test cases to insert are generated by applying Algorithm 1, taking any available annotation into account.

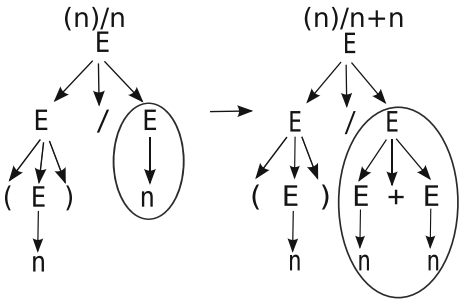


Fig. 6 Subtree mutation: a subtree (circled) is replaced with a new one generated from the grammar using Algorithm 1

TS Mutation 2: delete test cases: with a small probability γ a test case is removed from T. The test case which covers the least number of branches is selected for removal, so as to keep the most promising individuals in the test suite.

TS Crossover: Given two parent test suites T_1 and T_2, crossover results in offspring O_1 and O_2, each containing a portion of test cases from both parents. Specifically, the first $\delta|T_1|$ tests from T_1 and the last $(1-\delta)|T_2|$ tests from T_2 are assigned to O_1; while the first $\delta|T_2|$ tests from T_2 and the last $(1-\delta)|T_1|$ tests from T_1 are assigned to O_2, for $\delta \in [0, 1]$.

GP Operators

Subtree mutation: Subtree mutation is performed by replacing a subtree in the tree representation of the individual with a new subtree, generated from the underlying stochastic grammar by means of Algorithm 1. Figure 6 shows an example of subtree mutation applied to a test case.

If grammar annotation is activated, when a subtree is selected for mutation and when a replacement subtree is generated, the GP operator must ensure that type annotations are not violated:

- Type annotations associated with the root of the subtree being replaced need to be propagated to the new subtree that will be generated to replace it.
- The pool of declared identifier names in the tree being mutated needs to be available to the new subtree.
- Subtrees containing declarations of identifiers are never replaced, to avoid the situation in which any later uses of those identifiers are invalidated.

Subtree crossover: Figure 7 shows an example of subtree crossover between two test cases in a test suite T. Two subtrees rooted at the same non terminal are selected in the parent trees and swapped, so as to originate two new offspring trees.

If grammar annotation is activated, in addition to being associated with the same non-terminal, the two subtree nodes chosen as points of exchange/crossover must have the same type annotation.

4.2.3 Fitness Evaluation

The GA evaluates each individual (test suite) by computing its fitness value. For this purpose, the tree representation of the test cases in the suite is unparsed to a string, which is passed to the SUT as input. The GA determines the fitness value by running the SUT with

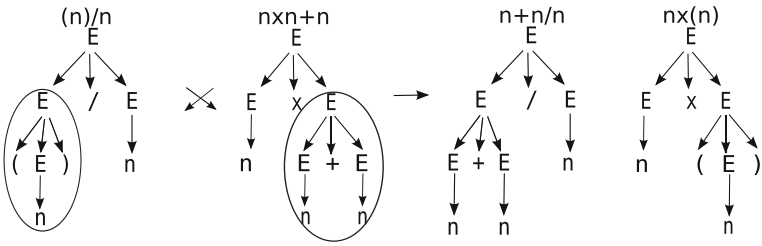


Fig. 7 Subtree crossover: subtrees of the same type (*circled*) from *parents* are exchanged to create *children*

all unparsed trees from the suite and by measuring the amount of branches that are covered, as well as the distance from covering the uncovered branches.

During fitness evaluation, branch distances (McMinn 2004) are computed from all possible branches in the SUT, spanning over multiple classes. The fitness of the suite is the sum of all such branch distances. This fitness function is an extended form of the one employed by Fraser and Arcuri (2013) for unit testing of classes. GA uses Equation 2 to compute the fitness value of a test suite T, where $|M|$ is the total number of methods in the SUT; $|M_T|$ is the number of methods executed by T (hence $|M - M_T|$ accounts for the entry branches of the methods that are never executed); $d(b_k, T)$ is the minimum branch distance computed for the branch b_k; a value of 0 means the branch is covered.

$$fitness(T) = |M| - |M_T| + \sum_{b_k \in B} d(b_k, T) \tag{2}$$

4.2.4 Implementation

We implemented the proposed approach in a prototype by extending the EvoSuite test generation framework (Fraser and Arcuri 2011). In particular, we extended EvoSuite with: (1) a new parse-tree based representation of individuals; (2) a new initialization method, which resorts to stochastic grammar based sentence derivation; (3) new GP operators which manipulate parse tree representations of individuals; (4) support for handling the proposed type annotation of grammars. Moreover, the top-level algorithm has been modified to accommodate the two levels (GA and GP) required by our approach. For each SUT, we assume that there is a system level entry point through which it can be invoked. In cases where such entry point is missing, we define one, acting as a test driver for invoking the core functionalities of the SUT.

To learn rule probabilities from a corpus, we used an existing implementation of the inside-outside algorithm[2]. Given a grammar and a set of sentences, this implementation of the algorithm produces as output a probability for each rule in the grammar. We also implemented a tool that performs all necessary transformations on the input grammar (Grune and Jacobs 1990), so as to turn it into a format acceptable for the inside-outside implementation. We also implemented a tool for tokenizing the sentences in the corpora that we used for learning. Since the tool expects a stream of tokens, rather than a stream of characters, it is necessary to tokenize the sentences into streams of tokens so that the learning algorithm can make use of them.

[2] http://web.science.mq.edu.au/~mjohnson/Software.htm

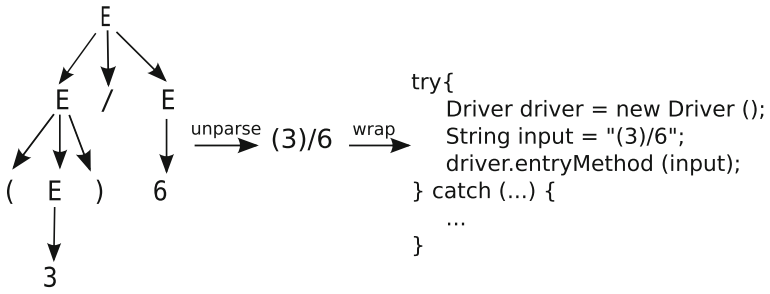


Fig. 8 During fitness evaluation, tree representations are unparsed and wrapped into sequences of Java statements

During fitness evaluation, the tree representation of each individual test case is unparsed to a string which is then wrapped into a sequence of Java statements. These sequences of Java statements are then executed against the instrumented SUT. Figure 8 shows a simplified example of this process.

