Grammar-Based Test Generation: New Tools and Techniques

by

Hong-Yi Wang
B.Sc., University of Victoria, 2007
M.Sc., University of Victoria, 2009

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Grammar-Based Test Generation: New Tools and Techniques

by

Hong-Yi Wang
B.Sc., University of Victoria, 2007
M.Sc., University of Victoria, 2009

Supervisory Committee

Dr. Dan Hoffman, Supervisor
(Department of Computer Science)

Dr. Daniel German, Departmental Member
(Department of Computer Science)

Dr. Tim Pelton, Outside Member
(Department of Curriculum and Instruction)

Supervisory Committee

Dr. Dan Hoffman, Supervisor
(Department of Computer Science)

Dr. Daniel German, Departmental Member
(Department of Computer Science)

Dr. Tim Pelton, Outside Member
(Department of Curriculum and Instruction)

ABSTRACT

Automated testing is superior to manual testing because it is both faster to execute and achieves greater test coverage. Typical test generators are implemented in a programming language of the tester's choice. Because most programming languages have complex syntax and semantics, the test generators are often difficult to develop and maintain. Context-free grammars are much simpler: they can describe complex test inputs in just a few lines of code. Therefore, Grammar-Based Test Generation (GBTG) has received considerable attention over the years. However, questions about certain aspects of GBTG still remain, preventing its wider application. This thesis addresses these questions using YouGen_NG, an experimental framework that incorporates some of the most useful extra-grammatical features found in the GBTG literature. In particular, the thesis describes the mechanisms for (1) eliminating the combinations of less importance generated by a grammar, (2) creating a grammar that generates combinations of correct and error values, (3) generating GUI playback scripts through GBTG, (4) visualizing the language generation process in a complex grammar, and (5) applying GBTG to testing an Really Simple Syndication (RSS) feed parser and a web application called Code Activator (CA).

Contents

List of Figures

ACKNOWLEDGEMENTS

I offer my sincerest gratitude to my supervisor, Professor Dan Hoffman, who guided me through my research with his patience and knowledge. The thesis would not have been completed without his encouragement and support. I would also like to thank all members in the lab for working with me on the various problems and making my graduate study a memorable experience. Finally, I would like to thank my wife, Man Cui, for her understanding and love during the past few years, and my parents for their continual support and advises.

Chapter 1

Introduction

Software is indispensable in today's society. It is embedded in the equipment that people use on a daily basis, such as computers, cell phones, and cars. With software being such an important technology, people expect a high level of reliability. However, software reliability has not met people's expectations. A list of software failures that result in the loss of billions of dollars can be found in the *Software Hall of Shame* [15].

Even frequently used software is not problem free. Miller et al. [35] provided a story of how one of his colleagues had difficulty accessing his Unix workstation from home on a stormy night. Communication between his workstation and home was established by a dial-up line which is susceptible to bad weather. As a result, the commands he typed were often garbled which, to his surprise, caused some of the Unix utilities to crash. Intrigued by this discovery, Miller et al. developed a series of tests to measure the reliability of Unix utilities across various platforms. The test results show that many Unix utilities do crash or hang on illegal inputs.

The reason for many software failures is insufficient testing. While most organizations understand the importance of testing, many of the software products that they produce are not properly tested before release. This is so because software testing is a time-consuming task. Faced with limited resources and tremendous pressure to deliver software on time, organizations often cut corners by reducing the number of tests. As a result, the software that they produce is not reliable. In short, a practical testing technique not only has to have good test coverage but also fit within the resource and time constraints.

1.1 Software Testing

In an effort to increase software reliability, many testing techniques have been proposed. While many techniques exist, most of them can be categorized as either manual or automated testing.

1.1.1 Manual Testing

With manual testing, testers first create a set of test scripts. The tests are intended for human execution and therefore are written in natural language. Each test case contains a short description of execution steps and the expected outputs. The testers then execute the test scripts and compare the program outputs with the expected outputs. For each test case, the testers record either pass or fail into a log which is then given to the software developers for bug fixing. After applying the bug fix, the testers carry out the tests again. This process continues until the number of bugs is less than some predefined threshold.

The disadvantages of manual testing are threefold. First, it takes a lot of time to execute one test case because the tests are carried out by humans. Second, the test coverage is poor as it is significantly limited by the rate at which the test cases are executed. Third, it is expensive to repeat the tests. Because of its poor coverage and time intensiveness, manual testing is inadequate for a lot of testing scenarios. However, manual test scripts are relatively easy to develop because they do not require programming skills. As a result, manual testing is adopted by many organizations.

1.1.2 Automated Testing

In contrast to manual testing, automated testing is carried out by computers and therefore the test scripts are written in a programming language. Automated testing is a three-step process. First, a test generator generates test inputs. Second, the test inputs are passed to the code under test (CUT) and the outputs are captured. Finally, a test oracle gives a verdict for each test case. It does so by comparing the expected output and the actual output. A test case passes if and only if these two outputs agree and fails otherwise. The test results are recorded into a log file for bug fixes. As with manual testing, the tests are carried out repeatedly until the CUT meets the quality assurance standard.

Automated testing is superior to manual testing in two ways. First, computers

```
def generate():
    for CallerOS in ['Mac','Win']:
        for ServerOS in ['Lin','Sun','Win']:
            for CalleeOS in ['Mac','Win']:
                yield CallerOS + ' ' + ServerOS + ' ' + CalleeOS
```

Figure 1.1: An example VoIP test generator

```
Mac Lin Mac
Mac Lin Win
Mac Sun Mac
Mac Sun Win
Mac Win Mac
Mac Win Win
Win Lin Mac
Win Lin Win
Win Sun Mac
Win Sun Win
Win Win Mac
Win Win Win
```

Figure 1.2: The output of the `generate` function

take much less time than humans to execute a test case and therefore more test cases can be executed in a given amount of time. Second, better test coverage can be achieved as result of faster execution. However, implementing the test scripts is a time consuming task. It also demands programming skills which many testers do not have. As a result, automated testing is the less preferred approach for many organizations.

Another problem with automated testing is that the test scripts are difficult to understand. Readability is an important issue as the test scripts may need modifications from time to time, either because of changes in software requirements or occurrences of new bugs. The problem is illustrated by an example testing scenario with the code under test being a Voice over Internet Protocol (VoIP) application.

1.1.3 Example Testing Scenario

Let us consider a testing scenario where the objective is to test the connection setup phase of a VoIP call which consists of a caller, a server, and a callee. Each of the

```
Call ::= CallerOS ServerOS CalleeOS;

CallerOS ::= 'Mac';
CallerOS ::= 'Win';

ServerOS ::= 'Lin';
ServerOS ::= 'Sun';
ServerOS ::= 'Win';

CalleeOS ::= 'Mac';
CalleeOS ::= 'Win';
```

Figure 1.3: `Call` grammar

three machines runs its own operating system. Both the caller and callee run either Macintosh (`Mac`) or Windows (`Win`) while the server runs Linux (`Lin`), SunOS (`Sun`), or `Win`. As a result, the number of test cases is $2 \times 3 \times 2 = 12$. Figure 1.1 shows the test generator: a Python function called `generate` that generates all 12 test cases. The function consists of three `for` loops which, from top to bottom, iterates the operating system alternatives of the caller, server, and callee. Figure 1.2 shows the output of the `generate` function.

Although it is easy to create a function to generate all 12 test cases, doing so programmatically has three disadvantages. First, the implementation is very verbose. If more machines are involved and each machine has more operating system alternatives, the function can expand significantly. Second, the structure of the test inputs is obscure. It takes some effort to understand that the `for` loops are used for filling in all the operating system alternatives. Third, writing the code for the general case of n parameters is tricky. Fortunately, these three problems can easily be solved with context-free grammars.

Figure 1.3 shows an improved version of the test generator that is implemented in a context-free grammar. The first rule defines the structure of the test inputs while the rest define the operating system alternatives for each machine. Test generation is done by generating the language of the grammar. Adding a new machine is easy. Suppose the new machine to be added is `X` and it has the same operating system alternatives as the caller and callee. The modifications required are twofold. First, add `X` to the right-hand side of the first rule. Second, add the following two rules: `X ::= 'Mac';` and `X ::= 'Win';`. Similarly, adding a new operating system alternative is easy.

Suppose the caller is to include one additional operating system alternative `QNX`. This can be achieved by adding the following rule: `CallerOS ::= 'QNX';`.

The lesson learned from the testing scenario is that, for some testing problems, context-free grammars are more suitable for implementing test generators than programming languages. The resulting implementation is not only easy to understand but also easy to modify and expand.

1.2 Grammar-Based Test Generation

1.2.1 Elements of a Grammar

A context-free grammar consists of a set of nonterminals, a set of terminals, a set of rules, and a start symbol. Each rule consists of a left-hand side and a right-hand side, separated by the production symbol `::=` and terminated with a semicolon. The left-hand side is a nonterminal. The right-hand side is a list of terms. A term is either a nonterminal or a terminal. The start symbol is the nonterminal that appears on the left-hand side of the first rule. By convention, a grammar is named after its start symbol. The language of a grammar G, denoted as $L(G)$, is all strings that can be derived from the grammar. An example grammar is shown in Figure 1.3:

- The nonterminals are {`Call`, `CallerOS`, `ServerOS`, `CalleeOS`}.
- The terminals are {`'Mac'`, `'Win'`, `'Lin'`, `'Sun'`}.
- The rules are:
  - `Call ::= CallerOS ServerOS CalleeOS;`
  - `CallerOS ::= 'Mac';`
  - `CallerOS ::= 'Win';`
  - `ServerOS ::= 'Lin';`
  - `ServerOS ::= 'Sun';`
  - `ServerOS ::= 'Win';`
  - `CalleeOS ::= 'Mac';`
  - `CalleeOS ::= 'Win';`
- The start symbol is `Call`.

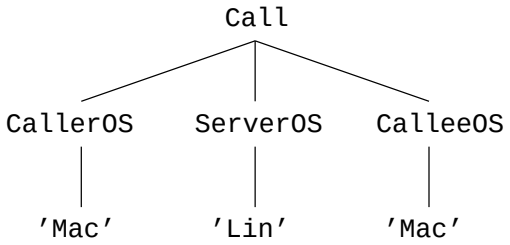


Figure 1.4: Parse tree of `'Mac' 'Lin' 'Mac'`

- The grammar is referred to as the `Call` grammar.
- The language of the grammar is shown in Figure 1.2.

1.2.2 Parsing

Context-free grammars are commonly used in parsing: given a string and a grammar, determine whether the string is in the language of the grammar. In the `Call` grammar, the string `Mac Lin Mac` is in the language because it can be derived from the grammar's start nonterminal `Call`, as evidenced by the parse tree shown in Figure 1.4. In contrast, the string `Mac Lin Sun` is not in the language because it is not derivable from `Call`.

1.2.3 Test Generation

Unlike parsing, the goal of grammar-based test generation (GBTG) is to generate the language of a grammar, not to determine if a string is a member in the language. Used in test generation, context-free grammars are used to describe the structure of the test inputs. Each string in the language of the grammar is a test case. The language of the grammar constitutes the test space. The following is a list of terms that are commonly used in GBTG.

- *Sentential form*: a list of terms where each term is either terminal or nonterminal. Figure 1.5 (a) shows five sentential forms separated by $\Rightarrow$. The last sentential form is ground because it consists of only terminals.
- *Derivation step*: the process of replacing a single nonterminal with a matching rule's right-hand side. Given a nonterminal N, its matching rules are the ones that have N on their left-hand side. Figure 1.5 (a) shows four derivation steps.

```
Call ⇒ CallerOS ServerOS CalleeOS
     ⇒ 'Mac' ServerOS CalleeOS
     ⇒ 'Mac' 'Lin' CalleeOS
     ⇒ 'Mac' 'Lin' 'Mac'
```

(a) Leftmost derivation of `'Mac' 'Lin' 'Mac'`

```
Call ⇒ CallerOS ServerOS CalleeOS
     ⇒ CallerOS ServerOS 'Mac'
     ⇒ CallerOS 'Lin' 'Mac'
     ⇒ 'Mac' 'Lin' 'Mac'
```

(b) Rightmost derivation of `'Mac' 'Lin' 'Mac'`

Figure 1.5: Derivations from the `Call` grammar

- *Derivation*: the process of generating a string in the language. The process starts with a sentential form that contains only the start symbol and applies one or more derivation steps until the sentential form becomes ground. Depending on which nonterminal is replaced at each derivation step, a derivation can be classified as either leftmost derivation or rightmost derivation.
- *Leftmost derivation*: chooses the leftmost nonterminal for replacement at each derivation step. Figure 1.5 (a) shows a leftmost derivation for string `Mac Lin Mac`.
- *Rightmost derivation*: chooses the rightmost nonterminal for replacement at each derivation step. Figure 1.5 (b) shows a rightmost derivation for string `Mac Lin Mac`.

1.2.4 Generation Tree

In a complex grammar, it is often difficult to understand how each string in the language is derived. A generation tree is a visualization tool that facilitates understanding of language generation. Figure 1.6 shows the generation tree for the `Call` grammar. Each node is a sentential form. A pair of nodes connected by an arc represents a derivation step. Language generation starts from the root which is a sentential form consisting of only the start symbol; in this case, the nonterminal `Call`. A path

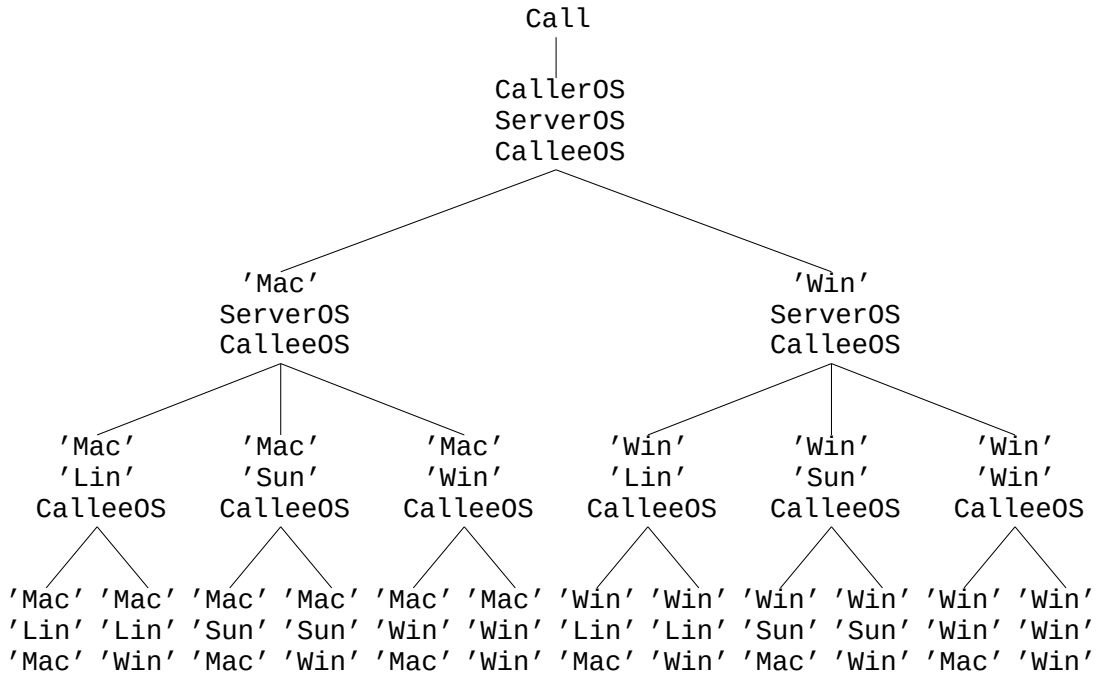


Figure 1.6: Generation tree of `Call` grammar

Version	header length	TOS	total length	id	flags	offset	TTL	protocol	source	destination
0	0	0	0	0	0	0	0	0	0x00000000	0x00000000
1	4	1	19	1	1	1	1	1	0xc0a86401	0xc0a8644d
4	5	3	20	65534	2	1479	2	2	0xc0a8644d	0xc0a86401
6	6	5	1499	65535	4	1480	3	3	0xc0a864ff	0xc0a864ff
7	14	7	1500		7	1481	31	5	0xffffffff	0xffffffff
	15		1501		3	1499	32	6	0xffffffff	0xffffffff
			3001		5	1500	33	7		
			65534		6	1501	63	16		
			65535			8190	64	17		
						8191	65	18		
							127	37		
							128	59		
							129	64		
							254	65		
							255	254		
								255		

Figure 1.7: IP parameters provided by Hoffman et al. [22]

from the root to a leaf represents a derivation. For example, the string `Mac Lin Mac` is derived by following the leftmost path in the generation tree.

1.3 Covering Arrays

Although grammars allow testers to create complex test cases with only a few lines of code, they tend to generate too many. As a result, the tests often take too long to execute. An example of GBTG's shortcomings is shown by Hoffman et al. [22]. The objective was to test a device under test (DUT)'s ability to withstand attack, given a large number of packets decorated with valid and invalid Internet Protocol (IP) parameter values. Figure 1.7 shows the IP parameter values that they used. If a grammar were constructed to cover all combinations of parameter values, the total number of packets generated would be $5 \times 6 \times 5 \times 9 \times 4 \times 8 \times 10 \times 15 \times 16 \times 6 \times 6 = 3,732,480,000$. Even with a packet sending rate of $1,000,000$ per second, it would still take about 1 hour to execute the test. Apparently, the test must be significantly reduced. An effective strategy to reduce the number of test cases is covering arrays.

Covering arrays are useful for restricting the size of parameterized tests. For example, Figure 1.8 (a) shows function `f` with three parameters `a`, `b`, and `c`. Each of these three parameters can take a number of possible values as shown in Figure 1.8 (b). Figure 1.8 (c) shows the test set that is required to exhaustively test function `f`. The test set is a Cartesian product of the three parameters. The Cartesian product is 3-ary because there are in total 3 parameters. A covering array is a reduced version

```
def f(a,b,c):
```

(a) Code under test: a function with three parameters

P_a	P_b	P_c
a_0	b_0	c_0
a_1	b_1	c_1
	b_2	

(b) Test parameters

P_a	P_b	P_c
a_0	b_0	c_0
a_0	b_0	c_1
a_0	b_1	c_0
a_0	b_1	c_1
a_0	b_2	c_0
a_0	b_2	c_1
a_1	b_0	c_0
a_1	b_0	c_1
a_1	b_1	c_0
a_1	b_1	c_1
a_1	b_2	c_0
a_1	b_2	c_1

(c) Cartesian product of test parameters

Figure 1.8: An example test scenario

of the test set because it only covers n-ary Cartesian product where n, in the case of testing function `f`, is less than 3.

1.3.1 Coverage Specification

Given a coverage specification, a covering array algorithm generates a covering array which is a subset of the Cartesian product of the parameters. A coverage specification has the form (P, n). P is the list of parameters to apply the covering array algorithm. The parameters are specified by parameter indexes which are zero relative. n is the coverage strength, an integer that specifies the n-ary Cartesian product of the parameters that must appear in the covering array. Use Figure 1.8 (b) as an example, `([0,2],2)` specifies a covering array of strength 2 over parameters P_a and P_c. As another example, `([1],1)` specifies a covering array of strength 1 over parameter P_b.

P_a	P_b	P_c
a_0	b_0	c_0
a_1	b_1	c_1
a_1	b_2	c_1

Figure 1.9: A one-cover: satisfies `([0,1,2],1)`

1.3.2 One-cover

A one-cover is a covering array of strength 1. For each parameter, its values must appear in the covering array at least once. Figure 1.9 shows a one-cover that satisfies `([0,1,2],1)`. Note that the covering array includes all values in P_a, P_b, and P_c.

1.3.3 Two-cover

A two-cover is a covering array of strength 2. For each pair of parameters, their Cartesian product must appear in the covering array at least once. Figure 1.10 (a) shows a two-cover that satisfies `([0,1,2],2)`. The numbers at the left of each row are used to identify the tuples in the covering array. To prove that the covering array satisfies `([0,1,2],2)`, let us consider the pairs of parameters: (P_a, P_b), (P_b, P_c), and (P_a, P_c). The Cartesian product of the pairs are shown in Figure 1.10 (b). The numbers at the left of each row specify the tuples in which the values appear. For example, the pair a_0 and b_0 appears in tuple 1. As another example, the pair a_0 and b_1 appears in tuple 3. Because all pairs of values can be found in the covering array, the coverage specification is satisfied.

1.3.4 Mixed-strength

A mixed-strength covering array satisfies two or more coverage specifications of varying strength. Figure 1.11 (a) shows a covering array that satisfies both `([0,2],2)` and `([1],1)`. This is achieved by concatenating two covering arrays, each satisfying one coverage specification. The top four tuples form a covering array that satisfies `([0,2],2)`. The bottom three tuples form a covering array that satisfies `([1],1)`. In total, 7 tuples are needed to satisfy `([0,2],2)` and `([1],1)`.

However, these two coverage specifications can be satisfied by fewer tuples. Figure 1.11 (b) shows a covering array that has only 4 tuples but still satisfies both coverage specifications. `([0,2],2)` is satisfied because all pairs of values in $P_a \times P_c$

	P_a	P_b	P_c
1	a_0	b_0	c_0
2	a_1	b_1	c_0
3	a_0	b_1	c_1
4	a_0	b_2	c_0
5	a_1	b_0	c_1
6	a_1	b_2	c_1

(a) A two-cover: satisfies `([0,1,2],2)`

	P_a	P_b
1	a_0	b_0
3	a_0	b_1
4	a_0	b_2
5	a_1	b_0
2	a_1	b_1
6	a_1	b_2

	P_b	P_c
1	b_0	c_0
5	b_0	c_1
2	b_1	c_0
3	b_1	c_1
4	b_2	c_0
6	b_2	c_1

	P_a	P_c
1	a_0	c_0
3	a_0	c_1
2	a_1	c_0
5	a_1	c_1

(b) Cartesian products of $P_a \times P_b$, $P_b \times P_c$, and $P_a \times P_c$

Figure 1.10: An example two-cover

P_a	P_b	P_c
a_0	b_0	c_0
a_0	b_0	c_1
a_1	b_0	c_0
a_1	b_0	c_1
a_0	b_0	c_0
a_0	b_1	c_0
a_0	b_2	c_0

(a) A mixed-strength cover: satisfies `([0,2],2)` and `([1],1)`

P_a	P_b	P_c
a_0	b_0	c_0
a_0	b_1	c_1
a_1	b_2	c_0
a_1	b_2	c_1

(b) Same covering array with less tuples

Figure 1.11: An example mixed-strength cover

are present. `([1],1)` is also satisfied because all values in P_b are present.

Although mixed-strength covering arrays with fewer tuples are generally preferred, generating mixed-strength covering arrays that have the least number of tuples is a difficult problem. Therefore, we generate mixed-strength covering arrays by concatenation as illustrated in Figure 1.11 (a).

1.4 YouGen, YouGen_NG, and Dervish

Sobotkiewicz [42, 21] developed a prototype called YouGen to demonstrate the power of GBTG. The prototype takes a grammar as input and outputs a language generator. The language generator, when invoked, generates the language of the grammar. Extra-grammatical features are facilitated by the use of tags. For example, covering array tags can be applied to grammar rules such that certain combinations are not generated.

However, YouGen has several shortcomings. First, a grammar can only be created by following the YouGen-defined syntax and saved into a file. If a new grammar feature is desired, then new syntax has to be invented and subsequently the YouGen parser has to be modified. Second, the semantics of the tags are not always clear. Given a grammar with multiple tags, it is not easy even for the grammar author to derive the language of the grammar. Third, there was very limited application of YouGen. In fact, YouGen was used only for firewall testing [42] and performance testing of XML processors [44]. Finally, it is difficult to understand the language generation process. This is so because YouGen does not output generation trees, such as the one shown in Figure 1.6 for the `Call` grammar. Without generation trees, a recurring theme is that whenever YouGen generates strings that the user does not expect, he or she has to draw a generation tree to validate the YouGen output.

These shortcomings are addressed in this thesis by the introduction of a new experimental framework called YouGen_NG [24]. First, an application programming interface (API) is provided for creating grammars programmatically. Second, the tag semantics are extensively revised to provide the grammar author with clear and easy-to-understand definitions. Third, successful applications of YouGen_NG are demonstrated in two new case studies. Finally, the language generation process is logged so that the resulting log can be used by Dervish for automatic drawing of generation trees.

Dervish is another prototype developed by Ly-Gagnon [29] to make the power of

GBTG accessible to testers. Because developing test cases using GBTG requires programming skills which typical testers do not have, Dervish assists testers by providing a Graphical User Interface that facilitates language generation, tag manipulation, and learning of GBTG.

In short, the contributions of this thesis are threefold. First, the thesis introduces the notion of API grammars and their usefulness in experimenting with new GBTG features. Second, the tag semantics are improved so that the ambiguities in grammars with more than one tag are eliminated. Third, the case studies demonstrate how GBTG can be applied to two dissimilar testing domains.

1.5 Motivations of Our Research

This thesis is motivated by the following research questions:

1. Combinatorial explosion occurs when a grammar consists of many nonterminals and each nonterminal has many matching rules. As a result, even grammars of only moderate complexity tend to generate too many combinations to be useful. How can we eliminate the combinations of less importance generated by a grammar?

2. Many testing scenarios require the generation of test cases that contain combinations of correct and error values in order to test the robustness of the Code Under Test (CUT). The current GBTG literature, however, does not address the issue of creating a grammar that fulfills such a requirement. How can we provide a mechanism for creating a grammar that generates combinations of correct and error values?

3. Although GUI playback enables automatic GUI testing, the generation of such scripts is largely manual. As a result, the testers have to create combinations of test parameters by hand, a task that is easily achievable through GBTG. How can we apply GBTG in the automatic generation of GUI playback scripts?

4. Even for moderately complex grammars, the language generation process can be hard to visualize. As a result, the grammar author often significantly underestimates the size of the language generated by the grammar which he or she creates. How can we help the grammar author to visualize the language generation process in a complex grammar?

5. The existing GBTG implementations are either proprietary, lost, or lack the extra-grammatical features that we need to answer the questions above. How can we apply GBTG to some testing scenarios and at the same time demonstrate the usefulness of the extra-grammatical features?

1.6 Thesis Organization

The rest of the thesis is organized as follows: Chapters 2 and 3 introduce the two different approaches of creating a grammar, Chapter 4 describes the YouGen_NG design and implementation, Chapter 5 demonstrates how generation trees can facilitate understanding of language generation, Chapter 6 provides a case study on the testing of an RSS feed parser, Chapter 7 provides another case study on the testing of a quiz generator, Chapter 8 presents a summary of the related work on GBTG and covering arrays, and Chapter 9 presents the conclusion of my research work plus a few areas for future research.

Chapter 2

YouGen_NG Requirements: Text Grammar

Text grammars, in YouGen_NG terminology, refer to the grammars that are written in Backus Naur Form (BNF). Given a text grammar, generating its language involves a two-step process. First, the text grammar is translated into Python code. Second, the Python code is executed to produce the language of the grammar, one string per line. Translating a text grammar into an executable Python code is accomplished by running the YouGen_NG parser against the text grammar. Language generation is accomplished by invoking the YouGen_NG runtime library through the Python code.

