Dervish: A New GUI for Grammar-Based Test Generation

by

David Ly-Gagnon
B.Eng., McGill University, Montreal, Canada, 2008

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Dervish: A New GUI for Grammar-Based Test Generation

by

David Ly-Gagnon
B.Eng., McGill University, Montreal, Canada, 2008

Supervisory Committee

Dr. Daniel M. Hoffman, Co-Supervisor
(Department of Computer Science)

Dr. Micaela Serra, Co-Supervisor
(Department of Computer Science)

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

Supervisory Committee

Dr. Daniel M. Hoffman, Co-Supervisor
(Department of Computer Science)

Dr. Micaela Serra, Co-Supervisor
(Department of Computer Science)

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

Abstract

Because software testing is a repetitive and time-intensive task, a practical solution is to turn to automation. Test automation, however, requires programming skills. Testers, who typically know a lot about the application under test, often do not have the programming skills to automate the testing effort. Grammar-based test generation (GBTG) uses context-free grammars to generate strings in the language described by the grammar. Given a grammar, a GBTG algorithm can produce test cases for the application under test. Since testers typically have little programming skills and are not likely to develop the grammars needed for practical testing, the power of GBTG is unavailable to many testing practitioners.

To help address this problem, we have developed Dervish, a graphical user interface which allows testers to use the power of GBTG. Our new tool allows testers to modify parts of a grammar, generate test cases, and visualize generation trees. To demonstrate the benefits of Dervish, we present the results of three case studies.

Table of Contents

List of Figures

Acknowledgements

I owe my deepest gratitude to my supervisors, Dan Hoffman and Micaela Serra, whose encouragement, guidance and support helped me throughout this thesis. It is with their encouragement and effort that I have been able to complete this thesis. I would also like to thank my lab mates for keeping me company and making the lab an enjoyable place to work. Finally, I would like to thank Hoi Ying for leaving Montreal and taking this journey with me, and my parents for their continual support and advice.

Chapter 1
Introduction

For many years, software developers have been successful at creating a variety of software products. For example, while the years behind us are filled with success stories, they are also filled with a long history of software failures. In the past two decades, many organizations have been victims of software failures. The *Software Hall of Shame* [3], a listing of companies who have reported important software failures from 1992 to 2005, shows that software projects are not always a success. Among the most common factors of failure in software projects are poor development practices, use of immature and untested technology, and commercial pressure.

1.1 Software Testing

Introducing software testing early in the development process can have a considerable impact on the success of a project. For instance, detecting errors in the design phase may prevent the same errors from appearing in the implementation phase. While software testing may have a positive impact on a project, it also comes at a price. Software testing is known to be a time-intensive task. Since organizations usually have deadline, resource, and budget constraints, they must turn to software testing practices which can adequately test their products within these constraints.

1.2 Test Automation

Faced with short deadlines and the pressure to do more with less, organizations consider automation. Automating tests can provide several benefits. Dustin et al. [5] have identified significant benefits of automating the testing effort. For instance, some of the benefits observed are that test automation can improve some of the tasks of testing such as test development, test execution, and analysis of test results. Typically, automation requires programming skills. For example, creating scripts to automate

the execution of test cases, and the comparison of actual and expected output are often difficult tasks. Testers, who usually know a lot about the application under test, often do not have the programming skills or knowledge to automate the testing process. Nonetheless, testers play an important role in the testing process because they usually know the test cases that are likely to expose errors in the application under test.

1.3 Grammar-Based Test Generation

If testers were provided with an approach to automate the generation of test cases which did not require programming skills, they could use automation more effectively to create the test cases that are likely to reveal errors in the application under test. Grammar-based test generation (GBTG) is an approach to test generation. As opposed to using context-free grammars to determine if a string is in the language accepted by the grammar, GBTG uses context-free grammars to generate strings in that language. A context-free grammar specifies the syntax of a language. For example, a grammar may specify the syntax of the input to an application under test. Using a grammar, a generator can produce test cases for the application under test. Many tools have been designed and implemented to demonstrate the effectiveness of GBTG [1, 6, 22]. Although the tools have helped testers and researchers learn about GBTG, most of the time the tools are difficult to use except by the tool developers. In addition, creating complex grammars requires programming skills and understanding the language of a grammar may be a challenging task for the testers.

1.4 Application Domains of GBTG

For more than 30 years, we have seen extensive use of GBTG in many application domains. The first domain in which GBTG has been used is compiler testing. In 1970, Hanford used a grammar to generate test cases for a PL/1 compiler [6]. His work is the earliest known application of context-free grammars to testing. Hanford inspired other researchers, such as Bird and Munoz, who later also applied GBTG to test compiler programs, graphical output applications and sort/merge programs [1].

A major improvement to the work of Hanford is that they provided a mechanism to determine whether a test case has passed or failed by predicting its execution and comparing its execution with the predicted one at run time. Years later, Sirer applied GBTG to the testing of Java Virtual Machine implementations [22]. Their work describe *lava*, a language for specifying grammars which can be used to construct complex test cases for testing JVMs. As a result of their work, faults were found in the Sun JDK1.0.2 and Microsoft Java virtual machine implementations.

Compiler testing is not the only domain in which GBTG has been applied. Payne generated test programs using grammars for the testing of overload conditions in real time systems [20]. Duncan described how complex test cases could be generated from a grammar for the testing of a text processing application [4]. Kaksonen developed PROTOS, a tool to systematically test implementations of network protocols using attribute grammars which model input syntax [13]. GBTG has also been used in firewall testing [10, 23] and web testing [28].

More recently, GBTG has been applied to the testing of XML and XPath applications. Hoffman et al. applied GBTG in the testing of RSS clients for HTML injection vulnerabilities [9]. Wang used GBTG to generate large XML documents and corresponding XPath queries [25] and MacNamara used GBTG in the testing of XPath interpreters [17].

1.5 Dervish: A New GUI for GBTG Tools

Our work focuses on providing support for testers so that they may use and understand GBTG. To achieve this, we have designed and implemented Dervish, a graphical user interface (GUI), which allows testers to interact with YouGen [23], a GBTG tool. First, Dervish provides testers with the ability to use GBTG tools without programming skills. Dervish allows testers to modify parts of a grammar and invoke YouGen to generate test cases through a GUI. Dervish also provides help in understanding complex grammars. By displaying generation trees, which are a useful visual representation of derivations in a grammar, Dervish helps testers understand how test cases are generated.

In this work, we identify three roles in the testing effort:

- The *tool developers* are experienced with GBTG and its application in different domains. They are responsible for creating application programming interfaces (API's) or frameworks for using GBTG tools.

- *The grammar developers* are responsible for using the API or framework and applying GBTG to a specific domain. For instance, they may choose to use a GBTG tool to test network protocols. They are also in charge of creating the grammars which will be used by the testers to generate the test cases.

- The *testers* are in control of the test execution. They usually know a lot about the application under test and the test cases which are likely to reveal errors, but they do not necessarily have programming skills or knowledge to automate the testing effort.

1.6 Case Studies

To observe the benefits of Dervish, we have conducted three case studies. The first case study focuses on showing how to use Dervish to support the learning of GBTG. In this case study, we present three simple grammars and show how testers may interact with Dervish to learn about GBTG. The second case study focuses on showing that Dervish may help testers understanding complex grammars. In this case study, we present examples of complex grammars which generate a large number of XML documents and show how Dervish can help testers in understanding the language of a grammar by exploring generation trees. Finally, the third case study focuses on using Dervish and applying GBTG to the testing of XPath interpreters. In this case study, we are interested in observing whether testers can use Dervish to modify parts of a grammar to generate test cases which may reveal errors in the XPath interpreters.

1.7 Thesis Organization

Chapter 2 provides the reader with background material on GBTG and YouGen. Chapter 3 presents Dervish and its main features. Chapter 4 describes the implementation of Dervish. Chapter 5 and 6 discusses the application of Dervish as a tool to support the learning of GBTG and the understanding of complex grammars. Chapter 7 applies Dervish to the testing of XPath interpreters. Chapter 8 presents an overview of the related work in the field of test generation. Finally, Chapter 9 ends with the conclusions of this work and give suggestions for future work based on the proposed solutions and the current results obtained.

Chapter 2
Background on GBTG and YouGen

This chapter introduces the background material of Dervish. In particular, a review of GBTG terminology and concepts is presented as well as a description of the YouGen tool and features.

2.1 Context Free Grammars

Natural languages and programming languages consist of words. For example, Python contains words such as *x*, =, *, +, or *1*. The list of words form the vocabulary of a language. When words are combined, they can create syntactically valid sentences. For example, a syntactically valid Python statement is

x = x + 1

However, a syntactically invalid Python statement is

x = x + 1*

The role of context-free grammars is to describe languages. A common use of context-free grammars is to specify which sentences are syntactically valid or invalid.

2.1.1 Nonterminals, Terminals, and Rules

A context-free grammar is defined as $G = \langle N, T, R, S \rangle$ where $S \in N$.

- N is a finite set of non-terminal symbols.
- T is a finite set of terminal symbols.
- R is a finite set of rules where each rule consists of a left-hand side, the symbol `::=`, and a right-hand side.
- S is the start symbol.

A rule's left-hand side is in N. A rule's right-hand side consists of terminal or non-terminal symbols in N or T. A non-terminal symbol may be replaced with a

```
T ::= A B;

A ::= '0';
A ::= '1';

B ::= '0';
B ::= '1';
```

Figure 2.1: `TwoBit` grammar

rule's right-hand side if the symbol and the rule's left-hand side are identical. A terminal is a literal and cannot be replaced with a rule's right-hand side.

Figure 2.1 shows an example of the `TwoBit` grammar, where:

- $N = \{$`T`, `A`, `B`$\}$
- $T = \{$`'0'`, `'1'`$\}$
- $R = \{$`T ::= A B`, `A ::= '0'`, `A ::= '1'`, `B ::= '0'`, `B ::= '1'`$\}$
- `T` is the start symbol, S

2.1.2 Derivation

A derivation results in a string in the language being generated. A derivation consists of derivation steps. It begins with the start symbol and ends with only terminal symbols. A derivation may be characterized as a leftmost or a rightmost derivation.

2.1.2.1 Derivation Step

A derivation step is one of the steps in a derivation. A derivation step replaces a non-terminal with a rule's right-hand side.

2.1.2.2 Sentential Form

A sentential form is a sequence of non-terminals and terminals derived in zero or more derivation steps from the start symbol of the grammar. A *ground* sentential

```
T  ⇒  A B
   ⇒  0 B
   ⇒  0 0
```

Figure 2.2: Leftmost derivation of string `0 0`

```
T  ⇒  A B
   ⇒  A 0
   ⇒  0 0
```

Figure 2.3: Rightmost derivation of string `0 0`

form is a sentential form that contains only terminals and represents a string in the language.

2.1.2.3 Leftmost Derivation

A leftmost derivation chooses the leftmost non-terminal for replacement. For example, Figure 2.2 shows a leftmost derivation for the `TwoBit` grammar from Figure 2.1.

2.1.2.4 Rightmost Derivation

A rightmost derivation chooses the rightmost non-terminal for replacement. For example, Figure 2.3 shows a rightmost derivation for the `TwoBit` grammar from Figure 2.1.

2.1.3 Language of a Grammar

The language of a grammar G, $L(G)$, is the set of all strings generated by this grammar. For example, Figure 2.4 shows the language generated by the `TwoBit` grammar from Figure 2.1.

2.1.4 Recursive Grammar

A recursive grammar generates an infinite number of derivations. Figure 2.5 shows an example of a recursive grammar. Nonterminal `Zeros` appears on both the right-

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

Figure 2.4: The language of the `TwoBit` grammar

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

Figure 2.5: `Zeros` grammar

hand side and the left-hand side of the second rule, which makes this grammar recursive.

2.2 String Generation Using Grammars and YouGen

Parsing is the process used to determine if a string is in a language. This process can be reversed. From a context-free grammar, it is possible to generate strings in the language described by the grammar.

YouGen takes a grammar *G* and a non-terminal *N*, and generates *L(G,N)*: the set of all strings which can be derived from *N* using the rules of *G*. At each derivation step, two important decisions are taken into consideration:

- In a sentential form with more than one non-terminal, which one is selected for replacement?

  YouGen selects the leftmost non-terminal for replacement. For example, Figure 2.2 shows the derivation steps of the string with 2 zeros from the `TwoBit` grammar in Figure 2.1. First, non-terminal `T` is chosen and replaced with non-terminals `A B`. Next, non-terminal `A` is chosen for replacement since it is the leftmost non-terminal appearing in the sentential form. Nonterminal `A` is replaced with terminal `0`. Next, non-terminal `B` is chosen for replacement. It is replaced with terminal `0` which leads to the derivation of the string `0 0`.

- Which rule is selected when two or more rules can be used for replacement?

The first rule selected is the one appearing first in the grammar from top to bottom. For example, Figure 2.1 shows two rules with non-terminal `A` on the left-hand side. YouGen first selects `A ::= '0'` and then `A ::= '1'`.

2.3 Generation Trees

A generation tree is a useful representation of derivations in a grammar. In a generation tree, each node is a sentential form. The root node contains the start symbol and each leaf node contains a string in the language of the grammar. Figure 2.6 shows the generation tree for the `TwoBit` grammar from Figure 2.1.

A generation tree is visited in depth first traversal order. Each path from the root to a leaf corresponds to a derivation. Each arc is annotated with a rule identifier to specify which rule alternative is used in the derivation step associated with this arc. A rule identifier is constructed by taking a rule's left-hand side and adding an integer. For example, reading the `TwoBit` grammar in Figure 2.1 from top to bottom, the first rule which starts with non-terminal A has rule identifier `A0`, the second rule which starts with non-terminal A has rule identifier A1, and so on. A generation tree is different from a parse tree because a generation tree displays multiple derivations in a single tree. A parse tree displays the derivation of a single string in a grammar.