As recommended by Arcuri et al. (2010), our implementation takes advantage of accidental coverage. If the execution of a test case covers a coverage target which was not covered so far, such a test case is kept as a solution, regardless of the survival of the test suite it belongs to. At the end of the search, such test cases are merged with the best suite evolved by the search. In this way, test cases that exercise uncovered targets but are not part of the final "best" test suite are not lost.

We implemented a random generation technique (RND hereafter) as a baseline for comparing the performance of the proposed approach. RND is implemented as follows: generate a random test case (either using stochastic grammars or annotated grammars), execute it against the SUT, and collect all covered branches (Arcuri et al. 2010). RND stops either when full coverage is reached or the search budget is finished.

The implementation of the annotation scheme used for our empirical experiments and related resources are available for download at the following address: http://selab.fbk.eu/kifetew/gbt.html.

5 Experimental Results

To evaluate the effectiveness of annotations and learning in combination with GP, we carried out experiments on 6 open source grammar based systems with varying levels of complexity. Effectiveness was assessed in terms of generation of semantically valid inputs, coverage and fault detection. Specifically, we formulated the following research questions:

RQ1 (valid sentences): Does the use of annotations or learning increase the number of semantically valid sentences that are generated by a stochastic grammar?

RQ2 (combination): Does the combination of GP with annotations or learning increase the level of coverage achieved during test case generation?

RQ3 (comparison): Which technique between annotations and learning provides the most effective combination with GP in terms of coverage? Are the two compared techniques complementary or largely overlapping?

RQ4 (fault detection): What is the fault detection capability of RND/GP with either annotations or learning?

RQ5 (annotator bias): What is the sensitivity of the stochastic grammar to changes in the annotations introduced by the annotator?

5.1 Metrics

To answer RQ1, we compute the proportion of semantically valid sentences generated by a stochastic grammar using annotations or learning, in comparison with a baseline consisting of a stochastic grammar that implements the 80/20 rule described in Section 3. For RQ2 and RQ3, the metrics used to measure the *effectiveness* of the techniques being compared is *branch coverage at the system level*, computed as the number of branches covered out of the total number of branches in the SUT. To answer the second part of RQ3, we compute set intersection and set difference between the branches covered by GP with annotations vs. GP with learning. For RQ4 we consider artificial faults (mutants) injected into the SUT by the mutation tool PIT[3]. We measure the *mutation score*, i.e., the proportion of mutants that are *killed* by the generated test cases. A mutant is considered as killed if the original and mutated programs produce different outputs when the generated test cases are executed. To answer RQ5, we measure the coverage reached at different levels of annotation, by dropping annotations from the annotated grammar. We then compute the delta coverage with respect to the baseline (no annotations dropped), when an increasing proportion of annotations is dropped.

5.2 Subjects

The subjects used in our experiments are 6 open source Java systems that accept structured input based on a grammar. `Calc`[4] is an expression evaluator that accepts an input language including variable declarations and arbitrary expressions. `MDSL`[5] is an interpreter for the Minimalistic Domain Specific Language (MDSL), a language including programming constructs such as functions, loops, conditionals etc. `Rhino`[6] is a JavaScript compiler/interpreter. `Basic`[7] is an interpreter for a BASIC-like language called COCOA. `Kahlua`[8] is a compiler for the Lua language. `Javascal`[9] is a Pascal to Java compiler.

Considering the complexity of the input structure (specifically, the associated grammar) they accept, these subjects are representative of a wide range of grammar based systems. The smallest is `Calc` and the largest `Rhino`, while the others are in between. Table 1 reports the size in LOC [10] (Lines Of Code) of the source code (excluding comments) and the number of productions in the respective grammars. Terminal productions, accounting for the lexical structure of the tokens, are excluded. These grammars are far more complex than those typically found in the GP literature and contain several nested and recursive definitions. Hence, they represent a significant challenge for the automated generation of test data.

[3] http://www.pitest.org

[4] https://github.com/cmhulett/ANTLR-java-calculator/

[5] http://mdsl.sourceforge.net/

[6] http://www.mozilla.org/rhino (version 1.7R4)

[7] http://www.mcmanis.com/chuck/java/cocoa/

[8] https://code.google.com/p/kahlua/

[9] http://javascal.sourceforge.net/

[10] counted by CLOC - http://cloc.sourceforge.net/

Table 1 Subjects used in our experimental study

Subject	Size(LOC)	# Prods	# Annot prods	# Types	# Selectors	# Propagators
Calc	1,736	38	5 (13.16 %)	1	3	5
Basic	4,866	108	50 (46.30 %)	3	48	16
Kahlua	8,093	132	51 (38.64 %)	4	22	43
MDSL	10,008	161	50 (31.06 %)	12	27	48
Javascal	13,486	270	86 (31.85 %)	11	65	34
Rhino	56,849	331	82 (24.77%)	11	39	60

The corpus used for learning the probabilities of stochastic grammars is composed of sentences we selected from the test suites distributed with each SUT (for `Calc`, `MDSL`, `Kahlua`), from the V8 JavaScript Engine benchmark[11] (for `Rhino`), and from code examples freely available on the Internet (for `Basic` and `Javascal`). Detailed information about the size of the corpus used for probability learning is shown in Table 2.

Also shown in Table 1 are details related to the annotations we performed on the grammars. Between 13 % (`Calc`) and 46 % (`Basic`) of the productions contain some form of annotation, i.e., *selector* annotations or *propagator* annotations (see Section 4.1.2). Table 1 further reports the number of each specific type of annotation introduced by the annotations. The annotation of the subject grammars was performed by one of the authors of this paper. The time spent for performing the annotations ranges between 2 and 8 hours, depending on the grammar. As the annotator is not a developer of any of the subject programs nor has detailed prior knowledge of the grammars, most of the time was spent in understanding the grammar and the intended semantics of the underlying language. Presumably, the developers/testers of the respective programs know very well the grammar and the corresponding language. Hence in practice, the quality of the annotations could potentially be much higher than in our experiments. The amount of time it takes to perform the annotations would also be lower. Consequently, the experiments and the results reported in this section are not necessarily indicative of the best case scenario. Rather, they show the potential of the annotation scheme introduced.

5.3 Procedure and Settings

Since the approaches being compared (GP and RND, with or without annotations) are based on stochastic grammars, they heavily rely on non-deterministic choices. Therefore, we repeated each experiment 10 times and measured statistical significance of the differences using the Wilcoxon non parametric test.

Based on some preliminary sensitivity experiments, we assigned the following values to the main parameters of our algorithm: population size = 50, crossover rate = 0.75, subtree crossover rate $\alpha = 0.1$, new test insertion rate $\beta = 0.1$. For the other parameters we kept the default values set by the EvoSuite tool. Since the subjects used in our experiments differ significantly in size and complexity, giving the same search budget to all would not be fair. Hence, we resorted to the following heuristic rule for budget assignment: we give each SUT a budget of $n * |branches|$, where $|branches|$ is the number of branches in the SUT. Based on a few preliminary experiments, we chose the value $n = 5$.