To illustrate, Figure 2.1 shows an example text grammar called `TwoBit`. Running the YouGen_NG parser against the `TwoBit` grammar generates the Python code shown in Figure 2.2. The first line in the figure is used to import the YouGen_NG runtime library. Running the Python code produces the language of the `TwoBit` grammar as shown in Figure 2.3. Because it is inconvenient for a tester to go through the two-step process each time he or she wishes to execute a text grammar, a script called `genex.sh` was developed which takes a text grammar as input and outputs the

```
TwoBit ::= Bit Bit;

Bit ::= '0';
Bit ::= '1';
```

Figure 2.1: `TwoBit` grammar

```
from youGen_NG import *

G = Grammar()
G.append_rule( Rule( {}, 'TwoBit', [V('Bit'),V('Bit')] ) )
G.append_rule( Rule( {}, 'Bit', [T('0')] ) )
G.append_rule( Rule( {}, 'Bit', [T('1')] ) )
G.set_gentree_file('twobit.xml')

if __name__ == '__main__':
    for s in generate_language(G,'TwoBit'):
        print s
```

Figure 2.2: Code generated for `TwoBit` grammar

```
0 0
0 1
1 0
1 1
```

Figure 2.3: Output of running Figure 2.2

language of the grammar.

The following sections explain the elements of a grammar, the output formats, and the generation tree formats.

2.1 Rules

2.1.1 Syntax

A rule has the form:

lhs `::=` *rhs* `;`

lhs, which stands for left-hand side, is a nonterminal and *rhs*, which stands for right-hand side, is a list of terms. A term is either a nonterminal or terminal. A nonterminal is denoted by a sequence of characters that contain only lowercase letters, uppercase letters, and underscore. A terminal, on the other hand, is denoted by a sequence of characters enclosed by a pair of single quotes. YouGen_NG introduces an extra-grammatical feature called tags to restrict language generation. Tags are explained in the next section.

2.2 Tags

2.2.1 Syntax

The syntax of a tag is:

`{` *tag_name tag_parameter* `}`

There are two kinds of tags: covering-array and derivation-limiting. Covering-array tags are attached to rules while derivation-limiting tags are attached to non-terminals. YouGen_NG defines four tags: `cov`, `rdepth`, `depth`, and `count`. `cov` is a covering-array tag. `rdepth`, `depth`, and `count` are derivation-limiting tags.

2.2.2 `cov`

The syntax of a `cov` tag is: `{ cov [`$C_0, ..., C_n$`] }` where C_i is a coverage specification. Each coverage specification is a 2-tuple where the first element is a list of parameters and the second element is a strength. Domains are expressed as indexes. Each index refers to a term on a rule's right-hand side. For example, Figure 2.4(a) shows the `Call` grammar. The first rule has `CallerOS`, `ServerOS`, and `CalleeOS` on its right-hand side. They are indexed as 0, 1, and 2 respectively.

When a `cov` tag is attached to a rule, the language of the rule is reduced to a covering-array that satisfies all of the `cov` tag's coverage specifications.

For example, Figure 2.4(b) shows the output of running the `Call` grammar when {`cov [ ([0,1,2],2) ]`} is attached to the first rule. The output is a covering-array of strength 2 over the parameters {`CallerOS,ServerOS,CalleeOS`}. To prove this, let us consider the parameters {`CallerOS,CalleeOS`}. The cross product of these two parameters are {⟨`'Mac','Mac'`⟩, ⟨`'Mac','Win'`⟩, ⟨`'Win','Mac'`⟩, ⟨`'Win','Win'`⟩} which appear in lines 1, 3, 2, and 5 respectively. To complete the proof, the same exercise must be carried out for {`CallerOS,ServerOS`} and {`ServerOS,CalleeOS`}.

As another example, Figure 2.4(c) shows the output of running the `Call` grammar when {`cov [ ([0,2],2), ([1],1) ]`} is attached to the first rule. Lines 1 through 4 of the output is a covering-array of strength 2 over the parameters {`CallerOS,CalleeOS`}. The cross product of these two parameters, {⟨`'Mac','Mac'`⟩, ⟨`'Mac','Win'`⟩, ⟨`'Win','Mac'`⟩, ⟨`'Win','Win'`⟩}, appear in lines 1, 2, 3, and 4 respectively. Similarly, lines 5 through 7 of the output is a covering-array of strength 1 over the parameter {`ServerOS`}. The elements of the parameter, {⟨`'Lin'`⟩, ⟨`'Sun'`⟩, ⟨`'Win'`⟩}, appear in lines 5, 6, and 7 respectively. The output as a whole is a mixed-strength

```
Call ::= CallerOS ServerOS CalleeOS;

CallerOS ::= 'Mac';
CallerOS ::= 'Win';

ServerOS ::= 'Lin';
ServerOS ::= 'Sun';
ServerOS ::= 'Win';

CalleeOS ::= 'Mac';
CalleeOS ::= 'Win';
```

(a) `Call` grammar

```
1  Mac Lin Mac
2  Win Sun Mac
3  Mac Sun Win
4  Mac Win Mac
5  Win Lin Win
6  Win Win Win
```

(b) Output of running `Call` grammar with {`cov [([0,1,2],2)]`}

```
1  Mac Lin Mac
2  Mac Lin Win
3  Win Lin Mac
4  Win Lin Win
5  Mac Lin Mac
6  Mac Sun Mac
7  Mac Win Mac
```

(c) Output of running `Call` grammar with {`cov [([0,2],2),([1],1)]`}

Figure 2.4: `Call` grammar and its outputs

```
{rdepth 3}  Zeros;
            Zeros ::= '0';
            Zeros ::= '0' Zeros;
```

(a) `Zeros` grammar with `rdepth` tag

```
{depth 3}  Zeros;
           Zeros ::= '0';
           Zeros ::= '0' Zeros;
```

(b) `Zeros` grammar with `depth` tag

```
{count 3}  Zeros;
           Zeros ::= '0';
           Zeros ::= '0' Zeros;
```

(c) `Zeros` grammar with `count` tag

Figure 2.5: `Zeros` grammars with tags

covering array that satisfies {`cov [ ([0,2],2), ([1],1) ]`}.

2.2.3 `rdepth`

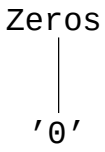
The syntax of a `rdepth` tag is: `{ rdepth` N `}`. N specifies the maximum number of times the nonterminal to which the `rdepth` tag is attached can appear on any path in the parse tree.

For example, Figure 2.5(a) shows the `Zeros` grammar with {`rdepth 3`} attached to the `Zeros` nonterminal. Figures 2.6 (a), (b), and (c) show the parse trees for one, two, and three zeros respectively. They conform to {`rdepth 3`} because they have at most three `Zeros` nonterminals in any parse tree path. Figure 2.6 (d), however, violates {`rdepth 3`} as it has four `Zeros` nonterminals on its rightmost parse tree path, exceeding the specified limit by one.

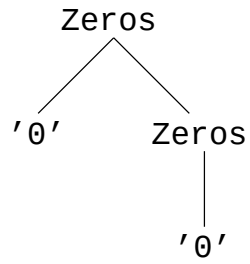
2.2.4 `depth`

The syntax of a `depth` tag is: `{ depth` N `}`. N specifies the maximum depth of any parse subtree rooted at the nonterminal to which the `depth` tag is attached.

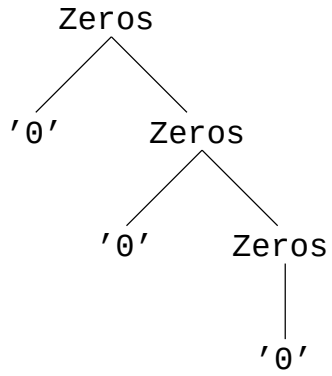
For example, Figure 2.5(b) shows the `Zeros` grammar with {`depth 3`} attached to the `Zeros` nonterminal. Figures 2.6 (a), (b), (c), and (d) show the parse trees for the first four strings generated by the `Zeros` grammar without tags. As shown in the



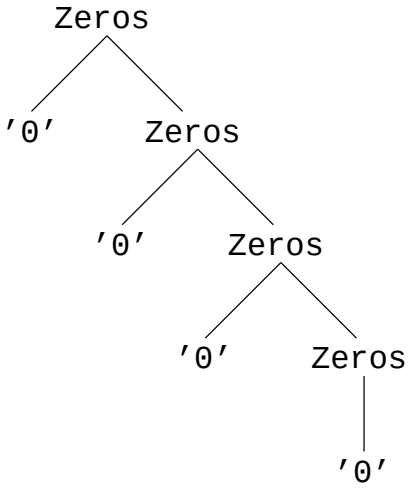
(a) Parse tree for `'0'`



(b) Parse tree for `'0' '0'`



(c) Parse tree for `'0' '0' '0'`



(d) Parse tree for `'0' '0' '0' '0'`

Figure 2.6: Parse trees for the first four strings generated by the `Zeros` grammar without tags

figure, all of these parse trees are rooted at the `Zeros` nonterminal. The first three strings conform to {`depth 3`} because counting from the root of their respective parse trees, the maximum tree depth is three. The fourth string, however, violates {`depth 3`} as its parse tree shows that the number of edges from the root to the rightmost leaf is four, exceeding the specified limit by one.

2.2.5 `count`

The syntax of a `count` tag is: `{ count` C `}`. Let N be a nonterminal to which a `count` tag of value C is attached. Let S be a sentential form in which N is the leftmost nonterminal and therefore is chosen for replacement. Then C specifies the maximum number of strings that can be derived from S.

For example, Figure 2.5(c) shows the `Zeros` grammar with {`count 3`} attached to the `Zeros` nonterminal. Since the start sentential form consists of only a `Zeros` nonterminal and therefore the `Zeros` nonterminal is the leftmost nonterminal, the sentential form can only be used to derive at most three strings.

2.3 Embedded Code

YouGen_NG allows user-defined code to be inserted into a text grammar. These code blocks are useful for:

1. limiting language generation,
2. manipulating strings,
3. setting up test environments, and
4. tearing down test environments.

2.3.1 Syntax

The syntax of an embedded code block is:

`{` *code_type code_block* `}`

code_type specifies the type of the embedded code and *code_block* contains the embedded code. The embedded code must be written in Python and therefore must be indented properly. YouGen_NG defines four types of embedded code: `precode`,

`postcode`, `global_precode`, and `global_postcode`. `precode` and `postcode` are attached to rules while `global_precode` and `global_postcode` are not attached to anything. Also, variables defined in one embedded code block are not accessible in another. The only exception to this rule is the variables that are defined in the `global_precode` block. These variables can be accessed in any embedded code block.

2.3.2 Definitions

- rule id: Each rule is assigned unique identifier. Let R be a rule with nonterminal N on its left-hand side. R has identifier of the form N_i where i is the number of rules that appear textually before R that also have N on their left-hand side. For example, Figure 2.7 shows the `Zeros` grammar with two rules. The first rule, `Zeros ::= '0';`, is referred to as $Zeros0$. The second rule, `Zeros ::= '0' Zeros;`, is referred to as $Zeros1$.

- yield: A rule's yield is the parse subtree created by replacing the rule's left-hand side with its right-hand side.

- ground: A rule's yield is ground when it consists only of terminals and not ground otherwise.

2.3.3 `precode`

Let R be a rule with precode attached. R's precode is referred to as R.precode. It is executed whenever R is chosen for application. The purpose of precode is to decide whether to proceed with rule application, that is, R is applied if and only if R.precode returns `True`. To illustrate, Figure 2.7 shows the `Zeros` grammar decorated with precode blocks. Because the language of the grammar is infinite, the `Zeros` nonterminal is attached with {`rdepth 3`} to restrict language generation.

Figure 2.8 shows the generation tree of the `Zeros` grammar. The generation tree is visited in depth-first traversal order. First, $Zeros0$.precode is executed. Because it returns `True`, rule $Zeros0$ is applied. After application of rule $Zeros0$, the first string of this grammar is generated. Second, $Zeros1$.precode is executed. Because it returns `True`, rule $Zeros1$ is applied. Third, $Zeros0$.precode is executed. Because it also returns `True`, rule $Zeros0$ is applied. After application of rule $Zeros0$, the second string of this grammar is generated. The generation of the third string proceeds in a similar fashion. Figure 2.9 shows the output of running this grammar.

```
{rdepth 3}
Zeros;

{precode
    print 'pre Zeros0'
    return True
}
Zeros ::= '0';

{precode
    print 'pre Zeros1'
    return True
}
Zeros ::= '0' Zeros;
```

Figure 2.7: `Zeros` grammar with `rdepth` and `precode` tags

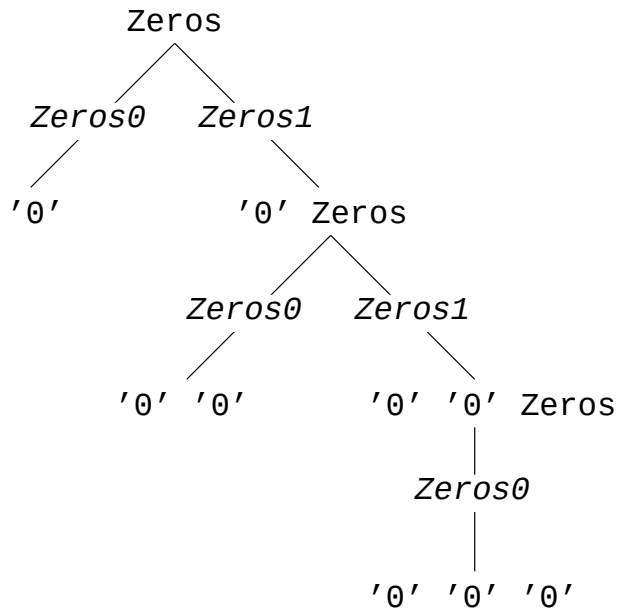


Figure 2.8: Generation tree for Figure 2.7

```
pre Zeros0
0
pre Zeros1
pre Zeros0
0 0
pre Zeros1
pre Zeros0
0 0 0
pre Zeros1
```

Figure 2.9: Output of running Figure 2.7

```
{rdepth 3}
Zeros;

{postcode
    print 'post Zeros0:', s
}
Zeros ::= '0';

{postcode
    print 'post Zeros1:', s
}
Zeros ::= '0' Zeros;
```

Figure 2.10: `Zeros` grammar with `rdepth` and `postcode` tags

2.3.4 `postcode`

Let R be a rule with postcode attached. R's postcode is referred to as R.postcode. It is executed when R's yield becomes ground. The purpose of postcode is to manipulate strings. To achieve that, YouGen_NG provides a variable called `s` that is accessible inside postcode blocks. `s` is a list containing a rule's yield. For each term on a rule's right-hand side, `s[0]` refers to the leftmost term, `s[1]` refers to the second leftmost term, and so on. To illustrate, Figure 2.10 shows the `Zeros` grammar decorated with postcode blocks.

Figure 2.11 shows the generation tree of the `Zeros` grammar. The generation tree is visited in depth-first traversal order. First, rule $Zeros0$ is applied. Because $Zeros0$'s yield is ground, $Zeros0$.postcode is executed. After $Zeros0$.postcode finishes, the first string of this grammar is generated. Second, rule $Zeros1$ is applied.

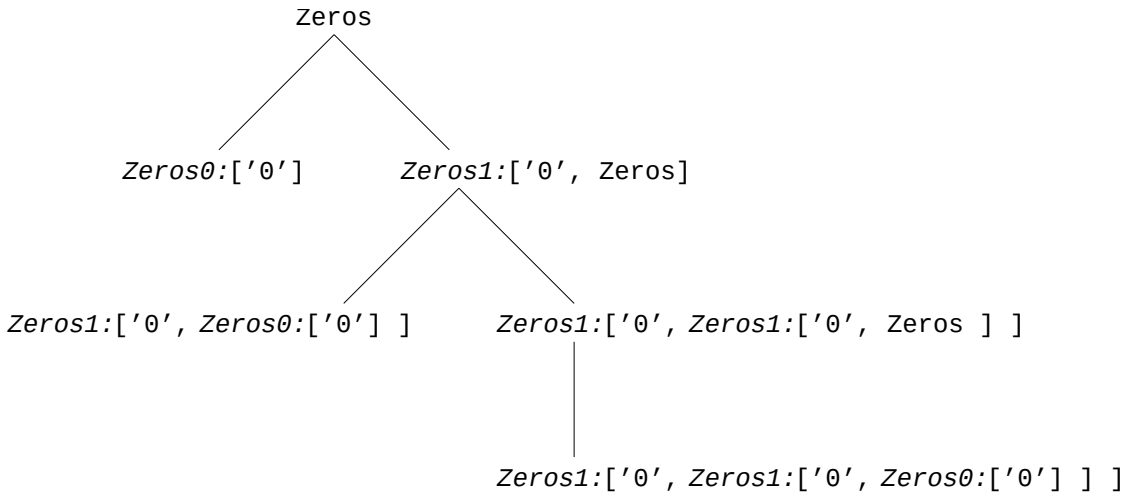


Figure 2.11: Generation tree for Figure 2.10

```
post Zeros0: ['0']
0
post Zeros0: ['0']
post Zeros1: ['0', ['0']]
0 0
post Zeros0: ['0']
post Zeros1: ['0', ['0']]
post Zeros1: ['0', ['0', ['0']]]
0 0 0
```

Figure 2.12: Output of running Figure 2.10

Because *Zeros*1's yield is not ground, *Zeros*1.postcode is not executed. Third, rule *Zeros*0 is applied. Because *Zeros*0's yield is ground, *Zeros*0.postcode is executed. At this point, *Zeros*1's yield also becomes ground and therefore *Zeros*1.postcode is executed. After *Zeros*1.postcode finishes, the second string of this grammar is generated. The generation of the third string proceeds in the similar fashion. Figure 2.12 shows the output of running this grammar.

2.3.5 `global_precode`

`global_precode` is executed before language generation. Its purpose is to set up the test environment.

2.3.6 `global_postcode`

`global_postcode` is executed after language generation. Its purpose is to tear down the test environment.

2.4 Terminal Generators

Let R be a rule with a terminal generator G. G takes a list of parameters as input and generates a terminal whenever R is invoked. A terminal generator is equivalent to multiple rule alternatives with the same left-hand side.

2.4.1 Syntax

A terminal generator is placed on the right-hand side of a rule and has the syntax:
generator_name `(` *parameter*1, *parameter*2, ... , *parameterN* `)`

2.4.2 The `Range` Terminal Generator

- Syntax: `Range(`*start*`,` *skip*`,` *count*`)`
- Semantics: Generates a range of integers that starts with *start* and incremented by *skip*. The total number of integers generated is specified by *count*.
- Example: Figure 2.13 (a) generates integers 0, 1, and 2.

2.4.3 The `List` Terminal Generator

- Syntax: `List(`*terminal*1`,` *terminal*2`,` ... `,` *terminalN*`)`
- Semantics: Generates *terminal*1, *terminal*2, ..., *terminalN*.
- Example: Figure 2.13 (b) generates strings `a`, `b`, and `c`.

2.4.4 The `File` Terminal Generator

- Syntax: `File(`*file_name*`)`. *file_name* is the path to a file that contains newline-separated strings.
- Semantics: Reads the file and generates one terminal for each line that is read.

```
S ::= Range(0, 1, 3);
```

(a) Example of `Range` terminal generator

```
S ::= List('a', 'b', 'c');
```

(b) Example of `List` terminal generator

```
S ::= File('m.txt');
```

(c) Example of `File` terminal generator

```
{global_precode
class Fibonacci(Terminal_generator):
    def __init__(self,param_list):
        self.param_list = param_list

    def generate(self):
        N = self.param_list[0]
        S = [0,1] # stores integer sequence

        for i in range(N):
            if i >= 2:
                S.append( S[-1] + S[-2] )
            yield S[i]
}
S ::= Fibonacci(6);
```

(d) Example of custom terminal generator

Figure 2.13: Examples of terminal generators

- Example: Let `m.txt` be a file that contains three strings `a`, `b`, and `c`, each string in its own line. Figure 2.13 (c) generates strings `a`, `b`, and `c`.

2.4.5 The Custom Terminal Generator

Custom terminal generators can be created by implementing a Python class in the `global_precode` block. The class must be a subclass of `Terminal_generator` and must contain two methods. The first method is `__init__`, the class constructor with parameter `param_list`, a list containing the parameters for the custom terminal generator. The second method is `generate`, invoked whenever the rule containing the custom terminal generator is applied. The generated terminals are returned by using the Python keyword `yield`. Figure 2.13 (d) shows a custom terminal generator that generates the first N Fibonacci numbers where N is obtained from `param_list`. The rule below the custom terminal generator generates the first six Fibonacci numbers.

2.5 Output Formats

The format of grammar output is specified by `G.set_output_format(`*format*`)` in the `global_precode` block where *format* can be any one of the following:

- `None`: Generates no output. This is useful when the output is intended for computer processing.
- `Flatten`: Outputs one string per line, where each line contains space-separated terminals. Figure 2.14(a) shows the flattened output of the `TwoBit` grammar.
- `Nested`: Outputs one string per line, each line contains a bracketed expression of the string generated. The brackets are used to show the parse tree structure of the strings. Figure 2.14(b) shows the nested output of the `TwoBit` grammar.
- `Custom`: Generates customized output as specified by a user-defined function F. F takes a nested list of terms as its input parameter and returns a formatted output. It is passed to the `G.set_output_format` method as the second parameter.

```
0 0
0 1
1 0
1 1
```

(a) Flattened

```
[['0'], ['0']]
[['0'], ['1']]
[['1'], ['0']]
[['1'], ['1']]
```

(b) Nested

Figure 2.14: Different formats for `TwoBit` output

2.6 Generation Tree Formats

The format of a generation tree is specified by `G.set_gentree_format(`*format*`)` in the `global_precode` block where *format* can be any one of the following:

- `None`: Skips tree generation. This can be used to speed up language generation as creating a generation tree involves writing a file to disk.
- `XML`: Creates a generation tree in eXtensible Markup Language (XML) format and saves it into a file. Each node in the generation tree is denoted by a `gentree` element. Each `gentree` element contains four different child elements: `s`, `id`, zero or more `gentree`s, and `count`. The `s` element contains a sentential form, the `id` element contains the identifier of the rule used to derive the sentential form, and the `count` element contains a positive integer specifying the number of strings that can be derived from the sentential form. Figure 2.15 shows the XML-based generation tree for the `TwoBit` grammar. Note that the `id` of the top `gentree` element is `None`. This is because no rule was used to derive the start sentential form `[TwoBit]`.

```
<gentree>
    <id>None</id>
    <s>[TwoBit]</s>
    <gentree>
        <id>TwoBit0</id>
        <s>[[Bit, Bit]]</s>
        <gentree>
            <id>Bit0</id>
            <s>[[['0'], Bit]]</s>
            <gentree>
                <id>Bit0</id>
                <s>[[['0'], ['0']]]</s>
                <count>1</count>
            </gentree>
            <gentree>
                <id>Bit1</id>
                <s>[[['0'], ['1']]]</s>
                <count>1</count>
            </gentree>
            <count>2</count>
        </gentree>
        <gentree>
            <id>Bit1</id>
            <s>[[['1'], Bit]]</s>
            <gentree>
                <id>Bit0</id>
                <s>[[['1'], ['0']]]</s>
                <count>1</count>
            </gentree>
            <gentree>
                <id>Bit1</id>
                <s>[[['1'], ['1']]]</s>
                <count>1</count>
            </gentree>
            <count>2</count>
        </gentree>
        <count>4</count>
    </gentree>
    <count>4</count>
</gentree>
```

Figure 2.15: Generation tree for `TwoBit` grammar

Chapter 3

YouGen_NG Requirements: API Grammar

API grammars, in YouGen_NG terminology, refer to the grammars that are written in Python. An API grammar is developed by using the API provided by the YouGen_NG runtime library, hereafter referred to as YouGen_NG API. Because API grammars are written in Python, they are executable by themselves and therefore do not need to be translated in order to generate their languages. Instead, language generation is accomplished by running the Python interpreter against an API grammar. Also, YouGen_NG API has all the features that are supported by text grammars. Given a text grammar, one can develop an equivalent grammar in API form by using YouGen_NG API.

This chapter focuses on the development of API grammars, in particular how to incorporate the features that are supported by text grammars. The syntax and semantics of these features are skipped here because they were explained in the previous chapter.

3.1 Context-Free Grammar

A context-free grammar is a grammar without tags, embedded code, and terminal generators. Figure 3.1(a) shows a context-free grammar called `TwoBit`. It is used as an introductory example for API grammars because of its simplicity. Figure 3.1(b) shows the `TwoBit` grammar in API form. This grammar is equivalent to the text grammar shown in Figure 3.1(a). First, YouGen_NG API is imported as shown in

```
TwoBit ::= Bit Bit;

Bit ::= '0';
Bit ::= '1';
```

(a) `TwoBit` text grammar

```
1  from youGen_NG import *
2
3  G = Grammar()
4  G.append_rule( Rule( {}, 'TwoBit', [V('Bit'),V('Bit')] ) )
5  G.append_rule( Rule( {}, 'Bit', [T('0')] ) )
6  G.append_rule( Rule( {}, 'Bit', [T('1')] ) )
7
8  for s in generate_language(G,'TwoBit'):
9      print s
```

(b) `TwoBit` API grammar

Figure 3.1: `TwoBit` text and API grammars

line 1. With this import statement, the current namespace is merged with that of the `YouGen_NG` module. Therefore, invoking a function or class defined in the `YouGen_NG` module does not need to be prefixed with `YouGen_NG.`.

Second, a `Grammar` object named `G` is created as shown in line 3. A `Grammar` object is a container for rules. It is created by invoking the `Grammar` constructor which takes no parameter.

Third, three rules are created in lines 4 through 6 and added to the `Grammar` object `G` by invoking `G.append_rule`. A rule is created by invoking the `Rule` constructor which takes three parameters. The first parameter is a dictionary of tags where the keys are tag names and the values are tag parameters. The second parameter is a rule's left-hand side, a nonterminal. The third parameter is a rule's right-hand side, a list containing instances of terminals and nonterminals. Terminals are instantiated by calling the `T` constructor while nonterminals are instantiated by calling the `V` constructor. For example, `TwoBit ::= Bit Bit;` is created by invoking the `Rule` constructor with parameter one being an empty dictionary because the rule has no tags, parameter two being a nonterminal `TwoBit`, and parameter three being a list containing two instances of nonterminal `Bit`. Similarly, `Bit ::= '0';` is created by

```
1  from youGen_NG import *
2
3  G = Grammar()
4  G.tag_nonterminal({'rdepth':3},V('Zeros'))
5  G.append_rule( Rule( {}, 'Zeros', [T('0')] ) )
6  G.append_rule( Rule( {}, 'Zeros', [T('0'),V('Zeros')] ) )
7
8  for s in generate_language(G,'Zeros'):
9      print s
```

Figure 3.2: `Zeros` API grammar with `rdepth` tag

invoking the `Rule` constructor with parameter one being an empty dictionary because the rule has no tags, parameter two being a nonterminal `Bit`, and parameter three being a list containing one instance of terminal `'0'`.