2.4 XML Generation Trees

When YouGen generates the language of a grammar, it also logs the generation tree in an XML-based file format. Figure 2.7 shows the XML generation tree structure of the `TwoBit` grammar from Figure 2.1. The structure of an XML generation tree consists of `<gentree>` elements. The children of a `<gentree>` are `<id>`, `<s>`, zero or more nested `<gentree>`, and a `<count>`. The element text `<id>` is a rule identifier specifying which rule alternative has been used to get from the parent sentential form to the child sentential form associated with this rule. For example, on line 5 in Figure 2.7 rule identifier `T0` has been used to get from sentential form `[T]` (line 3) to sentential form `[[A,B]]` (line 6). The element text `<s>` is a sentential form and represents the parse tree using nested lists. The element text `<count>` specifies

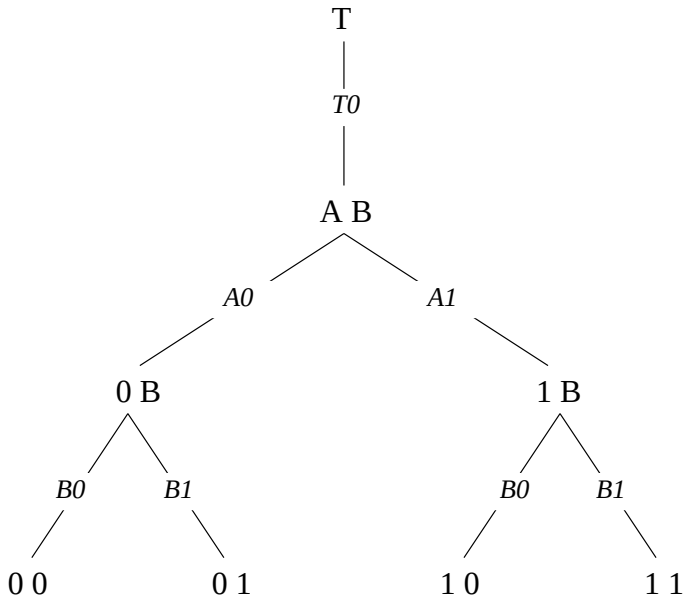

Figure 2.6: Generation tree of `TwoBit` grammar in Figure 2.1

the number of ground sentential forms in the subtree rooted at the `<gentree>` node containing the element `<count>`. A leaf node has element text `<count>` set to one since a leaf node contains a string in the language. For instance, `<gentree>` node from line 10 to 14 has element text `<count>` set to one and similarly for `<gentree>` node from line 15 to 19.

2.5 Count Trees

Although XML generation trees are useful at representing the structure of a generation tree, they are usually difficult to read. First, XML documents consist of markup and content. Strings which constitute markup element either starts with '`<`' and end with '`>`', or starts with '`&`' and end with '`;`'. XML generation trees contains several markup elements such `<gentree>`, `<id>`, `<s>` and `<count>` which makes it challenging to read. In addition, as a grammar gets larger, the XML generation tree produced also gets larger. Therefore, we introduce count trees which help visualizing generation trees. A count tree shows the same information as a generation tree, except that counts are explicitly shown. A node in a count tree consists of a rule identifier,

```
<gentree>
   <id>None</id>
   <s>[T]</s>
   <gentree>
      <id>T0</id>
      <s>[[A,B]]</s>
      <gentree>
         <id>A0</id>
         <s>[[['0'],B]]</s>
         <gentree>
            <id>B0</id>
            <s>[[['0'],['0']]]</s>
            <count>1</count>
         </gentree>
         <gentree>
            <id>B1</id>
            <s>[[['0'],['1']]]</s>
            <count>1</count>
         </gentree>
         <count>2</count>
      </gentree>
      <gentree>
         <id>A1</id>
         <s>[[['1'],B]]</s>
         <gentree>
            <id>B0</id>
            <s>[[['1'],['0']]]</s>
            <count>1</count>
         </gentree>
         <gentree>
            <id>B1</id>
            <s>[[['1'],['1']]]</s>
            <count>1</count>
         </gentree>
         <count>2</count>
      </gentree>
      <count>4</count>
   </gentree>
   <count>4</count>
</gentree>
```

Figure 2.7: XML generation tree of `TwoBit` grammar

```
None:4:[T]
        T0:4:[[A, B]]
                A0:2:[[['0'], B]]
                        B0:1:[[['0'], ['0']]]
                        B1:1:[[['0'], ['1']]]
                A1:2:[[['1'], B]]
                        B0:1:[[['1'], ['0']]]
                        B1:1:[[['1'], ['1']]]
```

(a) Count tree from root to bottommost level

```
T0:4:[[A, B]]
        A0:2:[[['0'], B]]
        A1:2:[[['1'], B]]
```

(b) Count tree from `T0` to depth 1

Figure 2.8: Example of count trees for `TwoBit` grammar

a count, and a sentential form, each separated by a colon. Figure 2.8 shows two examples of count trees for the `TwoBit` grammar from Figure 2.1. Each line consists of one node, where the node at the top of the tree is the root. Figure 2.8(a) shows the count tree from the root to the bottommost level. Figure 2.8(b) shows the count tree from `T0` to depth 1.

2.6 Tags

Tags are added to a grammar to reduce the size of the language generated by the grammar. YouGen supports several tags and allow zero or more tags to be attached to each grammar rule. Each tag has the syntax: {*tagName tagParameters*}.

2.6.1 Count Tag

When tag {`count C`} is associated with rule R for non-terminal N, YouGen does not allow more than C strings to be derived using R for non-terminal N. Figure 2.9(a) shows an example of the Zeros grammar with the `count` tag. From the first invocation of the tagged rule `Zeros`, no more than 2 strings can be generated so that it does not exceed the `count` tag limit of 2.

```
Zeros ::= '0';
{count 2}  Zeros ::= '0' Zeros;
```

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

```
Zeros ::= '0';
{rdepth 2}  Zeros ::= '0' Zeros;
```

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

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

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

Figure 2.9: `Zeros` grammars with tags

2.6.2 Recursive Depth Tag

When tag {`rdepth D`} is associated with rule R, then YouGen does not allow R to be applied more than D times on any path in the parse tree.

Figure 2.9(b) shows an example of applying the `rdepth` tag to second rule `Zeros`. Figure 2.10 shows the parse tree for 3 zeros. Since the second rule `Zeros` is applied at most 2 times on any path in the parse tree, which does not exceed the limit of 2 set by the `rdepth` tag, 3 zeros can be generated.

2.6.3 Depth Tag

When tag {`depth D`} is associated with rule R for non-terminal N, YouGen limits the paths generated by R in the parse tree below N to depth D.

Figure 2.9(c) shows a grammar applying the `depth` tag to the second rule `Zeros`. Figure 2.10 shows the parse tree for 3 zeros. Since the parse tree contains no more than 3 paths below non-terminal `Zeros`, which does not exceed the limit of 3 set by the `depth` tag, 3 zeros can be generated.

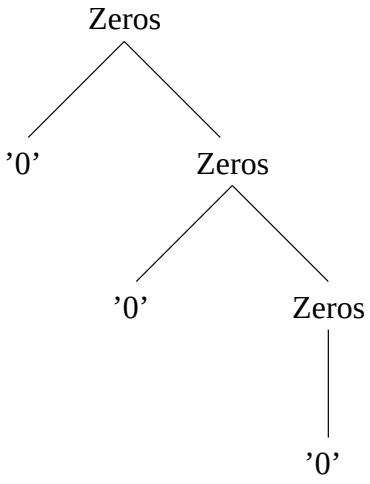

Figure 2.10: Parse tree of 3 zeros for the `Zeros` grammar

2.6.4 Covering-Array Tag

When a `cov` tag is associated with a rule, a covering-array algorithm is used in the generation strategy used by YouGen. A covering-array generates a subset of the cross product of the parameters given as inputs. The cross product of the input parameters is defined as the test space. When the test space is large, a lot of test cases are generated. Therefore, to reduce the number of test cases, a covering-array may be used.

The `cov` tag syntax is {`cov [`P_1`,...,`P_n`]`}, where each P_i is a coverage specification. Each coverage specification consists of an index set and a strength. An index set is a subset of $[0..N-1]$, where N denotes the number of parameters. The parameters consist of the non-terminals and terminals appearing on the right-hand side of the rule associated with a `cov` tag.

For example, Figure 2.11 shows an example of the `Call` grammar and Figure 2.12 shows the language generated by the `Call` grammar. When the rule with non-terminal `Call` on the left-hand side is associated with the tag {`cov ([0,1,2],2)`}, YouGen generates the strings shown in italics. The `cov` tag has one coverage specification `([0,1,2],2)`. The index set is `[0,1,2]` and specifies that `CallerOS`, `ServerOS`, and `CalleeOS` are the parameters to the covering-array algorithm. The strength is 2 and specifies that the combinations of each of the three pairs of parameters

```
Call ::= CallerOS ServerOS CalleeOS;

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

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

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

Figure 2.11: `Call` grammar

	CallerOS	ServerOS	CalleeOS
1	*Mac*	*Lin*	*Mac*
2	Mac	Lin	Win
3	Mac	Sun	Mac
4	*Mac*	*Sun*	*Win*
5	Mac	Win	Mac
6	*Mac*	*Win*	*Win*
7	Win	Lin	Mac
8	*Win*	*Lin*	*Win*
9	*Win*	*Sun*	*Mac*
10	Win	Sun	Win
11	*Win*	*Win*	*Mac*
12	Win	Win	Win

Figure 2.12: Language of `Call` grammar

must be considered. For example, the cross product of `CallerOS` and `CalleeOS` is: $\{\langle \texttt{Mac}, \texttt{Mac}\rangle, \langle \texttt{Mac}, \texttt{Win}\rangle, \langle \texttt{Win}, \texttt{Mac}\rangle, \langle \texttt{Win}, \texttt{Win}\rangle\}$. The first pair appears in row 1 of Figure 2.12. The other pairs are found in row 4, 8, and 9. The same exercise must be executed for the pairs of parameters: `CallerOS` and `ServerOS`, and `ServerOS` and `CalleeOS`.

2.7 Embedded Code

2.7.1 Global Precode

The `global_precode` block is executed once before the generation begins. It often contains declarations and initializations of global variables, and system actions, such as opening files [8]. The syntax is: {`global_precode` *arbitrary_python_code_block*}

2.7.2 Global Postcode

The `global_postcode` block is executed once after generation ends. It is typically used for exit tasks, such as closing a file [8]. The syntax is: {`global_postcode` *arbitrary_python_code_block*}

2.7.3 Precode

The code which is executed just before invocation of a rule to which the precode is attached [8]. The syntax is: {`precode` *arbitrary_python_code_block*}

2.7.4 Postcode

The code which is executed as soon as a string derived from a rule is available [8]. The syntax is: {`postcode` *arbitrary_python_code_block*}

2.8 Generator Nonterminals

Generator non-terminals are used when a non-terminal must be used to specify a long sequence of terminal alternatives. For example, to create a grammar representing the *Roman* alphabet, 52 separate rules would be required to express the lowercase and uppercase letters of the alphabet. Instead, a generator non-terminal may be used as a shorthand to generate each letter of the alphabet.

Grammar developers can create their own generator non-terminals, but YouGen also provides three built-in generator non-terminals: `Range`, `List`, and `File`. The syntax and semantics are defined in Sobotkiewicz [23] and are revisited below.

```
G ::= Range(0,1,10);
```

Figure 2.13: Example of grammar using the `Range` generator nonterminal

```
G ::= List('hello','world','this','is','a','list','generator');
```

Figure 2.14: Example of grammar using the `List` generator nonterminal

2.8.1 Range

- Syntax: `Range(`*start, skip, count*`)`
- Semantics: Generates *count* integers from *start*, incrementing the value by *skip* after each derivation.
- Example: Figure 2.13 shows a grammar which generates the integers 0 to 9.

2.8.2 List

- Syntax: `List(`*item1, item2, ..., itemN*`)`

  Each item is a string enclosed in quotation marks.
- Semantics: Generates each item in the list from left to right.
- Example: Figure 2.14 shows a grammar which generates 7 words.

2.8.3 File

- Syntax: `File(`*filename*`)`

  filename is a path to a file which contains a list of strings separated by newlines.
- Semantics: Generates one string for each line in *filename*.

Chapter 3
Dervish: A New GUI for GBTG tools

This chapter introduces the new tool, Dervish, which allows testers to visualize generation trees and provides them with support to generate the test cases they need. Dervish consists of a text-based and a GUI-based version. The text-based version was implemented as a first attempt to display generation trees and its main features and functions were ported to a GUI-based version. The GUI-based version provides support for test case generation in addition to allowing testers to visualize generation trees. The features and functions of the text-based version are presented first, followed by the ones of the GUI-based version. The next chapter will describe the implementation of each version.

3.1 Text-Based Version

The core feature of the text-based version is to display generation trees. It has been designed as a command line interpreter that parses an XML generation tree, reads commands issued by the user, and displays count trees as output.

The text-based version is implemented using Python. To run the tool, type the following command in a terminal window.

```
./dervish.py <filename>
```

`dervish.py` is a configuration script specifying a path to the directory where Dervish is located and `<filename>` is an XML generation tree file.

The following subsections present the syntax and semantics of each command supported in the text-based version and provides some examples on how to invoke each command.

3.1.1 `explore` Command

- Syntax: `explore` [`-q`] [`-d` *n*] [*path*]

```
$:explore -q
None:4
        T0:4
                A0:2
                        B0:1
                        B1:1
                A1:2
                        B0:1
                        B1:1
```

Figure 3.1: Count tree of `TwoBit` grammar with no sentential forms displayed

n is a nonnegative integer and *path* is a list of rule ids, each separated by a slash, that appears in a generation tree path.

- Semantics: The `explore` command takes three optional parameters and allows users to display count trees from an XML generation tree. The first parameter is the quiet flag: `-q`. When supplied, sentential forms are not displayed in the count tree. The second parameter is the depth flag: `-d` followed by *n*. When supplied, *n* specifies the number of levels to be displayed in the tree. The third parameter is a *path* and represents the path to a node. Specifically, it shows the list of all its ancestors, plus the node name, separated by slashes. If the *path* is omitted, the path to the root is used as *path*. Node paths are not unique. If two or more nodes using the same rule id are at the same level in a count tree, `an index`, enclosed within square brackets, can be specified to select one of the rule alternatives.

- Example: `explore -q`

  Figure 3.1 shows the count tree for the Twobit grammar from Figure 2.1, which results from typing the command with the quiet flag: `-q`. The sentential form which would appear as third element at each node in the count tree is omitted.

- Example: `explore -d 2`

```
$:explore -d 2
None:4:[T]
        T0:4:[[A, B]]
                A0:2:[[['0'], B]]
                A1:2:[[['1'], B]]
```

Figure 3.2: Count tree of `TwoBit` grammar from root to depth 2

```
$:explore /T0/A0
A0:2:[[['0'], B]]
        B0:1:[[['0'], ['0']]]
        B1:1:[[['0'], ['1']]]
```

Figure 3.3: Count tree of `TwoBit` grammar from `A0` to bottommost level

Figure 3.2 shows a count tree for the Twobit grammar which results from typing the command with the depth flag: `-d`. Since the command is supplied with depth d set to 2, no more than 2 levels are displayed below the node specified in the *path* parameter. In this example, the *path* parameter is omitted, therefore the root is used as the initial level displayed in the count tree.

- Example: `explore /T0/A0`

  Figure 3.3 shows a count tree for the Twobit grammar which results from typing the command with a *path*. This command indicates that the node named `A0` can be found immediately below the node named `T0`, which is at toplevel.

3.1.2 `setindent` Command

- Syntax: `setindent` n

  n is a nonnegative integer.

- Semantics: The `setindent` command takes a nonnegative integer n and specifies the number of spaces of horizontal indentation between a parent node and each of its children.

- Example: `setindent 2`

```
$:setindent 2
$:explore
None:4:[T]
  T0:4:[[A, B]]
    A0:2:[[['0'], B]]
      B0:1:[[['0'], ['0']]]
      B1:1:[[['0'], ['1']]]
    A1:2:[[['1'], B]]
      B0:1:[[['1'], ['0']]]
      B1:1:[[['1'], ['1']]]
```

Figure 3.4: Count tree of `TwoBit` grammar with indentation n set to two spaces.