[11] https://code.google.com/p/v8/

Table 2 Number of sentences and tokens in the corpus used for each subject during learning. Also shown is the average number of tokens per sentence

Subject	# Sentences	# Tokens	Tokens per sentence
Calc	19	189	9.95
Basic	19	1186	62.42
Kahlua	9	1583	175.89
MDSL	65	5285	81.31
Javascal	5	618	123.60
Rhino	990	32752	33.08

To measure the proportion of semantically valid sentences, we generated $s = n * |branches|$ sentences (with $n = 1$) when applying each approach and counted how many sentences are semantically valid. By *semantically valid* sentences we mean sentences that do not make the SUT emit a custom error message (either related to the input language syntax or semantics). Since deciding whether an input is semantically valid or not in an automatic manner is a non trivial task, we adopted the following approximation. If for a given input sentence the SUT throws a *custom exception* (i.e., an exception class defined in the SUT itself), the input is considered invalid (i.e the SUT recognizes that the input does not respect the specifications of the language). Otherwise it is considered valid (i.e., no exception is thrown or the thrown exception is a standard Java exception, such as a NullPointerException).

To measure the sensitivity of the stochastic grammar to changes in the annotations, we performed simulated experiments in which we randomly dropped a fraction of the annotations from the annotated grammar. Since the type of annotation that determines whether a production is selected or not during sentence derivation is the *selector annotation*, we decided to drop selector annotations for this experiment. By dropping the selector annotation for a given production, we intend to simulate the situation in which the developer did not annotate the production, either intentionally or unintentionally. The number of each type of annotation for each subject in the experiment is presented in Table 1. As can be seen from the table, the selector annotations for the smallest subject (`Calc`) are very few (3), and hence not suitable for this analysis. For this reason, we excluded `Calc` and carried out the experiment on the remaining 5 subjects. We experimented with five different configurations in which we dropped 5 %, 10 %, 15 %, 20 %, and 25 % of the annotations. For each configuration, we executed both GP and RND 10 times on each subject, every time changing the set of annotations dropped.

5.4 Results

Table 3 reports the proportion of well-formed sentences generated by the various strategies (columns 5-6-7). In addition, the number of unique sentences is also reported in columns 2-3-4, accompanied (in brackets) by the percentage of unique sentences over the total number of sentences that have been generated. Such percentage is indicative of the proportion of duplicate sentences generated by the various strategies (in fact, the complement of the reported percentages is the percentage of duplicate sentences). The proportion of well-formed sentences is computed with respect to the *unique sentences* generated by each technique.

Table 3 Unique sentences and proportion of valid sentences produced by the various sentence generation strategies. In the last 3 columns, highest values in boldface differ from the second highest in a statistically significant way according to the Wilcoxon test, at significance level 0.05

	Unique sentences generated			Valid unique sentences		
Subject	8020	LRN	AN	8020	LRN	AN
Calc	212 (78.83 %)	212 (76.29 %)	212 (81.57 %)	100.00 %	100.00 %	100.00 %
Basic	907 (78.29 %)	907 (72.90 %)	907 (80.48 %)	22.01 %	**56.12 %**	51.58 %
Kahlua	1191 (56.15 %)	1191 (90.70 %)	1191 (48.27 %)	64.53 %	71.94 %	**86.26 %**
MDSL	2630 (60.97 %)	2630 (86.90 %)	2630 (43.77 %)	25.46 %	75.13 %	**97.86 %**
Javascal	730 (95.74 %)	730 (97.41 %)	730 (57.34 %)	14.38 %	41.04 %	**95.07 %**
Rhino	6138 (43.80 %)	6138 (60.26 %)	6138 (10.29 %)	84.70 %	55.52 %	**97.29 %**

From the results, we can notice that annotation (AN hereafter) produces the largest proportion of well-formed sentences in four subjects (`Kahlua`, `MDSL`, `Javascal`, and `Rhino`). For subject `Basic`, learning (LRN hereafter) produces the highest proportion of valid sentences, while for `Calc` they are equal. In fact, for `Calc` all three strategies generate 100 % valid sentences. This is because the approximation we applied to determine whether a sentence is valid or not relies on the SUT throwing custom defined exceptions, while in the case of `Calc` no custom exceptions are defined. Furthermore, the language defined by the grammar has a relatively simple structure and malformed sentences generated by the considered techniques are quite rare, and whenever they are found, they manifest themselves as standard Java exceptions (e.g NullPointerException). We manually verified that the proportion of such exceptions is negligible and does not affect the results in any substantial way.

We can notice that AN is effective in generating the highest proportion of valid sentences in the larger subjects (both in terms of grammar size and LOC). AN generates also the largest proportion of duplicates for such subjects (see Table 3, columns 2-3-4). We argue that annotations introduce constraints that on one hand ensure the generation of valid sentences almost always, but on the other hand they limit the possibility of exploring alternative generation paths, hence resulting in repeated generation of the same sentences.

We can answer **RQ1** positively. The annotation scheme significantly increases the proportion of semantically valid sentences as compared to learning in four out of five subjects (in one subject all techniques behave the same).

Table 4 Branch coverage and p-values obtained from the Wilcoxon test comparing LRN and AN. Statistically significant values (at significance level 0.05) are shown in boldface under columns LRN and AN; the highest values are in gray background

	RANDOM				GP			
Subject	**8020**	**LRN**	**AN**	**p**	**8020**	**LRN**	**AN**	**p**
Calc	80.57%	80.24%	**80.57%**	0.0054	80.57%	80.38%	80.67%	0.0587
Basic	66.24%	**67.96%**	56.94%	0.0002	67.38%	**69.14%**	67.90%	0.0098
Kahlua	77.92%	**78.85%**	75.57%	0.0002	78.95%	79.50%	79.83%	0.2253
MDSL	73.32%	**72.58%**	60.47%	0.0002	79.99%	78.62%	**79.33%**	0.0028
Javascal	8.49%	18.42%	19.92%	0.1730	12.09%	19.59%	22.21%	0.1770
Rhino	25.86%	23.03%	**25.76%**	0.0002	48.81%	46.35%	**49.16%**	0.0001