Finally, lines 8 and 9 generate the language of `TwoBit` and print it to the standard output, one string per line. Language generation is accomplished by invoking the `generate_language` function which takes two parameters: a `Grammar` object and a start symbol. In the case of the `TwoBit` grammar shown in Figure 3.1(b), the `Grammar` object is `G` and the start symbol is `TwoBit`.

3.2 Derivation-Limiting Tags

Derivation-limiting tags attach to nonterminals. Tags that are derivation-limiting are `rdepth`, `depth`, and `count`.

Attaching a derivation-limiting tag to a nonterminal is accomplished by invoking `G.tag_nonterminal` where `G` is a `Grammar` object. The `tag_nonterminal` method takes two parameters. The first parameter is a dictionary of tags where the keys are tag names and the values are tag parameters. The second parameter is an instance of a nonterminal. Figure 3.2 shows the `Zeros` grammar with {`rdepth 3`} attached to nonterminal `Zeros` in line 4.

3.3 Covering-Array Tags

Covering-Array tags attach to rules. The only tag that generates covering-array is `cov`.

```
1   from youGen_NG import *
2
3   G = Grammar()
4   G.append_rule( Rule( {'cov':[ ([0,1,2],2) ]},
5       'Call', [V('CallerOS'),V('ServerOS'),V('CalleeOS')] ) )
6   G.append_rule( Rule( {}, 'CallerOS', [T('Mac')] ) )
7   G.append_rule( Rule( {}, 'CallerOS', [T('Win')] ) )
8   G.append_rule( Rule( {}, 'ServerOS', [T('Lin')] ) )
9   G.append_rule( Rule( {}, 'ServerOS', [T('Sun')] ) )
10  G.append_rule( Rule( {}, 'ServerOS', [T('Win')] ) )
11  G.append_rule( Rule( {}, 'CalleeOS', [T('Mac')] ) )
12  G.append_rule( Rule( {}, 'CalleeOS', [T('Win')] ) )
13
14  for s in generate_language(G,'Call'):
15      print s
```

Figure 3.3: `Call` API grammar with `cov` tag

Attaching a covering-array tag to a rule is done at rule creation. As mentioned before, a rule is created by invoking the `Rule` constructor with the first parameter being a dictionary of tags. Let R be a rule with a covering-array tag of the form {`cov` $[C_0, ..., C_n]$} where C_i is a coverage specification. Creating rule R by invoking the `Rule` constructor with {`'cov'`:$[C_0, ..., C_n]$} as the first parameter attaches the covering-array tag to rule R. Figure 3.3 shows the `Call` grammar with {`cov [ ([0,1,2],2) ]`} attached to the `Call` rule in line 4.

3.4 Embedded Code

YouGen_NG defines four types of embedded code: `precode`, `postcode`, `global_precode`, and `global_postcode`. `precode` and `postcode` attach to rules while `global_precode` and `global_postcode` are standalone.

Attaching {`precode` C} to rule R is a two-step process. First, create a function named F with C as the function body. Function F takes no parameter. Second, create rule R by invoking the `Rule` constructor with {`'precode'`:F} as the first parameter. Figure 3.4 shows the `Zeros` grammar decorated with $Zeros0$`.precode` and $Zeros1$`.precode`. $Zeros0$`.precode` is defined in lines 8 through 10 and attached to rule $Zeros0$ in line 22. $Zeros1$`.precode` is defined in lines 11 through 15 and

attached to rule *Zeros*1 in line 25.

Similarly, attaching {`postcode` *C*} to rule *R* is a two-step process. First, create a function named *F* with *C* as the function body. Function *F* takes a nested list of strings as its parameter. Second, create rule *R* by invoking the `Rule` constructor with {`'postcode'`:*F*} as the first parameter. Figure 3.4 shows the `Zeros` grammar decorated with *Zeros*0`.postcode` and *Zeros*1`.postcode`. *Zeros*0`.postcode` is defined in lines 16 and 17 and attached to rule *Zeros*0 in line 22. *Zeros*1`.postcode` is defined in lines 18 and 19 and attached to rule *Zeros*1 in line 25.

{`global_precode` *C*} must be defined after grammar creation and before `precode` and `postcode` functions for two reasons. First, it needs to configure output formats and generation tree formats through the `Grammar` object. Second, it needs to define global variables for other embedded code blocks. Figure 3.4 shows the `global_precode` in lines 5 and 6.

{`global_postcode` *C*} must be defined after call to the `generate_language` function such that the `global_postcode` is executed after the language generation is complete. Figure 3.4 shows the `global_postcode` in line 31.

3.5 Terminal Generators

A terminal generator is located on the right-hand side of a rule. It takes a list of integers or strings as input and yields a terminal every time the rule to which the terminal generator is attached is applied. YouGen_NG defines four types of terminal generators: `Range`, `List`, `File`, and custom. Figure 3.5 shows a grammar that uses all four types of terminal generators:

- `Range` takes a list of three integers as parameter as shown in line 18.
- `List` takes a list of strings as parameter as shown in line 19.
- `File` takes a list of one string as parameter as shown in line 20.
- A custom terminal is created by implementing a Python class that inherits `Terminal_generator`. The class must contain two methods: `__init__` and `generate`. `__init__` is the class constructor that takes a list of integers or strings as parameter. `generate` is a Python generator function. Figure 3.5 shows a custom terminal generator named `Fibonacci` which takes a positive

```
1  from youGen_NG import *
2
3  G = Grammar()
4
5  n = 0
6  print '\tglobal_precode reached'
7
8  def Zeros0_precode():
9      print '\tZ0 precode reached'
10     return True
11 def Zeros1_precode():
12     global n
13     print '\tZ1 precode reached'
14     n += 1
15     return n < 3
16 def Zeros0_postcode(s):
17     print '\tZ0 postcode:',s
18 def Zeros1_postcode(s):
19     print '\tZ1 postcode:',s
20
21 G.append_rule( Rule(
22     {'precode':Zeros0_precode,'postcode':Zeros0_postcode},
23     'Zeros', [T('0')] ) )
24 G.append_rule( Rule(
25     {'precode':Zeros1_precode,'postcode':Zeros1_postcode},
26     'Zeros', [T('0'),V('Zeros')] ) )
27
28 for s in generate_language(G,'Zeros'):
29     print s
30
31 print '\tglobal_postcode reached'
```

Figure 3.4: `Zeros` API grammar with embedded code

```
1  from youGen_NG import *
2
3  G = Grammar()
4
5  class Fibonacci(Terminal_generator):
6      def __init__(self,param_list):
7          self.param_list = param_list
8
9      def generate(self):
10         N = self.param_list[0]
11         S = [0,1] # stores integer sequence
12
13         for i in range(N):
14             if i >= 2:
15                 S.append( S[-1] + S[-2] )
16             yield S[i]
17
18 G.append_rule( Rule( {}, 'S', [Range([0, 1, 3])] ) )
19 G.append_rule( Rule( {}, 'S', [List(['a', 'b', 'c'])] ) )
20 G.append_rule( Rule( {}, 'S', [File(['m.txt'])] ) )
21 G.append_rule( Rule( {}, 'S', [Fibonacci([6])] ) )
22
23 for s in generate_language(G,'S'):
24     print s
```

Figure 3.5: A grammar with terminal generators

integer N as input and yields the first N Fibonacci numbers. `Fibonacci` is defined in lines 5 through 16 and instantiated in line 21.

3.6 Output Formats

Output formats are specified by `G.set_output_format(`F`)` where `G` is a `Grammar` object and F is one of the supported formats. F can be `NONE`, `FLATTEN`, `NESTED`, or `CUSTOM`.

3.7 Generation Tree Formats

Generation tree formats are specified by `G.set_gentree_format(`F`)` where `G` is a `Grammar` object and F is one of the supported formats. F can be `NONE` or `XML`.

Chapter 4

YouGen_NG Design and Implementation

The implementation of YouGen_NG is divided into three components. The first component is grammar translation. It is responsible for parsing a text grammar and generating the equivalent API grammar. The second component is grammar creation. It provides the API to create a grammar. The third component is language generation. It is responsible for generating the language of a grammar, formatting the strings in the language, and logging the generation tree. The following sections explain the modules that are used to implement the three components. The modules' number of classes, functions, and lines of code are given followed by a short description of the classes and functions.

4.1 Grammar Translation

Translating a text grammar into its equivalent API grammar is a three-step process: lexical analysis, parsing, and code generation. Each of these three steps is implemented as a module. In addition, a main module is implemented that is responsible for reading the file path to a text grammar and initiating the three-step process.

4.1.1 Lexical Analyzer Module

Lines of code: 145
Classes: 1
Functions: 0

Lexical analysis is accomplished by the `Lexical_analyzer` class. `Lexical_analyzer` is instantiated by passing the entire text grammar as a string into its constructor. Tokens are retrieved one at a time by calling the `Lexical_analyzer.next` method. The method generates tokens by matching the text grammar against a set of regular expressions. Each token is a three-tuple: token value, token type, and line number. For example, a terminal `'0'` has token value of `0` and token type of `TERMINAL`. Line number refers to the line in which the token was found. It is useful for grammar developers whenever there is a parsing error.

4.1.2 Parser Module

Lines of code: 188
Classes: 1
Functions: 0

Parsing is accomplished by the `Parser` class. `Parser` is instantiated by passing the entire text grammar as a string into its constructor. Calling the `Parser.grammar` method initiates parsing. The method calls other helper methods to extract tags and rules from the text grammar. The outcome is a list of tags and rules stored in `Parser.tag_rule_list`. A tag is denoted by a three-tuple: ⟨`tag`, *tag_name*, *tag_parameter*⟩. For example, {`count 3`} has the form ⟨`tag`, `count`, `3`⟩. A rule is also denoted by a three-tuple: ⟨`rule`, *lhs*, *rhs*⟩ where *lhs* refers to the rule's left-hand side and *rhs* refers to the rule's right-hand side. For example, `Zeros ::= '0';` has the form ⟨`rule`, `Zeros`, [⟨`'0'`, `TERMINAL`⟩]⟩.

4.1.3 Code Generator Module

Lines of code: 215
Classes: 1
Functions: 0

Code generation is accomplished by the `Code_generator` class. `Code_generator` is instantiated by passing `Parser.tag_rule_list` into its constructor. Calling the `Code_generator.generate` method initiates code generation. The method calls other

helper methods to generate code for embedded code blocks, tags and rules. Upon completion, a string that contains the generated code is returned.

4.1.4 Main Module

Lines of code: 23
Classes: 0
Functions: 0

This module parses a text grammar and generates its equivalent API grammar. First, a text grammar is read from a file. A text grammar file has extension `.gr`. Second, the text grammar is parsed by calling `Parser.grammar`. Third, an equivalent API grammar is generated by calling `Code_generator.generate`. Last, the API grammar is written into a file. It has the same file path as the text grammar except that its file extension is `.py` instead of `.gr`.

4.1.5 Translation Procedure

The procedure for grammar translation is illustrated by the call graph shown in Figure 4.1. The call graph is laid out such that (1) the called functions or methods are indented with respect to their calling functions or methods, (2) the functions or methods with the same indentation levels are executed sequentially from top to bottom, and (3) the three phases of grammar translations are shown in italics.

Grammar translation starts by reading a text grammar from a file into a string. The string is passed to `Parser`'s constructor which instantiates `Lexical_analyzer`. The `Parser.grammar` method in line 8 starts the lexical analysis and parsing phases. The method calls its helper methods to extract tags and rules from the text grammar. A tag is extracted by `Parser.tag` whereas a rule is extracted by `Parser.rule`. `Parser.rule` calls two methods: `Parser.terminal_generator` and `Parser.term`. The former extracts a terminal generator while the latter extracts a nonterminal or terminal.

`Parser.get_token` in line 9 retrieves a token by calling `Lexical_analyzer.next`. `Lexical_analyzer.next_embedded_code` is called by `Lexical_analyzer.next` whenever there is an embedded code block in the text grammar. If the token returned by `Parser.get_token` is not as expected, the `Parser.error` method is called. This method prints the found token, the expected token, and the line in which the found token was encounterd to the standard output.

```
1       main
2           open
3           read
4           Parser.__init__
5               Lexical_analyzer.__init__
6           close
7   Lexical Analysis and Parsing
8           Parser.grammar
9               Parser.get_token
10                  Lexical_analyzer.next
11                      Lexical_analyzer.next_embedded_code
12              Parser.tag
13              Parser.rule
14                  Parser.terminal_generator
15                  Parser.term
16              Parser.error
17  Code Generation
18          Code_generator.__init__
19          open
20          Code_generator.generate
21              generate_global_precode
22              generate_global_postcode
23              generate_precodes
24              generate_postcodes
25              generate_rules
26          write
27          close
```

Figure 4.1: Call graph of grammar translation

The `Code_generator.generate` method in line 20 starts the code generation phase. The method, together with its helper methods, generates code for defining embedded code blocks, creating a grammar object, appending rules to the grammar object, and attaching tags to the rules. The `write` function in line 26 writes the generated code into a file.

4.2 Grammar Creation

4.2.1 Grammar Module

Lines of code: 140
Classes: 1
Functions: 0

Creating a grammar instance is accomplished by instantiating the `Grammar` class. `Grammar.append_rule` appends a rule to the grammar. `Grammar.tag_nonterminal` attaches a tag to a nonterminal. The `Grammar` class also contains methods for configuring the output and generation tree formats. These are accomplished by `Grammar.set_output_format` and `Grammar.set_gentree_format` respectively. The file used for saving the generation tree is specified by `Grammar.set_gentree_file`.

4.2.2 Rule Module

Lines of code: 124
Classes: 8
Functions: 0

A rule R is created by instantiating the `Rule` class with three parameters: a set of tags, R's left-hand side, and R's right-hand side. R's right-hand side is a list of terms or terminal generators. A term is either nonterminal or terminal. A nonterminal is created by instantiating the `NT` class while a terminal is created by instantiating the `T` class. Both `NT` and `T` are subclasses of `Term`. A terminal generator is created by instantiating one of the following classes: `Range`, `List`, `File`, and custom defined. All four of these classes are subclasses of `Terminal_generator` and therefore must override `Terminal_generator.generate`: a method that yields terminals one at a time.

4.3 Language Generation

4.3.1 Tags Module

Lines of code: 588
Classes: 2
Functions: 7

As mentioned in the previous chapters, there are two kinds of tags: derivation-limiting and covering-array. Limiting a derivation is accomplished by the `prune_tree` function. The function returns `True` if the derivation should stop and `False` otherwise. `prune_tree` is invoked just before the application of a rule. Covering-arrays are generated by the `generate_cov` function. The function takes the list of terms on a rule's right-hand side as parameters and returns a n-cover over the selected parameters. n can be 1, 2, 3, or N where N is the number of terms on the rule's right-hand side. `generate_cov` is invoked whenever the rule to apply has a `cov` tag.

4.3.2 Output Formatting Module

Lines of code: 36
Classes: 0
Functions: 2

A string in the language is formatted by two functions. `flatten` generates space-separated terminals. `nested` formats the string into a bracketed expression. The brackets are used to show the parse tree structure.

4.3.3 Generation Tree Logging Module

Lines of code: 177
Classes: 2
Functions: 0

This module has two classes: `Gentree_xml` and `Gentree_default`. The former logs the generation tree in XML format while the latter does nothing. Both of these classes implements the following methods:

- `__init__`: Opens the log file for writing.
- `destructor`: Closes the log file.

- `log_start`: Logs the start of rule application. This method logs both the rule identifier and the sentential form after rule application.
- `log_end`: Logs the end of rule application.

4.3.4 Derivation Algorithm

Deriving the strings in the language is accomplished by the `generate_language` function as shown in Figure 4.2. The notations used in the function are defined as follows:

- G: A `Grammar` object that contains a set of rules.
- S: A sentential form.
- S': Derived by performing a deep copy on S. A deep copy is needed to prevent backtracking.
- R`.lhs`: Refers to rule R's left-hand side: a nonterminal.
- R`.rhs`: Refers to rule R's right-hand side: a list of nonterminals, terminals, and terminal generators.

`generate_language` is a recursive function as it calls itself on lines 9, 16, and 20. The recursion stops only when sentential form S is ground as shown in lines 2 and 3. At this point, the function returns the ground sentential form which is essentially a string in the language. Otherwise, a deep copy is performed on S and saved into S' as shown in line 4. A deep copy is needed so that changes to S' do not affect S, the sentential form that is also used by the previous recursive call to the `generate_language` function. How the rest of the function proceeds depends on N, the leftmost terminal generator or nonterminal extracted from S'. If N is a terminal generator, N`.generate()` is invoked to generate a list of terminals as shown in line 7. For each of these terminals T, N in S' is replaced by T and the language generation proceeds by a recursive call to the `generate_language` function. If N is a nonterminal, rules that start with N are extracted from the grammar object G as shown in line 12. For each of these rules R, R is checked for the existence of covering-array tags. If a covering-array tag is present, the `generate_cov` function is invoked to generate a subset of $L(N)$. Otherwise, the complete $L(N)$ is generated.

As shown in line 14 of Figure 4.2, the `generate_cov` function takes a grammar object G and a rule R as input and returns a set of tuples. The rule R has a `cov`

```
1   generate_language(G,S)
2       if S is ground
3           return S
4       S′ = deep copy of S
5       N = leftmost terminal generator or nonterminal in S′
6       if N is a terminal generator
7           for terminal T in N.generate()
8               replace N in S′ with T
9               for s in generate_language(G,S′)
10                  yield s
11      else
12          for rule R in G where R.lhs == N
13              if R has cov tag
14                  for tuple T in generate_cov(G,R)
15                      replace N in S′ with T
16                      for s in generate_language(G,S′)
17                          yield s
18              else
19                  replace N in S′ with R.rhs
20                  for s in generate_language(G,S′)
21                      yield s
```

Figure 4.2: Derivation algorithm

```
1   generate_cov(G,R)
2       N = len(R.rhs)
3       P = parameter trees of size N
4       for i in 0..N - 1
5           S = sentential form consists of only R.rhs[i]
6           for s in generate_language(G,S)
7               P[i].append(s)
8       for coverage specification C in R.cov
9           for tuple T in cov(P,C)
10              yield T
```

Figure 4.3: Covering array algorithm

tag attached to it as already verified in line 13. The algorithm for the `generate_cov` function is detailed in Figure 4.3. First, parameter trees of size N are created where N is equal to the number of terms appearing on rule R's right-hand side. Second, each parameter tree is populated with the language of the corresponding term. Finally, a function called `cov` takes the parameter trees and a coverage specification as input and returns a set of tuples. Because a `cov` tag may contain more than one coverage specification, a `for` loop is used in line 8 in order to iterate through all coverage specifications. With this approach, the output of the `generate_cov` is the concatenation of the covering arrays created from multiple coverage specifications.

Chapter 5

Generation Tree Analysis

Most grammars used for GBTG are complex. Testers often have difficulty understanding how each string in the language is generated, let alone controlling the size of the language. As a result, the grammars tend to generate too many test cases to be useful. For example, Figure 5.1 shows a moderately complex grammar that generates book catalogs. A catalog consists of one or more books. Each book has one title and three chapters. A book title can be empty, `TTT`, `ttt`, or omitted. Each chapter consists of one or more sections. Each section has a name which can be empty, `SSS`, `sss`, or omitted. The grammar is decorated with *BooksTag* and *ChaptersTag* which are place holders for tags. For example, replacing *BooksTag* with {`rdepth 2`} and *ChaptersTag* with nothing generates 65,792 strings. It may be hard to believe that such a small grammar generates so many strings.

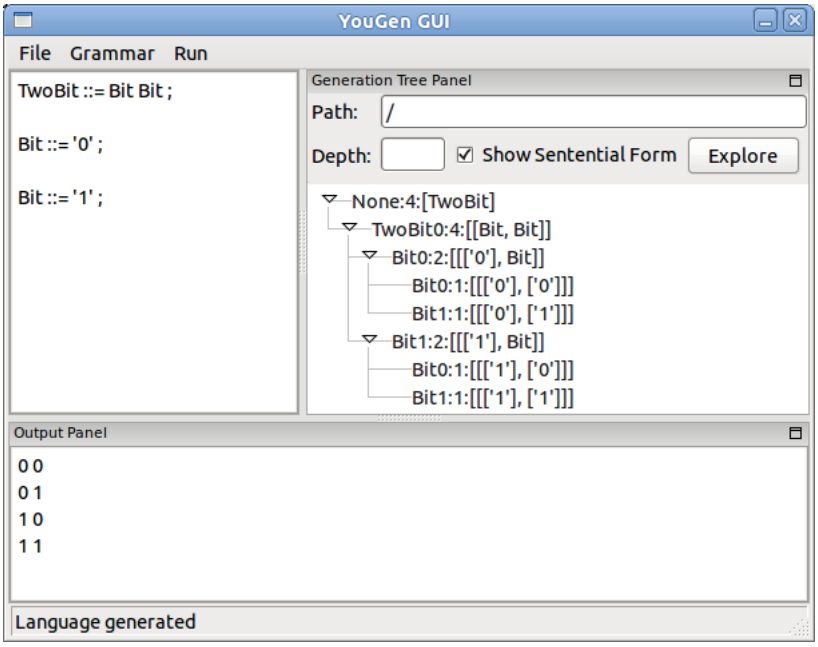
Generation trees can help testers understand the language generation process. As mentioned before, YouGen_NG logs generation trees in XML format. For each rule application, a generation tree log displays the identifier of the rule applied, the sentential form resulting from the application of the rule, and the number of strings that can be derived from the sentential form. By looking at a generation tree log, testers not only able to visualize the derivation steps for each string generated but also identify the rules that generate the most strings. Testers can then attach tags to the rules or nonterminals of their choice to reduce the size of the language. However, generation tree logs—written in XML format—are intended for computer processing and therefore are difficult for humans to read, especially when the logs are large. The following sections show that this problem is overcome by Dervish [29], an application that not only displays generation tree logs in human-readable form but also allows testers to zoom in on the generation subtrees of interest.

```
                Catalog ::= '<BOOKS>' Books '</BOOKS>';

BooksTag        Books;
                Books ::= Book;
                Books ::= Book Books;

                Book ::= '<BOOK>' Title Chapters '</BOOK>';

                Title ::= '<TITLE>' List('','TTT','ttt') '</TITLE>';
                Title ::= '';

ChaptersTag     Chapters ::= Chapter Chapter Chapter;
                Chapter ::= '<CHAPTER>' Sections '</CHAPTER>';

                Sections ::= Section;
                Section ::= '<SECTION>' Name '</SECTION>';

                Name ::= '<NAME>' List('','SSS','sss') '</NAME>';
                Name ::= '';
```

Figure 5.1: `Catalog` Grammar

```
TwoBit ::= Bit Bit;

Bit ::= '0';
Bit ::= '1';
```

(a) `TwoBit` grammar



(b) `TwoBit` generation tree

Figure 5.2: `TwoBit` grammar and generation tree

5.1 Dervish

Dervish is a GBTG learning tool that facilitates exploration of generation trees. It takes a grammar as input and displays its language and generation tree by invoking YouGen_NG runtime library. It also allows testers to manipulate a grammar by adding tags, deleting tags, and modifying tag parameters. Dervish is developed by David Ly-Gagnon while the facility for generation tree logging is developed by me. Figure 5.2 (b) shows the `TwoBit` grammar displayed by Dervish. The GUI consists of three panels: the left panel displays the grammar, the right panel displays the generation tree, and the bottom panel displays the language of the grammar. The generation tree can be expanded or contracted by clicking the triangles that appear to the left-hand of the tree nodes.

Above the generation tree are three GUI widgets that control the display of the

generation tree. The first widget is a textbox labelled as `Path`. It specifies the path to the root of the generation subtree to explore. The second widget is a textbox labelled as `Depth`. It specifies the depth of the generation subtree to explore. The third widget is a checkbox labelled as `Show Sentential Form`. It toggles the display of sentential forms that appear in the tree nodes. The settings come into effect when the `Explore` button is clicked. For example, Figure 5.2 (b) shows the `TwoBit` grammar with `Path` set to `/`, `Depth` set to empty, and `Show Sentential Form` set to checked. As a result, the generation tree of the `TwoBit` grammar is displayed from root to bottom with sentential forms displayed in each tree node.

The following subsections explain the generation tree displayed by Dervish. Each node in the generation tree shows the identifier of the rule applied, the sentential form resulting from the application of the rule, and the number of strings that can be derived from the sentential form.

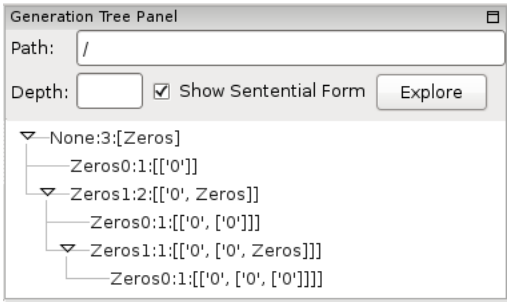
5.1.1 `TwoBit` Grammar

The right panel in Figure 5.2 (b) shows the generation tree for the `TwoBit` grammar shown in Figure 5.2 (a). The root of the generation tree is `None:4:[TwoBit]`. `None` means that none of the rules was applied to derive the start sentential form `[TwoBit]`. Also, the number of children for each tree node is equal to the number of rule alternatives for the nonterminal chosen for replacement. For example, the nonterminal in `[TwoBit]` chosen for replacement is `TwoBit` which has one rule alternative: `TwoBit0`. Therefore, node `None:4:[TwoBit]` has only one child node `TwoBit0:4:[[Bit, Bit]]`. As another example, the nonterminal in `[[Bit, Bit]]` chosen for replacement is `Bit` which has two rule alternatives: `Bit0` and `Bit1`. Therefore, node `TwoBit0:4:[[Bit, Bit]]` has two child nodes `Bit0:2:[[['0'], Bit]` and `Bit1:2:[[['1'], Bit]]`.

Language generation proceeds in depth-first traversal order. The following explains the steps taken to derive the first two strings, starting from the sentential form `[TwoBit]`. First, rule `TwoBit0` is applied to derive `[[Bit, Bit]]`. Second, rule `Bit0` is applied to derive `[[['0'], Bit]]`. Third, rule `Bit0` is applied to derive `[[['0'], ['0']]]`. At this point, the first string is generated. The generation of the second string starts from the sentential form `[[['0'], Bit]]`. Rule `Bit1` is applied to derive `[[['0'], ['1']]]`. At this point, the second string is generated.

```
{rdepth 3}  Zeros;
            Zeros ::= '0';
            Zeros ::= '0' Zeros;
```

(a) `Zeros` grammar



(b) `Zeros` generation tree

Figure 5.3: `Zeros` grammar and generation tree

5.1.2 `Zeros` Grammar

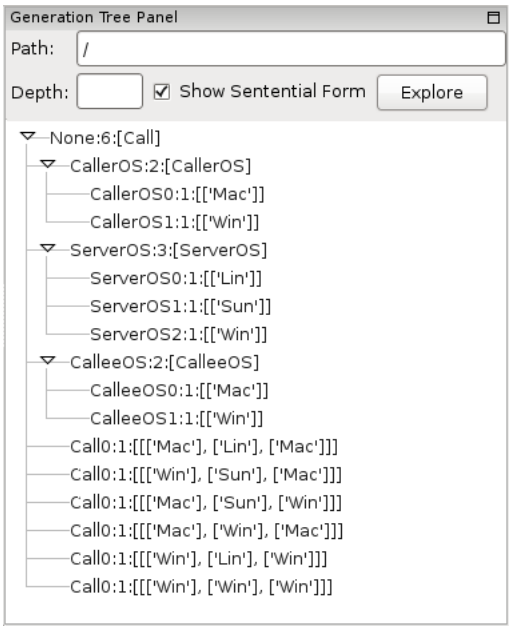
Figure 5.3 (b) shows the generation tree for the `Zeros` grammar shown in Figure 5.3 (a). The grammar has {`rdepth 3`} attached to the `Zeros` nonterminal and therefore generates three strings. Again, language generation proceeds in depth-first traversal order. The following explains the steps taken to derive the first two strings, starting from the sentential form `[Zeros]`. First, rule `Zeros0` is applied to derive `[['0']]`. At this point, the first string is generated. The generation of the second string starts from the sentential form `[Zeros]`. First, rule `Zeros1` is applied to derive `[['0', Zeros]]`. Second, rule `Zeros0` is applied to derive `[['0', ['0']]]`. At this point, the second string is generated.