```
$:help

Documented commands (type help <topic>):
========================================
exit  explore  help  setindent
```

Figure 3.5: Example of typing the `help` command and the result returned

Figure 3.4 shows the count tree for the Twobit grammar which results from setting indentation n to two spaces.

3.1.3 `help` Command

- Syntax: `help` [*command*]

  command is the name of a command supported in Dervish.

- Semantics: The tool provides a `help` command which gives access to the commands available and their documentation.

- Example: `help`

  Figure 3.5 shows the result from typing the command with no arguments. The list of commands supported by the text-based version is displayed.

3.2 GUI-Based Version

The GUI-based version provides support for test cases generation and visualization of generation trees. Specifically, it allows testers to customize grammars by manipulating tags and generator non-terminals to generate the test cases they need. Also, the GUI-based version provides support to understand test case generation by allowing testers to explore generation trees.

Dervish's GUI is implemented using wxPython, a GUI toolkit for Python. Before running Dervish, the configuration script `dervish.py` must specify the path to the directory where YouGen is located. Then, Dervish can be run by typing the following command in a terminal window.

```
dervish.py gui
```

The following subsections first present an overview of the main window followed by the main features and functions of Dervish. The features and functions are divided into three subsections and are presented in the following order: Menu Features, Grammar Features, and Generation Tree Features.

3.2.1 Overview of The Main Window

Figure 3.6 shows the main window of Dervish applied to the grammar in Figure 2.5. The menu bar, located at the top of the window, provides access to functions such as opening or modifying a grammar file. The left panel shows grammar rules as read-only and tags as clickable underlined text. The right panel shows the generation tree and the bottom panel shows the language generated by the grammar. A status bar, located at the bottom of the window, provides feedback to the user when interacting with the application. In this example, the grammar generates three strings. Dervish displays each one of the strings on a separate line, in the order they were generated by YouGen.

3.2.2 Menu Features

This section presents the operations available in the menu bar. The menu bar places commands under three different menus: *File*, *Grammar*, and *Run*.

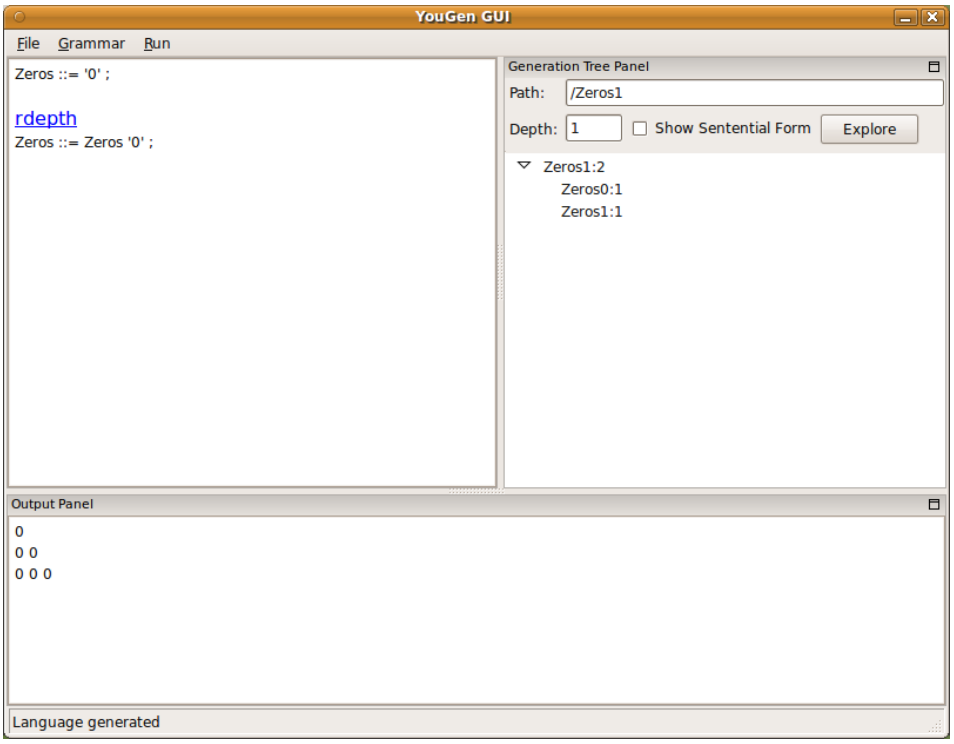

Figure 3.6: Dervish screenshot of `Zeros` grammar

3.2.2.1 *File* Menu

The file menu contains two menu items: *Open File* and *Quit*, shown in Figure 3.7. Menu items are assigned a shortcut key to facilitate interation with the tool.

The *Open File* option allows users to select a grammar file ending with suffix `'.gr'` or `'.py'`. When a grammar file is opened, Dervish displays the grammar rules as shown in the left panel of Figure 3.6. Both terminals and non-terminals appear in normal font; terminals are enclosed with single quotes. Underlined text is clickable and represents tags and generator non-terminals. For example, Figure 3.6 shows the `rdepth` tag applied to the second `Zeros` rule.

3.2.2.2 *Grammar* Menu

The *Grammar* menu provides an option to add tags to grammar rules as shown in Figure 3.8. This option allows users to associate a `count`, `cov`, `depth`, or `rdepth` tag with a rule. For example, Figure 3.9 shows the window that appears when users

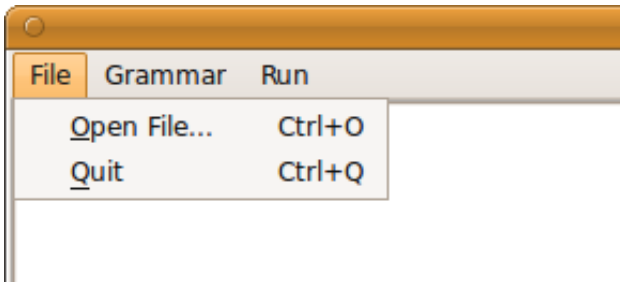

Figure 3.7: *File* menu options and their associated shortcut keys

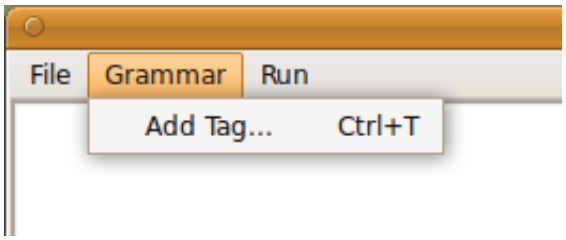

Figure 3.8: *Grammar* menu option and its associated shortcut key

select the *Add Tag* menu option. The top portion of the window shows a list of grammar rules and allows users to select a rule to which a tag will be added. The highlighted item specifies the rule selected to add a tag. The bottom portion of the window allows users to select the tag to be associated with the highlighted rule. Once a tag is selected, an entry becomes visible next to the tag to add the tag parameter. In this example, the user is adding tag {`count 1`} to the second `Zeros` rule.

3.2.2.3 *Run* Menu

The *Run* menu contains one command and two options, as shown in Figure 3.10. When selecting the *Run* command, Dervish invokes YouGen to generate the language of the grammar. If the *Show Output* option is checked, the language generated is displayed in the bottom panel. If the *Log Generation Tree* option is checked, YouGen logs the generation tree to an XML file which is later parsed and displayed by Dervish.

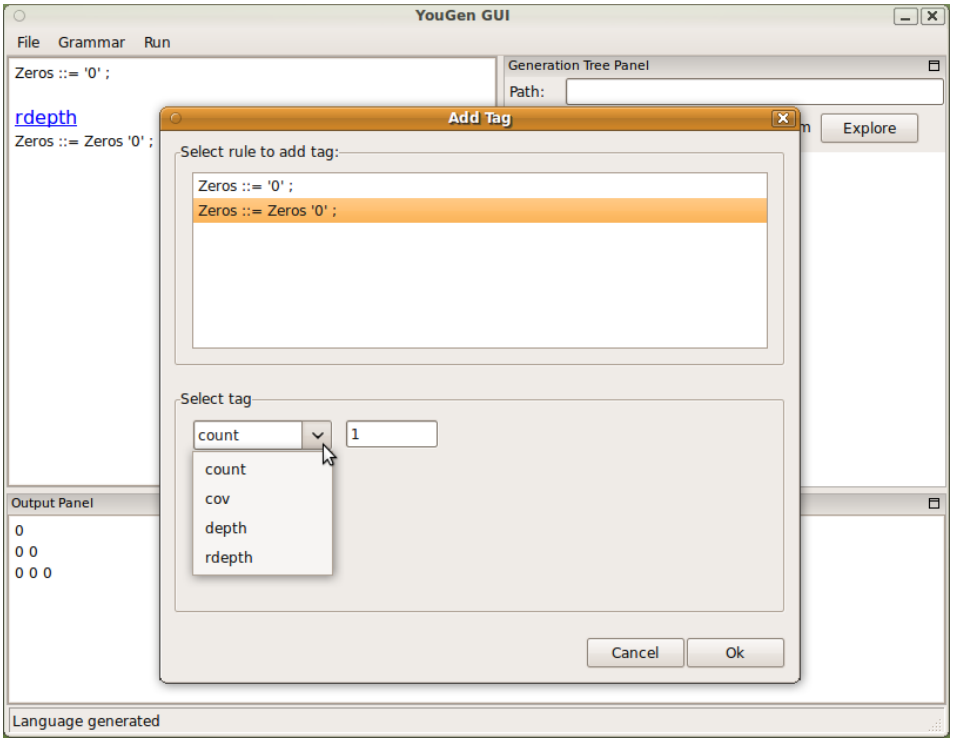

Figure 3.9: Dervish screenshot of *Add Tag* window

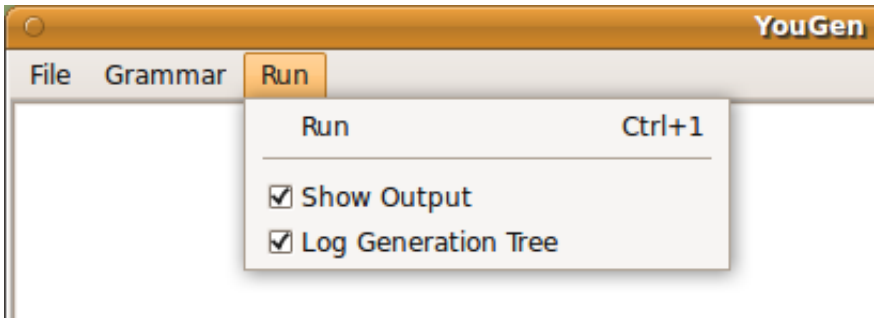

Figure 3.10: *Run* menu options and its associated shortcut key

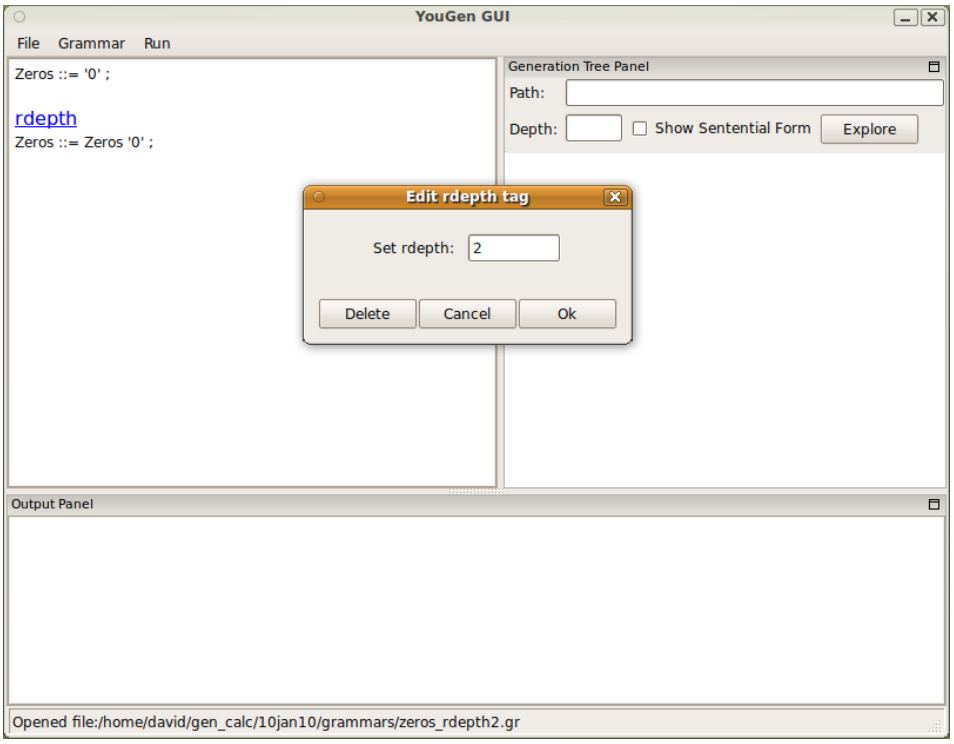

Figure 3.11: Dervish screenshot of `rdepth` tag dialog box

3.2.3 Grammar Features

Dervish allows users to modify tags and generator non-terminals.

3.2.3.1 Tags

In the grammar panel, tags are shown as clickable underlined text and are associated with the rule that appears on the following line. Only tag names, rather than the full tag text, are displayed. When a tag's underlined text is clicked, a dialog box displaying the tag parameters appears and allows users to delete the tag, modify the tag parameter, or cancel the modification. For instance, Figure 3.11 shows the dialog box that appears when the user clicks on the `rdepth` tag.

The dialog boxes for the `count` and `depth` tags are similar to the one used for `rdepth` tag.

When the user clicks on a `cov` tag, a dialog box appears as shown in Figure 3.12. The user can select a coverage specification and modify its parameters and strength.

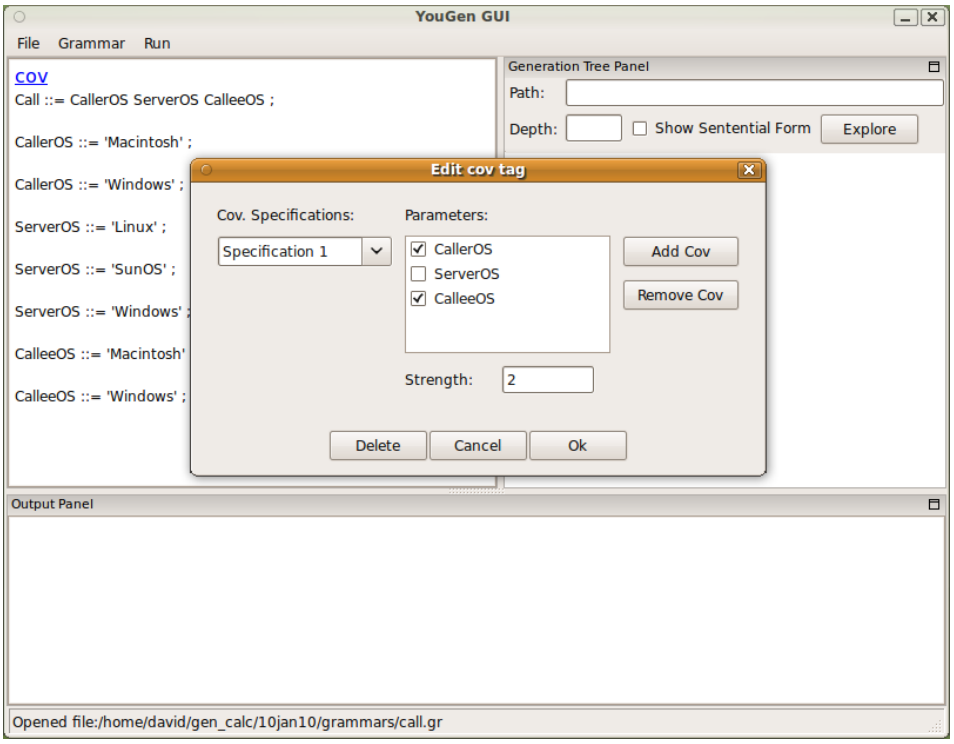

Figure 3.12: Dervish screenshot of `cov` dialog box

Coverage specifications can also be added or removed by clicking on the *Add Cov* or *Remove Cov* buttons. Users can remove the `cov` tag using the delete button. Otherwise, modifications to the `cov` tag are applied when the user clicks on the *Ok* button.

3.2.3.2 Generator Non-Terminals

Similar to tags, generator non-terminals are displayed as clickable underlined text. In the grammar panel, only their names, rather than the full generator non-terminal text, are displayed. Therefore, to accesss their parameters, the user must click on the generator name. For example, Figure 3.13 shows the dialog box that appears when the user clicks on the `Range` generator non-terminal. The user can modify the *start*, *skip*, and *end* parameters.

Currently, YouGen provides three generator non-terminals, `Range`, `List`, and `File`. Since grammar developers can also create their own generators, Dervish has

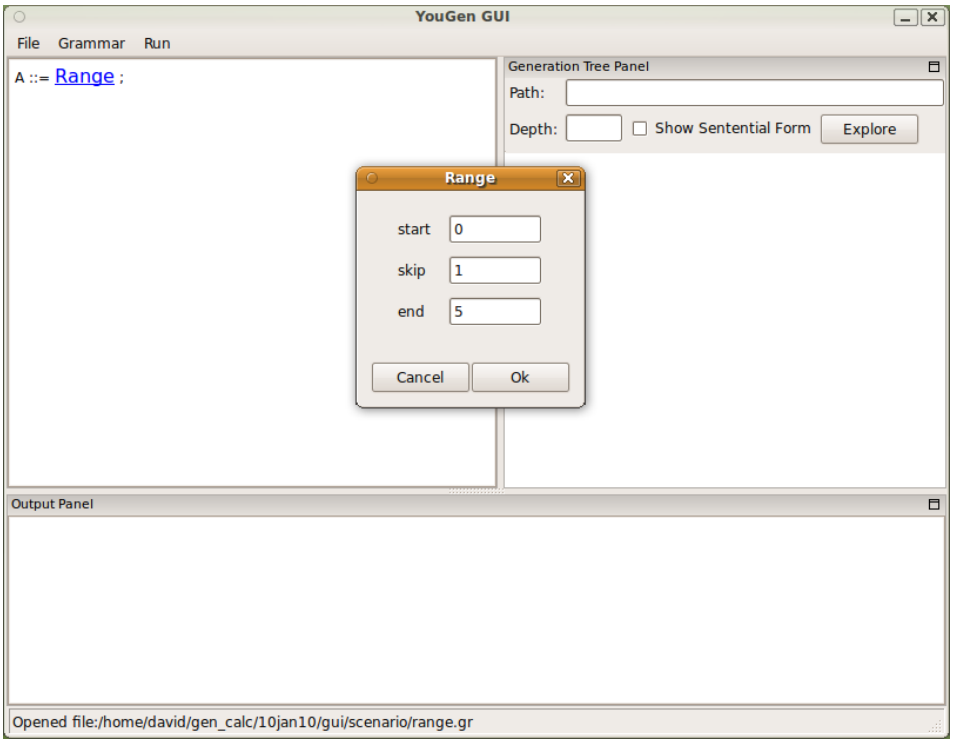

Figure 3.13: Dervish screenshot of `Range` generator non-terminal

been designed to support custom generators. The steps to create a custom generator are similar to the ones presented in YouGen [23]. A sub-class of YouGen's `Terminal_generator` class is created in the `global_precode` section. However, instead of providing two methods, the class must now provide at least four methods. The four methods to be provided by the class are:

- `__init__(self,`*param_list*`):` a constructor. *param_list* is a list containing all parameters specified for this generator non-terminal.
- `generate(self)`: a generator function which is called when the rule containing the generator non-terminal is derived.
- `get_parameters(self)`: a function which returns a list of triples. Each triple consists of a label, type, and validating function object. Label is a string used as parameter name to be displayed in the dialog box. Type is the data type of the parameter and the validating function object is a function to validate the

parameter when the user clicks on the *Ok* button in the dialog box.

- `validate_param(self,`*param*`):` This function is used to validate a parameter of the generator non-terminal. If validation fails, it must return a string specifying the message to display to the user, or return None otherwise. If necessary, the grammar developer can specify more than one validating function, one for each generator non-terminal parameter.

Figure 3.14 shows a grammar that uses a generator non-terminal to generate N positive even integers starting at *Start*, and Figure 3.15 shows this grammar in Dervish. The `EvenNumbers` generator non-terminal button has been pressed, bringing up a dialog box which graphically displays the generator non-terminal `EvenNumbers(5,2)`.

3.2.3.3 Embedded Code

Dervish does not display or allow modification of embedded code. If embedded code is present in a grammar file, it will not appear in the grammar panel of Dervish. For example, Figure 3.16 shows the Zeros grammar with embedded code, and Figure 3.17 shows the result of displaying this grammar in Dervish.

3.2.4 Generation Tree Features

Similar to the text-based version, the GUI also supports the display of generation trees. Figure 3.18 shows the generation tree of the Zeros grammar with tag {`rdepth 4`}. As shown in this figure, the top portion of the generation tree panel consists of four items: two textboxes, one checkbox and one button. The textboxes are used to specify the path and depth of the generation tree and the checkbox is used to show sentential forms. When the user clicks on the *Explore* button, the generation tree is displayed in the bottom portion of the panel. In this figure, the generation tree from path `/Zeros1` to depth 2 is shown. A tree widget is used to display the generation tree. This widget is similar to the ones used to display directory structures in file manager applications such as *Windows Explorer*. It allows nodes to be expanded or