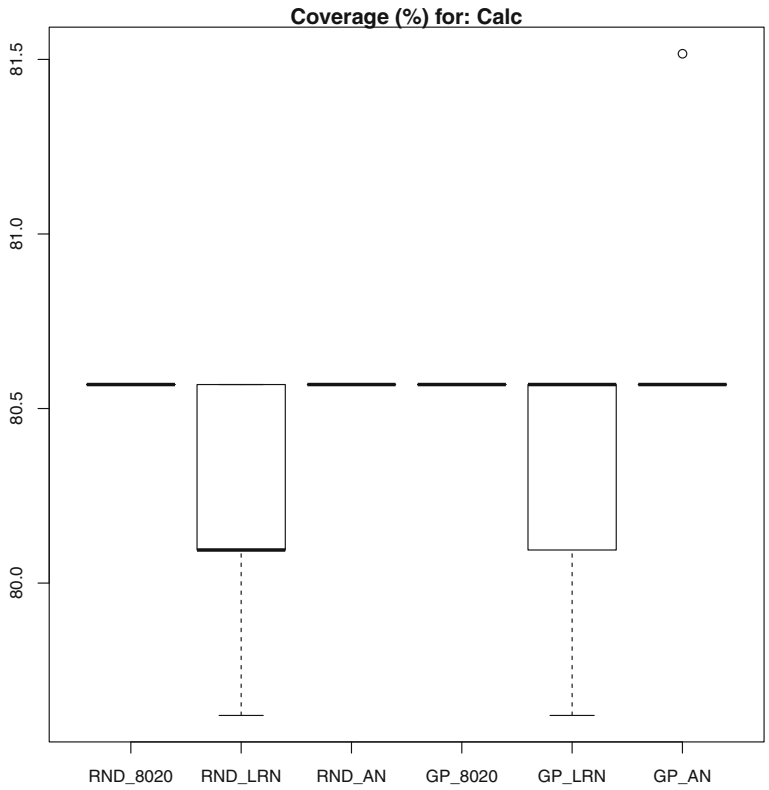


Fig. 9 Coverage box plots under various configurations for the subject `Calc`

The level of branch coverage achieved by each technique is shown in Table 4; together with the results of the Wilcoxon test of significance (p-values) comparing LRN and AN (under column p). The corresponding box plots are shown in Fig. 9 for the subject `Calc`, Fig. 10 for the subject `Basic`, Fig. 11 for the subject `Kahlua`, Fig. 12 for the subject `MDSL`, Fig. 13 for the subject `Javascal`, and Fig. 14 for the subject `Rhino`. These results confirm the findings reported in our previous work (Kifetew et al. 2014), where in most cases the combination of GP and grammar learning outperforms and in a few cases it is equivalent to the baseline 80/20 random generator.

Let us consider GP-based test case generation, which results in the highest coverage levels reached on all subjects. With the exception of `MDSL`, on which 80/20 is marginally but not significantly better, on all the other subjects either learning (GP_LRN) or annotations (GP_AN) give the highest coverage. Specifically, for subject `Basic`, LRN gives significantly higher coverage than AN. For the other subjects, AN gives either significantly better coverage (this is the case of `Rhino` and `MDSL`) or better coverage without statistical significance (this is the case of `Calc`, `Kahlua` and `Javascal`) as compared to LRN.

When combined with random test case generation, all strategies achieve coverage scores which are lower than the same strategy combined with GP. It can be noticed that, within the random generation strategy, LRN achieves better coverage than AN on three subjects (`Basic`, `Kahlua`, and `MDSL`) while AN achieves better coverage on the remaining

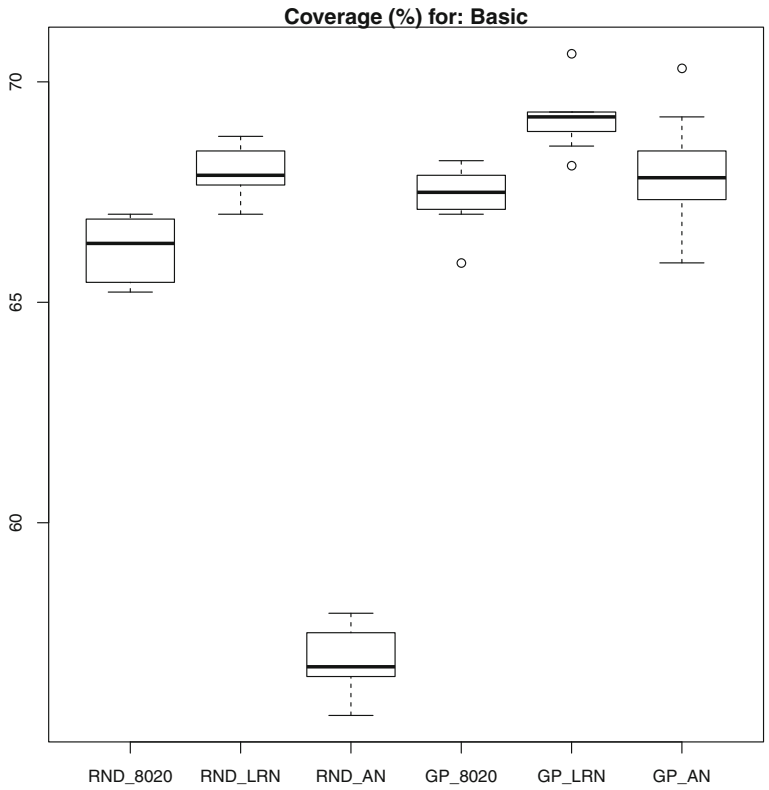


Fig. 10 Coverage box plots under various configurations for the subject `Basic`

three, although in the case of `Javascal` the difference is not statistically significant. The baseline (80/20) achieves a (relatively) high coverage on `Calc`, `MDSL` and `Rhino`.

We can notice that on `Calc` all techniques achieve similar levels of coverage. Since this subject is the smallest one, both in terms of source code and grammar, it is relatively easy for all techniques to reach quickly maximum coverage.

For what concerns **RQ2**, GP search combined with any of the three strategies (80/20, LRN, AN) outperforms random search. Within GP, using either LRN or AN gives higher coverage in five out of the six subjects, with just a marginal difference on the sixth subject. Hence we can answer positively **RQ2**, i.e., the combination of GP with either annotations (AN) or learning (LRN) increases coverage.

In Table 5 we report the overlaps in the branches covered by GP when combined with annotations and learning. Specifically, column *AN* shows the number of branches covered by the sentences generated by GP with annotations, column *LRN* shows the number of branches covered by the sentences generated by GP with learning. Column $AN \cap LRN$ shows the number of branches covered by both (intersection). Columns $AN \setminus LRN$ and $LRN \setminus AN$ show the differences, i. e., branches covered by one but not by the other. Column *Similarity* shows the Jaccard similarity between the set of branches covered by the two strategies, computed as $|AN \cap LRN|/|AN \cup LRN|$.