5.1.3 `Call` Grammar

The language of the `Call` grammar shown in Figure 5.4 (a) can be expressed as $L(\texttt{CallerOS}) \times L(\texttt{ServerOS}) \times L(\texttt{CalleeOS})$: the size of $L(\texttt{CallerOS})$ is 2, the size of $L(\texttt{ServerOS})$ is 3, and the size of $L(\texttt{CalleeOS})$ is 2. The size of $L(\texttt{Call})$ is therefore $2 \times 3 \times 2 = 12$. The following example shows that $L(\texttt{Call})$ can be reduced by the use of a covering-array tag. Figure 5.4 (b) shows the generation tree for the `Call` grammar with {`cov [ ([0,1,2],2) ]`} attached to rule `Call0`. Language generation starts from the sentential form `[Call]`. Because rule `Call0` is attached with a covering-

```
Call ::= CallerOS ServerOS CalleeOS;

CallerOS ::= 'Mac';
CallerOS ::= 'Win';

ServerOS ::= 'Lin';
ServerOS ::= 'Sun';
ServerOS ::= 'Win';

CalleeOS ::= 'Mac';
CalleeOS ::= 'Win';
```

(a) `Call` grammar



(b) Generation tree for `Call` grammar with {`cov [ ([0,1,2],2) ]`}

Figure 5.4: `Call` grammar and generation tree

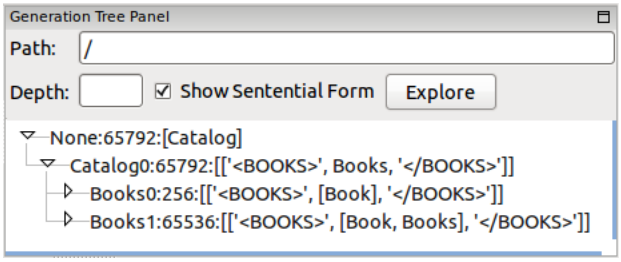


Figure 5.5: Generation tree for `Catalog` grammar with {`rdepth 2`}; tree is displayed from root to depth 2

array tag, language generation proceeds differently. First, L(`CallerOS`) is generated. Two rule alternatives for nonterminal `CallerOS` are applied to derive `[['Mac']]` and `[['Win']]`. Second, L(`ServerOS`) is generated. Three rule alternatives for nonterminal `ServerOS` are applied to derive `[['Lin']]`, `[['Sun']]`, and `[['Win']]`. Third, L(`CalleeOS`) is generated. Two rule alternatives for nonterminal `CalleeOS` are applied to derive `[['Mac']]` and `[['Win']]`. Finally, the three languages together with the coverage specification `([0,1,2],2)` are passed to a covering-array algorithm which returns six sentential forms as shown in the bottom of Figure 5.4 (b). As a result, the size of L(`call`) is reduced from twelve to six.

5.2 `Catalog` Grammar

As mentioned before, when *BooksTag* is set to {`rdepth 2`} and *ChaptersTag* to empty, the `Catalog` grammar generates 65,792 strings. Figure 5.5 shows the generation tree from root to depth 2. Rule `Books0` generates 256 strings while rule `Books1` generates 65,536 strings. To understand why rule `Books1` generates 65,536 strings, it is imperative to first examine rule `Books0`.

The following subsections are organized as follows. First, the grammar with *BooksTag* set to {`rdepth 1`} is explained. Setting `rdepth` to 1 eliminates the generation subtree starting from `Books1:65536`, which allows us to focus on rule `Books0`. Second, the grammar with *BooksTag* set to {`rdepth 2`} is explained by using the knowledge learned from {`rdepth 1`}. Finally, the grammar with *ChaptersTag* set to {`cov [ ([0,1,2],2) ]`} is presented to show the power of covering-array tags.

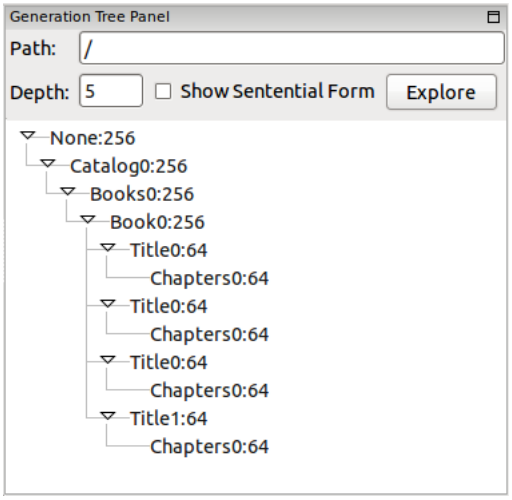


Figure 5.6: Generation tree for `Catalog` grammar with {`rdepth 1`}; tree is displayed from root to depth 5

5.2.1 *BooksTag* set to {`rdepth 1`}

When *BooksTag* is set to {`rdepth 1`} and *ChaptersTag* to empty, the grammar generates 256 strings. To understand why 256 strings are generated, let us examine the generation tree from root to depth 5 as shown in Figure 5.6. As shown in the generation tree, each `Title` rule generates 64 strings. This is so because there are 4 rule alternatives for nonterminal `Title` and therefore each rule alternative generates $256/4 = 64$ strings.

Immediately below the `Title` rules are the `Chapters0` rules. Each of the rules generates 64 strings as indicated by `Chapters0:64` in Figure 5.6. To understand why 64 strings are generated, let us examine the generation subtree starting from `Chapters0:64` as shown in Figure 5.7. Starting from `Chapters0:64`, rule `Chapter0` is applied three times. This is so because application of rule `Chapters0` yields three `Chapter` nonterminals. Each of the nonterminals is later replaced by rule `Chapter0`'s right-hand side. Also, the size of L(`Chapter`) is 4 because application of rule `Chapter0` yields a `Sections` nonterminal, application of rule `Sections0` yields a `Section` nonterminal, application of rule `Section0` yields a `Name` nonterminal, and a `Name` nonterminal has 4 rule alternatives. As a result, rule `Chapters0` generates $4 \times 4 \times 4 = 64$ strings.

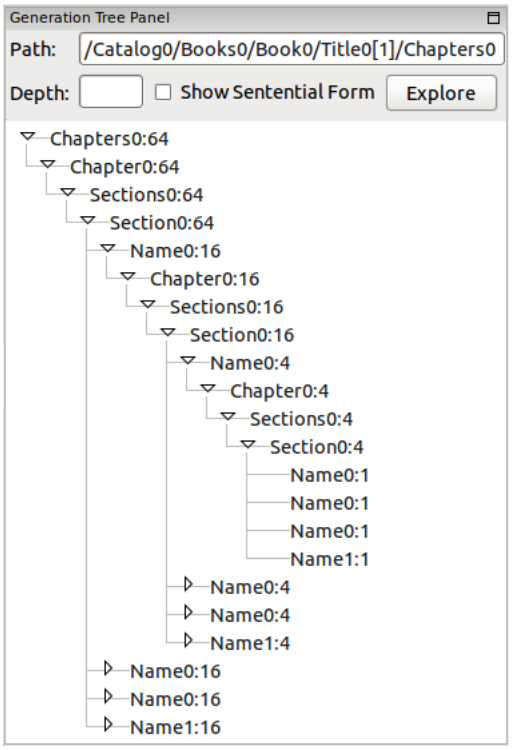


Figure 5.7: Generation tree for `Catalog` grammar with {`rdepth 1`}; tree is displayed from `Chapter0` to bottom

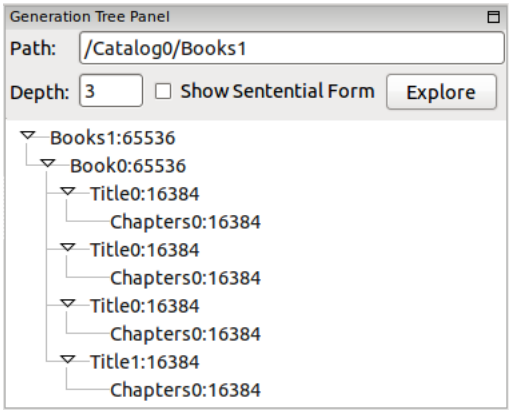


Figure 5.8: Generation tree for `Catalog` grammar with {`rdepth 2`}; tree is displayed from `Books1` to depth 3

5.2.2 *BooksTag* set to {`rdepth 2`}

When *BooksTag* is set to {`rdepth 2`} and *ChaptersTag* to empty, the grammar generates 65,792 strings, as indicated by `None:65792` in Figure 5.5. To understand why 65,792 strings are generated, let us examine the generation tree from root to depth 2 as shown in Figure 5.5. As shown in the generation tree, rule `Books0` generates 256 strings while rule `Books1` generates 65,536 strings. The two rules combined generate $256 + 65536 = 65792$ strings. What remains to be understood is why rule `Books1` generates 65,536 strings.

The sentential forms shown in Figure 5.5 help us answer this question. Application of rule `Books0` yields a `Book` nonterminal. Therefore, a catalog generated by rule `Books0` consists only one book. In contrast to rule `Books0`, rule `Books1` yields a `Book` nonterminal and a `Books` nonterminal. The `Books` nonterminal is later replaced by a `Book` nonterminal. Therefore, a catalog generated by rule `Books1` consists of two books. Since there are in total 256 book variations, rule `Books1` generates $256 \times 256 = 65,536$ strings.

The generation subtree starting from rule `Books1` has similar structure as that of rule `Books0`. Figure 5.8 shows the generation subtree from `Books1` to depth 3. Generation of the first book starts with application of rule `Books0`. Each `Title` rule generates $65,536/4 = 16,384$ strings as there are 4 rule alternatives for nonterminal `Title`. Immediately below the `Title` rules are the `Chapters0` rules. Figure 5.9 shows the generation subtree from `Chapters0` to depth 12. Since a book consists of three chapters and each chapter has 4 variations, there are in total $4 \times 4 \times 4 = 64$ chapter

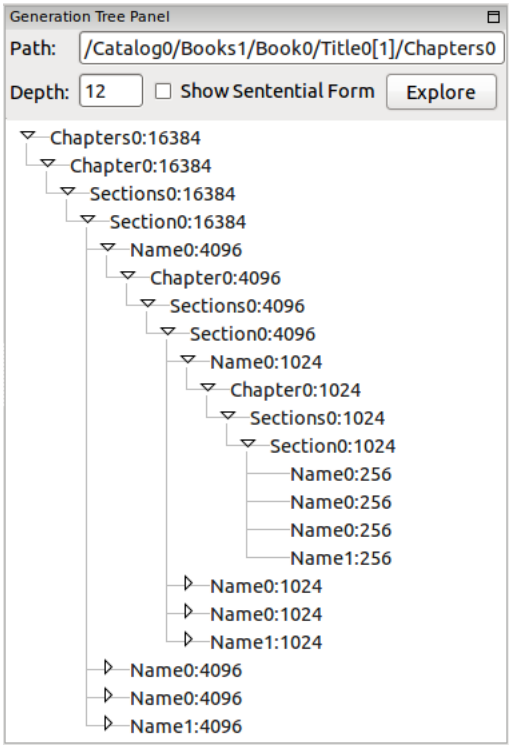


Figure 5.9: Generation tree for `Catalog` grammar with {`rdepth 2`}; tree is displayed from `Chapters0` to depth 12

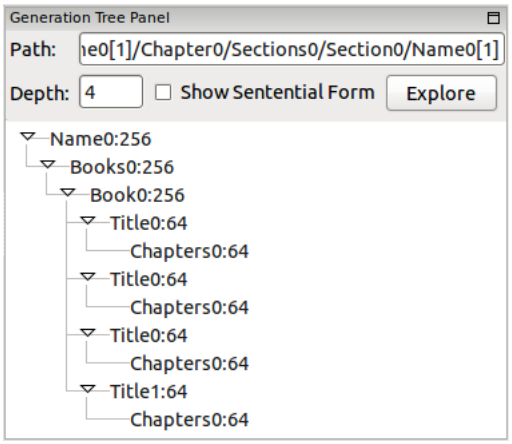


Figure 5.10: Generation tree for `Catalog` grammar with {`rdepth 2`}; tree is displayed from `Name0` to depth 4

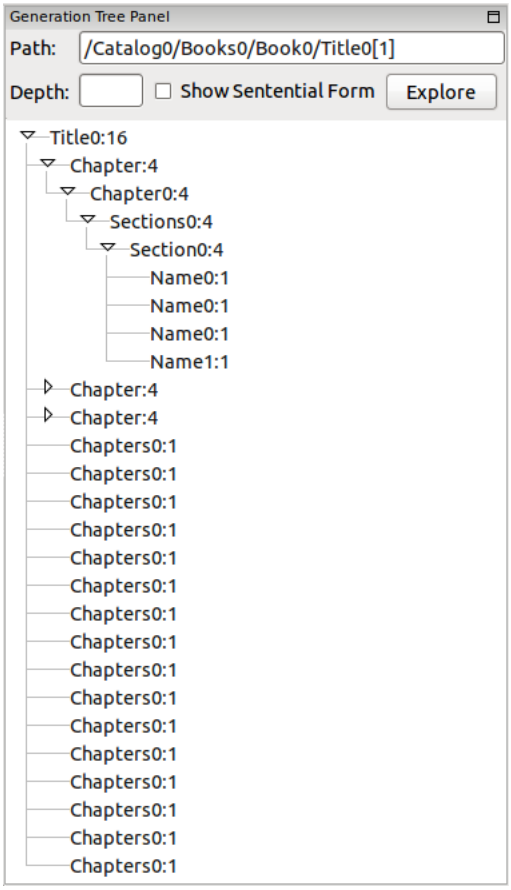


Figure 5.11: Generation tree for `Catalog` grammar with {`rdepth 1`} and {`cov [ ([0,1,2],2) ]`}; tree is displayed from `Title0[1]` to bottom

combinations. Generation of the first book ends with application of rules `Name0` or `Name1`. Figure 5.10 shows the generation subtree from `Name0` to depth 4. Generation of the second book starts with application of rule `Books0`.

5.2.3 *ChaptersTag* set to {`cov [ ([0,1,2],2) ]`}

When *BooksTag* is set to {`rdepth 1`} and *ChaptersTag* to {`cov [ ([0,1,2],2) ]`}, the grammar generates 64 strings. Figure 5.11 shows the generation subtree from `Title0[1]` to bottom. Since rule `Chapters0` is attached with a covering-array tag, a covering-array over three chapters is generated instead of the cartesian product. As a result, application of rule `Chapters0` yields 16 chapter combinations, as shown by 16 `Chapters0:1` nodes. Since there are 4 title alternatives and 16 chapter combinations, the total number of strings generated by the grammar is $4 \times 16 = 64$.

Chapter 6

Case Study: RSS Tests

Many testing scenarios involve generation of test inputs that are mostly correct except for a few errors in various places. With GBTG, it is easy to create a grammar that generates correct test inputs. Inserting errors in various places, however, is not as straightforward. One naive approach is to interleave correct and error rules. Figure 6.1 (a) shows a grammar that generates simple XML documents with errors inserted in open and close tags. The goal is to test an XML processor's ability to handle malformed XML documents such as mismatched open and close tag names, open tags that end with `/>`, and close tags that start with `<//`.

The grammar generates 81 documents in total. In order to understand why, let us consider the language size for each of the 6 nonterminals: `A`, `A_open`, `A_close`, `B`, `B_open`, and `B_close`. First, each of the four `List` terminal generators generates three terminals: `'x'`, `'/'`, and `''`. Since each rule that starts with `A_open`, `A_close`, `B_open`, or `B_close` has a `List` terminal generator on the rule's right-hand side, each nonterminal has language of size 3. Second, $L(\texttt{B})$ can be expressed as $L(\texttt{B\_open}) \times L(\texttt{B\_close})$ and therefore the size of $L(\texttt{B})$ is $3 \times 3 = 9$. Finally, $L(\texttt{A})$ can be expressed as $L(\texttt{A\_open}) \times L(\texttt{B}) \times L(\texttt{A\_close})$ and therefore the size of $L(\texttt{A})$ is $3 \times 9 \times 3 = 81$. Of the 81 documents generated, only 1 is correct because a correct document is generated only when all four `List` terminal generators generate `''`.

For the example above, the rules that generate valid open and close tags are referred to as correct rules whereas the rules that generate error values are referred to as error rules. As shown in Figure 6.1 (a), there is no distinction between correct and error rules, that is, they are interleaved. Interleaving correct and error rules causes two difficulties. First, the grammar is hard to read because the structure of correct test inputs is not immediately clear. Readability is important as testers need to

```
1  A ::= A_open B A_close;
2  B ::= B_open B_close;
3
4  A_open ::= '<a' List('x', '/', '') '>';
5  A_close ::= '</' List('x', '/', '') 'a>';
6
7  B_open ::= '<b' List('x', '/', '') '>';
8  B_close ::= '</' List('x', '/', '') 'b>';
```

(a) Interleaved correct and error rules

```
1  {cov [ ([0,1,2,3],2) ]}
2  A ::= A_open B_open B_close A_close;
3
4  A_open ::= '<a' List('x', '/', '') '>';
5  A_close ::= '</' List('x', '/', '') 'a>';
6
7  B_open ::= '<b' List('x', '/', '') '>';
8  B_close ::= '</' List('x', '/', '') 'b>';
```

(b) Apply `cov` tag

Figure 6.1: Examples of generating nearly correct test inputs

change the grammar whenever there are new test requirements. Second, reducing the number of XML documents is difficult. One possibility is to combine the rules in lines 1 and 2 into one rule and attach a `cov` tag to it. Figure 6.1 (b) shows the modified grammar. The rule in line 2 has four nonterminals on its right-hand side: `A_open`, `A_close`, `B_open`, and `B_close`. As mentioned before, each of these nonterminals has language of size 3. Therefore, applying a coverage tag with strength 2 reduces the number of XML documents to 9. Although this approach works, it requires a substantial change to the grammar, inconvenient when the grammar is large.

In order to solve these problems, it is necessary to introduce a new paradigm such that all grammar developers have an example to follow. The new paradigm that we will introduce in the next section is coined as template/probe paradigm.

6.1 Template/Probe Paradigm

The template/probe paradigm, as the name suggests, divides the rules into two categories: template and probe rules. Template rules generate templates: correct test inputs with place holders for error insertion. Probe rules generate probes, where each probe is a set of values for an error insertion point. A test input is generated by substituting the place holders in a template with the corresponding probes.

6.1.1 Usage

Figure 6.2 shows version of the grammar shown in Figure 6.1 (a) using template/probe approach. The template rules are shown in lines 11 through 13. There are in total four place holders, each of which is marked with `$`. Invoking the template rule in line 11 generates a template. The probe rules are shown in lines 15 through 19. There are in total four probes: `P0` is used to substitute `$0`, `P1` is used to substitute `$1`, etc. Invoking the probe rule in line 15 generates four probe values. For each template/probe values pair generated, the postcode is executed. The template is stored in `s[0]` while the probe values are stored in `s[1]`. In particular, probe values for `P0`, `P1`, `P2`, and `P3` are stored in `s[1][0]`, `s[1][1]`, `s[1][2]`, and `s[1][3]` respectively. Since both `s[0]` and `s[1]` contain bracketed expressions, the `flatten` function is invoked to convert them into strings. For example, line 2 flattens the template stored in `s[0]` and saves the result into `instance`. Also, the `replace` function is invoked to replace a place holder with the corresponding probe. For example, line 3 replaces place holder `$0` in

```
1   {postcode
2       instance = flatten(s[0])
3       instance = instance.replace('$0',flatten(s[1][0]))
4       instance = instance.replace('$1',flatten(s[1][1]))
5       instance = instance.replace('$2',flatten(s[1][2]))
6       instance = instance.replace('$3',flatten(s[1][3]))
7       print instance
8   }
9   S ::= D P;
10
11  D ::= A;
12  A ::= '<a' '$0' '>' B '</' '$3' 'a>';
13  B ::= '<b' '$1' '>' '</' '$2' 'b>';
14
15  P ::= P0 P1 P2 P3;
16  P0 ::= List('x', '/', '');
17  P1 ::= List('x', '/', '');
18  P2 ::= List('x', '/', '');
19  P3 ::= List('x', '/', '');
```

Figure 6.2: Template/probe approach

`instance` with the probe value stored in `s[1][0]`. After executing lines 2 through 6, `instance` contains a complete XML document decorated with error values. Line 7 prints `instance` to the standard output so that it can be used as a test input for the XML processor under test.

6.1.2 Advantages

As shown in Figure 6.2, adopting the template/probe paradigm has several advantages. First, the grammar is easier to understand because the structure of the correct test inputs are immediately clear. Good readability allows the testers to maintain the grammar with ease. Second, it is easy to reduce the number of XML documents. For example, attaching {`cov [ ([0,1,2,3],2) ]`} to line 15 reduces the number of XML documents from 81 to 9. Keeping the number of changes to a minimum also increases a grammar's maintainability.

It is understandable that the testing scenario provided above is too simple to demonstrate the real benefits of template/probe paradigm. Therefore, we provide a case study on testing an RSS parser. This requires generation of test inputs that

not only have a lot more complicated structures but also have a variety of different versions.

6.2 RSS Background

Many people access web content on a daily basis, one of the most popular being news headlines. To read the latest news, the traditional approach involves a two-step process. First, the Uniform Resource Locator (URL) of a news website is typed into a browser. Second, the news headline of interest is clicked to view its content. To read other news, one would go back to the news website and click on the next news headline of interest. Reading news online with the traditional approach is inconvenient because the reader is required to type in the URL of the news website. Also, it is not easy to find news that has not been read. Moreover, the web page needs to be refreshed in order to get the latest updates.

These problems can easily be solved by Really Simple Syndication (RSS). RSS is a family of XML-based web feed formats for syndicating web contents automatically. It is widely supported by web content publishers, such as news websites. To support RSS, a news website publishes an RSS feed which contains a list of links, one for each news headline. The reader subscribes to the RSS feed by using an aggregator, a program that runs on the reader's machine. An aggregator presents the news headlines from multiple news sources in a graphical user interface (GUI) which is refreshed whenever updates are available.

6.2.1 Structure of an RSS Feed

There are many RSS versions, the most popular being Atom 1.0, RSS 1.0 and 2.0. In the following, an example RSS feed is presented for each of the three RSS versions. For simplicity, the example RSS feeds contain only the required elements as described in the RSS specifications [1, 5, 6].

1. *Atom 1.0*: An Atom 1.0 feed consists of a `feed`. The children of a `feed` are `id`, `title`, `link`, `updated`, `rights`, and a variable number of `entry`s. Each `entry` has its own `id`, `title`, `link`, `updated`, and `author`. The `author` element in turn contains a `name` element. Figure 6.3 shows an example Atom 1.0 feed. Line 1 is the opening tag for `feed` with an attribute specifying the namespace for Atom 1.0. The values of feed `id`, feed `title`, feed `link`, feed `updated`, and

```
1   <feed xmlns="http://www.w3.org/2005/Atom">
2       <id>i</id>
3       <title>t</title>
4       <link href="http://l" />
5       <updated>u</updated>
6       <rights>r</rights>
7       <entry>
8           <id>i0</id>
9           <title>t0</title>
10          <link href="http://l/t0" />
11          <updated>u0</updated>
12          <author>
13              <name>n0</name>
14          </author>
15      </entry>
16  </feed>
```

Figure 6.3: An example Atom 1.0 feed

feed `rights` are `i`, `t`, `http://l`, `u`, and `r` respectively. The values of entry `id`, entry `title`, entry `link`, entry `updated`, and entry author `name` are `i0`, `t0`, `http://l/t0`, `u0`, and `n0` respectively. Note that the values of feed `link` and entry `link` are specified in element attribute while the rest are specified in element text.

