Grammar-Based Test Generation with YouGen

Daniel Hoffman¹, David Ly-Gagnon¹, Paul Strooper², and Hong-Yi Wang¹

¹ Dept. of Computer Science, University of Victoria, PO Box 3055 STN CSC, Victoria, B.C. V8W 3P6, Canada
² The University of Queensland, School of Inf. Tech. and Elec. Eng., Brisbane, Qld. 4072, Australia

SUMMARY

Grammars are traditionally used to recognize or parse sentences in a language, but they can also be used to generate sentences. In grammar-based test generation (GBTG), context-free grammars are used to generate sentences that are interpreted as test cases. A generator reads a grammar $G$ and generates $L(G)$, the language accepted by the grammar. Often $L(G)$ is so large that it is not practical to execute all of the generated cases. Therefore, GBTG tools support “tags”: extra-grammatical annotations which restrict the generation. Since its introduction in the early 1970s, GBTG has become well-established: proven on industrial projects and widely published in academic venues. Despite the demonstrated effectiveness, the tool support is uneven; some tools target specific domains, e.g., compiler testing, while others are proprietary. The tools can be difficult to use and the precise meaning of the tags are sometimes unclear. As a result, while many testing practitioners and researchers are aware of GBTG, few have detailed knowledge or experience.

We present YouGen, a new GBTG tool supporting many of the tags provided by previous tools. In addition, YouGen incorporates covering-array tags, which support a generalized form of pairwise testing. These tags add considerable power to GBTG tools and have been available only in limited form in previous GBTG tools. We provide semantics for the YouGen tags using parse trees and a new construct, generation trees. We illustrate YouGen with both simple examples and a number of industrial case studies.

KEY WORDS: Automated testing, grammar-based test generation, covering array, firewall testing, RSS, XML

1. INTRODUCTION

With grammar-based test generation (GBTG), test cases are specified with a context-free grammar $G$. A generator reads $G$ and uses derivations from the rules of $G$ to produce $L(G)$, the language accepted by $G$. Because $L(G)$ is often too large, GBTG tools support “tags”: extra-grammatical annotations which restrict the generation. GBTG is well-suited to generating...
structured test inputs, e.g., programs for compiler testing [1], packets for protocol testing [2], and XML files for web testing [3].

Since its introduction in the early 1970s [4], GBTG has become well-established. Despite the demonstrated effectiveness, however, the tool support is uneven; some tools target specific domains, e.g., Java machine code generation [5], while other tools are proprietary [6]. Some tools are hard to use except by the tool developers. In particular, the precise meaning of the tags is sometimes unclear. As a result, while many testing practitioners and researchers are aware of GBTG, few have detailed knowledge or experience.

We present YouGen [7], a new GBTG tool supporting the best of the features provided by previous tools. We divide the features into four groups:

1. Derivation-limiting tags reduce the generated output by limiting the number of derivations or by setting limits on derivation length.
2. Covering-array tags select derivations using a generalized form of pairwise testing [8].
3. Procedural code combined with grammar rules permits snippets of code in a procedural language to be attached to grammar rules.
4. Generation tree logging and viewing aids grammar development by logging generation trees to files and providing GUI support for tree traversal.

The covering-array tags are particularly useful, adding considerable power to a GBTG tool.

It is important to have predictable semantics for the tool features. Otherwise, tool users are forced to use trial and error. The semantics are also needed to control tool development. Implementing any one feature is straightforward; an implementation where the features work well in combination is much harder. We provide precise prose semantics for the YouGen tags using conventional parse trees and a new construct: generation trees. We also provide tool support for logging and viewing of generation trees, so that users can accurately predict and control grammar outputs.

In Section 2, we describe generation in a simplified setting, using context-free grammars without tags. We also present the fundamentals of traditional covering arrays followed by the generalization we use: mixed-strength covering arrays. In Section 3, we present the YouGen tool using simple examples.

To apply GBTG effectively, tool support alone is insufficient: standard techniques and examples are needed. Grammar developers need answers to questions such as:

1. How should the tags be used in combination?
2. What work should be done by the grammar versus the embedded code?
3. Should the grammar execute the test cases immediately or save them for later execution?
4. How do I debug a complex grammar?

Sections 4 and 5 address these questions. In Section 4, we use YouGen examples to present three GBTG techniques. In Section 5, we describe our experience in applying YouGen to test suites for (1) an XML web application, (2) XPath interpreters, (3) XPath/XQuery performance testing, and (4) an Internet firewall. In Section 6, we present the related work in GBTG and covering arrays.
2. BACKGROUND

2.1. Context-free grammars

A context-free grammar consists of a set of rules [9]. Rule syntax is simple: each rule has a single nonterminal (variable) on the left-hand side and a list of nonterminals or terminals (quoted strings) on the right-hand side. A derivation is a step-by-step application of the grammar rules using a simple substitution strategy:

1. select a nonterminal \( N \),
2. find a rule \( R \) with \( N \) on the left, and
3. replace \( N \) with the right-hand side of \( R \).

We illustrate the main concepts with a simplified test suite for software providing Internet telephony services. Suppose that field error reports have shown the software to be sensitive to combinations of the operating system versions used by the calling phone, the VoIP server, and the called phone. Figure 1(a) shows a grammar in which each test case is a CallerOS/ServerOS/CalleeOS triple.

Figure 2(a) shows a four-step derivation from the grammar in Figure 1(a), yielding the triple shown in the first row of Figure 1(b). In each step, one nonterminal is replaced by the right-hand side of the first matching rule. Figure 2(b) shows a second derivation, identical to the first one, except that, in step 4, the second rule with CalleeOS on the left is selected, yielding the triple shown in the second row of Figure 1(b). The 12 strings comprising the language of the Call grammar are shown in Figure 1(b).

For generation, we extend the definition of \( L(G) \) slightly. A GBTG tool takes a context-free grammar \( G \) and a nonterminal \( N \), and produces \( L(G; N) \): the set of all strings which can be derived from \( N \) using the rules of \( G \). If \( N \) is omitted, then the nonterminal on the left-hand side of the first rule is assumed.

GBTG tools operate by generating one derivation for each element of \( L(G; N) \). Two important aspects of derivations are not completely determined by the grammar:

1. In a sentential form with two or more nonterminals, which one is selected for replacement?
2. If nonterminal \( N \) is selected and there are two or more rules with \( N \) on the left-hand side, which rule is selected for the substitution?