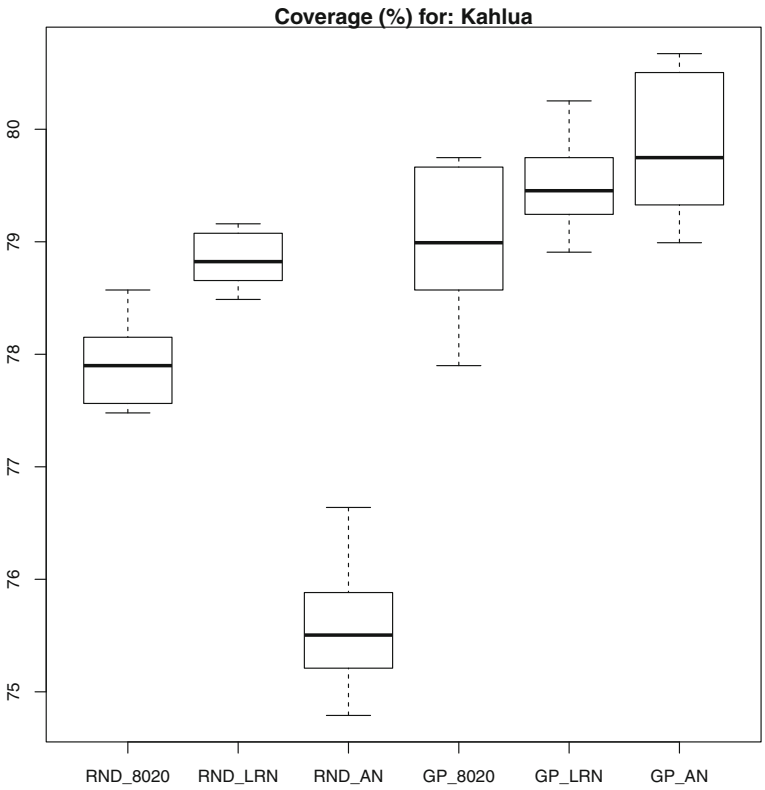


Fig. 11 Coverage box plots under various configurations for the subject `Kahlua`

From Table 5, we can see that the similarity between the set of branches covered by GP with either annotations or learning is quite high in the majority of the subjects. In some of the subjects, in particular `Rhino` and `Kahlua`, the number of branches covered by annotations but not by learning (column *AN*∖*LRN*) is relatively high (in comparison with column *LRN*∖*AN*). For the rest of the subjects, differences (*AN*∖*LRN*, *LRN*∖*AN*) are small and comparable, with the exception of `Javascal`, for which LRN and AN are largely complementary (with Jaccard index equal to 0.57 only and set differences of size 60, 61).

For what concerns **RQ3**, AN achieves the highest coverage (either statistically significantly or not) in four out of six subjects while LRN gives the highest coverage in one case. Hence, in terms of coverage AN is slightly superior to LRN. However, since the number of statistically significant improvements is low (2 in favor of AN and 1 in favor of LRN), the gap may not be meaningful. Hence we can answer **RQ3** as follows:

> For what concerns **RQ3**, the statistical evidence is not strong enough to conclude that either combination (i.e., GP_AN or GP_LRN) is better than the other. Both are however better than the baseline (GP_80/20). Furthermore, the overlap between the set of branches covered by GP_AN and GP_LRN is high, with just a few cases in which GP_AN covers a large number of branches not covered by GP_LRN.

To cope with the resource intensive nature of mutation analysis, we carried out experiments on a selected subset of classes from each SUT. In particular, we selected classes that

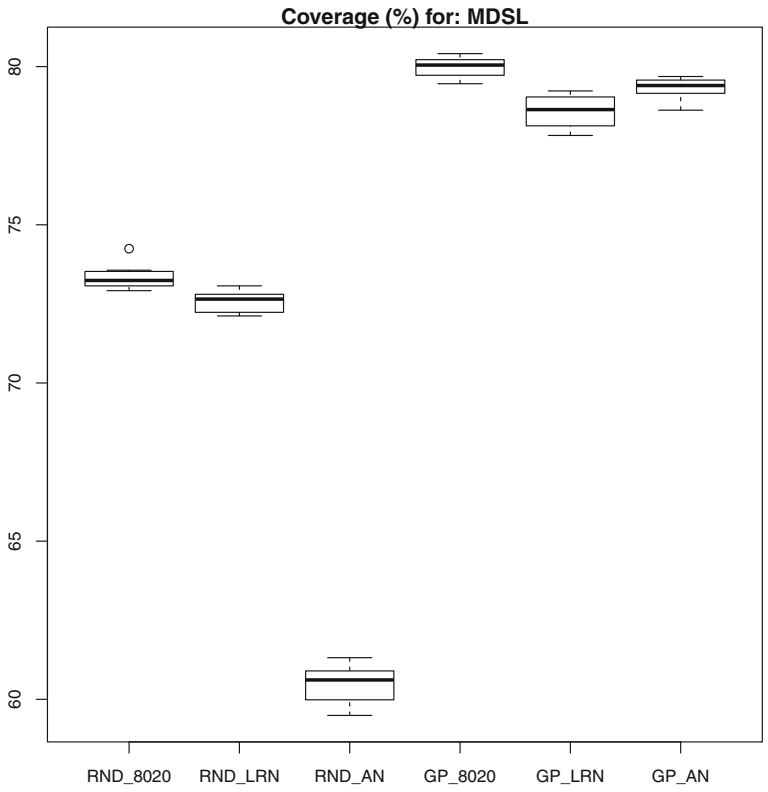


Fig. 12 Coverage box plots under various configurations for the subject `MDSL`

are involved in deep computations inside the SUT. This means that to reach these classes the input must be well formed and meaningful. For instance, in `Rhino` the input JavaScript program needs to pass lexical and syntax checking before reaching the `Interpreter` or `CodeGenerator`.

Table 6 reports the mutation scores. The mutation score for a given test suite is computed as the proportion of mutants killed by the suite out of the total number of mutants. We generated a maximum of 100 mutants per class. For each SUT we used the test suites resulting from the 10 executions carried out to measure coverage. The values reported in Table 6 are averages over the 10 test suites. We do not report mutation scores for the subject `Javascal` because the mutation tool we used was not able to measure the mutation score for this particular subject.

From the results in Table 6 we can see that there is a statistically significant difference between AN and LRN only in a few cases. Specifically, in 7 cases AN is superior (see boldface values under RND_AN or GP_AN in Table 6), while in 5 cases LRN has significantly higher mutation score. In the majority of the cases (10 remaining cases), the two strategies (AN vs. LRN) achieve comparable levels of mutation scores. For what concerns the benefits coming from GP, we can notice that they become visible, in terms of mutation score, as the size and grammar complexity of the subjects increase (see highest mutation scores, shown in gray background in Table 6, where subjects are sorted by increasing size/grammar complexity).