2. *RSS 1.0*: An RSS 1.0 feed has a significantly different structure than an Atom 1.0 feed. An RSS 1.0 feed consists of a `rdf:RDF` which in turn consists of a `channel` and a variable number of `item`s. The children of a `channel` are `title`, `link`, `description`, `items`, and `dc:rights`. The children of an `item` are `title`, `link`, and `description`. Figure 6.4 shows an example RSS 1.0 feed. Line 1 is the opening tag for `rdf:RDF` with three attributes specifying the namespace for `rdf`, `dc`, and RSS 1.0. `rdf:RDF` has two children: `channel` and `item`. The values of channel `title`, channel `link`, channel `description`, and channel `dc:rights` are `t`, `http://l`, `d`, and `r` respectively. The values of item `title`, item `link`, and item `description` are `t0`, `http://l/t0`, and `d0` respectively. Note that the element attribute of `rdf:li`, as shown in line 10, contains the URL of the `item` element.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns="http://purl.org/rss/1.0/">
    <channel>
         <title>t</title>
         <link>http://l</link>
         <description>d</description>
         <items>
              <rdf:Seq>
                   <rdf:li resource="http://l/t0" />
              </rdf:Seq>
         </items>
         <dc:rights>r</dc:rights>
    </channel>
    <item>
         <title>t0</title>
         <link>http://l/t0</link>
         <description>d0</description>
    </item>
</rdf:RDF>
```

Figure 6.4: An example RSS 1.0 feed

```
<rss version="2.0">
    <channel>
         <title>t</title>
         <link>http://l</link>
         <description>d</description>
         <copyright>c</copyright>
         <item>
              <title>t0</title>
              <link>http://l/t0</link>
              <description>d0</description>
         </item>
    </channel>
</rss>
```

Figure 6.5: An example RSS 2.0 feed

3. *RSS 2.0*: An RSS 2.0 feed has a similar structure to that of an RSS 1.0 feed except for three differences. In an RSS 2.0 feed, the root element is `rss` instead of `rdf:RDF`, `channel` does not contain `items` and `dc:rights`, and `item` is a child of `channel` instead of a sibling. Figure 6.5 shows an example RSS 2.0 feed. Line 1 is the opening tag for `rss` with an attribute specifying the version of RSS; in this case, RSS 2.0. `rss` has one child `channel` which has one child `item`. The element texts for channel `title`, channel `link`, and channel `description` are `t`, `http://l`, and `d` respectively. The element texts for item `title`, item `link`, and item `description` are `t0`, `http://l/t0`, and `d0` respectively.

6.2.2 Security Risks

It is common to have HTML elements in the element text for presentation and navigation purposes. For example, replacing line 3 in Figure 6.5 with

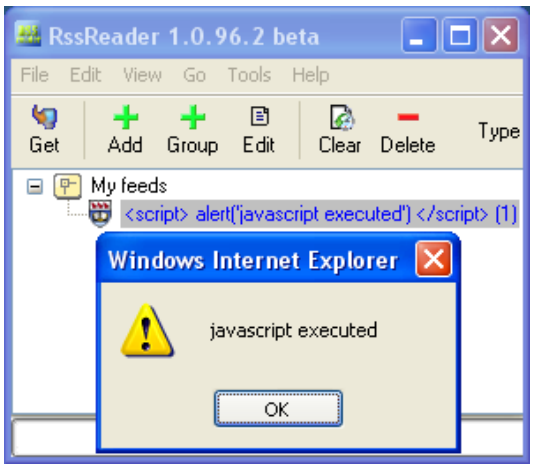
```
<title type="html">&lt;b&gt; t &lt;/b&gt;</title>
```

makes the aggregator display the channel title in bold. Note that the channel `title` has attribute `type="html"` which tells the aggregator that the element text of channel `title` contains HTML. Also, the `<` and `>` characters in the HTML tags are escaped with `<` and `>` respectively so that the HTML tags are not mistakenly interpreted as RSS tags.

Allowing HTML elements in RSS feeds can create security risks. As pointed out by Auger et al. [12], an aggregator can be used as a platform for launching cross-site scripting attacks. This is a form of attack that is carried out by injecting malicious code into web contents. Any client that accesses the dangerous web contents will then be compromised. In the case of RSS, the malicious code can be injected into HTML element text or attributes. For example, `script` is an HTML element that can contain javascript code in the element text. Figure 6.6 (a) shows a fragment of an RSS 2.0 feed that contains `script` in the channel `title`. The code in line 5, when executed, will pop up a dialog box displaying the message '`javascript executed`'. Figure 6.6 (b) shows the result of subscribing an aggregator named RSS Reader [7] to the feed. Note that the code was executed because the dialog box was displayed. If the code were malicious, the machine that runs the aggegator—hereafter referred to as the client side—would be compromised. Because of the potential damage that the `javascript` code can cause to the client side, the `script` HTML element is considered unsafe.

```
1    <rss version="2.0">
2         <channel>
3              <title>
4                   &lt;script&gt;
5                   alert('javascript executed')
6                   &lt;/script&gt;
7              </title>
               ...
13        </channel>
14   </rss>
```

(a) A fragment of an RSS feed



(b) Sanitization error in RSS Reader

Figure 6.6: An RSS feed and sanitization error

Malicious code can also be injected into HTML attributes. Figure 6.7 shows an RSS attack provided by Auger et al. [12]. Within the feed fragment is an HTML element named `img`, as shown in lines 5 and 6. The `img` element has a `src` attribute which in turn contains a link to a shopping site at `store.example.com`. When interpreted, the `src` attribute will trick the aggregator to make a purchase request to the shopping site. If the client were logged into the shopping site when the purchase request was made, the request would be considered valid and therefore processed, even though the user never made such a request.

To protect the client side from unsafe HTML elements and attributes, it is the responsibility of an aggregator to *sanitize* an RSS feed before interpretting the embedded HTML. Sanitization is the process of filtering out the unsafe HTML elements

```
1    <item rdf:about="http://host/about.foo">
2         <title>My Story Title</title>
3         <link>http://host/story.php</link>
4         <description>
5              &lt;img src="https://store.example.com/
6              buy?item=stamps&quantity=100"&gt;
7         </description>
8    </item>
```

Figure 6.7: An RSS attack example provided by Auger et al. [12]

and attributes. Failure to filter out the unsafe HTML elements and attributes, and only those elements and attributes, is considered a sanitization error.

6.2.3 Problems of Testing for Sanitization Errors

Testing an aggregator for sanitization errors is challenging for several reasons:

1. *Many RSS versions*: There are about 10 RSS versions in existence today and more are likely to be introduced in the future. Some of them have significantly different structures while others have very similar structures. For example, Atom 1.0 and RSS 1.0 have significantly different structures as shown in Figures 6.3 and 6.4. RSS 1.0 and 2.0 have very similar structures as shown in Figures 6.4 and 6.5.

2. *Many HTML injection points*: Any RSS element text is a possible place for injecting HTML. Some RSS elements appear only in one version. For example, `rights` is unique to Atom 1.0, `dc:rights` is unique to RSS 1.0, and `copyright` is unique to RSS 2.0. Some RSS elements are common to many RSS versions. For example, Atom 1.0, RSS 1.0 and 2.0 all have `title`.

3. *Many safe and unsafe HTML elements and attributes*: There are currently 91 elements and 564 attributes as defined in the HTML specification [2].

As a result, a proper test involves generation of test inputs that contain a combination of RSS versions, HTML injection points, and safe and unsafe HTML elements and attributes. It is obvious that an ad-hoc test approach is ineffective in this case. The following sections explain how GBTG can be used to systematically test an aggregator.

```
1    template ::= 'atom1.0' atom1;
2    template ::= 'rss1.0' rss1;
3    template ::= 'rss2.0' rss2;
```

Figure 6.8: Beginning of the template grammar

6.3 Test Approach

The feedparser [9] is the code under test. It takes an RSS feed as input and generates a parse tree. To avoid HTML injection vulnerabilities, the parse tree is sanitized by using the whitelist approach, that is, only those HTML elements and attributes that are considered safe by the feedparser documentation remain while the rest are stripped. There are many RSS versions supported by the feedparser. In this case study, we chose Atom 1.0 [1], RSS 1.0 [5] and RSS 2.0 [6] for testing. A grammar, developed by following the template/probe paradigm, is used to generate the test inputs: RSS feeds that contain safe and unsafe HTML elements and attributes.

6.3.1 Template Grammar

The template portion of the grammar starts with the rules shown in Figure 6.8. Each rule has two terms on its right-hand side. The first term is a terminal specifying the version of the RSS feed being generated. The version information is essential for output checking as different RSS versions have different HTML injection points. The second term is a nonterminal that serves as the start symbol for an RSS grammar. In particular, `atom1` is the start symbol for the Atom 1.0 grammar, `rss1` is the start symbol for the RSS 1.0 grammar, and `rss2` is the start symbol for the RSS 2.0 grammar. All three grammars are part of the template grammar.

The grammars for generating Atom 1.0, RSS 1.0 and 2.0 feed templates are shown in Figures 6.9, 6.10, and 6.11 respectively. Each of the three grammars generates an RSS feed template. Because the cause of sanitization errors is likely to be independent of the number of items, a feed template has only one item. Also, each feed template is decorated with two place holders: Atom 1.0 has `$rights_probe` and `$title_probe`, RSS 1.0 has `$dc_rights_probe` and `$title_probe`, and RSS 2.0 has `$copyright_probe` and `$title_probe`. Note that all RSS elements that contain a place holder have the `>` character omitted in their opening tags. This is so because we leave it to the probes portion of the grammar to decide whether to insert

```
1   atom1 ::= '<feed xmlns="http://www.w3.org/2005/Atom">'
2    atom1_channel_id channel_title atom1_channel_link
3    atom1_channel_updated atom1_rights atom1_entry '</feed>';
4
5   atom1_channel_id ::= '<id>' 'channel_id' '</id>';
6
7   atom1_channel_link ::= '<link href="' 'channel_link' '"/>';
8
9   atom1_channel_updated ::= '<updated>' 'channel_updated' '</updated>';
10
11  atom1_rights ::= '<rights' '$rights_probe' '</rights>';
12
13  atom1_entry ::= '<entry>' atom1_item_id item_title atom1_item_link
14   atom1_item_updated atom1_item_author '</entry>';
15
16  atom1_item_id ::= '<id>' 'item_id' '</id>';
17
18  atom1_item_link ::= '<link href="' 'item_link' '"/>';
19
20  atom1_item_updated ::= '<updated>' 'item_updated' '</updated>';
21
22  atom1_item_author ::= '<author>' atom1_item_author_name '</author>';
23
24  atom1_item_author_name ::= '<name>' 'item_author_name' '</name>';
25
26  channel_title ::= '<title' '$title_probe' '</title>';
27
28  item_title ::= '<title>' 'item_title' '</title>';
```

Figure 6.9: Atom 1.0 grammar

```
1   rss1 ::= '<rdf:RDF'
2    ' xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"'
3    ' xmlns:dc="http://purl.org/dc/elements/1.1/"'
4    ' xmlns="http://purl.org/rss/1.0/">'
5    rss1_channel rss1_item '</rdf:RDF>';
6
7   rss1_channel ::= '<channel>' channel_title rss_channel_link
8    rss_channel_description rss1_channel_items rss1_dc_rights
9    '</channel>';
10
11  rss1_channel_items ::= '<items>' rss1_channel_rdf_seq '</items>';
12
13  rss1_channel_rdf_seq ::= '<rdf:Seq>' rss1_channel_rdf_li '</rdf:Seq>';
14
15  rss1_channel_rdf_li ::= '<rdf:li resource="item_link" />';
16
17  rss1_dc_rights ::= '<dc:rights' '$dc_rights_probe' '</dc:rights>';
18
19  rss1_item ::= '<item>' item_title rss_item_link rss_item_description
20   '</item>';
21
22  rss_channel_link ::= '<link>' 'channel_link' '</link>';
23
24  rss_channel_description ::= '<description>' 'channel_description'
25   '</description>';
26
27  rss_item_link ::= '<link>' 'item_link' '</link>';
28
29  rss_item_description ::= '<description>' 'item_description'
30   '</description>';
31
32  channel_title ::= '<title' '$title_probe' '</title>';
33
34  item_title ::= '<title>' 'item_title' '</title>';
```

Figure 6.10: RSS 1.0 grammar

```
1   rss2 ::= '<rss version="2.0">' rss2_channel '</rss>';
2
3   rss2_channel ::= '<channel>' channel_title rss_channel_link
4    rss_channel_description rss2_copyright rss2_item '</channel>';
5
6   rss2_copyright ::= '<copyright' '$copyright_probe' '</copyright>';
7
8   rss2_item ::= '<item>' item_title rss_item_link rss_item_description
9    '</item>';
10
11  rss_channel_link ::= '<link>' 'channel_link' '</link>';
12
13  rss_channel_description ::= '<description>' 'channel_description'
14   '</description>';
15
16  rss_item_link ::= '<link>' 'item_link' '</link>';
17
18  rss_item_description ::= '<description>' 'item_description'
19   '</description>';
20
21  channel_title ::= '<title' '$title_probe' '</title>';
22
23  item_title ::= '<title>' 'item_title' '</title>';
```

Figure 6.11: RSS 2.0 grammar

`type="html"` in RSS open tags. Since there are three feed templates generated, the number of place holders is therefore $2 \times 3 = 6$. Of the six place holders, only four are unique. This is so because for the two place holders in each template, one is unique to an RSS version and the other one is shared by all three RSS versions. The ones that are unique are `$rights_probe`, `$dc_rights_probe`, and `$copyright_probe` which appear in the `rights` element of Atom 1.0, the `dc:rights` element of RSS 1.0, and the `copyright` element of RSS 2.0 respectively. The one that is shared is `$title_probe` which appears in the `title` element.

As mentioned before, the three RSS versions have some common features. For example, all three RSS versions have channel `title` and item `title`. Many elements that appear in an RSS 1.0 feed also appear in an RSS 2.0 feed. Figures 6.9, 6.10, and 6.11 show that the rules that generate the common features are shared. For example, the rules that start with `channel_title` and `item_title` are used by all three RSS grammars. The rules that start with `rss_` are used by RSS 1.0 and 2.0 grammars. In contrast, the features that are unique to one RSS version are generated by the rules that are used by only one of the grammars. The rules that start with `atom1_` are used only by the Atom 1.0 grammar, the rules that start with `rss1_` are used only by the RSS 1.0 grammar, and the rules that start with `rss2_` are used only by the RSS 2.0 grammar.

6.3.2 Probes Grammar

The probes portion of the grammar generates one probe per place holder. As shown in Figure 6.12, there are four unique place holders, four probes are generated: `title_probe`, `rights_probe`, `dc_rights_probe`, and `copyright_probe`. The italicized texts that appear to the left of each probe are place holders for covering-array tags. These texts are for illustration purposes only and are not part of the grammar. A probe value, a string in $L(P)$ where P is any one of the four probes, consists of a type and two HTML elements. The type is the `type` attribute that appears in the open tag of an RSS element. It is used to specify the type of element text. As shown in Figure 6.12, the type is either `html` or omitted. The two HTML elements are used to test the combination of safe and unsafe elements. A total of six different HTML element tags are tested: one safe and five unsafe. According to the feedparser documentation, the `a` element is safe while the `script`, `applet`, `embed`, `object`, and `meta` elements are unsafe. The attribute for each HTML element tag is `abbr`, `style`, or omitted. Ac-

```
probes ::= title_probe rights_probe dc_rights_probe
 copyright_probe;
```

TtileProbeTag

```
title_probe ::= type html html;
```

RightsProbeTag

```
rights_probe ::= type html html;
```

DcRightsProbeTag

```
dc_rights_probe ::= type html html;
```

CopyrightProbeTag

```
copyright_probe ::= type html html;

type ::= List('>','type="html">');

lt ::= '&lt;';

gt ::= '&gt;';

html ::= lt 'a' attr gt lt '/a' gt;
html ::= lt 'script' attr gt lt '/script' gt;
html ::= lt 'applet' attr gt lt '/applet' gt;
html ::= lt 'embed' attr gt lt '/embed' gt;
html ::= lt 'object' attr gt lt '/object' gt;
html ::= lt 'meta' attr gt lt '/meta' gt;

attr ::= 'abbr' value_empty;
attr ::= 'style' value_empty;
attr ::= '';

value_empty ::= '=' '""';
```

Figure 6.12: Probes grammar

```
1 type="html">
2 &lt; a abbr="" &gt; &lt; /a &gt;
3 &lt; script style="" &gt; &lt; /script &gt;
```

Figure 6.13: An example probe value

cording to the feedparser documentation, the `abbr` attribute is safe while the `style` attribute is unsafe. Figure 6.13 shows an example probe value. The type is `html` and the two HTML elements are `a` and `script`. The `a` element has `abbr` attribute and the `script` element has `style` attribute.

To summarize, a probe value consists of a type and two HTML elements. A type has two variations: `html` or omitted. An HTML element has $6 \times 3 = 18$ variations as there are six different element tags under test and each element tag has three different attributes. As a result, the number of possible values for a probe is $2 \times 18 \times 18 = 648$. Since there are in total four different probes, the size of $L(\texttt{probes})$ is therefore $648^4 \approx 176$ billion, which is too big to test.

6.3.3 Reducing Probe Values

Our approach of reducing the number of probe/value combinations is based on one assumption: a sanitization error that occurs in one probe location is independent of a sanitization error that occurs in another probe location. This assumption is reasonable as it is likely that the same sanitization procedure is invoked in all four probe locations. Testing the combination of probe values in different probe locations is therefore not necessary. Based on this assumption, the test is set up as follows. The grammar is run four times, each focusing on testing a probe location. For each run, the target probe location is tested against all possible probe values while the other three probe locations are tested against only two probe values. Of the two probe values, one has the `type` attribute equal to `html` and the other has the `type` attribute omitted.

As a result, the tester is required to change the probes grammar as shown in Figure 6.12. For example, testing the probe location marked by `$title_probe` involves removing *TitleProbeTag* and substituting *RightsProbeTag*, *DcRightsProbeTag*, and *CopyrightProbeTag* with {`cov [([0],1)]`}. As another example, testing the probe location marked by `$rights_probe` involves removing *RightsProbeTag* and substituting *TitleProbeTag*, *DcRightsProbeTag*, and *CopyrightProbeTag* with {`cov [([0],1)]`}. Testing the other two probe locations, as marked by `$dc_rights_probe` and `$copyright_probe`, proceeds in a similar fashion. With this approach, the number of probe values for one run is 5,184.

In order to understand how we arrived at this number, let us consider the start symbol of the probes grammar: the `probes` nonterminal. Figure 6.12 shows that

```
1    def tag_stripped(rss_text,html_tag):
2         return rss_text.find('</' + html_tag + '>') == -1
3
4    def attr_stripped(rss_text,html_attr):
5         if re.search(html_attr+'\s*=\s*""',rss_text):
6              return False
7         return True
```

Figure 6.14: Functions for checking the parse trees produced by the feedparser

the rule that starts with `probes` has four probe nonterminals on its right-hand side. $L($`probes`$)$ is therefore the cartesian product of all four probes. For each run, the probe under focus does not have a `cov` tag attached and therefore generates 648 probe values. The other three probes have {`cov [([0],1)]`} attached and therefore generate 2 probe values each. The number of probe value combinations is therefore $648 \times 2 \times 2 \times 2 = 5,184$. Since there are in total four runs, the total number of probe value combinations is therefore $5,184 \times 4 = 20,736$. Running the test on our test PC—a Dell Precision 390 with an 2.13 GHz Intel Core2 Duo and 2 gigabytes of RAM—requires approximately 2 minutes and 12 seconds.

6.3.4 Output Checking

Executing a test case is a four-step process. First, a feed is generated by substituting the place holders in a template with the corresponding probe values. Second, the feedparser is invoked, taking the feed as input and returning a parse tree. Third, the parse tree is checked for the existence of injected HTML elements and attributes. Finally, the result of the checking process is recorded in a log file formatted in XML.

A variable named `P` is created for storing the parse tree produced by the feedparser. The parse tree is a nested dictionary where the keys are RSS element tags and the values are RSS element texts. For example, the children of `feed` are stored in `P['feed']`. Because the feedparser does not distinguish between `feed` and `channel`, the children of `channel` are also stored `P['feed']`. In the case of an RSS 2.0 feed, the channel `title` element text is stored in `P['feed']['title']` whereas the channel `rights` element text is stored in `P['feed']['rights']`. Also, the children of `entry` and `item` are stored in `P['entries']`. Since an RSS feed can have multiple `entry`s or `item`s, `P['entries']` is a list containing the information for the `entry`s or `item`s. In the case of an RSS 2.0 feed with only one `item`, the item `link` element text is

```
1  <test>
2     <feed id="1" version="atom1.0">
3        <probe name="$title_probe" type="">
4           <html_tag name="script" safe="False"
5           stripped="False"/>
6           <html_tag name="script" safe="False"
7           stripped="False">
8              <html_attr name="style" safe="False"
9              stripped="False"/>
10          </html_tag>
11       </probe>
12       <probe name="$rights_probe" type="html">
13          <html_tag name="script" safe="False"
14          stripped="False"/>
15          <html_tag name="script" safe="False"
16          stripped="False"/>
17       </probe>
18    </feed>
19 </test>
```

Figure 6.15: An example log file

stored in `P['entries'][0]['link']`. Finally, the HTML elements within the parse tree have `<` and `>` in their open and close tags, even though these characters were represented by `<` and `>` in the template grammar.

Checking whether or not an HTML element or attribute is stripped is accomplished by the two functions shown in Figure 6.14. The first function, `tag_stripped`, checks whether an HTML element is stripped. This is accomplished by checking the existence of the HTML element's close tag in the RSS element text. The function returns true if the HTML element being checked is stripped and false otherwise. The second function, `attr_stripped`, checks whether an HTML attribute is stripped. This is accomplished by checking the existence of the HTML attribute assignment in the RSS element text. The function returns true if the HTML attribute being checked is stripped and false otherwise.

6.3.5 Output Logging

For each probe location, the log file records (1) the name of the probe, (2) the `type` of the RSS element, (3) the safeness of the injected HTML elements and attributes,

and (4) the existence of the injected HTML elements and attributes in the parse tree: whether or not they are stripped.

Figure 6.15 shows an example log file that contains the test result of one feed. The root element of the log is named `test`. It has only one child named `feed` because the log contains only the test result of one feed. In our experiment, there are many feeds under test and therefore our log contains a lot more `feed`s. An `feed` element has two attributes: `id` is an unique identifier for a feed and `version` is the RSS version under test. Line 2 in Figure 6.15 shows that the feed `id` is `1` and the feed `version` is `atom1.0`. The children of the `feed` element are two `probe` elements. The `probe` element also has two attributes: `name` is the name of the probe and `type` is the type of the RSS element text. If `type` is empty, then the RSS element that contains the probe does not have the `type` attribute. However, if `type` is `html`, then the RSS element that contains the probe has a `type` attribute equal to `html`. The first probe, shown in lines 2 through 11, has `name` equal to `$title_probe` and `type` empty. The second probe, as shown in lines 12 through 17, has `name` equal to `$rights_probe` and `type` equal to `html`.

Within the first probe are two HTML elements. The first HTML element is named `script`. It is unsafe but not stripped. The second HTML element is named `script`. It is also unsafe but not stripped. In addition, the second HTML element has attribute named `style`. It is unsafe but not stripped. The content of the second probe is identical to the first except that the second HTML element does not have an attribute.

6.4 Test Results

Examining the log file manually is a time consuming process as the log file is large: slightly over 1 megabyte. Therefore, an XQuery script was created to analyze the log file. Figure 6.16 shows the XQuery script which answers two questions. The first question, implemented in lines 2 through 15, asks whether there exists an HTML element which is either safe but stripped or unsafe but not stripped, if the `type` is `html`. The second question, implemented in lines 18 through 32, asks whether there exists an HTML attribute which is safe but stripped or unsafe but not stripped, if the `type` is `html`. The result of executing the script shows that:

1. An HTML element or attribute is stripped when it is unsafe and not stripped

```
1  <query_result> {
2     for $html_tag in /test/feed/probe/html_tag
3     let
4        $probe := $html_tag/..,
5        $feed := $probe/..
6     where
7        $probe/@type = 'html' and
8        (($html_tag/@safe = 'True' and
9        $html_tag/@stripped = 'True') or
10       ($html_tag/@safe = 'False' and
11       $html_tag/@stripped = 'False'))
12    return
13       <feed id="{$feed/@id}"
14       probe_name="{$probe/@name}"
15       html_tag_name="{$html_tag/@name}"/>,
16
17    for $html_attr in /test/feed/probe/html_tag/html_attr
18    let
19       $html_tag := $html_attr/..,
20       $probe := $html_tag/..,
21       $feed := $probe/..
22    where
23       $html_tag/@stripped = 'False' and
24       $probe/@type = 'html' and
25       (($html_attr/@safe = 'True' and
26       $html_attr/@stripped = 'True') or
27       ($html_attr/@safe = 'False' and
28       $html_attr/@stripped = 'False'))
29    return
30       <feed id="{$feed/@id}"
31       probe_name="{$probe/@name}"
32       html_attr_name="{$html_attr/@name}"/>
33 } </query_result>
```

Figure 6.16: XQuery script

```
26  channel_title ::= '<title>' 'channel_title' '</title>';
```

(a) Atom 1.0 grammar changes

```
17  rss1_dc_rights ::= '<dc:rights>' 'channel_rights' '</dc:rights>';

27  rss_item_link ::= '<link' '$rss1_link_probe' '</link>';

32  channel_title ::= '<title>' 'channel_title' '</title>';
```

(b) RSS 1.0 grammar changes

```
6   rss2_copyright ::= '<copyright>' 'channel_copyright' '</copyright>';

16  rss_item_link ::= '<link' '$rss2_link_probe' '</link>';

21  channel_title ::= '<title>' 'channel_title' '</title>';
```

(c) RSS 2.0 grammar changes

Figure 6.17: Template grammar changes

otherwise.

2. When an HTML element is unsafe, it is stripped regardless of whether or not its attributes are safe.

3. When an HTML element is safe, only the unsafe HTML attributes are stripped.

It is interesting to see whether the omission of the `type` attribute has any effect on the sanitization. To answer this question, the script was executed again with lines 7 and 24 removed. The result shows that when the `type` attribute is omitted, no HTML element or attribute—safe or unsafe—is stripped.

Another interesting aspect of the feedparser is that it does not sanitize all RSS elements, according to the feedparser documentation. An example RSS element that is not sanitized by the feedparser is item `link`. As shown in Figures 6.3, 6.4, and 6.5, the item `link` element appears in all three RSS versions under test.

To test whether the item `link` element is sanitized, a new grammar was created that is nearly identical to the old one. Figure 6.17 shows the changes in the template portion of the grammar. The left-hand side shows the line numbers in which the

```
                    probes ::= rss1_link_probe rss2_link_probe;
Rss1LinkProbeTag    rss1_link_probe ::= type html html;
Rss2LinkProbeTag    rss2_link_probe ::= type html html;
                    ...
```

Figure 6.18: Probes grammar changes

changes take place and the right-hand side shows the new grammar rules as a result of these changes. The `$title_probe` place holder is removed from all three grammars because the channel `title` element is not the test objective. For the RSS 1.0 grammar, the `$dc_rights_probe` place holder in the `dc:rights` element is removed and the `$rss1_link_probe` place holder is added to the item `link` element. For the RSS 2.0 grammar, the `$copyright_probe` place holder in the `copyright` element is removed and the `$rss2_link_probe` place holder is added to the item `link` element. The item `link` element for Atom 1.0 was not tested because the link is embedded in an attribute rather than an element text, as shown in Figure 6.3. Since the grammar was designed to test sanitization errors in the element texts, testing the item `link` element for Atom 1.0 requires substantial changes to the probes portion of the grammar and therefore the test was skipped. Figure 6.18 shows the changes in the probes portion of the grammar. The changes are threefold. First, the `title_probe` and `rights_probe` nonterminals are removed. Second, the `dc_rights_probe` nonterminal is replaced with the `rss1_link_probe` nonterminal. Third, the `copyright_probe` nonterminal is replaced with the `rss2_link_probe` nonterminal.

As with the old grammar, running the new grammar generates a log file. Examining the log file with the script shown in Figure 6.16 shows that no HTML element or attribute—safe or unsafe—is stripped, if the `type` is `html`. Running the same script again with lines 7 and 24 removed shows that no HTML element or attribute—safe or unsafe—is stripped, if the `type` is omitted. In other words, the feedparser does not sanitize the item `link` element, regardless whether the `type` attribute is specified as `html` or omitted.

6.5 Discussion

The feedparser has HTML injection vulnerabilities for two reasons. First, the feedparser sanitizes only certain RSS elements according to the feedparser documentation. Second, the feedparser strips unsafe HTML elements and attributes only when the `type` attribute is specified as `html`. Because of these, a malicious web content publisher can launch an attack by (1) inserting dangerous HTML into those RSS elements that will not be sanitized by the feedparser or (2) omitting the `type` attribute. However, the feedparser behaves correctly according to its specification.

This case study demonstrates that a grammar, developed by using the template/probe approach, can easily be changed to satisfy new test requirements because:

1. The template portion of the grammar is modular, that is, the rules that generate the common features of the RSS versions are shared. For example, the rules that start with `channel_title` and `item_title` are used by all three RSS grammars. The rules that start with `rss_` are used by RSS 1.0 and 2.0 grammars. If the test were to expand to include a new RSS version, some of the existing rules can be used towards generating the template for the new version.

2. Probes can be placed at arbitrary locations. As shown in the example, creating a new grammar to test item `link` instead of channel `rights` and `copyright` requires only three minor changes: (1) the place holders are moved from channel `rights` and `copyright` into item `link`, (2) the place holders are renamed to reflect the changes in their locations, and (3) the probes are also renamed such that they have the same names as their corresponding place holders. Extending the grammar to test additional HTML injection points can be carried out in the similar fashion.

3. Probe values can be added easily. As mentioned before, the grammar generates combinations of `type` attribute, HTML elements, and HTML attributes. The following shows generating additional combinations requires only minor changes to the probes portion of the grammar. Figure 6.19 (a) shows a portion of the probes grammar in Figure 6.12 decorated with line numbers. Adding the rule in Figure 6.19 (b) to line 12 adds a new `type` attribute of value `text`. Adding the rule in Figure 6.19 (c) to line 23 adds a new HTML element named `input`. Adding the rule in Figure 6.19 (d) to line 27 adds a new HTML attribute named `onclick`.