```
{global_precode
class EvenNumbers(Terminal_generator):
        def __init__(self,param_list):
                # constructor
                self.param_list = param_list

        def generate(self):
                # generates the first N positive even integers
                # starting at start
                N = self.param_list[0]
                start = self.param_list[1]

                if start % 2 != 0:
                        start += 1

                for i in range(0,N):
                        yield start
                        start += 2

        def get_parameters(self):
                # returns the list of parameters
                return [('N (how many):',int,self.validate_param),\
                        ('Start:',int,self.validate_param)]

        def validate_param(self,param):
                # validates param is greater or equal than 0
                if param < 0:
                        return 'Illegal value for parameter '
                return None
}
A ::= EvenNumbers(5,2);
```

Figure 3.14: Example of a Custom Generator Non-Terminal

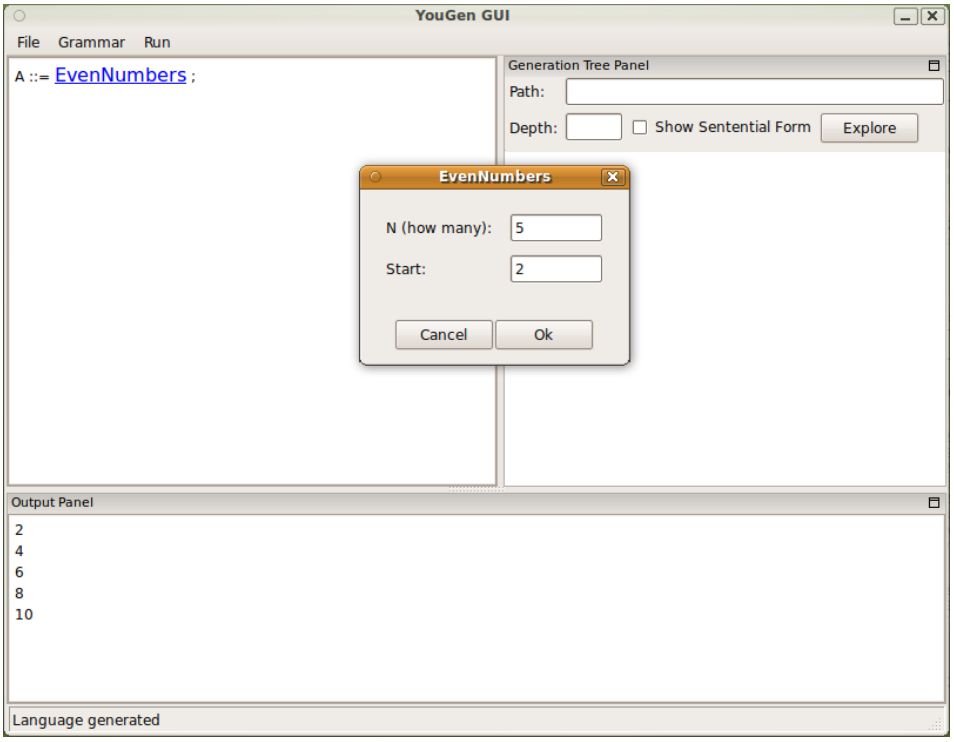

Figure 3.15: Dervish screenshot of EvenNumbers grammar

```
{global_precode
x = 'hello_world'
def foo(y):
        return y+y
}
{postcode
        print 'post Zeros0:', s
}
Zeros ::= '0';