While the answers to these questions are somewhat arbitrary, there are two important concerns in the GBTG context:

1. In many grammars, there are strings which have more than one derivation. A naive generation algorithm will generate duplicate test cases.
2. It is useful to the grammar developer if the generation order is repeatable and predictable.

As an example, the following strategy satisfies these concerns: (1) select the leftmost nonterminal for replacement and (2) select the rule alternatives in the order they appear textually in the grammar.

Figure 3 contains pseudocode for an algorithm following the generation strategy just described. The second parameter \( S \) in \( \text{generate}(G, S) \) is a sentential form: a sequence of
Call ::= CallerOS ServerOS CalleeOS;

CallerOS ::= 'Mac';
CallerOS ::= 'Win';
ServerOS ::= 'Lin';
ServerOS ::= 'Sun';
ServerOS ::= 'Win';
CalleeOS ::= 'Mac';
CalleeOS ::= 'Win';

(a) Call grammar; Call is the start nonterminal

<table>
<thead>
<tr>
<th>CallerOS</th>
<th>ServerOS</th>
<th>CalleeOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lin</td>
<td>Mac</td>
</tr>
<tr>
<td>2</td>
<td>Lin</td>
<td>Win</td>
</tr>
<tr>
<td>3</td>
<td>Sun</td>
<td>Mac</td>
</tr>
<tr>
<td>4</td>
<td>Sun</td>
<td>Win</td>
</tr>
<tr>
<td>5</td>
<td>Win</td>
<td>Mac</td>
</tr>
<tr>
<td>6</td>
<td>Win</td>
<td>Win</td>
</tr>
<tr>
<td>7</td>
<td>Lin</td>
<td>Mac</td>
</tr>
<tr>
<td>8</td>
<td>Lin</td>
<td>Win</td>
</tr>
<tr>
<td>9</td>
<td>Sun</td>
<td>Mac</td>
</tr>
<tr>
<td>10</td>
<td>Sun</td>
<td>Win</td>
</tr>
<tr>
<td>11</td>
<td>Win</td>
<td>Mac</td>
</tr>
<tr>
<td>12</td>
<td>Win</td>
<td>Win</td>
</tr>
</tbody>
</table>

(b) Language of Call grammar

Figure 1. Call grammar and its language

terminals and nonterminals. In Figure 3, substitute(S[i], R) returns a new sentential form with S[i] replaced by the right-hand side of R.

A generation tree displays multiple derivations in a single tree. Each node contains a list of terminals and nonterminals. Each path from the root to a leaf corresponds to a derivation. The two derivations in Figure 2 are shown in generation-tree form in Figure 4. From the perspective of generation trees, the algorithm in Figure 3 operates by performing a depth-first traversal of the generation tree whose leftmost two paths are shown in Figure 4.

Recursive grammars are useful for generating variable-length lists of items. In a recursive grammar, there are an infinite number of derivations. Figure 5(a) shows a simple recursive grammar. Figure 5(b) shows a generation tree containing the first three derivations produced by the algorithm in Figure 3.

Copyright © 2009 John Wiley & Sons, Ltd.
Prepared using speauth.cls
Call → CallerOS ServerOS CalleeOS
  ⇒ 'Mac' ServerOS CalleeOS
  ⇒ 'Mac' 'Lin' CalleeOS
  ⇒ 'Mac' 'Lin' 'Mac'

(a) Derivation of 'Mac' 'Lin' 'Mac'

Call → CallerOS ServerOS CalleeOS
  ⇒ 'Mac' ServerOS CalleeOS
  ⇒ 'Mac' 'Lin' CalleeOS
  ⇒ 'Mac' 'Lin' 'Win'

(b) Derivation of 'Mac' 'Lin' 'Win'

Figure 2. Derivations from the Call grammar

generate_language (G, S)
  index = position of leftmost nonterminal in S
  if index = -1
    yield S
  else
    for each rule matching S[index] in G
      generate_language (G, substitute(S,index,rule))

Figure 3. Generation pseudocode

2.2. Covering arrays and context-free grammars

With covering arrays, a test template is developed with $N$ parameters, each with a finite number of values. The requirements for the template are simple: it must allow each test case to be modeled as an $N$-tuple. The test space is the cross product of the parameters; each $N$-tuple in the test space specifies a test case. Usually the test space is so large that it is not practical to execute all the test cases. A covering-array algorithm takes a list of parameters and generates a subset of the test space which is much smaller and is still effective in revealing errors.

Revisiting the example shown in Figure 1 from the covering-array perspective, there are three parameters: CallerOS, ServerOS, and CalleeOS. Figure 1(b) shows the test space. Taken together, rows 1, 2, 6, 8, 9, and 11 are a two-cover of the test space [8]. This means that every pairwise combination of parameter values is present in at least one of those rows. To demonstrate this, we must consider the combinations of each of
Call

CallerOS ServerOS CalleeOS

'Mac' ServerOS CalleeOS

'Mac' 'Lin' CalleeOS

'Mac' 'Lin' 'Mac' 'Mac' 'Lin' 'Win'

Figure 4. Generation tree for Call grammar

zeros ::= '0';
zeros ::= '0' zeros;

(a) The Zeros grammar

(b) Generation tree for Zeros

Figure 5. The Zeros grammar
the three pairs of parameters. Consider the cross product of CallerOS and CalleeOS: \( \{ \text{Mac, Mac}, \text{Mac, Win}, \text{Win, Mac}, \text{Win, Win} \} \). The first pair is found in row 1 of Figure 1(b). The other pairs are found in rows 4, 8, and 9 of Figure 1(b). To complete the demonstration, the same exercise must be carried out for CallerOS \times ServerOS and for ServerOS \times CalleeOS.

Figure 1 shows the important connection between context-free grammars and covering arrays. A grammar rule specifies a cross product. For example, the first rule in Figure 1(a) specifies that the language of the grammar is the cross product of three sets: the strings which can be derived from (1) CallerOS, (2) ServerOS, and (3) CalleeOS. A covering array is a subset of the cross product of a list of sets. Therefore, it is meaningful to introduce a covering-array tag: a tag which specifies a subset of the cross product specified by the rule. While precise definitions for such tags are still needed, the opportunity is clear.