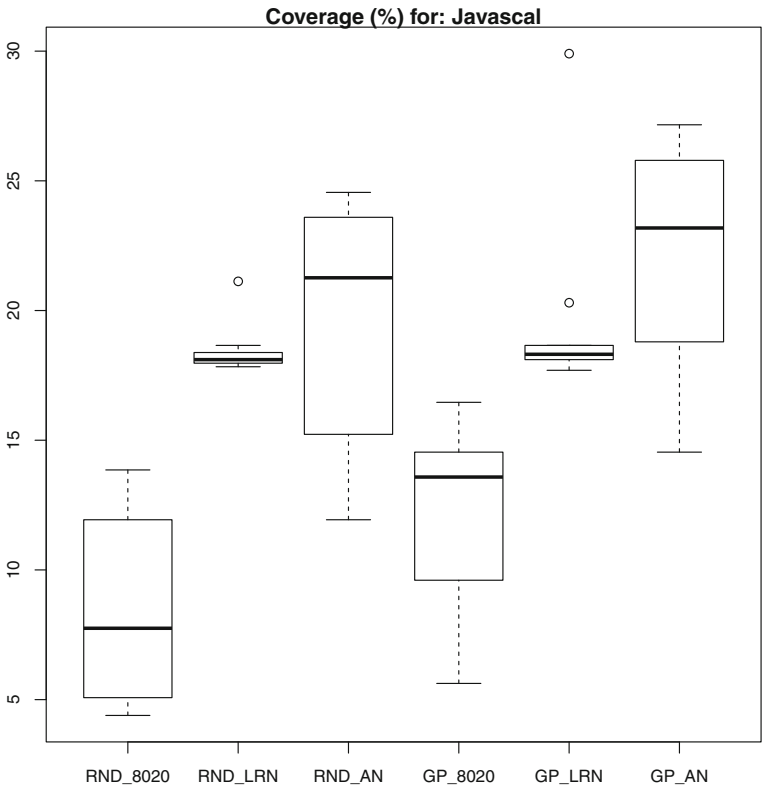


Fig. 13 Coverage box plots under various configurations for the subject `Javascal`

Regarding **RQ4**, mutation analysis shows no significant difference between annotations and learning. There is no statistically significant difference in most cases, while the remaining cases are split almost evenly between AN and LRN (7 vs. 5).

In Table 7, we report the average coverage achieved by RND and GP for each subject when a given percentage of annotations is dropped. Table 7 shows also the delta in pp (*percentage points*, i.e., difference between two percentage values) for each configuration (as compared to the baseline in which no annotations are dropped). In the majority of the cases, the variability in coverage (either positive or negative) is very low, mostly below 1pp. However, there are a few cases where dropping annotations has a noticeable effect on coverage. In particular for the subject `Javascal`, we observe a decrease in coverage as high as 5.33pp for GP, while for RND we observe an increase (up to 1.32pp) in coverage. This could be due to the strict structural requirements imposed by the underlying grammar (for the Pascal language). For the subject `Rhino`, we observe small decreases in coverage (up to 1pp) as the percentage of dropped annotations increases. For the subject `Basic`, RND showed an increase in coverage (up to 3.36pp) while variations in the coverage of GP remained quite low.

We can also observe from Table 7 that quite surprisingly some of the deltas are positive. This may indicate that the set of annotations used in the experiments may at times be too restrictive, hence preventing the techniques from exploring the space of sentences that could

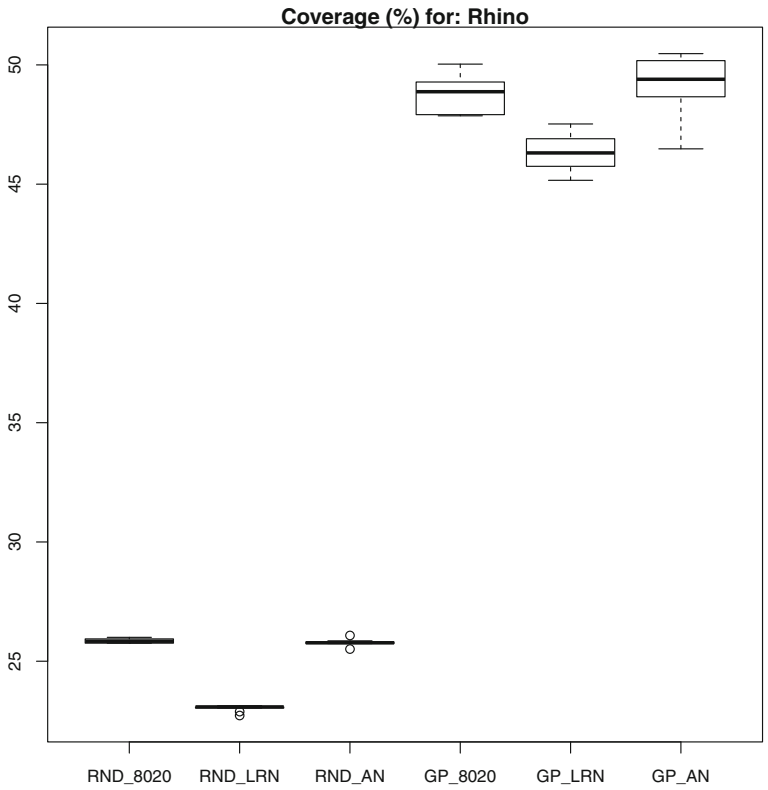


Fig. 14 Coverage box plots under various configurations for the subject `Rhino`

be derived from the grammar. Hence, dropping annotations in these cases led to increased coverage.

> Regarding **RQ5**, we can conclude that for the majority of the subjects on which we experimented, the performance of the stochastic grammar is robust with respect to annotation changes.