```
    ...
12  type ::= List('>','type="html">');
    ...
18  html ::= lt 'a' attr gt lt '/a' gt;
19  html ::= lt 'script' attr gt lt '/script' gt;
20  html ::= lt 'applet' attr gt lt '/applet' gt;
21  html ::= lt 'embed' attr gt lt '/embed' gt;
22  html ::= lt 'object' attr gt lt '/object' gt;
23  html ::= lt 'meta' attr gt lt '/meta' gt;
24
25  attr ::= 'abbr' value_empty;
26  attr ::= 'style' value_empty;
27  attr ::= '';
    ...
```

(a) A portion of the probes grammar in Figure 6.12

```
type ::= 'type="text">';
```

(b) A new `type` attribute of value `text`

```
html ::= lt 'input' attr gt lt '/input' gt;
```

(c) A new HTML element named `input`

```
attr ::= 'onclick' value_empty;
```

(d) A new HTML attribute named `onclick`

Figure 6.19: Probes grammar

Chapter 7

Case Study: CA Tests

The Code Activator [23], CA for short, is a software product line for creating web applications that facilitate quiz authoring and quiz taking. Authoring a quiz is a two-step process. First, a quiz author creates question templates from which questions can be generated. Second, the quiz author creates a quiz by specifying the set of questions to be included. The quiz specification is hosted on a web server. Taking a quiz is also a two-step process. First, a student points his or her web browser to the quiz. Second, the student answers the questions by filling in the hot spots. A hot spot is a text field that expects an integer, a floating-point number, or a string. The answers are logged into a file so that a mark can be generated when the quiz finishes.

To ensure CA behaves as expected, a set of test cases has been created. The test cases were developed in Selenium [26], a web application test tool that allows testers to emulate user inputs and to extract HTML fragments from web pages. The test cases covered only a small set of CA features. In this chapter, we are going to show (1) how GBTG can be used to cover the existing test cases with a few lines of code and (2) how GBTG can be used to extend the test coverage by testing a larger set of CA features and their combinations. Moreover, we are going to show how covering arrays can be used to counter the combinatorial explosion resulting from the greater test coverage.

7.1 CA Quiz Taking

Taking a quiz involves pointing a web browser to a quiz specification. Figure 7.1 shows an example quiz located at

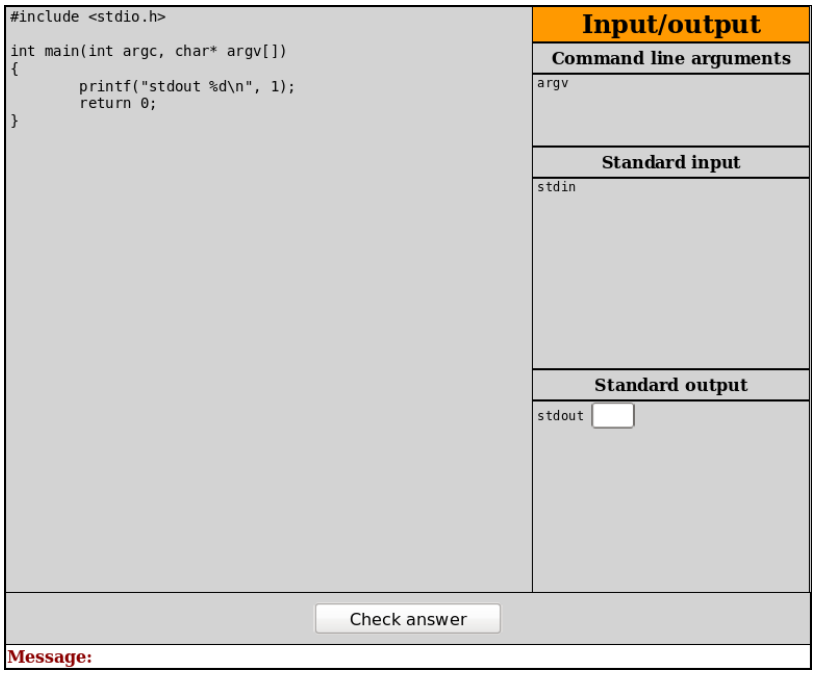


Figure 7.1: An example CA question of type *input-output*

`http://localhost:8080/cqg/quiz?spec=test_c_io`
where `test_c_io` is the name of the quiz specification. The quiz is configured such that (1) a student can check whether his or her answer is correct by clicking the `Check answer` button and (2) the quiz consists of only one question and therefore there are no navigation buttons such as `Previous` and `Next`.

7.1.1 Answering Questions

Figure 7.1 shows the basic features of a question including:

- *Code cell*: located on the left. It displays the code in question.
- *Question type*: displayed on the top right corner. It specifies the game rule of the question.
- *Command line arguments cell*: referred to as the argv cell. It displays the command line arguments passed to the code in question.
- *Standard input cell*: referred to as the stdin cell. It displays the standard input of the code in question.

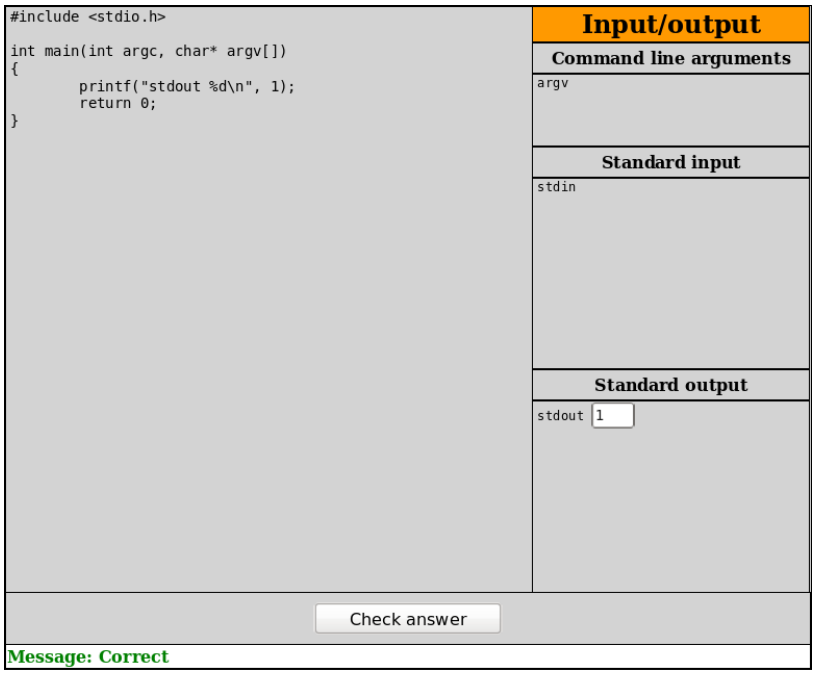


Figure 7.2: Solving the question shown in Figure 7.1

- *Standard output cell*: referred to as the stdout cell. It displays the standard output of the code in question.

Each of these four cells can contain one or more hot spots. In this example, the code cell contains a seven-line C program, the question type is *input-output*, the argv cell contains the string `argv`, the stdin cell contains the string `stdin`, and the stdout cell contains the string `stdout` followed by a hot spot.

The game rule of *input-output* is that filling in the hot spots such that the actual output produced by executing the code is identical to the content of the stdout cell. For example, executing the code in Figure 7.1 results in the string `stdout 1` being printed to the standard output. Therefore, the correct answer to fill in the hot spot is `1`. Figure 7.2 shows the result of filling the hot spot with `1` and clicking the `Check answer` button. Note that `Message: Correct` is displayed at the bottom to affirm the correctness of the answer. Also note that the contents of the argv cell and the stdin cell are not relevant in terms of answering this question. This is so because the code neither reads from the command line arguments nor the standard input.

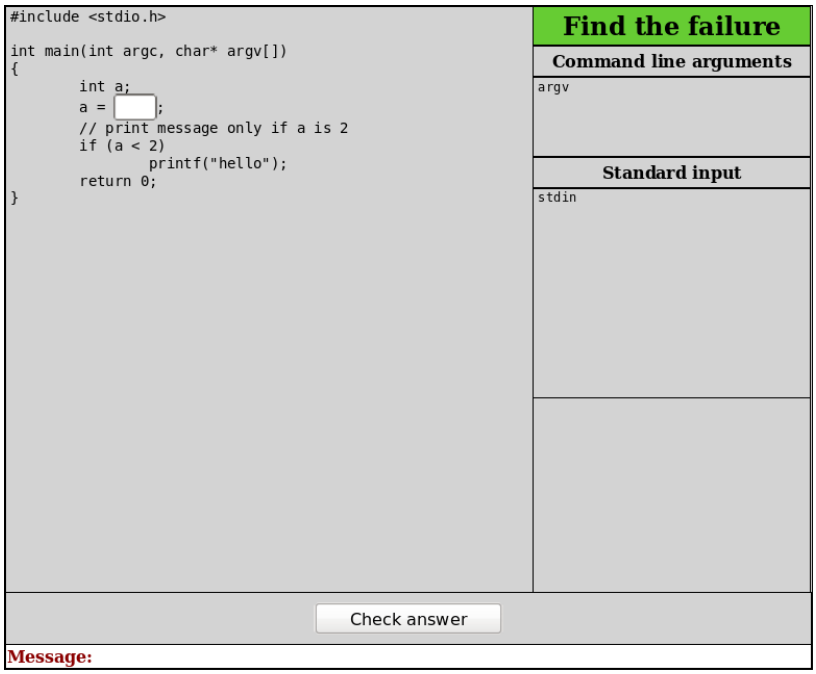


Figure 7.3: An example CA question of type *find-the-failure*

7.1.2 Question Types

In addition to *input-output*, CA supports two more question types: *find-the-failure* and *bullseye*. The game rule of *find-the-failure* is that given a specification and an implementation, the student must fill in the hot spots such that executing the code would produce an output that violates the specification. The game rule of *bullseye* is that the student must fill in the hot spots such that the highlighted code is executed.

Figure 7.3 shows an example *find-the-failure* question. The specification is located just below the hot spot. It specifies that the code prints the message `hello` to the standard output only when the variable `a` is equal to 2. However, the implementation is incorrect because the test condition in the `if` statement contains `<` rather than `==`. Filling the hot spot with `1` reveals the implementation error because (1) the value is not equal to 2 and (2) executing the code with the variable `a` assigned to `1` would result in the message `hello` being printed to the standard output. Since `1` reveals the implementation error, it is the correct answer. Note that a *find-the-failure* question does not have the stdout cell because it is not relevant in terms of answering the question.

Figure 7.4 shows an example *bullseye* question. As shown in the figure, the `printf`

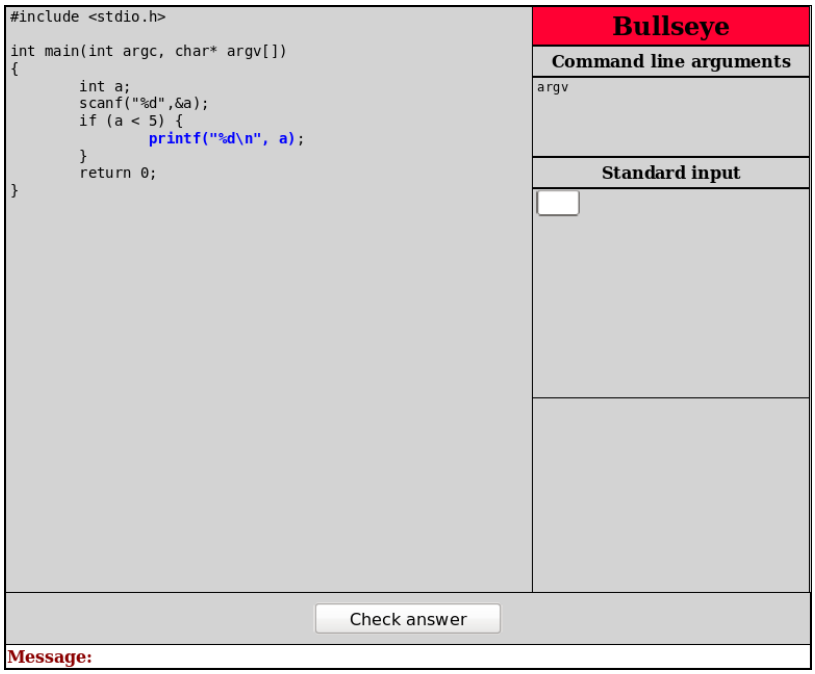


Figure 7.4: An example CA question of type *bullseye*

statement is highlighted in bold. It is executed when the variable `a` is less than 5, as specified by the test condition in the `if` statement. Filling the hot spot with `1` will cause the `printf` statement to be executed because (1) the value is less than 5 and (2) the value is assigned to the variable `a` through a call to `scanf` which reads the content of the standard input and assigns it to the variable `a`. Since `1` causes the `printf` statement to be executed, it is the correct answer. Note that a *bullseye* question does not have the stdout cell because it is not relevant in terms of answering the question.

7.1.3 Programming Languages

In addition to C, CA supports two more programming languages: Java and Python. Figure 7.5 shows an example Python question. The question is equivalent to the one shown in Figure 7.1. It is much shorter in comparison because it does not have the `main` function and the `return` statement as required by the C programming language.

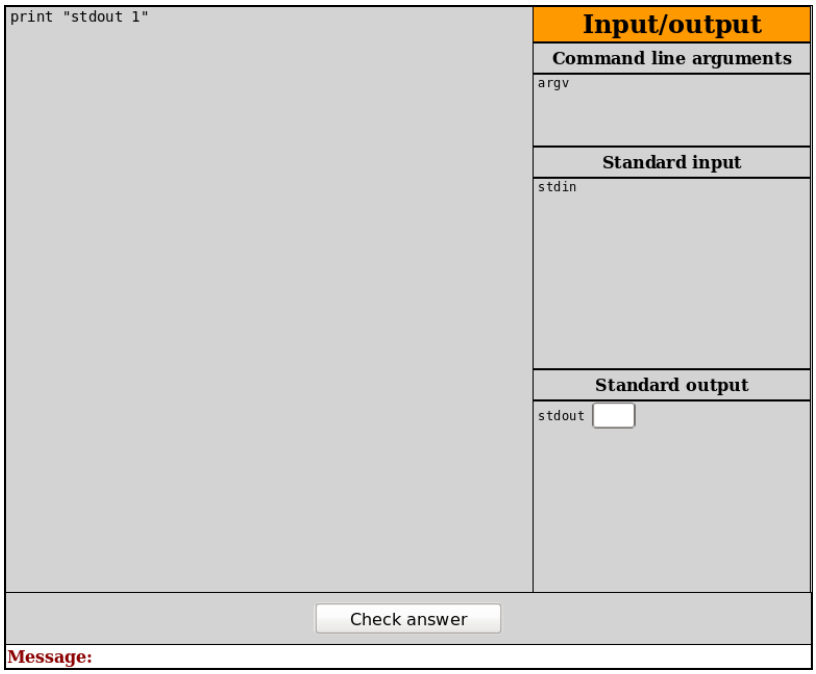


Figure 7.5: An example CA question written in Python

7.2 CA Quiz Authoring

Authoring a quiz involves creating one or more question templates, generating questions from the question templates, and creating one quiz specification.

7.2.1 Creating Question Templates

Figure 7.6 shows the question template that was used to generate the question shown in Figure 7.1. The template contains a set of variables including:

- `question_type`: specifies the question type.
- `source_language`: specifies the programming language.
- `parameter_list`: specifies the parameters used for the question. As shown in the figure, only one parameter is declared. It is identified as `$textbox` and has type of `int`.
- `tuple_list`: specifies the name of the generated questions and the value assigned to each parameter. A question name ends with an index. For example,

the question generated from this template has name `test_c_io_0`. The parameter values used for a question are specified by a list. For example, the parameter `$textbox` is specified as `None`, meaning that the parameter will appear as a hot spot.

- `global_code_template`: is used to import the libraries that are required to execute the code. The prefix `d` is used for those statements that will be displayed in the code cell, the prefix `x` is used for those statements that will be executed, and the prefix `dx` is used for those statements that will be displayed and executed. Note that `<`, as shown in line 13, is used to escape the `<` character in HTML.
- `main_code_template`: is where the code is stored. It uses the same prefix rules as `global_code_template`.
- `argv_template`: defines the contents of the command line arguments.
- `stdin_template`: defines the contents of the standard input.
- `stdout_template`: defines the contents of the standard output. As shown in the figure, the parameter `$textbox` is used here.

7.2.2 Generating Questions

Questions are generated by running a script called `generate.py` which takes a question template and the path to a question library as input. For each generated question, the script creates a folder in the question library, saves the code into the folder, and compiles the code. The folders are named after the names of the generated questions. For example, running `generate.py` against the question template shown in Figure 7.6 would generate a question stored in the `test_c_io_0` folder.

7.2.3 Creating Quiz Specification

Figure 7.7 shows the quiz specification that was used to generate the quiz shown in Figure 7.1. The specification contains a set of variables including:

```
question_type = 'input_output'
source_language = 'C'

parameter_list = [
    ['$textbox','int'],
]

tuple_list = [
    ['test_c_io_', [None]],
]

global_code_template = '''\
d   #include &lt;stdio.h>
x   #include <stdio.h>
dx
'''

main_code_template = '''\
dx      printf("stdout %d\\n", 1);
'''

argv_template = 'argv'

stdin_template = 'stdin'

stdout_template = '''\
stdout $textbox
'''
```

Figure 7.6: An example question template

```
question_list = [
#   (mark,  count,  [directories])
    (4,     1,      ['test_c_io_0']),
]
practice_mode = True
standalone = True
logged = False
log_dir = ''
```

Figure 7.7: An example quiz specification

- `question_list`: specifies the set of questions to include in the quiz. Each element in the list is a tuple containing three elements. The first element specifies the mark awarded for each correctly answered question. The second element specifies the number of questions to be sampled from the question pool. The third element specifies the directories that store the pool of questions. As shown in the figure, 4 marks are awarded for correctly answering the question `test_c_io_0`.

- `practice_mode`: is `True` when the quiz is to be taken in practice mode and `False` when the quiz is to be taken in marked mode. In practice mode, a student can check whether his or her answer is correct by clicking the `Check answer` button. In marked mode, a student can only submit his or her answer by clicking the `Submit answer` button. The student is not informed whether the answer is correct.

- `standalone`: is `True` when the quiz contains only one question and `False` otherwise.

- `logged`: is `True` when the quiz is to be recorded into a log file and `False` otherwise. A quiz log contains a student's answer and the resulting marks for each question attempted.

- `log_dir`: specifies the directory for storing the log.

7.3 Selenium Background

Selenium provides testers with the ability to automatically test a web application. Test automation is important for web application development. A typical web application undergoes a series of changes before release. Each change or group of changes is followed by running the full test suite to ensure that the changes do not inadvertently introduce bugs. Running the tests manually is costly as it is both time-consuming and error-prone. As a result, we make extensive use of Selenium for testing CA.

There are two different approaches for creating test cases in Selenium. The first approach involves the use of the eXtensible HyperText Markup Language (XHTML) [10]. For each test case, the tester creates an XHTML script that describes the list of commands to be executed and the expected outcome. The XHTML scripts are then sent to the Selenium Integrated Development Environment, Selenium IDE for short, for

execution. The second approach involves the use of the Selenium API. The Selenium API is available in Python and offers all the functionalities that are available to XHTML-based scripts. The next section describes how XHTML scripts were used for testing CA.

7.4 Script-Based Test Generation

With Selenium, the objectives of our tests are to ensure that quizzes can be invoked successfully regardless of question type and programming language and that questions are marked accurately. A successful quiz invocation is defined as the ability to accurately display all questions included in a quiz. An accurately marked question is defined as the ability to mark all correctly answered questions as correct and all incorrectly answered questions as incorrect. The following subsections explain the various items required for testing CA including the question templates, the quiz specifications, and the Selenium test cases. The quiz specifications and the Selenium test cases are generated by a script called `generate_tests.py` while the question templates are manually created.

7.4.1 Question Templates

One of the test objectives is to ensure that all questions, regardless of question type and programming language, can be displayed accurately. This calls for testing questions with different combinations of question type and programming language. Since CA supports 3 question types and 3 programming languages, a total of $3 \times 3 = 9$ questions are required for testing. As a result, 9 question templates were created. Each of them can be used to generate one question. A question template has name `test_`P`_`T where P is the programming language and T is the question type. For example, a question template that has C programming language and *input-output* question type is named as `test_c_io`. Note that, for brevity, all question types are denoted by two letters: `io` stands for *input-output*, `ff` stands for *find-the-failure*, and `be` stands for *bullseye*.

To make testing easier, the questions generated from the question templates have three features in common. First, each question has only one hot spot. Second, the hot spot is identified as `$textbox` and is of integer type. Third, `1` is the correct answer for the hot spot whereas `9` is the incorrect answer. Figure 7.6 shows the question

template of name `test_c_io`.

7.4.2 Quiz Specifications

A total of 9 quiz specifications are generated. Each of them selects only one question. The quiz specifications are named after the questions that they select without the trailing indexes. For example, a quiz specification that selects question `test_c_io_0` is named as `test_c_io`. Also, the quiz specifications have `practice_mode` set to `True`, `standalone` set to `True`, `logged` set to `True`, and `log_dir` set to an empty string. As a result, the quizzes generated by the quiz specifications would be in practice mode, each of which is a standalone quiz with answer logging disabled. Figure 7.7 shows the quiz specification that selects the question generated by the question template shown in Figure 7.6.

7.4.3 Selenium Test Cases

To meet the test objectives, combinations of question types, programming languages, and correct and incorrect answers are tested. As mentioned before, CA supports 3 different question types and 3 different programming languages. The number of test cases required is therefore $3 \times 3 \times 2 = 18$.

Figure 7.8 shows the template used for generating XHTML scripts where the place holders are shown in upper case. The `TEST_CASE` place holder in lines 7 and 12 is to be replaced by the name of the test case. For example, `TEST_CASE` is substituted by `c_io_correct` when the question under test has programming language C and *input-output* question type and the question is filled with a correct answer. The `TEST` place holder in line 16 is to be replaced by the name of the quiz specification. The `VALUE` place holder in line 22 is to be replaced by the answer to the question: `1` when the answer is correct and `9` when the answer is incorrect. The `MESSAGE` place holder in line 32 is to be replaced by the expected message that will appear when the `Check answer` button is clicked: `Correct` when the answer is correct and `Incorrect` when the answer is incorrect.

The commands performed by an XHTML script are defined in an HTML table, as shown in lines 10 through 34 of Figure 7.8. Each command is defined in a row with three columns. The first column is the command and the other two columns contain the command arguments. The following explains the action carried out by each command:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4  <head profile="http://selenium-ide.openqa.org/profiles/test-case">
5  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
6  <link rel="selenium.base" href="http://localhost:8080/" />
7  <title>TEST_CASE</title>
8  </head>
9  <body>
10 <table cellpadding="1" cellspacing="1" border="1">
11 <thead>
12 <tr><td rowspan="1" colspan="3">TEST_CASE</td></tr>
13 </thead><tbody>
14 <tr>
15     <td>open</td>
16     <td>/cqg/quiz?spec=TEST</td>
17     <td></td>
18 </tr>
19 <tr>
20     <td>type</td>
21     <td>name=$textbox</td>
22     <td>VALUE</td>
23 </tr>
24 <tr>
25     <td>clickAndWait</td>
26     <td>name=but</td>
27     <td></td>
28 </tr>
29 <tr>
30     <td>verifyText</td>
31     <td>id=message</td>
32     <td>MESSAGE</td>
33 </tr>
34 </tbody></table>
35 </body>
36 </html>
```

Figure 7.8: A template for Selenium test case

- `open`: points the web browser to the quiz specification to initiate a quiz.
- `type`: fills the hot spot with an answer. The hot spot is located by searching for an `input` element with attribute `name` equal to `$textbox`.
- `clickAndWait`: clicks the `Check answer` button. The button is located by searching for an `input` element with attribute `name` equal to `but`. After clicking the button, Selenium will declare session timeout if the requested web page does not load within a specified number of seconds.
- `verifyText`: determines whether the test passes or fails by comparing the message, displayed at the bottom of the question, with the text that replaces the `MESSAGE` place holder. The displayed message is located by searching for an element text that is embedded in the element that has attribute `id` equal to `message`.

7.4.4 Afterthought

Although the XHTML scripts that we generated were able to meet the test objectives, our test approach was not elegant. The Python script `generate_tests.py` that was used to generate the XHTML scripts was long and complicated. Most of the code was devoted to creating a template for the XHTML scripts and substituting the place holders with their corresponding values. As a result, it is difficult to see the combinations of CA features being tested. Readability is important as the testers may need to expand the tests later to cover more combinations. In the next section, we show how readability can be significantly improved by the use of GBTG.

Another weakness of our test approach is the use of XHTML scripts. First, each test case calls for an XHTML script. Since there were 18 test cases in our test setup, a total of 18 scripts was required. Managing the scripts would become a difficult task if we were to expand our tests to cover more combinations of CA features. Second, the XHTML scripts are verbose. Each script is 36 lines long and about half of the lines are irrelevant to testing.

7.5 Grammar-Based Test Generation

The strength of GBTG is its ability to generate many test combinations with only a few lines of code. In this section, we show a grammar that generates the same set of

test cases as the Python script `generate_tests.py` did. The Selenium API is used instead of XHTML scripts because of the problems associated with the latter.

7.5.1 Test Approach

Figure 7.9 shows the `Question` grammar. Combinations of programming languages, question types, and correct and incorrect answers are generated in lines 28 through 35. Note that each of the two `Answer` rules consists of two nonterminals. The first nonterminal specifies the answer. The second nonterminal specifies the expected message of the answer.

Each test case is driven by the Selenium API. The code that is used to start Selenium is defined in the `global_precode` block. Line 2 imports the Selenium library, line 5 invokes a test driver for the Firefox web browser, and line 6 specifies the number of seconds to wait for each HyperText Transfer Protocol (HTTP) request. Similarly, the code that is used to end Selenium is defined in the `global_postcode` block. `driver.quit()` is called to terminate the test driver.