{postcode
        print 'post Zeros1:', s
}
{rdepth 2}
Zeros ::= Zeros '0';
```

Figure 3.16: `Zeros` grammar with `global_precode` and `postcode` blocks

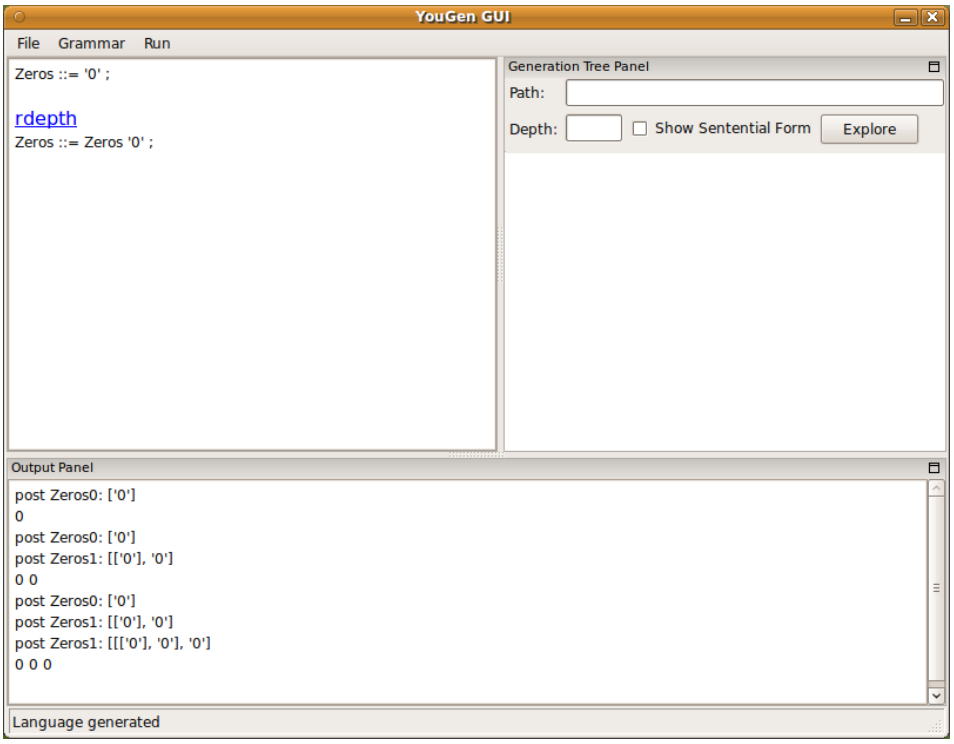

Figure 3.17: Dervish screenshot of `Zeros` grammar with embedded code

collapsed and takes minimal window space which is beneficial when displaying large generation trees.

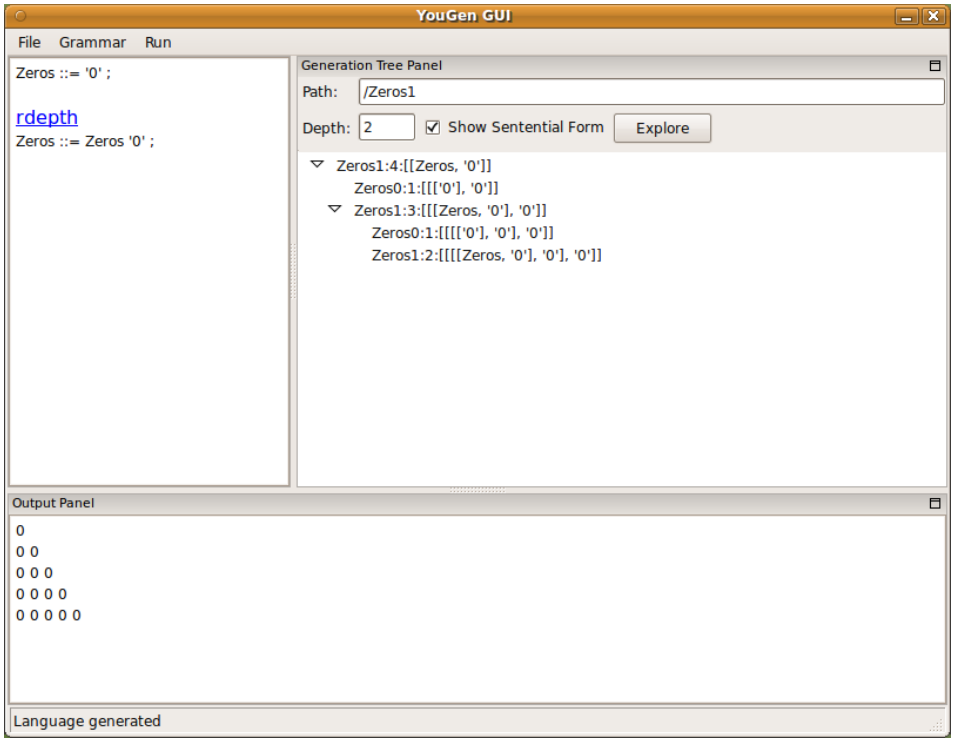

Figure 3.18: Dervish screenshot of `Zeros` grammar with `rdepth 4`

Chapter 4
Dervish Design and Implementation

This chapter explains the design and implementation of Dervish. First, the text-based version is presented, followed by the GUI version. In each version, the implementation is divided into a group of modules which are composed of classes and functions. For each version, a description of the modules is presented along with the number of classes, functions, and lines of code.

4.1 Text-Based Implementation

4.1.1 Modules

Lines of code: 104
Classes: 1
Functions: 3

The text-based version is divided into two modules: a main module and a generation tree module. The total number of lines of code, classes and functions is shown above.

4.1.1.1 Main Module

Lines of code: 54
Classes: 1
Functions: 0

The main module, which is composed of one class, is responsible for interpreting and executing commands specified by the user. The module executes code to create a loop, prompting the user to enter a command. When a command is entered, it is parsed and dispatched to its executing method. Four executing methods exist and are called when the user types `explore`, `help`, `setindent`, or `exit`.

The executing methods for each command are now presented:

- `explore` method: executes code to parse arguments passed to the `explore` command and to call the generation tree module which is responsible for displaying count trees on the screen.

- `help` method: executes code to parse arguments passed to the `help` command. If one argument is supplied and matches one of the four Dervish commands, this method executes code to display the command's documentation on the screen.

- `setindent` method: executes code for setting the number of spaces of indentation used between a parent node and its children when count trees are displayed.

- `exit` method: executes code to exit the loop and to gracefully terminate the program.

The main module also stores the XML generation tree in a data structure in memory using the `ElementTree` library available in Python's standard distribution. The data structure representing the XML generation tree is later used by the generation tree module to display count trees.

4.1.1.2 Generation Tree Module

Lines of code: 50
Classes: 0
Functions: 3

The generation tree module is responsible for traversing the internal representation of the XML generation tree and displaying count trees on the screen. When the main module receives the `explore` command as user input, this module is invoked. The module contains a routine which recursively visits each node of the tree exactly once and displays it on the screen. When the `explore` command is supplied with depth n, the routine uses n as a depth limit and stops displaying nodes after reaching that limit. When the `explore` command is supplied with path p, the routine first traverses the tree along p before displaying the nodes from the generation tree.

4.2 GUI Implementation

4.2.1 Modules

Lines of code: 655

Classes: 8

Functions: 5

The GUI version is divided into seven modules. The total number of lines of code, classes and functions is shown above.

4.2.1.1 Main Module

Lines of code: 227

Classes: 2

Functions: 0

The main module is composed of two classes: `DervishApp` and `DervishGUI`. It is responsible for creating the application window which contains three panels, a menu bar, and a status bar. The main module also declares event handler functions which are called when the user selects options from the menu bar, clicks on a tag or generator non-terminal from the grammar panel, or explores the generation tree.

This module is also responsible for invoking modules from YouGen to parse and generate the language of a grammar. When a grammar file with suffix `'.gr'` is opened, YouGen's parser module is invoked followed by a call to YouGen's code generator to translate the grammar into an executable Python module. The module is then imported into Dervish, giving access to the rule database. At this point, the main module calls the display rule module to show the rules in the grammar panel. When a grammar file with suffix `'.py'` is opened, only the last two steps are executed: the module is imported into Dervish and the display rule module is called.

To generate the language of a grammar, the main module invokes YouGen's `generate_language()` function and displays the strings generated in the bottom panel.

4.2.1.2 Display Rule Module

Lines of code: 38

Classes: 0

Functions: 3

The display rule module displays grammar rules to the grammar panel. While rules are displayed in normal text font and following YouGen's syntax, tags and generator non-terminals are displayed as clickable underlined text. Specifically, tag and generator non-terminal names are displayed rather than the full tag or generator text.

4.2.1.3 Tag Module

Lines of code: 148

Classes: 2

Functions: 2

The tag module is responsible for displaying and modifying tag parameter(s). This module is called when a user clicks on a tag in the grammar panel. A dialog box is created which allows users to delete the tag or modify the tag parameter(s). This module is also responsible for validating tag parameters. When a tag parameter is modified, the module checks whether the modification is valid. For example, when a user modifies a `rdepth` tag, this module verifies that the tag parameter is an integer equal or greater than zero. When tag parameters are validated, the module modifies the tag in the rule database. Otherwise, a pop-up window displays an error message specifying that the modification is not valid.

4.2.1.4 Generator Non-Terminal Module

Lines of code: 53

Classes: 1

Functions: 0

The generator non-terminal module is responsible for modifying and displaying generator non-terminal parameters. When the user clicks on a generator non-terminal button, the main module invokes this module to display a dialog box containing the

generator parameter(s). Since the number of parameters can vary from one generator non-terminal to another, this module calls the `get_parameters()` function from the generator instance to dynamically create widgets such as labels and textboxes needed to hold each parameter. The `get_parameters()` function is used to retrieve the number of parameters, the labels to be displayed in the dialog box, and the type and validating function of each parameter. This module then uses each parameter's validating function to test whether each parameter is valid. Once a parameter is validated, the module modifies the generator non-terminal in the rule database. Otherwise, a pop-up window displays an error message specifying which parameter is not valid.

4.2.1.5 Add Tag Module

Lines of code: 92
Classes: 1
Functions: 0

This module is responsible for displaying the dialog box that appears when the *Add Tag* option is selected from the *Grammar* menu and adding a tag to the rule database. When the dialog box is created, this module first displays the grammar rules in a list box. When a rule is selected, the module enables a drop down list containing the tags supported in YouGen. When a tag is selected, the module displays a textbox which allows the user to enter the tag parameter(s). When the user clicks the *Ok* button, the module calls functions from the tag module to validate the tag parameter. If the validation process succeeds, this module adds the newly created tag to the rule database. If the validation process fails, a pop-up window displays an error message specifying which parameter is not valid.

4.2.1.6 Generation Tree Module

Lines of code: 50
Classes: 0
Functions: 3

The generation tree module is called by the main module when the user clicks

on the *Explore* button. This module is similar to the generation tree module of the text-based version. While the text-based version displays generation trees as text, the GUI version uses a tree widget. The tree widget handles expansion and contraction of nodes.

Similar to the text-based version, this module contains a routine which recursively visits each node of the generation tree and displays it on the screen. If depth n is supplied, the routine stops displaying nodes after reaching that depth. If path p is supplied, the routine first traverses the generation tree along p before it starts displaying nodes from the generation tree.

4.2.1.7 Layout Module

Lines of code: 47
Classes: 2
Functions: 0

The layout module is responsible for two tasks: to lay out widgets on the screen and to provide a generic dialog box skeleton. This module is composed of two classes: `Grid` and `DialogBox`. When instantiated, a `Grid` object takes a list of widgets and displays them in a two-dimentional grid. The grid can then be placed and aligned on a window which is eventually displayed to the user. Although two-dimentional grids are not visible, they are used in dialog boxes to display tag and generator non-terminal labels and parameters. The `DialogBox` class is used as a base class to dialog box classes of tags and generator non-terminals. The `DialogBox` class contains widgets such as the *Ok* and *Cancel* buttons and a two-dimensional grid used to lay out widgets.

4.3 Call Graph

This section presents the call graph of Dervish's GUI, shown in Figure 4.1. The main operations, shown in italics, are executed by Dervish as a response to user events such as launching the application, opening a grammar file, exploring the generation tree, and others. Each line below an operation shows the name of a function and

each function calls the functions which are below it and tabbed to the right. For example, on startup `dervish.py` executes code to create the application instance by calling `DervishApp.__init__`, shown in line 2. Next, the application instance calls `DervishGUI.__init__` to create the main window, the menu and status bar, and the three panels. Then, the application instance enters a loop and waits for an event to occur. Each event is dispatched to an event handler function located in `DervishGUI`.

The role of the *open* operation (line 5) is to load and display a grammar file on the screen. First, when the user selects the *Open File* option from the *File* menu, the `DervishGUI.on_open()` function is triggered. Second, the grammar file is opened and read by calling function `DervishGUI.load_grammar()`. Third, YouGen's parser and code generator modules are called to translate the grammar into an executable Python module. Finally, the module is loaded and the function `display_grammar()` from the `DervishGUI` class accesses the rule database to display the grammar on the screen.

The *run* operation (line 11) is invoked from the *Run* menu option. This results in a call to the event handler function `DervishGUI.on_run()`. This function invokes YouGen to generate the language of the grammar by calling `generate_language()` from the `YouGen_NG` module.

The *modify tag* operation (line 14) is invoked when the user clicks on an underlined tag in the grammar panel. First, the `DervishGUI.on_click()` event handler function is called. Then, a dialog box instance is created by calling `TagDialog.__init__`. When the user presses the *Ok* button, the `TagDialog.on_ok()` event handler function is called. This function calls the `validate_tag()` function to check that the tag parameter entered is valid.

The *modify generator non-terminal* operation (line 19) is similar to the modify tag, with some differences in the number of functions called. The dialog box instance of a generator non-terminal calls the `get_parameter()` function of the generator non-terminal instance to retrieve the parameters to be displayed in the dialog box. When the user clicks the *Ok* button, the `validating_param()` function of each parameter is called.

The *add tag* operation (line 25) is invoked when the user selects the *Add Tag* option from the *Grammar* menu. The event handler function `DervishGUI.on_add_tag()` is invoked and creates a dialog box by calling the `AddTag.__init__` function. When the user presses the *Ok* button, the `AddTag.on_ok()` function is called. At this point, the tag parameter is validated by calling `validate_tag()`.

The *explore gentree* operation (line 30) is invoked when the user clicks on the *Explore* button in the generation tree panel. The first function called is the event handler `DervishGUI.on_explore()`. It retrieves the information entered in the textboxes labeled *path* and *depth*, and the checkbox labeled *show sentential form*. Next, the `parse_path()` function is called to split the *path* value into a list of rule ids. Then, the generation tree is traversed along the path before it is displayed on the screen.

Finally, lines 35 and 36 show the *exit* operation.

4.4 Testing of Dervish

Testing Dervish consisted of testing (1) the text-based version and (2) the GUI-based version. For the text-based version, a test case consisted of an XML generation tree file, a command and an expected output. The XML generation tree file is produced by running YouGen on a grammar. YouGen comes with a collection of test grammars which have been used to test Dervish and to create XML generation trees. Once an XML generation tree is created, it is used as input to the application under test, which is the text-based version of Dervish. Then, each command supported by the tool was executed with different combinations of parameters. The expected outputs were manually constructed and used in the comparison with the actual outputs.

For the GUI-based version, a test case consisted of a test grammar, one or more event(s) to be executed, and an expected output. For example, a test case consisted of the `TwoBit` grammar shown in Figure 2.1 and the following events to be executed:

1. Select *Open File* from the *File* menu and load the `TwoBit` grammar.

2. Generate the language of the `TwoBit` grammar by selecting *Run* from the menu.