2.3. Mixed-strength covering arrays

Compared to traditional pairwise tests, mixed-strength covering arrays are more general in two ways:

1. **Strength.** Suppose that there are \( N \) parameters. With a pairwise test, i.e., a covering array of strength two, we consider all pairs of the \( N \) parameters. If the strength is 3, we consider all triples of parameters, etc. For the italicized rows in Figure 1(b), for example, the strength is two, so we must consider all pairs of the three parameters.

2. **Scope.** With a pairwise test, all parameters are included. In a mixed-strength covering array, the scope may specify that only a subset of the parameters be included.

With mixed-strength covering arrays, multiple specifications are permitted, each with a different strength and each applied to a subset of the parameters.

More formally, given a non-empty list of parameters \( P_0, P_1, \ldots, P_{N-1} \), where each \( P_i \) is a non-empty, finite set:

- An **index set** is a subset of \( \{0, 1, \ldots, N - 1\} \).
- A **coverage specification** is a term of the form \((I, n)\) where \( I \) is an index set and \( n \in [1..|I|] \). \( I \) denotes the **scope** and \( n \) the **strength** of the coverage specification.
- Test set \( T \subseteq P_0 \times P_1 \times \ldots \times P_{N-1} \) satisfies coverage specification \((I, n)\) if, for each index set \( I_0 \subseteq I \) of size \( n \), extracting just the columns named in \( I_0 \) from \( T \) yields the cross product of the parameters in \( I_0 \). Formally:

\[
\forall I_0 \subseteq I, I_0 = \{i_0, i_1, \ldots, i_{n-1}\} \quad \pi_{i_0, i_1, \ldots, i_{n-1}}(T) = P_{i_0} \times P_{i_1} \times \cdots \times P_{i_{n-1}}
\]

where \( \pi \) is the projection operator from relational algebra.

Using the mixed-strength notation just introduced, the two-cover shown in the italicized rows of Figure 1(b) satisfies the specification \(([0, 1, 2], 2)\). Here, \( P_0 \) is CallerOS, \( P_1 \) is ServerOS and \( P_2 \) is CalleeOS, and the strength is 2.

Figure 6(a) shows a one-cover: every element of every parameter is present, but many combinations are missing. Here, the scope is again all three parameters, but the strength is 1.
Figure 6. Mixed strength-covering arrays

Figure 6(b) contains an array satisfying two mixed-strength covering-array specifications. The first specification restricts the scope to just two parameters and uses strength 2: all combinations of CallerOS and CalleeOS must be present. The second specification restricts the scope to a single parameter and uses strength 1: all elements in ServerOS must be present. Note that some elements in CallerOS, e.g., (Mac, Win), are missing.

3. THE YOUGEN TOOL

To describe grammar rules, YouGen uses the syntax shown in Figure 1(a). For generation, YouGen uses the algorithm described in the previous section. This section describes the extra-grammatical features, and the facilities for logging and later viewing of generation trees stored in XML files.

Tags are added to a grammar $G$ to reduce the size of $L(G)$ by pruning the generation tree for $G$. At this point, we extend the meaning of $L(G)$ to include tagged grammars: $L(G)$ is the language generated by the grammar rules controlled by the tags. YouGen supports a variety of tags and allows zero or more tags to be attached to each grammar nonterminal or rule. For a tag attached to a nonterminal the syntax is

\{tagName tagArguments\} nonterminal;

For a tag attached to a rule the syntax is

\{tagName tagArguments\} rule;


\begin{tabular}{|c|c|c|}
\hline
CallerOS & ServerOS & CalleeOS \\
\hline
Mac & Lin & Mac \\
Win & Sun & Win \\
Win & Win & Win \\
\hline
\end{tabular}

(a) A one-cover: satisfies $([0, 1, 2], 1)$

\begin{tabular}{|c|c|c|}
\hline
CallerOS & ServerOS & CalleeOS \\
\hline
Mac & Lin & Mac \\
Mac & Sun & Win \\
Win & Win & Mac \\
Win & Win & Win \\
\hline
\end{tabular}

(b) A mixed-strength cover: satisfies $([0, 2], 2)$ and $([1], 1)$
3.1. Derivation-limiting tags

There are three derivation-limiting tags, all of which can be attached only to nonterminals.

If the tag \{count \(C\)} is attached to nonterminal \(N\), then YouGen will not allow more than \(C\) strings to be derived for \(N\) after the invocation of a rule that has \(N\) on the right-hand side. Figure 7(a) shows the count tag in action. YouGen traverses the generation tree shown in Figure 5(b). From the first occurrence of zeros, no more than 3 strings can be generated.

If the tag \{rdepth \(D\)} is attached to a nonterminal \(N\), then YouGen will not allow more than \(D\) occurrences of \(N\) on any path in the parse tree. With the tag in Figure 7(b), YouGen generates 1 through 3 zeros. Figure 8 shows the parse tree for 3 zeros. Since zeros appears at most 3 times in all paths, not exceeding the rdepth limit of 3, 3 zeros are allowed. Extending the tree to generate 4 zeros would require a fourth occurrence of zeros.

If the tag \{depth \(D\)} is attached to a nonterminal \(N\), then YouGen will limit the parse tree below \(N\) to depth \(D\). With the tag in Figure 7(c), YouGen generates 1 through 3 zeros. Figure 8 shows the parse tree for 3 zeros. Since the parse subtree below the zeros at the root of the parse tree has depth 3, not exceeding the prescribed depth limit 3, 3 zeros are allowed. Extending the tree to generate 4 zeros would increase the tree depth to 4.

3.2. Covering-array tags

The covering-array tag can be attached only to rules. It prunes the generation tree in a different way. With the three tags just described, tree generation is halted when some limit has been exceeded, e.g., depth 5. If the cov tag is attached to a rule \(R\) for nonterminal \(N\), then a new, temporary tree is generated with \(N\) at the root and \(R\) as the rule used to generate strings.
The terminals and nonterminals on the right-hand side of $R$ are treated as the parameters to a covering-array algorithm, which is called when all the strings for the parameters have been generated.

The syntax of the cov tag is \{cov \[ P_1, \ldots, P_n \] \}, where each $P_i$ is a coverage specification.