Table 5 Branches covered by AN and LRN: intersection, differences and similarity (Jaccard index)

Subject	Branches	AN	LRN	AN ∩ LRN	AN ∖ LRN	LRN ∖ AN	Similarity
Calc	211	172	170	170	2	0	0.99
Basic	906	653	657	639	14	18	0.95
Kahlua	1190	977	968	964	13	4	0.98
MDSL	2629	2162	2160	2112	50	48	0.96
Javascal	729	218	219	158	60	61	0.57
Rhino	6138	3249	2984	2953	296	31	0.90

Table 6 Mutation scores achieved by RND and GP when using annotations (AN) and learning (LRN). Significantly better values are shown in boldface; the highest values are in gray background

Subject	class	RND_LRN	RND_AN	p	GP_LRN	GP_AN	p
Calc	**CalcLexer**	79.98%	**86.12%**	0.0040	**77.40%**	75.12%	0.0492
Calc	**CalcParser**	65.60%	66.40%	0.9093	62.90%	56.90%	0.9698
Basic	**Expression**	**9.18%**	7.81%	0.0281	9.59%	**12.05%**	0.0044
Basic	**Program**	8.00%	10.10%	0.1241	8.60%	9.00%	0.5783
Kahlua	**LexState**	59.70%	58.30%	0.0589	58.30%	58.90%	0.9382
Kahlua	**LuaState**	36.90%	38.30%	0.1463	32.70%	**36.70%**	0.0133
MDSL	**Dispatcher**	**57.83%**	44.29%	0.0002	56.42%	56.37%	1.0000
MDSL	**MiniLexer**	48.95%	49.64%	0.4454	51.75%	51.59%	0.8498
MDSL	**MiniParser**	**57.98%**	40.54%	0.0003	**58.50%**	54.85%	0.0398
Rhino	**CodeGenerator**	22.40%	**29.90%**	0.0001	54.60%	**59.30%**	0.0006
Rhino	**Interpreter**	12.20%	**16.20%**	0.0001	23.60%	**27.20%**	0.0083

5.5 Threats to Validity

The main threats to the validity of our results are related to internal, conclusion, and external validity.

Internal validity threats concern factors that may affect a dependent variable and were not considered in the study. In our case, different grammar based test data generation techniques could be used, with potentially varying effectiveness. We chose stochastic random

Table 7 Coverage achieved by RND and GP when 5, 10, 15, 20, and 25 % of the annotations are dropped. The corresponding loss or gain in coverage is also shown; values above 1pp are in highlighted background

Subject	Config	0%	5%	Δ(pp)	10%	Δ(pp)
Basic	**GP**	67.90	67.79	-0.11	67.37	-0.53
	RND	56.94	56.47	-0.47	58.25	1.30
Kahlua	**GP**	79.83	79.75	-0.08	79.34	-0.49
	RND	75.57	75.21	-0.36	75.21	-0.36
MDSL	**GP**	79.33	79.75	0.41	79.99	0.65
	RND	60.47	60.72	0.25	60.26	-0.21
Javascal	**GP**	22.21	18.61	-3.60	22.01	-0.20
	RND	19.92	21.19	1.27	20.10	0.19
Rhino	**GP**	49.16	49.36	0.20	49.60	0.43
	RND	25.76	25.82	0.06	25.98	0.22
Average	**GP**			-0.64		-0.03
	RND			0.15		0.23

Subject	Config	0%	15%	Δ(pp)	20%	Δ(pp)	25%	Δ(pp)
Basic	**GP**	67.90	67.69	-0.21	68.12	0.22	67.49	-0.41
	RND	56.94	59.08	2.14	60.30	3.36	59.90	2.96
Kahlua	**GP**	79.83	79.82	-0.01	79.76	-0.08	79.54	-0.29
	RND	75.57	75.56	-0.01	74.96	-0.61	75.82	0.25
MDSL	**GP**	79.33	79.86	0.52	79.97	0.64	79.94	0.61
	RND	60.47	60.79	0.32	60.54	0.07	60.97	0.50
Javascal	**GP**	22.21	17.11	-5.09	16.88	-5.33	18.94	-3.26
	RND	19.92	20.88	0.96	21.23	1.32	19.19	-0.73
Rhino	**GP**	49.16	48.47	-0.69	48.66	-0.51	48.16	-1.01
	RND	25.76	25.99	0.23	25.99	0.23	25.92	0.15
Average	**GP**			-1.10		-1.01		-0.87
	RND			0.73		0.87		0.63

generation as a baseline as it is representative of state-of-the-art techniques for random grammar based test generation.

Another potential threat could arise from the fact that one of the authors performed the annotation of the grammars, simulating the developer. This may result in under utilization of the capabilities of the annotation scheme due to lack of deeper understanding of the grammars (while the developer is expected to know very well the grammar and the characteristics of the language defined by it). On the other hand, the intended semantics of the annotation scheme was perfectly clear and familiar to the author, while it might not be so for a developer who has not been adequately trained for the job. Consequently, the performance of GP combined with our own annotations might have been marginally penalized as compared to the performance achievable by a well trained developer who is very familiar with the grammar.

Threats to *conclusion validity* concern the relationship between the treatments and outcomes. In addition to reporting the results of our experiments, whenever applicable, we have also tested them for statistical significance using the Wilcoxon non-parametric test, and drew conclusions in accordance with the results of the test.

External validity threats are related to the generalizability of results. We have chosen six subjects representative of grammar based systems of various size (both in terms of code and grammar size). Even though these subjects are quite diverse, generalization to other subjects should be done with care. Replication of the experiment on more subjects would further increase our confidence in the generalizability of the results.

6 Conclusions and Future Work

In this paper, we have compared two approaches for grammar-based test data generation. One approach combines GP with stochastic grammars learned from a corpus of manually written sentences. Learned stochastic grammars are effective in controlling recursion during sentence derivation and in promoting sentence structures similar to the valid sentences used for learning. However, learning comes with added costs, mainly associated with finding an appropriate corpus and adapting the grammar to the learning tool. The second approach combines GP with semantically annotated grammars. Annotations constrain the generation process so that only semantically valid sentences are produced.

Experimental results obtained on six grammar based systems of varying grammar complexities show that the two approaches are comparably effective, both in terms of coverage and fault exposure capability. Moreover, both approaches achieve maximum effectiveness when combined with GP. Depending on the difficulty of finding an appropriate corpus of valid sentences, as compared to the effort necessary for grammar annotation, developers may decide to adopt one or the other alternative approach. According to our empirical data, such a choice has minimal impact on the final coverage and fault detection capability of the resulting test suites, provided the test suite generation process is based on the GP evolutionary scheme.

Considering the complexity of the subject programs, the advantage of annotations over learned probabilities tends to increase as the complexity of the associated subject (and hence the grammar) increases. In our experiments, on the largest subject, `Rhino`, annotations gave significantly better results, both in terms of branch coverage and mutation killing. On the other hand, for the simplest subject, `Calc`, all techniques perform equally well. Consequently, developers may consider using annotations if the inputs to the system under

test are governed by a complex grammar (e.g. JavaScript) without risking a decrease in effectiveness. Moreover, since annotations could be iteratively improved, developers could start with an initial set of annotations and incrementally improve them, based on the resulting coverage or mutation score, until a satisfactory adequacy level is reached.

In future work, we will investigate mechanisms to promote diversity during sentence derivation from annotated grammars. In fact, preliminary observations (see, for instance, results on unique vs. duplicate sentences) seem to indicate that annotations tend to over-constrain the generation process, reducing the exploration of alternative sentence structures. We intend to introduce some form of history, recording the derivation choices made in the past, so as to promote different choices whenever possible. Furthermore, it could also be interesting to investigate whether there is any advantage in combining grammar annotations with probabilities learned from a corpus.