```
1   startup
2       DervishApp.__init__
3           DervishGUI.__init__
4       DervishApp.MainLoop()
5   open
6       DervishGUI.on_open()
7           DervishGUI.load_grammar()
8               parser.Parser()
9               parser.Code_generator()
10          DervishGUI.display_grammar()
11  run
12      DervishGUI.on_run()
13          YouGen_NG.generate_language()
14  modify tag
15      DervishGUI.on_click()
16          TagDialog.__init__
17              TagDialog.on_ok()
18                  validate_tag()
19  modify generator non-terminal
20      DervishGUI.on_click()
21          GNTDialog.__init__
22              get_parameters()
23              GNTDialog.on_ok()
24                  validate_param()
25  add tag
26      DervishGUI.on_add_tag()
27          AddTag.__init__
28              AddTag.on_ok()
29                  validate_tag()
30  explore gentree
31      DervishGUI.on_explore()
32          parse_path()
33          traverse_gentree()
34          display_root()
35  exit
36      DervishGUI.on_exit()
```

Figure 4.1: `Call` graph of Dervish GUI

3. Run YouGen on the `TwoBit` grammar from the command line to capture the expected output.

4. Compare the actual output shown in the *Output* panel of Dervish with the expected output provided by YouGen.

A similar exercise was conducted for testing each feature provided by the GUI-based version of Dervish.

Chapter 5
Case Study: A New GBTG Learning Tool Support

In this chapter, we show how to use Dervish to support the learning of grammar-based test generation (GBTG). Specifically, we show that Dervish helps testers use GBTG tools such as YouGen. To demonstrate how testers interact with Dervish, we present three simple grammar examples.

5.1 The Problem

For more than 30 years, many tools have been designed and implemented to demonstrate the effectiveness of GBTG [1, 6, 22]. Although the tools have helped testers and researchers learn about GBTG, many of them have been difficult to use except by the tool developer. Dervish was designed to allow the tester to modify a grammar to generate test cases while also considering the tester's capabilities. First, since testers usually know a lot about the application under test but less about programming, Dervish provides a GUI to help testers to use GBTG tools without any programming skills. Second, Dervish helps the interaction with YouGen to perform tasks such as generating test cases and understanding how test cases are created. In the following sections, we present three grammar examples to show that Dervish helps testers use the power of GBTG.

5.2 Example 1: `TwoBit` Grammar

Figure 5.1 shows a Dervish screenshot for the `TwoBit` grammar from Figure 2.1. The left panel shows the grammar rules as read-only, the bottom panel shows the language generated and the right panel shows the generation tree.

YouGen produces four strings, shown in the bottom panel. Although grammar developers may understand how this language was generated, testers, who are usually less familiar with GBTG, may need help to understand it. Since YouGen can generate

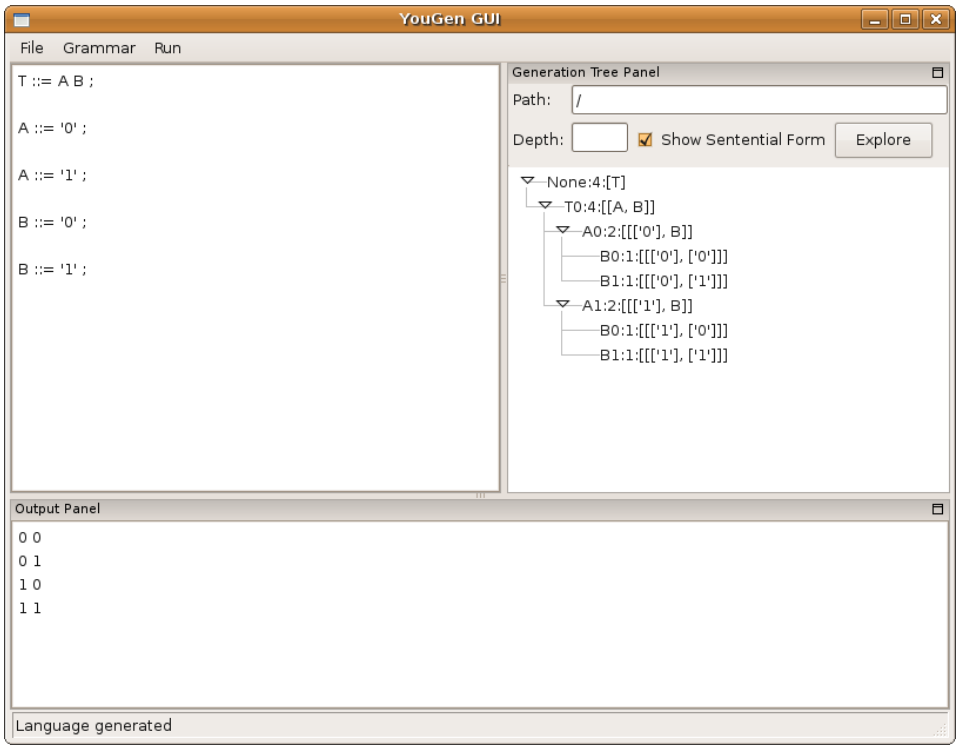

Figure 5.1: Dervish screenshot of `TwoBit` grammar

the test cases needed by a tester, understanding the generation strategy used by YouGen is important. Dervish has been designed to help testers understand how YouGen generates the language of a grammar. Since Dervish displays the generation tree, a tester may look at the derivation steps to understand how strings are generated.

For example, the derivation steps used by YouGen to generate the language of the `TwoBit` grammar are shown in the right panel. Node `None:4:[T]`, shown at depth 0, is the root of the generation tree. The start symbol for this grammar is `T` and the start sentential form is `[T]`. `None` means that none of the rules were used to get to the start sentential form `[T]`. `4` means that four ground sentential forms appear in the subtree rooted at node `None:4:[T]`. Next, YouGen selects the left-most non-terminal in the sentential form for replacement. Because the sentential form consists of `T`, `T` is chosen for replacement. The results are shown at depth one in the generation tree. Node `T0:4:[[A,B]]` shows that rule `T ::= A B` is used to replace non-terminal `T` with non-terminals `A B`, indicated by the modification of

parent sentential form `[T]` to child sentential form `[[A,B]]`. Next, YouGen selects non-terminal `A` for replacement since it is the left-most non-terminal in the sentential form. Because the `TwoBit` grammar contains two rules with non-terminal `A` on the left-hand side and that YouGen selects the rules in the order they appear textually in the grammar, rule `A ::= '0'` is selected for substitution. This is indicated by node `A0:2:[['0'],B]]` at depth two. Nonterminal `A` is replaced with terminal `'0'` which is indicated by the modification of parent sentential form `[[A,B]]` to child sentential form `[['0'],B]]`. Next, YouGen selects non-terminal `B` for replacement. Similar to non-terminal `A`, two rules with non-terminal `B` on the left-hand side appear in the grammar. YouGen selects the first rule `B` for substitution. The results are shown below node `A0:2:[['0'],B]]` at depth three in the generation tree. First, node `B0:1:[['0'],['0']]]` shows that first rule `B` is used to get from non-terminal `B` to terminal `'0'`, indicated by the modification of parent sentential form `[['0'],B]]` to ground sentential form `[['0'],['0']]]`. Second, node `B1:1:[['0'],['1']]]` shows that second rule `B` is used to replace non-terminal `B` with terminal `'1'`, indicated by the modification of parent sentential form `[['0'],B]]` to ground sentential form `[['0'],['1']]]`. At this point, YouGen has produced strings `00` and `01`, shown on the first two lines of the output panel. Then, similar derivation steps are applied at node `A1:2:[['1'],B]]` to produce strings `10` and `11`.

5.3 Example 2: Zeros Grammar

This section presents an example of a recursive grammar and also shows how Dervish helps users manipulate YouGen tags. Figure 5.2 shows a Dervish screenshot for the Zeros grammar from Figure 2.5. The `rdepth` tag text has been clicked, generating a dialog box which allows the user to modify the tag parameter. When the user clicks the `Ok` button, Dervish validates the tag parameter before applying the modification to the tag. Since testers usually have little programming experience, modifying tags through a GUI can be beneficial. First, because tags are manipulated through the GUI no emphasis is put on learning the tag syntax. Therefore, testers may focus on learning tag semantics rather than tag syntax. Second, since Dervish

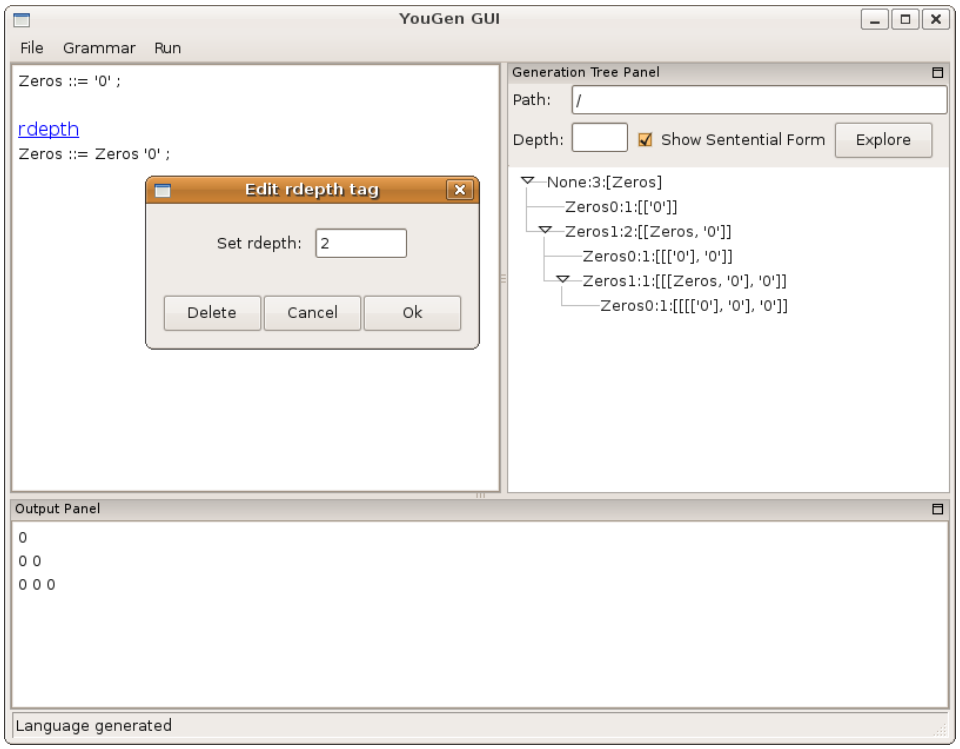

Figure 5.2: Dervish screenshot of `Zeros` grammar (left panel), generation tree (right panel), and language generated (bottom panel)

displays feedback to the user soon after incorrect tag parameter(s) are entered, invalid tag parameter(s) can be corrected before the grammar is parsed. Therefore, the users do not need to know nor learn the tag syntax which may help avoiding tag syntax errors.

When the tester is satisfied with the tag modifications, YouGen can be invoked from the *Run* menu to generate the language of the grammar. Then, through Dervish the tester may observe the effect that the tag modification has on the language generated. For example, attaching tag {`rdepth 2`} to second rule `Zeros` limits the number of strings generated to three. Again, the generation tree facilitates testers to understand visually how this language was generated. In addition, Dervish may help testers learn tag semantics with examples. Since Dervish allows tags to be inserted, modified, or deleted, testers may easily create several examples to understand the semantics of YouGen tags.

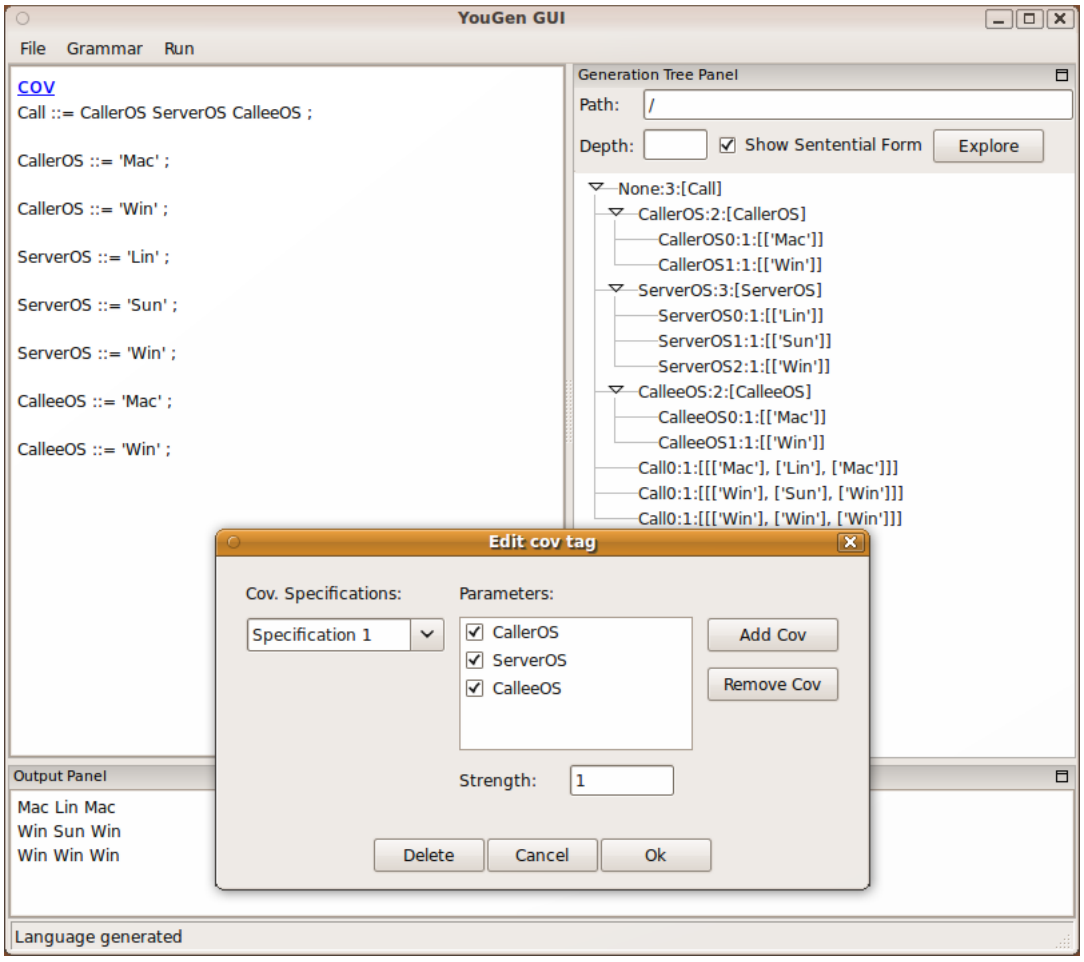

Figure 5.3: Dervish screenshot of `Call` grammar

5.4 Example 3: Call Grammar

This section describes the covering-array tag using an example of the Call grammar, shown in Figure 5.3. The grammar, the language generated, and the generation tree are shown in the left, bottom, and right panel respectively. If the `cov` tag is attached to a rule R for non-terminal N, then a new, temporary tree is generated with N as the root and R as the rule used to generate strings. The terminals and non-terminals on the right-hand side of R are treated as the parameters to a covering-array algorithm, which is called when all the strings for the parameters have been generated [8].

For example, a `cov` tag is attached to the rule with non-terminal `Call` on the left-

hand side. The underlined tag text has been clicked, bringing a dialog box showing the coverage specification. There is one coverage specification of strength 1 where parameters `CallerOS`, `ServerOS`, and `CalleeOS` have been selected. The generation tree for this grammar is shown in the right panel with node `None:3:[Call]` as root. Below the root are three parameter trees, one for each parameter. A parameter tree is a generation tree which shows the strings generated for each parameter. When all the strings of each parameter have been generated, they are passed to a covering-array algorithm which generates the set of tuples for this coverage specification. In this example, the strength is set to 1 and specifies that every element of each parameter must be present, but that combinations from Figure 2.12 may be missing. YouGen generates three strings, shown in the bottom panel.

Figure 5.4 shows a second example of a cov tag with two coverage specifications. Using YouGen's syntax, the `cov` tag parameter is {`cov [([0,2],2),([1],1)]`}. Coverage specification `([0,2],2])` specifies that all elements in `CallerOS x CalleeOS` must be present and coverage specification `([1],1)` specifies that all elements in `ServerOS` must be present. In this figure, the underlined tag text `cov` has been clicked, bringing a dialog box showing the second coverage specification where strength is 1 and parameter `ServerOS` is selected. The generation tree for this grammar is shown in the right panel with node `None:7:[Call]` as root. Below the root are three parameter trees which show the strings generated for each parameter. When all the strings of each parameter have been generated, they are passed to a covering-array algorithm which generates the set of tuples for the first coverage specification.

The coverage specification `([0,2],2)` indicates that all elements of `CallerOS x CalleeOS` must be present. The first four `Call0` nodes show the set of tuples for the coverage specification `([0,2],2)`. Next, the three parameter trees are regenerated since there is a second coverage specification. When all the strings of each parameter have been generated, they are passed to a covering-array algorithm which generates the set of tuples for the second coverage specification.

The second coverage specification `([1],1)` indicates that all elements of `ServerOS` must be present. The last three `Call0` nodes in the generation tree show the set of

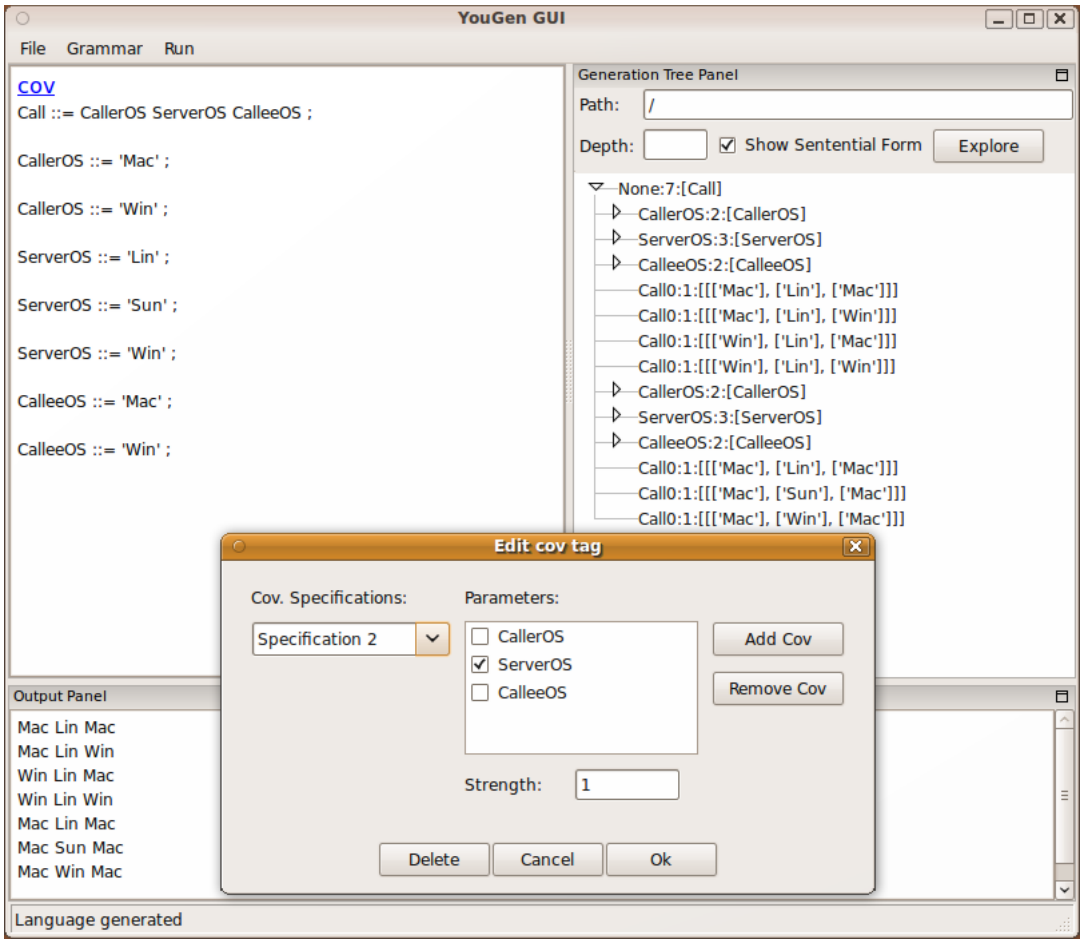

Figure 5.4: Dervish screenshot of the `Call` grammar with two coverage specifications

tuples for the coverage specification `([1],1)`. The bottom panel shows the language generated for this grammar.

Chapter 6
Case Study: Generation Tree Exploration