For example, attaching a cov tag to the first rule in the grammar in Figure 1 instructs YouGen to select a subset of the 12 triples shown in Figure 1. If the tag \{cov \[ ([0,1,2],1) \] \} is used, then every element of every domain must be present, but many combinations may be missing, as shown in Figure 6(a). If the tag \{cov \[ ([0,2],2), ([1],1) \] \} is used, then all elements in CallerOS × CalleeOS and all elements in ServerOS must be present, as shown in Figure 6(b). The YouGen output for this grammar consists of 7 cases: four to satisfy \{cov \[ ([0,2],2) \] \} and three to satisfy \{cov \[ ([1],1) \] \}. The YouGen output is correct but not optimal; Figure 6(b) satisfies both specifications with only 4 cases. A general solution is an open research problem: given a list of mixed-strength covering array specifications, generate a single covering array which satisfies all of them, with as few test cases as possible.

YouGen uses a state-of-the-art algorithm [10, 11, 12] to generate two-covers efficiently. In the future, we plan to use the same algorithm for generating three-covers, but at the moment YouGen uses a less efficient hill-climbing algorithm.

### 3.3. Combining procedural code and grammar rules

Many testing tasks are difficult to express in a context-free grammar but are easy to implement in a conventional programming language, especially an interpreted language.
3.3.1. Combining procedural code and grammar rules

YouGen provides four tags for embedding Python code in a grammar. Two tags are global:

1. **global precode** is executed once before generation begins. This section often contains declarations and initializations of global variables, and system actions, such as opening files.
2. **global postcode** is executed once after generation ends and is typically used for exit tasks, such as closing a file.

The other two tags are attached to rules.

1. **precode** is executed just before each invocation of a rule. A precode block must have a boolean return value; if the return value from a precode block is false, then the rule will not be invoked. A precode block usually contains a custom guard: a logical condition which blocks rule invocation in a way not supported by the `count`, `depth`, or `rdepth` tags.
2. **postcode** is executed as soon as a string derived from a rule is available. In test suites using an online strategy [13], test execution is often performed by postcode preceding the first grammar rule. This postcode is executed once for each string in $L(G)$.

In the embedded code shown in Figure 9(a), $n$ is initialized to 0 and then incremented each time the recursive rule is invoked. Because each embedded code section contains a print statement, the output will contain a trace of the rule invocations. YouGen once again traverses the tree shown in Figure 5(b). The complete output is shown in Figure 9(b).

In **postcode**, the terminals just derived from the tagged rule are available in global variable $s$, as shown in Figure 9(a). While the terminals could be stored as a list of strings or a single concatenated string, we use a nested list form which mirrors the parse tree. For example, Figure 10(a) shows the parse tree for the derivation of ('Mac', 'Lin', 'Mac'). Figure 10(b) shows the same parse tree in nested-list form, in which a pair of square brackets is introduced whenever a nonterminal is replaced by the right-hand side of a matching rule.

The embedded code in Figure 11 shows how to simulate weighted rules, not supported by YouGen directly. Each time the rule \[ A ::= B; \] is selected, $R$ is set randomly within $[0, 1]$. If $R$ is less than 0.66, then rule \[ B ::= '0'; \] is selected; otherwise that rule is skipped and \[ B ::= '1'; \] is selected. At that point, rule \[ A ::= A; \] is selected, introducing an $A$ in the sentential form and starting the process again, 10 times in total. The net effect is to generate a random sequence of zeros and ones, with twice as many zeros as ones.

3.3.2. Grammar objects

Like many GBTG tools, YouGen uses a two-phase approach. A parser reads the grammar file and generates a Python program. When executed, that program generates the language of the grammar. YouGen defines Python classes for terminals, nonterminals, tags, generators nonterminal, rules, and grammars. The generated code contains instances of these classes and imports the generation algorithm.
\{global_precode
    \n = 0
    print 'tglobals_precode reached'
\}
\{global_postcode
    print 'tglobals_postcode reached'
\}
\{precode
    print 'tZ0 precode reached'
    return True
\}
\{postcode
    print 'tZ0 postcode:',s
\}
Zeros ::= '0';
\{precode
    global n
    print 'tZ1 precode reached'
    n += 1
    return n < 3
\}
\{postcode
    print 'tZ1 postcode:',s
\}
Zeros ::= '0' Zeros;

(a) Zeros grammar with embedded code

globals_precode reached
Z0 precode reached
Z0 postcode: ['0']

0
Z1 precode reached
Z0 precode reached
Z0 postcode: ['0']
Z1 postcode: ['0', ['0']]

0 0
Z1 precode reached
Z0 precode reached
Z0 postcode: ['0']
Z1 postcode: ['0', ['0', ['0']]]
Z1 postcode: ['0', ['0', ['0', ['0']]]]

0 0 0
Z1 precode reached

(b) Output from generation

Figure 9. Embedded code
(a) Call parse tree

```
[ [ ['Mac'], ['Lin'], ['Mac'] ] ]
```

(b) Call parse tree in nested-list form

Figure 10. Call parse tree

```
{global_precode
import random
R = None
}

{precode
global R
R = random.random()
return True
}
A ::= B;

{count 10}
A ::= A;

{precode
global R
return R < 0.66
}
B ::= '0';

{precode
global R
return R >= 0.66
}
B ::= '1';
```

Figure 11. Simulated weighted rules using `precode`
G = Grammar()
G.append_rule(Rule( { }, 'zeros', [T('0')]))
G.append_rule(Rule( { }, 'zeros', [T('0'), V('zeros')]))
G.nonterminals['zeros']['count'] = 3
generate_language(G,'zeros')

(a) Zeros grammar in API form

import zeros
G = zeros.G

for count in [5,10,15]:
    G.nonterminals['zeros']['count'] = count
    generate_language(G,'zeros')

(b) Importing and modifying the Zeros grammar

Figure 12. Zeros grammar: API interface and dynamic grammar

The Python classes serve as a target for code generation and also as an API interface to YouGen, making it possible to skip the translation phase and write the generated code by hand. For example, Figure 12(a) contains an API version of the Zeros grammar in Figure 7(a). While this grammar was written manually, it is similar to the Python code YouGen would generate from the grammar in Figure 7(a).