For each combination generated by the grammar, the `postcode` is executed. Lines 13 through 16 extract the programming language, the question type, the answer, and the expected message. Line 19 creates the test name which is essentially the name of the quiz specification. The test name consists of the programming language and the question type. Line 22 points the web browser to the quiz specification to initiate a quiz. Line 23 fills the hot spot with the answer. Line 24 clicks the `Check answer` button. Lines 25 and 26 print an error message if the displayed message is different from the expected message.

7.5.2 Test Reduction

In addition to improved readability, GBTG provides testers the ability to adjust the number of test cases with covering arrays. The `Question` grammar shown in Figure 7.9 generates fewer test cases if a `cov` tag is attached to the `Question` rule. For example, applying the `cov` tag {`cov [([0,1],2)]`} would reduce the number of test cases to 9. The resulting test covers only the combinations of programming languages and question types. As another example, applying the `cov` tag {`cov [([2],1)]`} would reduce the number of test cases to 2. The resulting test covers only one correctly answered question and one incorrectly answered question. Although the test reduction presented here is not significant due to the small size of the test space,

```
1  {global_precode
2  from selenium import webdriver
3
4  # set up selenium
5  driver = webdriver.Firefox()
6  driver.implicitly_wait(3)
7  }
8
9  {postcode
10     global driver
11
12     # extract language, type, input, and expect
13     language = s[0][0]
14     type = s[1][0]
15     input = s[2][0]
16     expect = s[2][1]
17
18     # create test name
19     test_name = 'test_'+language+'_'+type
20
21     # execute test
22     driver.get('http://localhost:8080/cqg/quiz?spec='+test_name)
23     driver.find_element_by_name('$textbox').send_keys(input)
24     driver.find_element_by_name('but').click()
25     if driver.find_element_by_id('message').text != expect:
26         print test_name+' failed; expect:  '+expect
27 }
28 Question ::= Language Type Answer;
29
30 Language ::= List('c','java','python');
31
32 Type ::= List('be','ff','io');
33
34 Answer ::= '1' 'Correct';
35 Answer ::= '9' 'Incorrect';
36
37 {global_postcode
38 # tear down selenium
39 driver.quit()
40 }
```

Figure 7.9: `Question` grammar

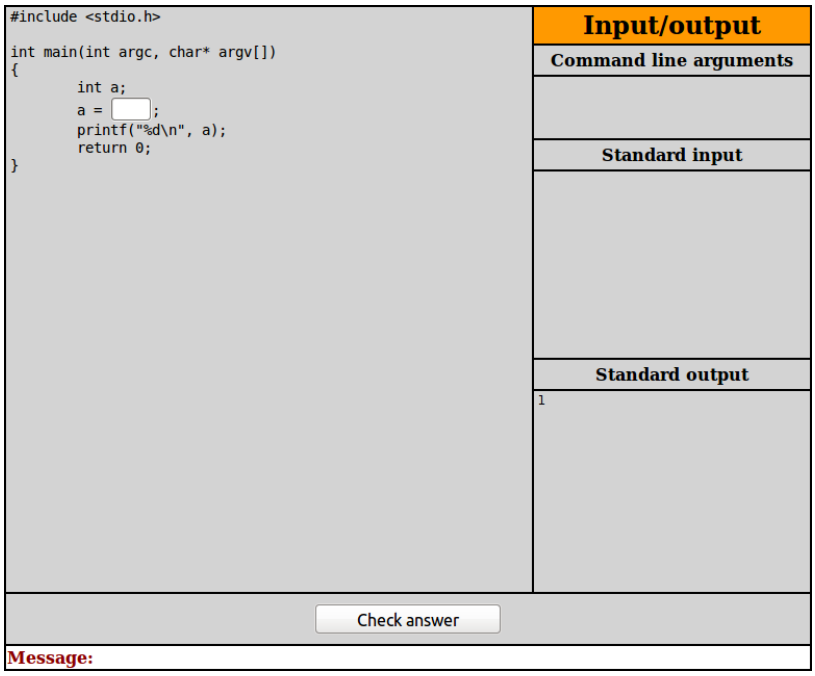


Figure 7.10: A CA question that has a hot spot of type integer

the `cov` tag proves extremely useful in later sections when we extend the test to cover more combinations of CA features.

7.6 Syntax Checking

Another aspect of CA that needs to be tested is the syntax checking of the inputs. As mentioned before, a student answers a question by filling all the hot spots in the question. Each hot spot expects an input of certain type. For example, Figure 7.10 shows a question with one hot spot. The input to the hot spot, when executed, is assigned to a variable named `a`. Since the variable is declared as an integer type, the hot spot therefore expects an integer. However, a student may not enter an integer as expected. A code fragment may be entered instead such that executing the displayed code will yield an unexpected output, a potential security loophole. Therefore, checking the syntax of the inputs is of paramount importance.

To avoid the aforementioned problem, CA performs syntax checks when a student clicks the `Check answer` button. It also strips out any unnecessary space that may exist in the inputs. This is so because a student may accidentally insert a

space while answering a question. With the extra space, what may have been a correct answer becomes incorrect. For example, the correct answer for the question shown in Figure 7.10 is `1`. Filling the hot spot with a space followed by `1` will cause `Message: Incorrect` to appear. Because spaces are difficult to notice in an HTML text field, it is not uncommon for a student to overlook the extra space that he or she accidentally inserted. As a result, a student may become frustrated with CA because he or she believes what has been entered ought to be marked as correct. For this reason, CA performs space stripping for all inputs, one rule for each input type. Currently, there are three input types supported by CA: `int`, `float`, and `string`.

7.6.1 Syntax and Space Stripping Rules

Below describes the syntax for each of the three input types:

- `int`: a string that starts with an optional plus or minus sign followed by one or more digits. The maximum number of digits is five.
- `float`: a string that starts with an optional plus or minus sign followed by one or more digits, a decimal point, and one or more digits. The maximum number of digits to the left of the decimal point is five and so is the maximum number of digits to the right of the decimal point.
- `string`: a string that consists of zero or more alphanumeric, underscore, and space characters.

With regard to the leading and trailing spaces, the space stripping rules are identical for all three input types, that is, CA strips all leading and trailing spaces. The `string` input type, however, has one extra rule with regard to the spaces that appear between words. For each pair of adjacent words, there can only be a space in between. Any extra space will be stripped by CA.

7.6.2 Testing Strategy

Initially, the problem of testing the syntax checking implementation is tackled as follows. First, three questions were manually created, one for each programming language. Each question contains three hot spots: one of type `int`, one of type `float`, and one of type `string`. Combinations of programming languages and question types were not tested because the same function, `clean_values`, is always invoked for syntax

Task	Running time
bring up Firefox	3.047 s
fetch web page	0.067 s
fill in hot spot	0.225 s
submit answer	0.171 s
verify message	0.044 s

Figure 7.11: Running time for function calls

checks, regardless of the programming language and the question type of the question being invoked. Second, Selenium was invoked to fill the questions with combinations of valid and invalid inputs. Finally, we checked to see if the syntax checks behave correctly. In particular, the displayed code should not be executed if the syntax checks fail.

This testing strategy does not scale because running a Selenium test case takes a lot of time. Using a Dell Precision 390 with an 2.13 GHz Intel Core2 Duo and 2 gigabytes of RAM, running the tests described in section 7.5 requires 15.008 seconds. Since there are in total 18 test cases being executed, this translates to about 0.834 seconds per test case. The reason why running a Selenium test case takes so much time is because doing so requires the test machine to bring up Firefox, fetch the web page that contains the code in question, fill in the hot spot(s), submit the answer by clicking the `Check answer` button, and verify the message. Figure 7.11 shows the time it takes to complete each task. Note that the most expensive task is bringing up Firefox which takes about 3 seconds. However, the test cases are executed as a batch and therefore Firefox only needs to be brought up once per test run. Nevertheless, it is obvious that Selenium was not suitable to run a large number of test cases as doing so would take too long.

Although Selenium tests do not scale, they are still essential because they ensure that the `clean_values` function is being invoked with the correct parameters at the right time. In contrast, it is not necessary to invoke Selenium in order to test whether the function behaves correctly. The latter should be performed standalone. As a result, we divided the tests into function-based and selenium-based. The function-based tests ensure that the `clean_values` function has the expected behavior while the selenium-based tests ensure that the function is invoked with the correct parameters at the right time. However, the existing `clean_values` function has strong coupling with the CA implementation, making it difficult to test stan-

Function name	Running time
`check_int`	4.525 μs
`check_float`	4.589 μs
`check_string`	2.863 μs
`strip_field`	0.935 μs
`normalize_string`	2.782 μs

Figure 7.12: Running time for function calls

dalone. To counter this problem, the syntax checking and space stripping code was lifted out from the `clean_values` function and converted into five functions.

The purposes of these five functions are described as follows. Each of these functions takes a parameter called `value` that is of `string` type.

- `check_int`: returns `True` when `value` satisfies the syntax rule of `int` and raises an exception otherwise.
- `check_float`: returns `True` when `value` satisfies the syntax rule of `float` and raises an exception otherwise.
- `check_string`: returns `True` when `value` satisfies the syntax rule of `string` and raises an exception otherwise.
- `strip_field`: returns `value` with the leading and trailing spaces stripped.
- `normalize_string`: returns `value` with the number of spaces between each pair of adjacent words reduced to one.

7.6.3 Function-Based Tests

Each one of these five functions is tested standalone. Figure 7.12 shows the time it takes to call each function. As shown in the figure, a function call ranges from 1 to 5 microseconds in duration. This is extremely fast compared with the time it takes to run a Selenium test case, making it ideal for testing a large varieties of inputs.

In order to test `check_int`, `check_float`, and `check_string`, valid and invalid inputs were manually created. Manual test generation was used instead of GBTG because the syntax for all three input types are quite simple. A human tester can easily generate sufficient inputs for testing. Figure 7.13 shows the valid inputs. As shown in the figure, the `int` column contains integers with or without the leading

int	float	string
"1"	"1.0"	"hello world"
"12345"	"1.23456"	"HeLLo"
"+1"	"12345.12345"	"x y"
"-1"	"+12345.12345"	"x y"
"-12345"	"-12345.12345"	"x y"
"+12345"		"x y z"
		"x y z"
		"1239"
		"123_45 3"

Figure 7.13: Valid inputs

sign and integers with one or five digits. The `float` column contains floating-point numbers with or without the leading sign, floating-point numbers with one or five digits in the integer part, and floating-point numbers with one or five digits in the fractional part. The `string` column contains strings with lower or upper case letters, strings with digits and underscores, and strings with more than one space between two adjacent words.

Figure 7.14 shows the invalid inputs. As shown in the figure, the `int` column contains integers with more than five digits, integers with periods and letters, integers with more than one sign, and an integer with no digits. The `float` column contains floating-point numbers with missing integer or fractional parts, floating-point numbers with letters and spaces, floating-point numbers with more than one sign, and a floating-point number with no digits. The `string` column contains strings with illegal characters.

The `strip_field` function was tested by inserting a random number of leading and trailing spaces to the valid inputs. The `normalize_string` function was tested by those valid strings with more than one space between two adjacent words. For example, `"x  y"` was used to test the `normalize_string` function because it has two spaces between words `x` and `y`.

7.6.4 Selenium-Based Tests

Three questions were manually created in order to perform the Selenium tests, one for each programming language. The names of the questions are `test_c_io_syntax`, `test_java_io_syntax`, and `test_python_io_syntax`. All three questions are identical

int	float	string
"123456"	"1."	"1.0"
"1.0"	"1.."	"+"
"-2-2"	".1"	"-"
"++2"	".000001"	'"asy?/"'
"+123456"	"10000.00001+"	"''"
"1d"	"++1.0"	"\0"
"xyz"	"-+1.0"	" a b ^^"
""	"1 3.0"	
	"1/2"	
	"23"	
	"234f"	
	""	
	"abc"	

Figure 7.14: Invalid inputs

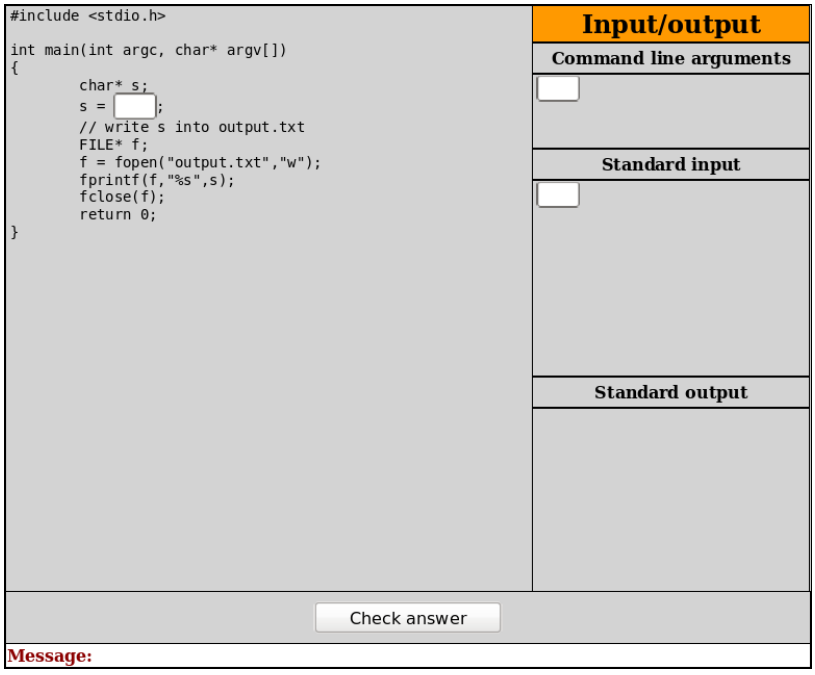


Figure 7.15: Question generated from question template `test_c_io_syntax`

```
1   Syntax_check ::= Question Input1 Input2 Input3;
2
3   Question ::= List('test_c_io_syntax','test_java_io_syntax',
     'test_python_io_syntax');
4
5   Input1 ::= 'valid' '123';
6   Input1 ::= 'valid' ' 123';
7   Input1 ::= 'invalid' '123456';
8
9   Input2 ::= 'valid' '123.123';
10  Input2 ::= 'valid' ' 123.123';
11  Input2 ::= 'invalid' '123456.123456';
12
13  Input3 ::= 'valid' 'X Y';
14  Input3 ::= 'valid' ' X Y ';
15  Input3 ::= 'invalid' 'X + Y';
```

Figure 7.16: `Syntax_check` grammar

except that the displayed codes are written in different programming languages. Figure 7.15 shows question `test_c_io_syntax`. The question has three hot spots, one for each input type. The hot spot in code cell, argv cell, and stdin cell expects a string, an integer, and a floating-point number respectively. The displayed code, when executed, writes the content of variable `s` into a file called `output.txt`. In other words, the presence of the file indicates that the displayed code is executed. Otherwise, the displayed code is not executed.

The test cases were generated and executed by running the grammar shown in Figure 7.16. The embedded code blocks are not shown here to avoid clutter. Each test case, as defined in line 1, consists of the name of a question followed by the inputs for the hot spots. Each input is of a different type and is decorated with a flag, indicating whether the input is valid. As shown in the grammar, three inputs are generated for each input type, two valid and one invalid. One of the valid inputs strictly follows the syntax rule for a specific input type while the other has one violation caused by one leading or trailing space.

For each test case generated, the `postcode` attached to the `Syntax_check` rule is executed. The `postcode`, when executed, points the web browser to the specified question, fills the hot spots with the generated inputs, submits the answer, and checks to see if the displayed code is executed. A test case is considered fail if (1) one of

the generated inputs is invalid and the displayed code is executed and (2) all of the generated inputs are valid but the displayed code is not executed due to leading or trailing spaces.

7.7 Extended Testing

In this section, we are going to show how GBTG is used to test more combinations of CA features. The combinations of question parameters being tested are:

- Programming languages C, Java, and Python.
- Question types *input-output*, *find-the-failure*, and *bullseye*.
- Zero, one, two, and three hot spots in argv cell.
- Zero, one, two, and three hot spots in stdin cell.
- Zero, one, two, and three hot spots in code cell.
- Correct and incorrect answers.

In addition, the combinations of quiz parameters being tested are:

- Zero, one, two, and three questions in `question_list`.
- `True` and `False` assigned to `practice_mode`.
- `True` and `False` assigned to `standalone`.

7.7.1 Test Approach

Our test approach involves two grammars. The first grammar is called `Generate`. It is responsible for generating question templates and generating questions from these question templates. The second grammar is called `Test`. It is responsible for generating quiz specifications and testing the quizzes through Selenium API. Figure 7.17 shows the `Generate` grammar. For brevity, the embedded code blocks are not displayed. The `Generate` rule specifies the question parameters being tested. This includes programming language, question type, and number of hot spots in the argv, code, and stdin cells. The rest of the rules specify the value assigned to each parameter.

```
1   Generate ::= Language Type Num_argv Num_code Num_stdin;
2
3   Language ::= List('c','java','python');
4
5   Type ::= List('be','ff','io');
6
7   Num_argv ::= Range(0,1,4);
8
9   Num_code ::= Range(0,1,4);
10
11  Num_stdin ::= Range(0,1,4);
```

Figure 7.17: `Generate` grammar

For each combination of question parameters generated, the `postcode` attached to the `Generate` rule is invoked. It extracts the value of each parameter and uses them to generate a question template of name `test_`P`_`T`_`A`_`C`_`S where:

- P is the programming language.
- T is the question type.
- A is the number of hot spots for argv cell.
- C is the number of hot spots for code cell.
- S is the number of hot spots for stdin cell.

A question template is generated by substituting the place holders in a template of question template, hereafter referred to as a template, with corresponding values. A total of nine templates are manually created, one for each combination of programming language and question type. Figure 7.18 shows the template for programming language C and question type *input-output* where the place holders are shown in upper case. The `PARAMETER_LIST` place holder in line 4 is replaced by a list of parameters, one for each hot spot. All the parameters are of type `int` and each has name `$t` followed by an index. For example, the first parameter in `parameter_list` is defined as `['$t0','int']` and the second parameter in `parameter_list` is defined as `['$t1','int']`. The `TUPLE_LIST` place holder in line 6 is replaced by the name of the question template followed by a list of parameter values. Since all the parameters in `parameter_list` are used for hot spots, their parameter values are `None`. The

NUM_ARGV, NUM_CODE, and NUM_STDIN place holders in lines 19, 22, and 25 are replaced by the number of hot spots in the argv, code, and stdin cells respectively. The CODE, ARGV, and STDIN place holders in lines 16, 34, and 36 are replaced by the names of the hot spots that appear in code, argv, and stdin cells respectively. The SUM place holder in line 38 is replaced by the total number of hot spots. To make testing easier, the questions generated from the question templates, regardless of programming language and question type, have one feature in common: 1 is the correct answer for the hot spots and 0 otherwise.

Figure 7.19 shows the Test grammar. As with the Generate grammar, the embedded code blocks are not displayed for brevity. Also, the italicized texts—*RdepthTag* and *CovTag*—are place holders for a rdepth tag and a cov tag respectively. These texts are for illustration purposes only and are not part of the grammar. The Test rule specifies the quiz parameters being tested such as practice_mode, standalone, and the questions to be included in question_list. The number of questions to be generated is specified by the rdepth tag that replaces the *RdepthTag* place holder. For example, replacing *RdepthTag* with {rdepth 4} would generate 0 through 3 questions. The Question rule specifies the question parameters being tested. This rule has the similar right-hand side as the Generate rule shown in Figure 7.17 except that the former has one additional nonterminal: Answer. As shown in Figure 7.19, there are two rules that start with the Answer nonterminal. Each of these two rules has two terminals on its right-hand side. The first terminal specifies the input for all the hot spots. The second terminal specifies the expected message for the input.

For each combination of question parameters generated, the postcode attached to, reading the grammar from top to bottom, the second Questions rule is invoked. It extracts the parameter values and uses them to generate the question name, the input, and the expected message. For each combination of quiz and question parameters generated, the postcode attached to the Test rule is invoked. It extracts the parameter values and uses them to generate a quiz specification. It then starts the quiz in order to check the following items:

- *message*: This is achieved by filling in the hot spots with the input extracted from the Questions rule, clicking the Check answer button, and comparing the displayed message with the expected message. Note that message checking is performed only when practice_mode is True.

- *fields*: Each question type has several fields where each field is displayed in

```
1  question_type = 'input_output'
2  source_language = 'C'
3
4  parameter_list = PARAMETER_LIST
5
6  tuple_list = TUPLE_LIST
7
8  global_code_template = '''\
9  d   #include &lt;stdio.h>
10 x   #include <stdio.h>
11 dx
12 '''
13
14 main_code_template = '''\
15 dx      int sum = 0;
16 dx      int code[] = {CODE};
17 dx
18 dx      int i;
19 dx      for (i = 1; i < NUM_ARGV+1; i++) {
20 dx          sum += atoi(argv[i]);
21 dx      }
22 dx      for (i = 0; i < NUM_CODE; i++) {
23 dx          sum += code[i];
24 dx      }
25 dx      for (i = 0; i < NUM_STDIN; i++) {
26 dx          int s;
27 dx          scanf("%d\\n", &s);
28 dx          sum += s;
29 dx      }
30 dx
31 dx      printf("%d\\n", sum);
32 '''
33
34 argv_template = '''ARGV'''
35
36 stdin_template = '''STDIN'''
37
38 stdout_template = '''SUM'''
```

Figure 7.18: The template for generating question templates with programming language C and question type *input-output*

```
Test ::= Quiz Questions;

Quiz ::= Practice_mode Standalone;

Practice_mode ::= 'True';
Practice_mode ::= 'False';

Standalone ::= 'True';
Standalone ::= 'False';

RdepthTag  Questions;
Questions ::= '';
Questions ::= Question Questions;

CovTag     Question ::= Language Type Num_argv Num_code Num_stdin
            Answer;

Language ::= List('c','java','python');

Type ::= List('be','ff','io');

Num_argv ::= Range(0,1,4);

Num_code ::= Range(0,1,4);

Num_stdin ::= Range(0,1,4);

Answer ::= '1' 'Correct';
Answer ::= '0' 'Incorrect';
```

Figure 7.19: `Test` grammar

one single table cell. An *input-output* question should display `Input/output`, `Command line arguments`, `Standard input`, and `Standard output`. A *bulls-eye* question should display `Bullseye`, `Command line arguments`, and `Standard input`. An *find-the-failure* question should display `Find the failure`, `Command line arguments`, and `Standard input`.

- *navigation buttons*: These refer to the `Previous` and `Next` buttons that are displayed when `standalone` is `False`. The `Previous` button should be disabled when the first question is displayed. The `Next` button should be disabled when the last question is displayed.

- *answer retention*: When a student uses the navigation buttons to go back to the questions he or she has answered, the hot spots must be filled with the values that were previously submitted.

7.7.2 Test Execution

In principle, the test can be executed by running the `Test` grammar with the *RdepthTag* place holder replaced with {`rdepth 4`} and the *CovTag* place holder replaced with an empty string. This configuration will test all combinations of quiz and question parameters where the number of questions per quiz ranges from 0 to 3. However, running the grammar as such would take too long as too many test cases are generated. For example, the combinations of question parameters can be expressed as $L(\texttt{Language})\times\ L(\texttt{Type})\times\ L(\texttt{Num\_argv})\times\ L(\texttt{Num\_code})\times\ L(\texttt{Num\_stdin})\times\ L(\texttt{Answer})$. The number of combinations is therefore $3 \times 3 \times 4 \times 4 \times 4 \times 2 = 1,152$. The combinations of quiz parameters can be expressed as $L(\texttt{Practice\_mode}) \times L(\texttt{Standalone})$. The number of combinations is therefore $2 \times 2 = 4$. Since the set of test cases generated can be expressed as $L(\texttt{Quiz}) \times L(\texttt{Questions})$, the number of test cases for 0 questions is $4 \times 1 = 4$, for 1 question is $4 \times 1,152 = 4,608$, for 2 questions is $4 \times 1,152 \times 1,152 = 5,308,416$, and for 3 questions is $4 \times 1,152 \times 1,152 \times 1,152 = 6,115,295,232$. As a result, the number of test cases for 0 through 3 questions is $4 + 4,608 + 5,308,416 + 6,115,295,232 = 6,120,608,260$. This translates to approximately 162 years of execution time as it takes 0.834 seconds to execute one Selenium test case as aforementioned.

In order to tackle this problem, the `Test` grammar is run twice. The first run involves replacing the *RdepthTag* place holder with {`rdepth 2`} and the *CovTag* place

holder with an empty string. The focus of this test is to exercise all combinations of quiz and question parameters while limiting the number of questions per quiz to 1. The resulting number of test cases is $4 \times (1 + 1,152) = 4,612$. The second run involves replacing the *RdepthTag* place holder with {`rdepth 4`} and the *CovTag* place holder with {`cov [ ([0,1,2,3,4,5],1) ]`}. The focus of this test is to exercise all combinations of quiz parameters with number of questions per quiz ranging from 0 to 3. Since all combinations of question parameters are already exercised in the first run, a `cov` tag was applied in order to reduce the number of combinations of question parameters. The resulting number of test cases is $4 \times (1 + 4 + 4^2 + 4^3) = 340$. With the number of test cases reduced to $4,612 + 340 = 4,952$, the test execution time becomes approximately 1 hour.

7.8 Discussion

Typical Selenium test scripts are manually created. Generating Selenium test scripts programmatically tends to be complicated and therefore the resulting test code is often difficult to maintain. Such is the case for `generate_tests.py` that generates Selenium test scripts for testing CA. By replacing the Python script with a grammar, we demonstrated how GBTG can be used to automatically generate Selenium test scripts. The resulting grammar is superior to `generate_tests.py` because the former is more readable. We also discussed how the grammar can be easily extended to cover more combinations of CA features. In particular, we demonstrated how covering arrays can be applied in order to counter the combinatorial explosion resulting from the greater test coverage. Moreover, we identified the testing scenarios where the use of GBTG is not appropriate. For example, GBTG has limited value in generating test inputs that do not involve combinations of different test parameters. Such test inputs can be easily created manually and therefore applying GBTG is not helpful.

Chapter 8

Related Work

8.1 GBTG Implementations

Over the past forty years, there has been a lot of research on GBTG. The following subsections present the language generation strategies used and the extra-grammatical features supported. Where applicable, a comparison to YouGen_NG is provided.

8.1.1 Language Generation Strategies

To our knowledge, there exist four language generation strategies in the GBTG literature. For each of these strategies, the behavior is described followed by the published works in which it appears.

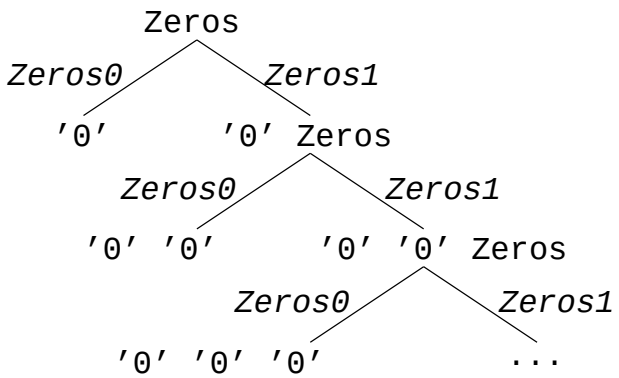
1. *Rule coverage*: requires that every rule in a grammar is used at least once. Figure 8.1 (a) shows the `Zeros` grammar with two rules. The italicized texts on the left are rule identifiers. They are used for illustration purposes and are not part of the grammar. Figure 8.1 (b) shows the `Zeros` generation tree. Each arc is marked with the identifier of the rule that was applied. As shown in the figure, the string `'00'` is derived by first applying rule `Zeros1` and then applying rule `Zeros0`. Since deriving the string requires the use of both rules, the language {`'00'`} satisfies rule coverage.