Another aspect of this work that we plan to address in our future work concerns the task of annotating grammars. In our empirical study (for research question RQ5), we tried to simulate how developers of varying level of expertise could perform the task of annotating a grammar. Consequently, from the outcome of the experiments, we got an insight into the robustness of the annotation scheme with respect to the variability of annotations. While this experiment serves as a starting point towards understanding the effect of variability from the annotator, better insight could be obtained by conducting a user study in which real world developers participate by annotating grammars.

Finally, we also plan to extend our empirical benchmark by including additional subjects, so as to further increase the external validity of our findings.

References

Arcuri A, Iqbal MZ, Briand L (2010) Formal analysis of the effectiveness and predictability of random testing. In: Proceedings of the 19th international symposium on software testing and analysis, ISSTA '10. ACM, New York, pp 219–230. doi:10.1145/1831708.1831736

Beyene M, Andrews JH (2012) Generating string test data for code coverage. In: Proceedings of the international conference on software testing, verification, and validation (ICST), pp 270–279

Booth TL, Thompson RA (1973) Applying probability measures to abstract languages. IEEE Trans Comput 100(5):442–450

Claessen K, Hughes J (2011) Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices 46(4):53–64

Duchon P, Flajolet P, Louchard G, Schaeffer G (2004) Boltzmann samplers for the random generation of combinatorial structures. Comb Probab Comput 13(4–5):577–625

Feldt R, Poulding S (2013) Finding test data with specific properties via metaheuristic search. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE). IEEE, pp 350–359

Fraser G, Arcuri A (2011) Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, ESEC/FSE '11. Szeged, Hungary, pp 416–419

Fraser G, Arcuri A (2013) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291. doi:10.1109/TSE.2012.14

Godefroid P, Kiezun A, Levin MY (2008) Grammar-based whitebox fuzzing. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI), pp 206–215

Grune D, Jacobs CJH (1990) Parsing techniques: a practical guide. Ellis Horwood Limited, Chichester

Guo HF, Qiu Z (2014) A dynamic stochastic model for automatic grammar-based test generation. Software: Practice and Experience

Hennessy M, Power JF (2005) An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, ASE '05. ACM, New York, pp 104–113. doi:10.1145/1101908.1101926

Kifetew FM, Tiella R, Tonella P (2014) Combining stochastic grammars and genetic programming for coverage testing at the system level. In: Proceedings of the 6th international symposium on search-based software engineering (SSBSE), pp 138–152

Lari K, Young SJ (1990) The estimation of stochastic context-free grammars using the inside-outside algorithm. Comput Speech Lang 4(1):35–56

Majumdar R, Xu RG (2007) Directed test generation using symbolic grammars. In: Proceedings of the 22nd IEEE/ACM international conference on automated software engineering (ASE), pp 134–143

Maurer PM (1990) Generating test data with enhanced context-free grammars. IEEE Softw 7(4):50–55

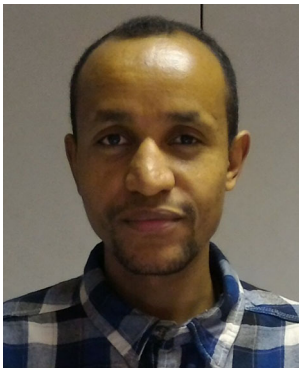
McKay RI, Hoai NX, Whigham PA, Shan Y, O'Neill M (2010) Grammar-based genetic programming: a survey. Genet Program Evolvable Mach 11(3–4):365–396

McMinn P (2004) Search-based software test data generation: a survey. J Softw Test Verification and Reliability (STVR) 14:105–156

Pargas R, Harrold MJ, Peck R (1999) Test-data generation using genetic algorithms. J Softw Test Verification and Reliability (STVR) 9:263–282

Poulding S, Alexander R, Clark JA, Hadley MJ (2013) The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing. In: Proceedings of the 15th annual conference on genetic and evolutionary computation, GECCO '13. ACM, New York, pp 1477–1484. doi:10.1145/2463372.2463550

Purdom P (1972) A sentence generator for testing parsers. BIT Numer Math 12:366–375. doi:10.1007/BF01932308



Fitsum Meshesha Kifetew received his BSc and MSc degrees in Computer Science from Addis Ababa University, Ethiopia in 2003 and 2005 respectively. From 2005 to 2009 he worked as a lecturer at the Addis Ababa University. In 2009 he joined the Software Engineering unit at Fondazione Bruno Kessler (FBK) in Trento, Italy as a software engineer. In 2011 he started studying at the University of Trento towards a PhD degree, sponsored by FBK. In 2015 he obtained the PhD degree after defending his thesis "Evolutionary Test Case Generation via Many Objective Optimization and Stochastic Grammars".

He is currently a postdoctoral researcher in the Software Engineering unit at FBK. His research focuses on applying search-based techniques to software engineering problems, such as test case generation. He served as program committee member of the Symposium on Search-Based Software Engineering (SSBSE) and as reviewer for the Journal of Software Testing, Verification and Reliability (STVR).

Roberto Tiella received his MSc degree in Mathematics from the University of Trento, Italy, in 1993. He is a researcher in the Software Engineering unit at Fondazione Bruno Kessler (FBK) since 2003. Previously he had a ten years long industrial experience in software development. His main publications are in the areas of recommender systems, software metrics, formal verification and evaluation, code generation. His recent research interests include testing transformations for model-based testing; testing techniques for grammar-based software systems; data obfuscation.

Paolo Tonella is head of the Software Engineering Research Unit at Fondazione Bruno Kessler (FBK), in Trento, Italy. He is also Honorary Professor at University College London (UCL). He received his PhD degree in Software Engineering from the University of Padova, in 1999, with the thesis "Code Analysis in Support to Software Maintenance". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is the author of "Reverse Engineering of Object Oriented Code", Springer, 2005, and of "Evolutionary Testing of Classes", ISSTA 2004. He participated in several industrial and EU projects on software analysis and testing. His H-index (according to Google Scholar) is 42.

Paolo Tonella was Program Chair of ICSM 2011 and ICPC 2007; General Chair of ISSTA 2010 and ICSM 2012. Among the others, he served in the program committees of ICSE, FSE, ICSM, ISSTA, ICST. He was ranked among the top-50 Software Engineering scholars by Communications of the ACM, vol. 50, n. 6, 2007. He is associate editor of TSE and he is in the editorial board of EMSE and JSEP. His current research interests include code analysis, web and object oriented testing, search based test case generation.