With grammar objects, YouGen supports dynamic grammars. Grammar object $G$ can be composed with arbitrary Python code. Figure 12(b) shows the Zeros grammar object invoked repeatedly, with increasing count value. Arbitrary changes to the grammar are permitted.

3.4. Generation tree logging and viewing

Debugging complex grammars is difficult. To assist in this task, YouGen logs the generation tree to an XML file for later analysis. Figure 13 contains the generation tree for the Zeros grammar in Figure 7(a), with count set to 1 to keep the output small.

Each $<gentree>$ element is a subtree in a generation tree. In tree node $N$:  
- the $<s>$ element contains a sentential form, representing the parse tree using nested lists,
- the $<id>$ element contains the identifier of the rule used to derive $N$’s sentential form from the sentential form in $N$’s parent ($None$ is used for the root of the tree, which does not have a parent), and
- the $<count>$ element contains the number of ground sentential forms in the subtree rooted at $N$.  

Copyright © 2009 John Wiley & Sons, Ltd.  
Prepared using speauth.cls
Working testers typically know a lot about the application under test and the test cases they would like to run. However, many testers have little programming skill and are not likely to develop the grammars and embedded code needed for practical testing. Consequently, the power of GBTG tools is unavailable to many testing practitioners. One solution to this problem relies on separating the tasks of (a) grammar development and (b) test control and execution. While task (a) requires programming skill, task (b) can often be accomplished by modifying the grammar tag parameters without changing the grammar rules.

We have developed a GUI which displays a parsed grammar and allows the user to modify the tag parameters to explore the generation tree. For example, Figure 14 shows the tool applied to the grammar in Figure 5(a). The left panel shows the grammar rules as read-only and tags as clickable underlined text. The bottom panel shows the language generated and the right panel shows the generation tree. The \texttt{rdepth} tag has been selected, bringing up a dialog box which graphically displays the tag \texttt{\{rdepth 3\}}. The tester may delete the tag or modify the tag parameter. In the right panel, the tester may supply a path and a depth to specify the root and depth of the generation tree to be displayed. For example, the right panel displays the generation tree rooted at tagged rule \texttt{Zeros} to depth 1.

4. THREE GBTG TECHNIQUES

4.1. Coverage and depth in combination

Used together, the recursion depth and coverage tags are powerful. Figure 15 shows three instances of a Book grammar which can generate simple XML documents. The test focus is bugs that arise when the first and last chapters have a different number of sections; three chapters are sufficient for this test focus. Because the Book grammar is recursive, its language is infinite.

The simplest way to reduce the generated output is with the count tag. If \texttt{\{count 3\}} is attached to the \texttt{Book} nonterminal as shown in Figure 15(a), there will be three books generated.
In all three books, the first two chapters will have one section each and the third chapter will have from one to three sections. Changing the count tag value will change the number of sections in the third chapter, but not in the first two chapters.

The recursion depth tag can spread the sections more evenly across the chapters. If \{rdepth 3\} is attached to the Sections nonterminal as shown in Figure 15(b), then there will be 27 books generated, with all combinations of one, two, or three sections for each of the three chapters. As the recursion depth increases linearly, the number of generated books increases exponentially.

Suppose that \{cov [ ([0,2],2), ([1],1) ]\} is attached to the Chapters rule and \{rdepth 3\} is attached to the Sections nonterminal as shown in Figure 15(c). YouGen will generate nine cases to cover all combinations of chapter sizes between the first and last chapters, and three cases to cover all three chapter sizes for the second chapter. In this grammar, the rdepth tag controls the number of sections and the cov tag controls the scope and strength of section interaction.

4.2. Generation tree analysis

Logging and viewing generation trees can facilitate a tester’s understanding of the language generated by a grammar. Figure 16 shows a tagged recursive grammar which generates a large number of strings, and the generation tree for this grammar. When nonterminal A is
(a) Book grammar with count tag

```plaintext
{count 3} Book;
    Book ::= '<BOOK>' Chapters '</BOOK>';
    Chapters ::= Chapter Chapter Chapter;
    Chapter ::= '<CHAPTER>' Sections '</CHAPTER>';
    Sections ::= Section;
    Sections ::= Section Sections;
    Section ::= '<SECTION> text ... </SECTION>';
```

(b) Book grammar with rdepth tag

```plaintext
{rdepth 3} Sections;
    Sections ::= Section;
    Sections ::= Section Sections;
    Section ::= '<SECTION> text ... </SECTION>';
```

(c) Book grammar with cov and rdepth tags

```plaintext
{cov [ ([0,2],2), ([1],1) ]} Chapters ::= Chapter Chapter Chapter;
    Chapter ::= '<CHAPTER>' Sections '</CHAPTER>';
{rdepth 3} Sections;
    Sections ::= Section;
    Sections ::= Section Sections;
    Section ::= '<SECTION> text ... </SECTION>';
```

Figure 15. Book grammars with tags
tagged with \texttt{rdepth 2} and the \texttt{cov} tag associated with rule \texttt{D} is omitted, the number of strings generated is 65792. By looking at the grammar, it is hard to arrive at this number. However, by looking at the selected paths in the generation tree, a tester may find the grammar rules which cause the language generated to be so large. For example, exploring the generation tree in Figure 16 from the root to depth 1, we see that the number of strings generated from recursive rule \texttt{A}, indicated by the count at node \texttt{A1:65536}, is much larger than the number of strings generated from non-recursive rule \texttt{A}, i.e., \texttt{A0:256}. Therefore, a first approach to reduce the number of strings generated is to decrease the value of the \texttt{rdepth} tag to 1. Note that it normally does not make sense to include a rule with an \texttt{rdepth} tag set to 1, but that this can be useful when exploring the grammar. In this case, the grammar produces $256 = 2^8$ strings and the generation tree consists of the subtree rooted at node \texttt{A0:256}. Exploring the tree from root node \texttt{B0:256} to depth 1 shows that there are four alternatives for rule \texttt{C}, and that each alternative produces $64 = 256/4$ strings. Exploring deeper in the tree shows how YouGen generates $64 = 2^6$ strings for each alternative of rule \texttt{C}. Since rule \texttt{E} has four alternatives, and rule \texttt{D} consists of three \texttt{E}s, $64 = 4 \times 4 \times 4$ strings are generated from the subtree rooted at \texttt{D0:64}. Therefore, $256 = 64 \times 4$ strings are generated in the subtree rooted at \texttt{A0:256}.