   Rule coverage was first introduced by Purdom [39] and later used by Malloy [30]. Its advantage is reducing the number of test cases. This is particularly true for testing programming languages whose grammars typically consist of hundreds

Zeros0 `Zeros ::= '0';`

Zeros1 `Zeros ::= '0' Zeros;`

(a) `Zeros` grammar



(b) `Zeros` generation tree

Figure 8.1: `Zeros` grammar and generation tree

of rules. For example, Malloy [30] used GBTG to test ANSI C and GNU C++ implementations. Because the grammars for the programming languages generate an infinite number of test cases, rule coverage was applied. For the ANSI C grammar that has 213 rules, 11 test cases were generated. For the GNU C++ grammar that has 824 rules, 81 test cases were generated. Rule coverage is seldom used as it generates too few test cases to be effective.

2. *Random traversal*: randomly selects a rule for application, given a nonterminal with multiple rule alternatives. For example, the nonterminal `Zeros` in Figure 8.1 (a) has two rule alternatives. When random traversal is used, one of the two rule alternatives is selected for application. Hanford [20], Bird et al. [13], Murali et al. [37], Homer et al. [25], and Maurer [34] incorporate random traversal in their GBTG tools.

3. *Depth-first traversal*: generates the language of a grammar in depth-first order. Figure 8.1 (b) shows the generation tree for the `Zeros` grammar shown in Figure 8.1 (a). When depth-first traversal is used, the generation tree is visited in depth-first order. YouGen_NG uses depth-first traversal.

4. *Breadth-first traversal*: generates the language of a grammar in breadth-first or-

der. As mentioned before, each string in the language has a parse tree structure. Breadth-first traversal generates the strings in the order of increasing parse tree depth. Figure 8.1 (b) shows the generation tree for the `Zeros` grammar shown in Figure 8.1 (a). `'0'` is generated before `'00'` because the former has smaller parse tree depth than the latter. Zaytsev [48] and Lammel et al. [27] incorporate breadth-first traversal in their GBTG tools.

8.1.2 Extra-Grammatical Features

The first known application of GBTG is compiler testing. In 1970, Hanford [20] developed a GBTG tool called a syntax machine to test a PL/1 compiler. While he applied GBTG to compiler testing, he encountered one difficulty. Although a context-free grammar is capable of generating syntactically correct test cases, it is not useful for testing compilers because typical programming languages are context-sensitive. Therefore, he introduced the notion of extra-grammatical rules: a set of GBTG features to support the development of context-sensitive grammars. The following is a list of GBTG features that have been introduced in the past. For each of these features, its behavior is described followed by the published works in which it appears.

- *Embedded code*: refers to code fragments that are attached to a grammar. Given a grammar, its code fragments are executed when the language of the grammar is generated. Celentano et al. [14] introduced the concept of actions. An action is a code fragment that is attached to a rule. When a rule is applied, its action is executed. The use of actions also appears in Duncan et al. [18]. YouGen_NG's `precode` and `postcode` play the role of actions except that the former is executed just before a rule is applied and the latter is executed after a rule's right-hand side becomes ground. YouGen_NG also supports `global_precode` and `global_postcode` which are executed before and after language generation respectively.

- *Derivation-limiting tags*: are used to limit the number of rule applications which in turn reduce the number of strings that can be generated by a grammar. A short description for each of the four derivation-limiting tags is provided as follows:

  - *Guard tag*: contains a boolean expression whose return value is used to determine whether a rule can be selected for application. If the boolean expression evaluates to true, then the rule to which the guard tag is attached can be applied. Otherwise, an alternative rule is selected for application. The use of guard tags appears in Duncan et al. [18] and Homer et al. [25]. The YouGen_NG equivalent of guard tag is `precode`.
  - *Count tag*: limits the number of times a rule can be applied. The use of count tags appears in Hanford [20] and Murali et al. [37]. The YouGen_NG equivalent of count tag is the `count` tag.
  - *Recursion tag*: limits the number of times a rule can be applied recursively. Figure 8.1 (b) shows an example recursive rule application. From top to bottom, the second application of rule `Zeros1` is recursive because it replaces the `Zeros` nonterminal that is yielded by the first application of rule `Zeros1`. The use of recursion tags appears in Hanford [20], Celentano et al. [14], and Bird et al. [13]. The YouGen_NG equivalent of recursion tag is the `rdepth` tag.
  - *Depth tag*: limits the depth of a parse tree after application of a rule. Let N be a nonterminal that has a depth tag of value D. The depth of the parse tree rooted at N should be no more than D. The use of depth tags appears in Zaytsev [48] and Lammel et al. [27]. The YouGen_NG equivalent of depth tag is the `depth` tag.

- *Weight tag*: usually used when random traversal is chosen as the language generation strategy. It specifies the probability of a rule being invoked. Let R_1 and R_2 be two rules with the same left-hand side. If R_1 has weight 1 and R_2 has weight 2, then R_2 is twice as likely to be applied as R_1. The use of weight tags appears in Hanford [20], Payne [38], and Bird et al. [13]. Figure 8.2 shows how YouGen_NG supports weight tag by the use of `precode`. The variable `R` is used to determine which of the two rules to apply. Whenever nonterminal `Zeros` is chosen for replacement, a random number with value ranging from 0.0 to 1.0 is generated and assigned to `R`. If `R` is within the range of 0.0 and 0.333, then rule `Zeros0` is applied. Otherwise, rule `Zeros1` is applied. Since `R` is twice as likely to be in [0.333..1.0] than in [0..0.333) otherwise, rule `Zeros1` is twice as likely to be applied as rule `Zeros0`.

```
{global_precode
import random
R = 0.0
}

{precode
        global R
        R = random.random()
        return 0.0 <= R < 0.333
}
Zeros ::= '0';

{precode
        global R
        return 0.333 <= R < 1.0
}
Zeros ::= '0' Zeros;
```

Figure 8.2: `Zeros` grammar with `precode` for emulating rule weight

- *Terminal generators*: are used to generate a set of rules that have the same left-hand side. For example, to develop a grammar that generates the English alphabet, one would create 26 rules where each rule's right-hand side is an English letter. The same can be achieved by creating a terminal generator that generates the English alphabet programmatically. The use of terminal generators appears in Homer et al. [25] and Maurer [33]. YouGen_NG supports four variants of terminal generators: `Range`, `List`, `File`, and custom.

- *Dynamic grammars*: allow the addition of rules while the language of a grammar is being generated. First introduced by Hanford [20], dynamic grammars are useful for testing the use of identifiers in programming languages. For example, following a declaration of an integer variable A, the rule `integer ::= A;` is added to the grammar so that A can be used in the statements that follow the variable declaration. YouGen_NG supports dynamic grammars by the use of API grammars, that is, grammar developers have the ability to programmatically create rules within embedded code blocks.

- *Dependence control*: limits the number of strings that can be generated by a rule. As mentioned before, a rule's right-hand side is a list of nonterminals

and terminals, each of which can generate a language of certain size. Without dependence control, the set of strings generated by a rule is simply the cartesian product of the languages of the nonterminals and terminals. Dependence control reduces the number of strings by taking a rule's right-hand side as a list of parameters and returning a covering-array. The use of dependence control appears in Zaytsev [48] and Lammel et al. [27]. YouGen_NG equivalent of dependence control is the `cov` tag.

- *Balance control*: limits the variations of the parse tree depth in a string. Introduced by Zaytsev [48] and Lammel et al. [27], the usefulness of balance control is not clear and therefore is not supported by YouGen_NG.

- *Variable*: provides a temporary place for storing a ground sentential form derived from a nonterminal so that it can be applied to other parts of the grammar. This feature is useful when a value that is generated early in the language generation process is needed later on. The use of variables appears in Sirer et al. [41] and Maurer [34]. Variables are not supported by YouGen_NG.

8.2 Applications of GBTG

GBTG has seen many different applications over the past forty years. We explain some of the most notable applications here.

8.2.1 Very Large-Scale Integrated Circuits

Verifying the correctness of Very Large-Scale Integrated (VLSI) circuits is a difficult task because typical circuits have poor controllability and observability, that is, it is not easy to pass the test inputs to the circuit under test and capture its outputs. Maurer [32, 34] introduced a novel approach for testing a math accelerator unit (MAU), a floating point processor that connects to a central processing unit (CPU) through a data bus. The authors solved the controllability and observability problems by creating the protocol and functional simulators. The protocol simulator emulates the interaction between the MAU and the CPU while the functional simulator emulates the MAU.

With improved controllability and observability, the authors then manually created a set of tests that exercise all of the MAU's control paths. The resulting test

set was able to catch most of the bugs. However, the tests were time consuming to create. It took the authors four months to create approximately 13,000 test cases. Also, some bugs can still be found even after the MAU passed all of the tests. Because of these, the authors turned to GBTG for help. They developed a GBTG tool that is customized for generating MAU test cases. Each test case involves an arithmetic operator, two floating point numbers, and the expected output. The floating point numbers are divided into five formats and each format is randomly generated. In total, 100,000 GBTG test cases were ran and the bugs that slipped through the manually created tests were uncovered.

In short, GBTG's advantage over manual test generation is apparent. First, it is easy for GBTG to generate a lot of test cases quickly. Second, GBTG is able to generate combinations of input parameters that can be easily overlooked by test developers and therefore reveal more bugs.

8.2.2 Java Virtual Machines

Sirer et al. [41] used GBTG to test the Java Virtual Machine (JVM). The test objective was to ensure that (1) any code that violates the security axioms is never executed and (2) the bytecode instructions are interpreted correctly. In the past, the test cases were often hand generated. Not only were they time consuming to create and maintain, but they also had poor test coverage. In order to improve test quality, the authors developed *lava*, a GBTG tool that was specifically designed for generating Java bytecode. As with YouGen_NG, *lava* is a code generator generator which takes a grammar as input and outputs a code generator. The code generator in turn generates the language of the grammar.

The oracle problem was tackled by using two approaches. The first approach is comparative testing, that is, multiple implementations of JVM were tested against the same set of test cases and their correctness depends on whether their outputs agree. However, this approach does not work when multiple implementations were not available or when all implementations fail on the same set of test cases. As a result, the second approach was proposed. It involves generation of one behavioural description for each test case generated. A behavioural description, when executed, produces the same output as its corresponding test case.

In all, the authors observed two advantages of GBTG over manual test generation. First, grammar generated test cases consistently cover more security axioms. Second,

the grammar used to generate test cases is much easier to create and maintain.

8.2.3 Firewalls

Sobotkiewicz [42] presented a case study on using GBTG to test firewall responses to Transmission Control Protocol (TCP) packets with invalid flag combinations. A TCP packet contains 6 flags, each of which can either be set or unset. The number of flag combinations is therefore $2^6 = 64$, of which 46 are invalid. A TCP packet with an invalid flag combination is considered as a bad packet. The test was configured such that bad packets were interleaved with legitimate packets at the connection setup and teardown phases. In total, 7 bad packet insertion points were identified.

The test was conducted by using the offline approach. First, a grammar was used to generate a set of connection descriptions and log them to a file. Each connection description is a sequence of flag combinations that are used in the connection setup and teardown phases. Second, a packet generator reads the file and assembles TCP packets accordingly. Finally, the packets are sent to the DUT and the responses from the DUT are captured.

Because there are 7 bad packet insertion points and each insertion point can contain any one of the 46 invalid flag combinations, the size of the test space is 46^7 which is about 435 billion. According to the author, this translates to about 138 years of test execution time. To reduce the test space size, covering arrays of two different specifications were applied. The first covering array was a 1-cover over all 7 bad packet insertion points. This reduced the size to 46. The second covering array was a partial 2-cover over 3 bad packet insertion points because a full 2-cover over all 7 bad packet insertion points took too long to generate. The second covering array reduced the size to $2,116$.

Traditionally, covering arrays and GBTG are applied to different testing problems. Covering arrays are used for tests with a list of parameters, each parameter can take a set of values. GBTG is used for tests with nested structure such as programming languages. This case study demonstrates the advantages of combining covering arrays with GBTG, that is, the test implementation is easy to understand and the test space can be easily reduced.

8.2.4 XML Processors

Wang [44] conducted two case studies to evaluate the performance of two XML processors: PyXML [4] and Saxon [8]. The first case study involves measuring the time it takes to run an XPath [11] query against a number of grammar-generated XML documents. The XPath query used has the expression `//y/text()` which searchs the entire document tree for an element named `y` and returns its element text. Each XML document generated has one element named `y` and the rest of the elements named `x`. The `y` element is placed on the bottom right of the document tree to maximize the search time.

Three grammars were used to create XML documents of three different structures, one for each structure. The structures that were tested are termed tall tree, wide tree, and complete tree. In a tall tree, every element—except for the leaf element—has only one child. In a wide tree, every element—except for the root element—is a child of the root. In a complete tree, an element at depth D has up to 2^{D+1} children. Running the XPath query against the three different structures shows that PyXML has better performance than Saxon when the XML documents being searched have less than 250 elements. On the contrary, Saxon has better performance than PyXML when the XML documents being searched have more than 250 elements.

This case study shows that GBTG is capable of generating XML documents of various structures. However, generating documents of specific sizes has proven difficult for GBTG. For example, GBTG cannot easily generate a document of size N without also generating all documents of size 1 through $N - 1$.

The second case study is similar to the first except that the queries and the documents are extracted from XMark [40]. The purpose of XMark is to provide a benchmark tool for evaluating the performance of XML processors. To achieve that, it provides an XML document generator `xmlgen` and 20 XQuery queries. `xmlgen` generates XML documents that are similar in structure to real world online auction databases. It takes a scaling factor as input and generates XML documents of the corresponding size. The 20 XQuery queries are used to test XML processors' ability to handle full text search, ordered access, and chasing references.

One of the drawbacks of XMark is the complexity of its implementation. First, it contains $7,479$ lines of C code of which $5,805$ are string constants: a set of real person names and emails that are used for generating buyers and sellers. Second, it is able to randomize the number of children and references that each element has.

Both of these are useful for approximating real world online auction but have limited value in terms of evaluating the performance of XML processors. As a result, Wang introduced a reduced version of the online auction databases which can be easily generated by using four grammars. The reduced version is a lot simpler in structure but still has the ability to test full text search, ordered access, and chasing references.

This case study shows that GBTG can be used to simplify XML document generation, given that there are no complex requirements on the structure of the document trees. The resulting implementation is not only a lot shorter but also easier to understand and maintain.

Following Wang's work, Ly-Gagnon [29] used GBTG to test four XPath interpreters: Libxml2 [47], Lxml [3], PyXML, and VTD-XML [46]. The test objective was to discover the resource limitations for each of the XPath interpreters. To test that, XML documents of extreme structures were created by two grammars. The structures include XML documents with long element text, XML documents with deeply nested elements, and XML documents with many sibling elements. The test result shows that VTD-XML fails when the element text exceeds $94,332,507$ characters, PyXML fails when the level of nesting exceeds 993, and LibXML and Lxml fail when the level of nesting exceeds 1025. Additional failures on combinations of extreme structures were also reported.

The distinctive feature of Ly-Gagnon's work is that the tests can be manipulated and executed without any programming knowledge. GBTG has not seen widespread adoption because one has to have programming skills in order to use it. However, testers who have in depth understanding of the code under test may not have any programming knowledge. As a result, the power of GBTG is unavailable to many testers. To address this problem, the author developed a GBTG tool called Dervish that facilitates grammar manipulation and execution through a graphical user interface (GUI). The idea was that testing by GBTG can be divided into two subtasks: grammar development and test execution. Grammar developers are responsible for the former task while testers are responsible for the latter. With the help of Dervish, testers alone can generate and execute test cases once the grammars are created.

8.3 Covering Arrays

Grindal et al. [19] conducted a survey on using combinatorial testing strategies to reduce the number of test cases. Over 30 papers published from 1985 to 2003 were

included in the survey. Among them is the work on orthogonal arrays which were introduced by Mandl [31]. Given a code under test with n input parameters and each parameter with d distinct values, an orthogonal array of strength t has two properties. First, it must contain all possible combinations of t input parameters. Second, each combination of t input parameters must appear the same number of times. Williams [45] argued that the second property is not necessary for typical testing scenarios. Therefore, he introduced covering arrays which have only the first property of orthogonal arrays.

However, Williams' work is applicable only when the code under test has same number of values for all input parameters. This limits the applicability of covering arrays because typical code under test has variable number of values for its input parameters. As a result, Moura et al. [36] proposed an algorithm for generalizing covering arrays to support variable size input parameters. The algorithm is able to generate covering arrays of strength 2 and 3. Because testing with covering arrays of strength 2 is commonplace, it has a special term called pair-wise testing.

Cohen et al. [17, 16] built a tool called Automatic Efficient Test Generator (AETG) to support pair-wise testing. Tai and Lei [43] also built a tool called PairTest that has the same functionality as AETG but using a different algorithm called in-parameter-order (IPO) for generating covering arrays. Five years later, an improved version of PairTest called FireEye was introduced by Lei et al. [28]. In contrast to PairTest which supports only pair-wise testing, FireEye is able to generate covering arrays of greater strength.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

GBTG has been applied to many testing problems since its introduction over forty years ago. Originally applied to compiler testing, GBTG is now used for testing VLSI circuits, Java virtual machines, firewalls, and XML processors. While these testing applications have demonstrated the power of GBTG, questions about certain aspects of GBTG still remain, preventing it from being applied to more testing domains. As a result, further research into these questions is imperative. The objective of this thesis is to address these research questions presented in Section 1.5. The following describes the approach taken to answer each of these research questions and the results of our research:

1. *How can we eliminate the combinations of less importance generated by a grammar?* Covering arrays are often used for reducing the test space size of a parameterized test. With covering arrays, testers can easily adjust the number of test cases by either increasing or decreasing the coverage strength. Because of their usefulness, covering arrays are incorporated into our experimental framework with the `cov` tag. By applying a `cov` tag to a rule, the grammar author can reduce the number of sentential forms generated by the rule and thus avoid combinatorial explosion.

2. *How can we provide a mechanism for creating a grammar that generates combinations of correct and error values?* Through RSS tests, we introduced the template-probe paradigm that facilitates insertion of safe and unsafe HTML elements and attributes into RSS element text. With the template-probe paradigm,

a grammar is divided into template and probes portions. The template portion generates feed templates that conform to the RSS specifications while the probes portion generates combinations of safe and unsafe HTML elements and attributes. Dividing a grammar into template and probes portions allows the grammar author to focus on one task at a time. Also, it is easy to move a probe into a different location whenever the test specifications change. Moreover, the combinations of probe values can be adjusted easily by the use of `cov` tags.

3. *How can we apply GBTG in the automatic generation of GUI playback scripts?* We used Selenium as a GUI playback tool for testing a web application called Code Activator (CA). To test CA, a grammar was created that generates combinations of test parameters while the grammar's `postcode` translates the test parameters into Selenium commands. Running the grammar generates the Selenium test scripts. The test results show that applying GBTG in the automatic generation of GUI playback scripts has a number of advantages. First, the need to manually create combinations of test parameters is eliminated. Second, the grammar is easy to read and maintain. Third, the number of test cases generated can be easily adjusted by the use of `cov` tags.

4. *How can we help the grammar author to visualize the language generation process in a complex grammar?* To our knowledge, generation trees are the best tool to visualize the language generation process. Therefore, we incorporated logging into the language generation process so that the resulting logs can be used by Dervish for automatic drawing of generation trees. Through Dervish, the grammar author can zoom in on the generation subtree of interest in order to understand (1) why so many strings are generated and (2) how each string is derived.

5. *How can we apply GBTG to some testing scenarios and at the same time demonstrate the usefulness of the extra-grammatical features?* We developed an experimental GBTG framework called YouGen_NG that incorporates some of the most useful extra-grammatical features found in the GBTG literature. Through RSS and CA tests, we were able to show that GBTG test scripts are easy to read and maintain and the extra-grammatical features are essential for restricting the language size of certain grammars and formatting the test inputs.

9.2 Future Work

9.2.1 n-error

Some testing scenarios involve testing an application's ability to handle invalid inputs. For example, a typical firewall inspects the header of every packet that it receives. A packet is dropped if the firewall discovers just one single error and forwarded otherwise. It is obvious that packets with lots of errors are less likely to successfully pass through a firewall than packets with few errors because the chance of discovering one error within the former is significantly higher than the latter. In order to test a firewall's error handling ability, testers are required to generate packets with n errors each where n, in most cases, equals to 1. With GBTG, limiting the number of errors to a fixed number is difficult to achieve using context-free grammars. Even with `precode` and `postcode` blocks, the problem is not any easier because correctly deriving the number of errors generated is largely dependent on the order in which the embedded code blocks are invoked. As a result, YouGen_NG support for generating n-error is preferred.

9.2.2 Variables

It is common in GBTG that the latter part of a string is dependent on the former part. Such is the case for generating Internet Protocol (IP) packets. For example, the amount of data that can be encapsulated in an IP packet depends on the Total Length field in the packet header. A grammar for generating IP packets can use a variable to save the value of the Total Length generated and later apply it to generate data of appropriate length. As with n-error, YouGen_NG support for variables is preferred.

Bibliography

[1] *The Atom Syndication Format*. http://www.ietf.org/rfc/rfc4287.txt.

[2] *HTML 4.01 Specification*. http://www.w3.org/TR/html4/.

[3] *Lxml*. http://codespeak.net/lxml/.

[4] *Python and XML Processing*. http://pyxml.sourceforge.net/topics/.

[5] *RDF Site Summary (RSS) 1.0*. http://web.resource.org/rss/1.0/spec.

[6] *RSS 2.0 Specification*. http://cyber.law.harvard.edu/rss/rss.html.

[7] *RssReader*. http://www.rssreader.com/.

[8] *Saxon: the XSLT and XQuery Processor*. http://saxon.sourceforge.net/.

[9] *Universal Feed Parser*. http://www.feedparser.org/.

[10] *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. http://www.w3.org/TR/xhtml1/.

[11] *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/xpath/.

[12] Robert Auger and Caleb Sima, editors. *Zero Day Subscriptions: Using RSS and Atom Feeds As Attack Delivery Systems*. SPI Dynamics, 2006. www.cgisecurity.com/papers/RSS-Security.ppt.

[13] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[14] A. Celentano, S. Crespi Reghizzi, P. Della Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*.

[15] R.N. Charette. Why software fails [software failure]. *Spectrum, IEEE*, 42(9):42–49, sept. 2005.

[16] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437 –444, jul 1997.

[17] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *Software, IEEE*, 13(5):83 –88, sep 1996.

[18] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 170–178, Piscataway, NJ, USA, 1981. IEEE Press.

[19] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15:167–199, 2005.

[20] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[21] D. Hoffman, L. Sobotkiewicz, Hong-Yi Wang, P. Strooper, G. Bazdell, and B. Stevens. Test generation with context free grammars and covering arrays. In *Testing: Academic and Industrial Conference - Practice and Research Techniques, 2009. TAIC PART '09.*, pages 43 –47, 4-6 2009.

[22] Daniel Hoffman, Chien Chang, Gary Bazdell, Brett Stevens, and Kevin Yoo. Bad pairs in software testing. In Leonardo Bottaci and Gordon Fraser, editors, *Testing Practice and Research Techniques*, volume 6303 of *Lecture Notes in Computer Science*, pages 39–55. Springer Berlin / Heidelberg, 2010.

[23] Daniel Malcolm Hoffman, Ming Lu, and Tim Pelton. A web-based generation and delivery system for active code reading. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 483–488, New York, NY, USA, 2011. ACM.

[24] Daniel Malcolm Hoffman, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang. Grammar-based test generation with yougen. *Softw. Pract. Exper.*, 41(4):427–447, April 2011.

[25] William Homer and Richard Schooler. Independent testing of compiler phases using a test case generator. *Software: Practice and Experience*, 19(1):53–62, 1989.

[26] Sauce Labs. *Selenium HQ: Web application testing system.* http://seleniumhq.org/.

[27] R. Lammel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proceedings of 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006)*, pages 19–38. Springer-Verlag, LNCS 3964, 2006.

[28] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. IPOG: a general strategy for t-way software testing. In *Proceedings of the 14th Annual International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 549–556, 2007.

[29] David Ly-Gagnon. Dervish: A new gui for grammar-based test generation. Master's thesis, University of Victoria, 2010.

[30] Brian A. Malloy. An interpretation of purdoms algorithm for automatic generation of test cases. In *In 1st Annual International Conference on Computer and Information Science*, pages 3–5, 2001.

[31] Robert Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Commun. ACM*, 28(10):1054–1058, 1985.

[32] Peter M. Maurer. Design verification of the we 32106 math accelerator unit. *IEEE Des. Test*, 5:11–21, May 1988.

[33] Peter M. Maurer. The design and implementation of a grammar-based data generator. *Softw. Pract. Exper.*, 22(3):223–244, 1992.

[34] P.M. Maurer. Generating test data with enhanced context-free grammars. *Software, IEEE*, 7(4):50 –55, jul 1990.

[35] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[36] L. Moura, J. Stardom, B. Stevens, and A. Williams. Covering arrays with mixed alphabet sizes. *Journal of Combinatorial Designs*, 11(6):413–432, 2003.

[37] V. Murali and R. K. Shyamasundar. A sentence generator for a compiler for pt, a pascal subset. *Software: Practice and Experience*, 13(9):857–869, 1983.

[38] A. J. Payne. A formalised technique for expressing compiler exercisers. *SIGPLAN Not.*, 13(1):59–69, 1978.

[39] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12:366–375, 1972. 10.1007/BF01932308.

[40] A. R. Schmidt, Florian Waas, Martin L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The xml benchmark project. Technical report, Amsterdam, The Netherlands, The Netherlands, 2001.

[41] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *PLAN '99: Proceedings of the 2nd conference on Domain-specific languages*, pages 1–13, New York, NY, USA, 1999. ACM.

[42] Lewis Sobotkiewicz. A new tool for grammar-based test case generation. Master's thesis, University of Victoria, 2008.

[43] K.C. Tai and Y. Lie. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002.

[44] Hong-Yi Wang. Grammar-based test generation for xpath queries. University of Victoria, 2008. Master's report.

[45] Alan W. Williams. Determination of test configurations for pair-wise interaction coverage. In *TestCom '00: Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems*, pages 59–74, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[46] XimpleWare. *VTD-XML: The future of XML processing.* http://vtd-xml.sourceforge.net.

[47] XMLSoft. *The XML C parser and toolkit of Gnome: Introduction.* http://www.xmlsoft.org/intro.html.

[48] V.V Zaytsev. Combinatorial test set generation: Concepts, implementation, case study. Master's thesis, Universiteit Twente, 2004.