In the previous case study, we showed that Dervish can be used as a tool to support the learning of grammar-based test generation. We extend this idea to show that Dervish also helps testers understand complex grammars.

6.1 The Problem

When working with GBTG tools, testers typically create complex grammars to generate the test cases needed to test software systems. Since the language generated by those grammars is often large, testers are frequently faced with difficulties understanding the test cases generated by a grammar. Logging and viewing generation trees however can facilitate a tester's understanding of the language generated by a grammar. For instance, by looking at nodes in the generation tree a tester may find which rules lead to the generation of a large number of test cases. Then, the rules generating a large number of test cases may be modified or associated with tags to reduce the number of test cases generated. Since Dervish can be used as a tool to visualize generation trees, this case study presents three variations of a complex grammar and shows how Dervish helps testers understand the language generated by complex grammars.

6.2 Catalog Grammar

Figure 6.1 shows the Catalog grammar: a tagged recursive grammar which generates simple XML documents. The strings *BooksTag* and *ChaptersTag* are used to illustrate several posssible tag placements. This grammar is recursive and its language is infinite.

When *BooksTag* is {`rdepth 1`} and *ChaptersTag* is empty, the number of documents generated is $2^8 + 2^{16} = 65792$. By looking at the grammar, it is hard to arrive

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

                Books ::= Book;
BooksTag        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 6.1: Catalog Grammar

at this number. By looking at selected paths in the generation tree however, a tester may find the grammar rules which cause the language generated to be so large. For example, Figure 6.2 shows the generation tree of the Catalog grammar from the root to depth 2. When exploring this generation tree, a tester may notice that the number of documents generated from the recursive rule `Books`, indicated by the count at node `Books1:65536`, is much larger than the number of documents generated from the non-recursive rule `Books`, i.e., `Books0:256`. Therefore, a first approach to reduce the number of documents generated is to decrease the value of the `rdepth` tag to 0. Note that it normally does not make sense to include a rule with `rdepth` tag set to 0, but that this can be useful when exploring the grammar. In this case, the grammar produces $256 = 2^8$ documents and the generation tree consists of the subtree rooted at node `Books0:256`. At this point, the tester knows by looking at the generation tree that the number of test cases can be reduced to 256 if *BooksTag* is {`rdepth 0`}. What remains to be understood is how this number was generated. The following section shows how Dervish helps answering this question.

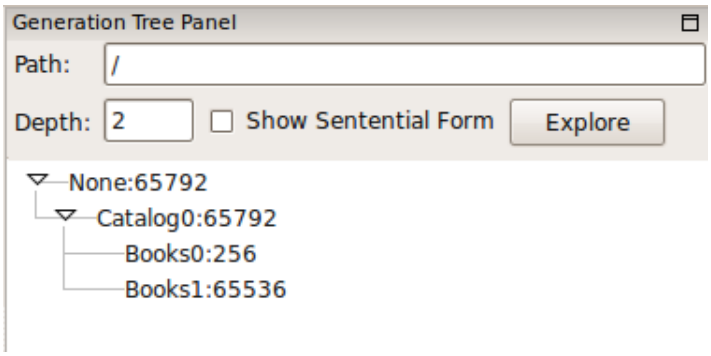

Figure 6.2: Generation tree from root to depth 2

6.3 Catalog Grammar with *BooksTag* set to {`rdepth 0`}

When *BooksTag* in Figure 6.1 is {`rdepth 0`} and *ChaptersTag* is empty, the size of `L(Catalog)` is limited to one book. Figure 6.3 shows the generation tree from root to depth 2 for this grammar. YouGen produces 256 documents, as indicated by the count at node `Catalog0:256`. To understand how this number was generated, Dervish may be used to explore the generation tree in greater depth. Figure 6.4 shows the generation tree from node `Books0:256` to depth 3. Since there are four title alternatives, indicated by three `Title0:64` and one `Title1:64` nodes, the number of books per title alternative is $64 = 256/4$. Since a book consists of one title and three chapters, exploring deeper in the generation tree shows that the `Chapters` rule consists of three chapters and that for each chapter there are four section name alternatives. Figure 6.5 shows the generation tree with `Chapters0:64` node expanded. Each of the nodes `Chapter0:64`, `Chapter0:16`, and `Chapter0:4` represents the three chapters. Since for each chapter there are four section name alternatives, there are $64 = 4 \times 4 \times 4 = 4^3$ section name alternatives per title alternative.

Therefore, the total number of documents generated is $256 = 4 \times 4 \times 4 \times 4 = 4^1 \times 4^3 = 2^8$ since there is one title with four alternatives and three chapters with four section name alternatives.

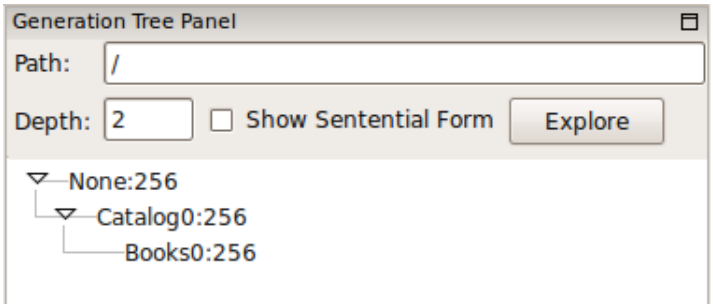

Figure 6.3: Generation tree from root to depth 2

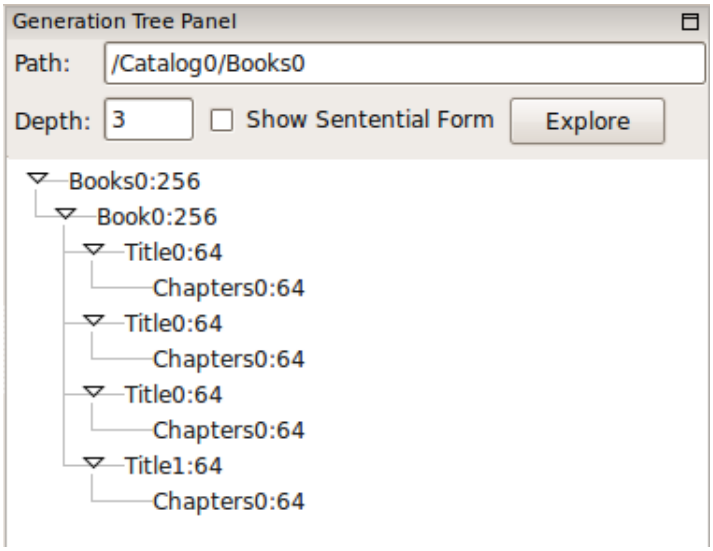

Figure 6.4: Generation tree from `Books0` to depth 3

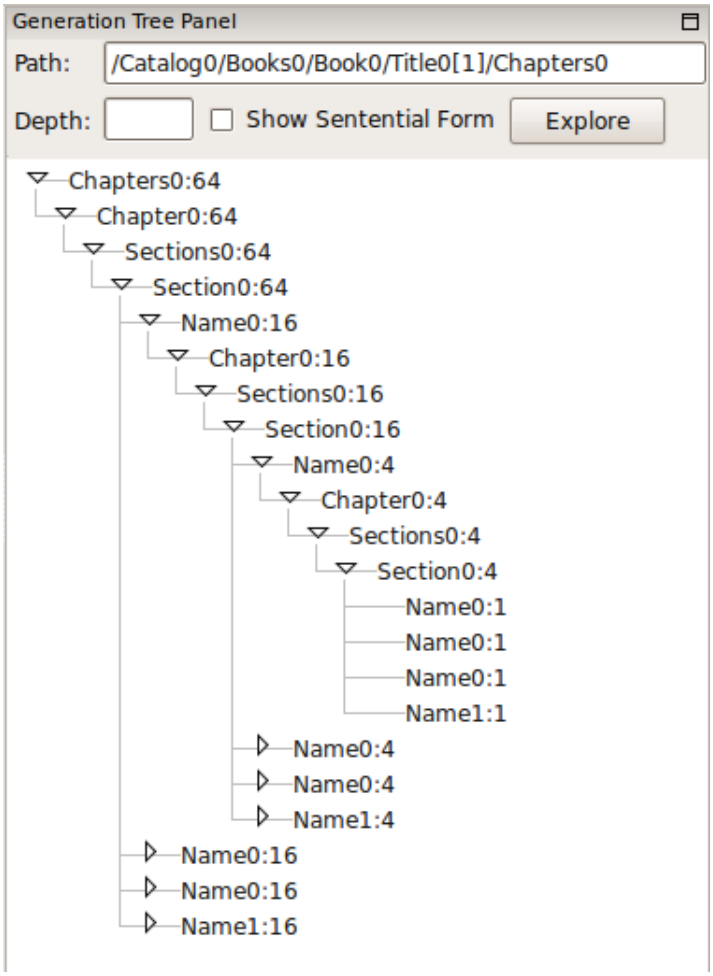

Figure 6.5: Generation tree from `Chapters0` to bottommost level

6.4 Catalog Grammar with *BooksTag* set to {`rdepth 1`}

When *BooksTag* is {`rdepth 1`} and *ChaptersTag* is empty, the Catalog grammar produces $65792 = 256 + 65536 = 2^8 + 2^{16}$ documents, as shown in Figure 6.2. The number of documents, $256 = 2^8$, from the path `/Catalog0/Books0` was explained in the previous section. Therefore, what remains to be understood is how $65536 = 2^{16}$ documents are generated from the path `/Catalog0/Books1`.

Using Dervish, a tester may explore the generation tree to a greater depth. Figure 6.6 shows the result of exploring the path `/Catalog0/Books1` to a depth of 7. There are four title alternatives, as indicated by three `Title0:16384` nodes and one `Title1:16384` node. Similar to the Catalog grammar when *BooksTag* is {`rdepth 0`} and *ChaptersTag* is empty, there are three chapters for each title alternative and each chapter has four section name alternatives. Unlike this grammar, the derivation steps of a second book follow the derivation steps of the first book which lead to the generation of documents with two books. Figure 6.7 shows the result of exploring the generation tree from the node where the derivation steps of the second book begins, which is along the path `/Catalog0/Books1/.../Books0`, to depth 3.

As a result, because there are two books, the number of documents generated consists of $65536 = 4^1 \times 4^3 \times 4^1 \times 4^3 = 256 \times 256 = 2^{16}$.

6.5 Catalog Grammar with *ChaptersTag*

A tester may notice that another approach to reduce the number of strings generated is to keep the `BooksTag` to {`rdepth 0`} and reduce the number of alternatives from the `Chapters` rule. Adding the tag {`cov [([0,1,2],2)]`} to *ChaptersTag* reduces the number of strings to $64 = 2^6$. There are three `Chapter` domains and each `Chapter` has one section which consists of four `Name` alternatives. YouGen generates a two-cover of 16 strings. Figure 6.8 shows how the 16 strings were generated. Next, because there are four title alternatives, the total number of documents generated is $64 = 16 \times 4$. Figure 6.9 shows the total number of documents generated, as indicated by node `Catalog0:64` and each one of the four title alternatives for this generation tree.

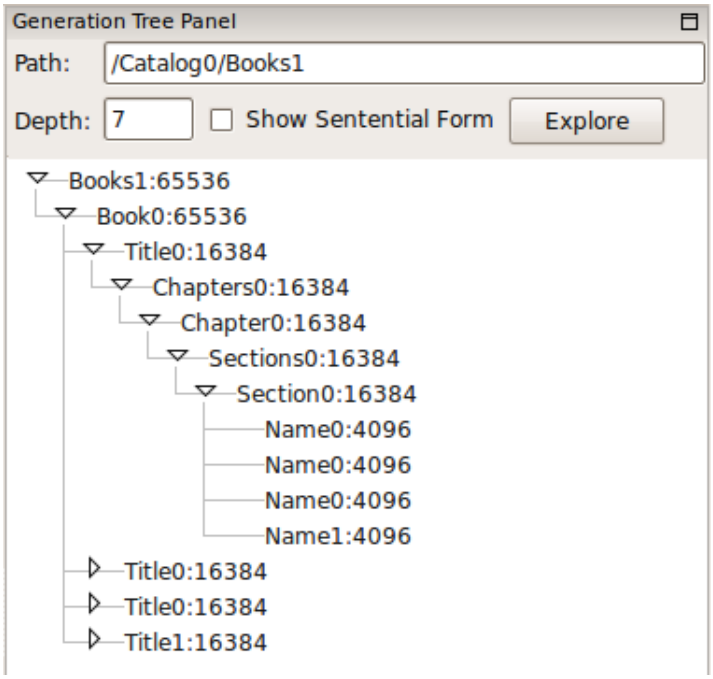

Figure 6.6: Generation tree from `Books1` to depth 7

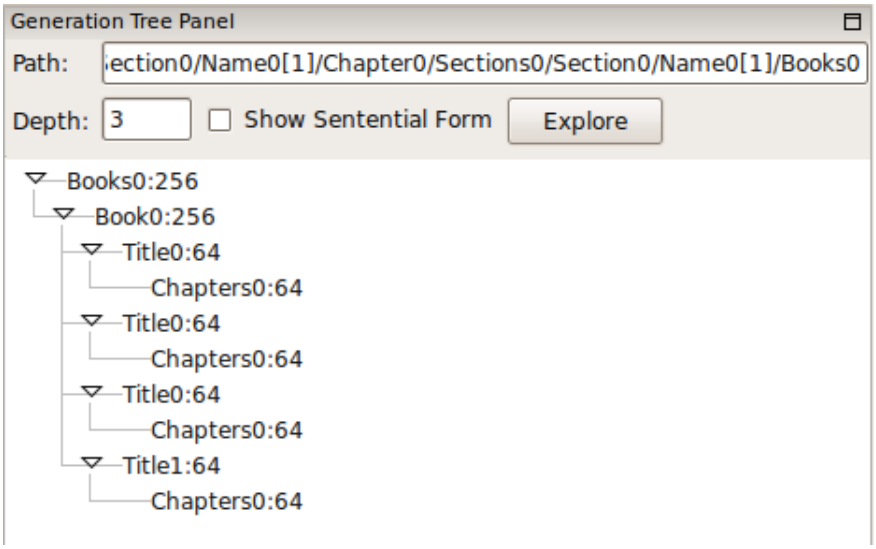

Figure 6.7: Generation tree from `Books0` to depth 3

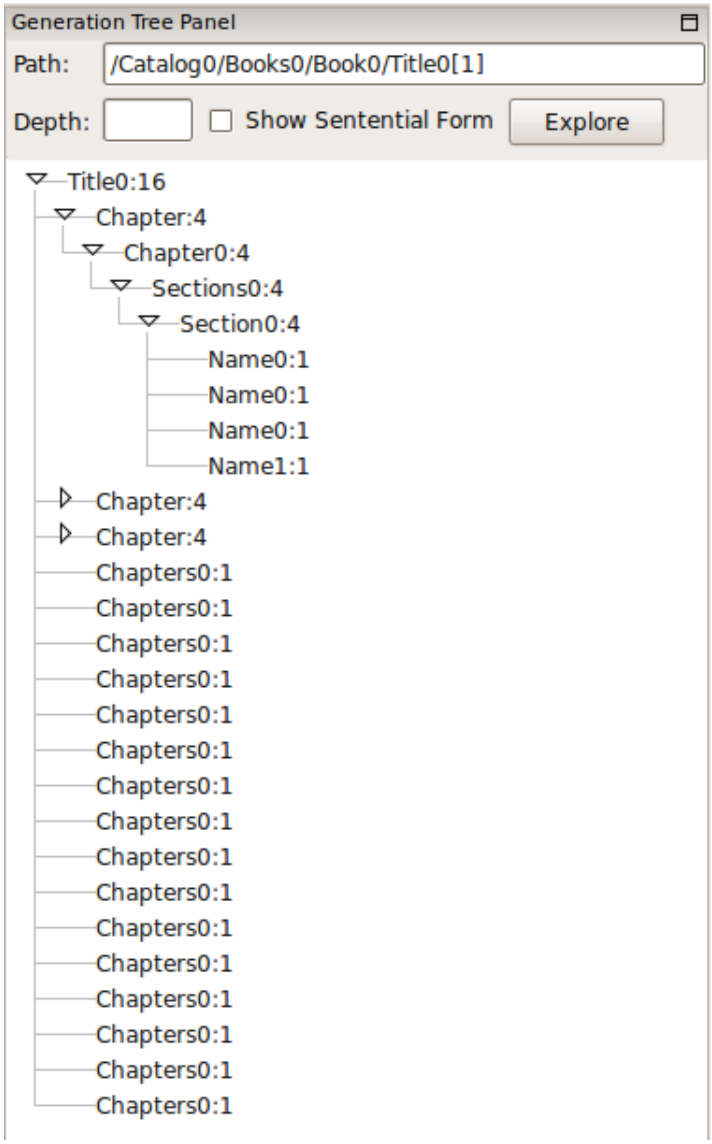

Figure 6.8: Generation tree from `Title0[1]` to bottommost level

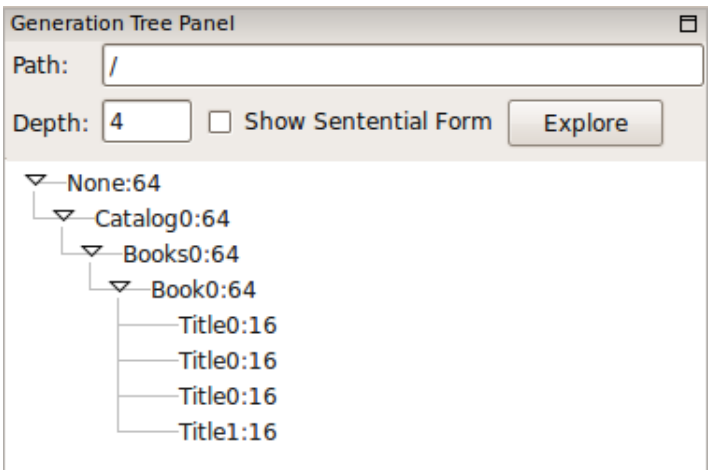

Figure 6.9: Generation tree from root to depth 4

Chapter 7
Case Study: Testing XPath Interpreters

In this case study, we are interested in observing the benefits of using Dervish as a testing tool. Since testers do not always have the programming skills to automate the testing process, we are interested in knowing whether Dervish can help testers generate and execute the test cases they need to test XPath interpreters. In the first section, we present the goal of the case study. Then, we review the background material on the XML [7] and the XPath [14] languages. This is followed by a description of the test approach and test execution. Then, we present the test results and conclude with a discussion.

7.1 Testing XPath Interpreters

Testers, who usually know a lot about the application under test, do not always have the programming skills or knowledge to automate the testing process. However, testers could use GBTG as an approach to automate the testing process if they were provided with a tool to support the control and execution of test cases. In this case study, we are interested in observing whether Dervish can be this tool and be used as a test bench to support the control and execution of test cases for testing XPath interpreters. We are interested in finding if testers, giving guidance through Dervish, can execute the tests they want to reveal errors in the XPath interpreters. Some of the key points we are observing are whether Dervish allows testers to modify parts of a grammar to generate the test cases they need to expose errors and whether testers can effectively guide the testing process.

This case study assumes that the testers are looking at failures that appear in interpreters which support the XPath language. More specifically, we assume that testers are looking at failures on three particular types of XML documents: XML documents with large element text length, XML documents with large depth, and

XML documents with large length. The code under test consists of four XPath interpreters: Libxml2 [27], Lxml [16], PyXML [21] and VTD-XML [26]. Each one of these software libraries provides functions which take an XML document and an XPath expression as input and return the result from applying the XPath expression to the document. In the next two sections, we present a short overview of the XML and the XPath languages which form the background material for this case study.

7.2 The XML Language

XML is used to label, structure and store information. The term XML stands for eXtensible Markup Language and was developed in 1996 before becoming an official World Wide Web Consortium (W3C) standard in 1998 [12]. Since its introduction, XML has been used in web technologies, in database systems, and in programming languages. Figure 7.1 shows an example of an XML document. In simple terms, XML is composed of markup symbols and data which form a tree structure. Markup symbols, called tags, are enclosed within angle brackets (<>). For example, `<addressbook>` and `<contacts>` are two tags. The text starting with `<addressbook>` and ending with `</addressbook>` is called an XML element. Elements must have a start tag and an end tag or must end with the characters `/>` if the element is empty. An element may be empty, may contain element text or may consist of other elements. For example, in Figure 7.1, elements `<name>` contain text element `Julia`, `James` and `Jill`, respectively. The element `<addressbook>` consists of an element `<contacts>`. Elements represent nodes in an XML tree and each element has relationships. For example, the elements `<addressbook>` and `<contacts>` form a parent-child relationship. Also, each one of the `<name>` elements are siblings and all elements have as parent the element `<contacts>` and as grand-parent the element `<addressbook>`.

7.3 The XPath Language

The XML Path Language (XPath) is used to search and to retrieve information from an XML document [14]. Using XPath expressions, a portion of an XML document can be retrieved. When an XPath expression is run against an XML document,