A tester may notice that another approach to reduce the number of strings generated is to reduce the number of alternatives from rule \texttt{D}. Adding the tag \{	exttt{cov } \{(0,1,2),2\}\} to grammar rule \texttt{D} reduces the number of strings to 64. There are three \texttt{E} domains and each domain \texttt{E} has four alternatives; YouGen generates a two-cover of 16 strings. Since there are four alternatives for rule \texttt{C}, the number of strings generated is $64 = 16 \times 4$.

### 4.3. The template/probe paradigm

With GBTG, the following situation often arises. A grammar is available which models correct inputs well, but the grammar is long and complex. The goal is to produce a large number of “nearly correct” inputs: correct inputs with one or a few errors inserted. A grammar with interleaved “correct” and “error” rules can work but the resulting grammar is hard to understand and maintain. Maintenance is a concern because small changes to the correct grammar occur frequently.

One solution is to write separate grammar rules for normal and error text. The two groups of rules can be developed and maintained separately and then joined by a single top-level rule. The normal rules generate template documents with insertion points marked for later substitution. The error rules provide values to fill the “holes” in the template.

Figure 17 shows a simple example of this approach. Each test case is a floating point number with two error insertion points: one between the left-side digits and the ‘.’ and the other one between the ‘.’ and the right-side digits. In the postcode, \texttt{flatten(s)} takes a nested sentential form \texttt{s} and returns the string corresponding to \texttt{s}. \texttt{x0.replace(x1,x2)} replaces each occurrence of \texttt{x1} in \texttt{x0} with \texttt{x2}. The template is stored in \texttt{s[0]} and the probe values are stored in \texttt{s[1]}. The probe values are generated separately: two incorrect and one correct. In total, 36 inputs are generated, 32 of which are incorrect.
5. EXPERIENCE

In this section, we briefly discuss a number of applications of YouGen on industrial and open-source software.

5.1. HTML injection vulnerabilities in RSS feeds

Really Simple Syndication (RSS) is the name for a family of XML-based formats for syndicating web content. There are many RSS versions. For example, an RSS 2.0 feed consists of a <channel>. The children of a <channel> are <title>, <link>, <description>, and a variable number of <item>s where each <item> has its own <title>, <link>, and <description>.

There are security risks associated with HTML injection in RSS feeds [14]. It is common that an RSS feed contains HTML elements in its channel title, channel link, and channel description element texts. These HTML elements are used for presentation and navigation.
purposes. Some HTML elements are considered “unsafe.” For example, the script HTML element is considered unsafe because it can contain malicious JavaScript code that will be executed on the client side. The process of filtering out the unsafe HTML elements is called sanitization. Failure to filter out the unsafe HTML elements, and only those elements, is considered a sanitization error.

YouGen has been used to test the Universal Feedparser [15] for HTML sanitization errors [16, 17]. Of the nine RSS versions supported by the feedparser, three were tested: Atom 1.0 [18], RSS 1.0 [19], and RSS 2.0 [20]. The grammar was developed using the template/probe paradigm. The template rules generate correct RSS documents in each of the three versions. The probes insert a variety of dangerous HTML constructs in a variety of positions.

Test cases are executed by postcode attached to the grammar starting rule. The postcode is invoked once for each template/probe pair. The code inserts the probe values in the document template and passes the resulting RSS document to the feed parser. The postcode checks for the presence of dangerous HTML elements and logs the results in an XML file. Because the log files were large (over 1MB), XQuery scripts were developed to analyze the files and provide summary results.

Sanitization errors were found for all three RSS versions. First, the feedparser does not sanitize all of the RSS elements. Second, for those RSS elements where sanitization is performed, the feedparser sanitizes the feed only when the type attribute is specified to be HTML. Therefore, it is possible to perform HTML injection by placing HTML in those RSS elements that would not be sanitized by the feedparser or by leaving the type attribute unspecifed.
5.2. Testing XPath implementations

XPath is a language for querying XML documents [21]. The XPath standard is rapidly evolving and many XPath implementations exist today. It is important that these implementations are tested for compliance against the standard. YouGen was used to generate and execute test cases, consisting of XML documents and corresponding XPath queries [22]. To deal with the test oracle problem, we used differential testing, in which we compared the output generated by two XPath tools, rather than determining the expected output for each test case.

We tested the following XPath features: root and element node selection, text node selection, attribute node selection, predicates and axes. To limit the number of test cases, we split up the testing of each feature into a number of test packages, each focusing on a particular aspect of that feature. Each package consists of a pair of grammars: one to specify XML documents and one to specify XPath queries for those documents. A script ran YouGen on both grammars, invoked the two XPath implementations on each XML document and XPath query combination, and compared the result after some minor reformatting of the output.

The XPath implementations that we tested were version 2.1 of VTD-XML (implemented in Java) [23] and version 2.6.16 of LibXML2 (implemented in C) [24]. Although both are software libraries, they provide XPath wrapper programs for download—referred to as the XPath tools. We wrote a total of 31 packages that generated over 345,000 test cases. Five bugs were found: LibXML2 was unable to handle XML documents longer than 1024 elements, would incorrectly treat whitespace as text nodes, and would incorrectly return the value of an attribute when asked for its name. VTD-XML was unable to handle XML documents longer than 256 elements and was unable to handle very large blocks of text.

5.3. Generation of large XML databases

XML is now being used to replace large relational databases. Therefore, performance testing of XQuery implementations [25] on very large documents is important. XMark [26] is an XML benchmark tool that consists of a document generator, xmlgen, and 20 XQuery scripts. The document generator is used to produce XML documents which mimic online auction databases. The XQuery scripts are used to reveal performance bottlenecks in typical XML processor implementations.

xmlgen takes a scaling factor as input and generates documents of sizes from 10MB to 10 GB. While every XML document has a tree structure, each xmlgen document also includes many references: strings stored in one part of the document used as lookup keys to other parts. The implementation of xmlgen consists of 7,479 lines of C code of which 1,674 lines are executable code. The remaining lines contain string constants used to generate realistic item text. XMark also includes a collection of XQuery scripts covering:

1. text search: searching the entire document for a target string,
2. ordered access: searching through lists of items, and
3. chasing references: following a reference to the referenced text.

To evaluate GBTG in this domain, we used YouGen to produce newxmlgen [27], a generator which produces documents somewhat simpler than those produced by xmlgen. newxmlgen is
implemented with a total of 274 lines. While both the newxmlgen implementation and output is relatively simple, it is still capable of producing very large documents (up to 9.6 MB) and useful performance results.

We evaluated two XPath implementations: PyXML [28] implemented in Python and Saxon [29] implemented in Java. PyXML is better than Saxon in terms of full-text search. On the other hand, Saxon has significantly better performance than PyXML on ordered access and chasing references.

5.4. TCP flag attacks on network firewalls

TCP Connections are established by a process known as the three-way handshake [30]. The solid lines in Figure 18 show the packets typically involved in establishing a TCP connection. Each TCP packet contains six boolean flags: synchronize (SYN), acknowledge (ACK), finish (FIN), push (PSH), reset (RST), and urgent (URG). The client node initiates the connection by sending a packet with the SYN flag set to the server. The server responds with a packet with both the SYN and ACK flags set. Finally, the client responds with a packet with just the ACK flag set. The connection is now established. Connection termination operates in a similar fashion, requiring three packets, two of which have the FIN flag set.

While there are $2^6 = 64$ possible flag combinations, only 18 of those are ever legal. For example, a packet with both SYN and FIN set is illegal. Because some TCP implementations are vulnerable to illegal flag combinations, hackers often purposely send bad flag packets: TCP packets with illegal flag combinations. Often a network firewall is present between the client and server, as shown by the heavy dashed line in Figure 18. Ideally, the firewall will drop all bad flag packets. In practice, most firewalls will forward some of these packets. It is useful to have a test suite to characterize the bad flag filtering behaviour of a given firewall. YouGen has been used to implement such a test suite [31, 17].

The test suite uses an offline approach [13]. A grammar generator writes abstract TCP packets to a file; a C++ executor reads the file and generates packets “on the wire.” The bad flag packets in the connection-establishment phase are shown by light dashed lines in Figure 18. Normal packets and bad flag packets are interleaved so that each normal packet is sandwiched between a pair of bad flag packets. Including the connection termination packets, there are 13 packets per TCP connection: 6 normal packets and 7 bad flag packets.

The grammar is relatively short and simple. Each abstract packet specifies values for the six TCP flags, a flag indicating whether the packet should be sent inbound or outbound, and a flag indicating whether the packet should be forwarded or dropped. A generator coded in Python is used for the bad flag combinations.

The executor converts each abstract packet to binary, adding fields to produce complete TCP, IP, and Ethernet headers [30]. The packet is written using raw sockets [32] because a TCP (stream) socket does not provide sufficient control of TCP flags. After writing each packet, the executor pauses to determine whether the packet has been forwarded and writes the results to a log file.

With 46 bad flag combinations and seven bad flag positions, there are $46^7 \approx 435,000,000,000$ possible connections to test. Since the executor can test roughly 50 connections per second, significant reduction is necessary. The cov tag is used heavily. Initially, strength 1 was used,
generating 46 connections. As we learned more about firewall behaviour, we selectively used strength 2, generating roughly 2500 connections. As with the RSS test suite, the log files were too large for manual analysis. Python scripts were written to analyze the log files and determine, for each bad flag position, the number of times each bad flag combination was forwarded in that position.

We found significant differences in bad flag filtering across the firewalls tested. An inexpensive small-office/home-office (SOHO) firewall forwarded nearly all bad flag packets. Most firewalls, even with carefully configured rule sets, forwarded some bad flag packets. Only the Linux iptables [33] firewall blocked all bad flag packets.

5.5. Discussion

When developing test suites, one traditional goal is minimizing the number of test cases while meeting some criteria, e.g., 100% statement coverage. With GBTG, the economics are different. We knowingly generate many test cases which are redundant because removing them would make the test suite much more complex. We accept the redundant cases but keep the execution cost per test case small by automating input generation, test execution, and output checking.
The new goal is to minimize test suite development cost, while meeting criteria such as 100% statement coverage.

We now view the YouGen applications just presented by revisiting the questions faced by grammar developer presented in Section 1:

1. **How should the tags be used in combination?** In our experience, the most useful tag combination is cov and rdepth. The cov tag generates useful sets of tuples from a list of parameter sets; the rdepth tag limits the size of those parameter sets. In newxmlgen, the two tags provide the means to control the total size of the generated document, as well as the relative sizes of the document sections. In the RSS feed test suite, the same tag combination is used. The template/probe paradigm plays a crucial role here; the top probe rule collects the parameters in a single place. When the cov tag is attached to rule R, it can name only the items appearing on the right-hand-side of R.

2. **What work should be done by the grammar versus the embedded code?** The simple answer is to recall that a context-free grammar is computationally weak, with far less power than a Turing machine. The grammar developer must recognize which computations are naturally context free, and implement only those with grammar rules. In the TCP flags test suite, the 46 bad flag combinations are hard to generate with grammar rules but are easily computed with Python code. In many GBTG test suites, the majority of the file is used for Python code, with the grammar rules serving as “glue.”

3. **Should the grammar execute the test cases immediately or save them for later execution?** In this case, the answer is dependent on the test domain. In the RSS test suite, each XML document is generated by the grammar and immediately passed to the feed parser for testing. In the XPath implementation test suite, the XML documents are written to files for later testing. In the TCP flags test suite, abstract test cases are written to files, with the concrete test cases generated later by C++ code.

4. **How do I debug a complex grammar?** Even moderately complex grammars can be tricky to debug. We have used three main approaches:

   (a) Begin with small settings for the most important tags. We often execute grammars with rdepth set to 1.
   (b) Insert debug print statements in precode and postcode.
   (c) With difficult bugs, we make heavy use of the GUI shown in Figure 14 to manipulate tag parameters and explore the generation tree.

6. **RELATED WORK**

6.1. **Covering arrays**

Grindal et al. [34] survey and classify combinatorial test strategies, including work on orthogonal and covering arrays. Many approaches [8, 35, 36] and tools [37, 38] have been proposed for calculating covering arrays. The fault-detection ability of covering arrays of various strengths has been evaluated by a number of authors [39, 40]. Variable strength covers, initially proposed by Cohen et al. [41], are a special form of mixed-strength covering arrays.
in which higher coverage is required among a subset of parameters and lower coverage for the entire set. Many of the above approaches only work for covering arrays in which all domain sizes are the same, which is not useful in testing applications because the domain sizes will typically vary in such applications. Algorithms for mixed-level covering arrays, in which domain sizes can vary, have been described by Moura et al. [10]. The YouGen tool described generates mixed-strength and mixed-level covering arrays up to strength three.

6.2. Grammar-based test generation

GBTG has long been used for testing programming language compilers and interpreters [4, 1, 42, 43, 5]. More recently, GBTG has been applied to the testing of network protocols [6, 2] and firewalls [44], VLSI simulators [45], and XML applications [3]. Like YouGen, many other GBTG tools have used recursive algorithms based on derivations and have provided tags to reduce the size of the generated language.

The generation tree construct appears to be new. A generation tree is quite different from the parse trees well known in the programming language community [9].

6.3. Incorporating covering arrays in grammar-based test generation

The connection between context-free grammar rules and the cartesian product has long been known. For example, one of the earliest GBTG tools was based on random traversal of the generation tree but provided a tag to request the full cartesian product [4].

More recent work by Lammel et al. [46] describes an implemented tool with a one-cover tag and proposes a general notation for covering-array tags. The YouGen cov tag uses scope/strength pairs. The Lammel covering-array specification is based on sets of integers corresponding to the scope used in YouGen. In the Lammel notation, however, the strength is implicit. Specifically, the strength is determined by the scope itself; the strength used is the same as the number of elements in the scope. A Lammel specification consists of a set of scope sets. For example, \{ \{0, 1\} \} is equivalent to the YouGen specification (\{0, 1\}, 2); \{ \{0, 1\}, \{0, 2\}, \{1, 2\} \} is equivalent to (\{0, 1, 2\}, 2). While the Lammel notation is elegant, it can be cumbersome. For example, the firewall test suite described in the previous section uses the specification (\{1, 3, 5, 7, 9, 11, 13\}, 2). The Lammel equivalent

\{\{1, 3\}, \{1, 5\}, \{1, 7\}, \{1, 9\}, \{1, 11\}, \{1, 13\}, \{3, 5\}, \{3, 7\}, \{3, 9\}, \{3, 11\}, \{3, 13\}, \ldots, \{11, 13\}\}

contains 21 pairs.


In this subsection, we organize the related work by generator features. We provide a short description of each feature and identify the corresponding YouGen feature, if any.

- A count limit restricts the total number of strings generated by a rule or grammar. This feature was first developed by Hanford [4] and later used by Murali et al. [47]. Count limits are provided in YouGen with the count tag.
• **Rule weights** randomize the selection of rule alternatives by attaching weights to the rules. Rule weights were supported by Hanford [4], Payne [48], and Bird et al. [1]. While rule weights are not directly provided by YouGen, they can be simulated using `precode`, as shown in Figure 11.

• A **rule guard** is a boolean expression which allows or prevents a rule from being selected for expansion. Rule guards have been provided by Duncan et al. [49] and Homer et al. [42]. In YouGen, `precode` tags can be used as guards.

• An **action** is a code fragment associated with a grammar rule. Actions were supported by Celentano et al. [50] and Duncan et al. [49]. In YouGen, the `precode` and `postcode` tags allow arbitrary Python code to be associated with a rule.

• A **recursion limit** restricts the number of times a recursive rule can be selected in a single derivation. This feature has been widely used and is found in the work of Hanford [4], Celentano et al. [50], and Bird et al. [1]. In YouGen, the `rdepth` tag provides recursion limits.

• A **depth limit** limits the number of steps in any derivation. Depth limits appear to have been introduced only recently in the work of Zaytsev [3] and Lammel et al. [46]. YouGen provides depth limits using the `depth` tag.

• Balance control restricts the generation tree in a way somewhat like height-balanced binary trees. Zaytsev [3] and Lammel et al. [46] have used this feature in their work. Balance controls are not implemented in YouGen.

• Covering arrays features generate subsets of the cross product of the strings generated by the right-hand-side of a rule. Zaytsev [3] and Lammel et al. [46] provide some covering array support. YouGen generates covering arrays with the `cov` tag.

• Random traversal selects rule alternatives randomly. This is by far the most common approach to GBTG [4, 1, 47, 42, 51]. Random traversal order is not supported by YouGen.

• Breadth-first traversal traverses the generation tree breadthfirst. Zaytsev [3] and Lammel et al. [46] used this traversal approach in their work. They claim that, with breadth-first traversal, there is no need for recursion or depth limits. Breadth-first traversal is, however, more complex to implement and uses more memory than depth first. YouGen does not support breadth-first traversal.

• Dynamic grammars allow rules to be added to the grammar during generation. Hanford [4] appears to be first one to introduce this feature. Dynamic grammars are supported in YouGen by using `precode` or `postcode` to control grammar objects.

7. **CONCLUSIONS**

Since its introduction in the early 1970s, GBTG has become well-established. Despite the demonstrated effectiveness, however, the tool support is uneven; some tools target specific domains while other tools are proprietary. Some tools are hard to use except by the tool developers. In particular, the precise meaning of the tags is sometimes unclear. As a result, while many testing practitioners and researchers are aware of GBTG, few have detailed knowledge or experience.
We have presented YouGen, a GBTG tool providing the best of the features provided by previous tools. Along with the standard features, YouGen supports covering-array tags. GBTG and covering arrays are among the most powerful and widely used black-box test generation methods available. YouGen allows them to be combined seamlessly.

We have provided detailed examples of several GBTG techniques as well as four short case studies demonstrating YouGen on more realistic tasks.

ACKNOWLEDGEMENTS

We thank Mitch Chang, David Ly-Gagnon, and Lewis Sobotkiewicz for their assistance with implementing YouGen and some of the case studies; Craig MacNamara for testing the XPath implementations; Gary Bazdell and Brett Stevens for providing the two-cover algorithm used in YouGen; and Gordon Fraser for providing feedback on an earlier version of this paper.

REFERENCES