```
<addressbook>
        <contacts>
                <name>Julia</name>
                <name>James</name>
                <name>Jill</name>
        </contacts>
</addressbook>
```

Figure 7.1: Example of an XML document

```
/addressbook/contacts/name[1]/text()
```

Figure 7.2: XPath expression that retrieves element text `Julia`

the nodes matching the expression are selected and returned to the caller. The result returned may be a collection of nodes, a string, a floating-point number or a boolean value [12].

Although different types of XPath expression exist, this case study focuses on only one specific type, the location path. A location path is similar to the path to a file on disk. However, a location path may specify additional information. For instance, at each step along a path, one or more tests, called predicates, may be performed to select specific nodes. Since the complete syntax of the location path is complex, only the XPath expressions used in this case study are described. Figure 7.2 shows an example of an XPath expression which results in retrieving the element text `Julia` from the first element `<name>` appearing in Figure 7.1.

7.4 Code Under Test

Four interpreters were selected as code under test: Libxml2, Lxml, PyXML and VTD-XML. They are briefly described in this section.

1. Libxml2

   Libxml2 is an XML parser and toolkit developed for the Gnome project [27]. The library is written in C but language bindings are available for other envi-

ronments such as Java, Python, and Perl. The version used in this case study is 2.6.32.

2. Lxml

   Lxml is a library built on top of two C libraries: Libxml2 and Libxslt. It is intended to provide an API which is well documented and provides an intuitive interface to Python developers [16]. The version used in this case study is 2.1.1.

3. PyXML

   PyXML is another XML library for Python which consists of a collection of functions and utility modules for the XML and XPath language [21]. The version used in this case study is 0.8.4.

4. VTD-XML

   VTD-XML is a less traditional XML library which keeps pointers in memory to parts of a document as opposed to converting the XML document to a tree in memory. The version used in this case study is 2.1.

7.5 Test Approach

In this case study, a test case consists of the combination of an XML document and an XPath expression. We assume that testers are interested in failures appearing on the three particular types of XML documents, as presented in [17]: XML documents with large string as element text, XML documents with large depth, and XML documents with large length. The first type of test cases focus on creating XML documents with element text of large length, i.e., with a large number of characters. Figure 7.3 shows an XML document with an element text which is, however, quite short. Testers will be looking at creating similar XML documents with element text of large length, perhaps in the millions of characters.

The second type of test cases focus on creating XML documents of large depth, perhaps in the thousands. The depth of an XML document is defined to be the largest

```
<a>element_text</a>
```

Figure 7.3: XML document that contains an element text of small length

```
<a>
        <a>
                <a>nested element text</a>
        </a>
</a>
```

Figure 7.4: XML document that contains a nested element text at depth 3

number of ancestors an element can have. Figure 7.4 shows an XML document of small depth, 3.

The third type of test cases focus on creating documents of large length, perhaps in the thousands. The length of an XML document is defined to be the largest number of children an element can have. Figure 7.5 shows an XML document of small length, 3. In the next section, we are exploring how Dervish helps testers in generating test cases for the three possible scenarios.

7.5.1 Test Case Generation

The role of the grammar developer is to create the grammars to be used by the testers for testing the XPath interpreters. After a grammar file is developed, the testers may use Dervish's graphical interface to modify the parameters of the grammar tags and generator non-terminals to generate the desired test cases. For this case study, two grammars were created: (1) a grammar to generate XML documents

```
<root>
        <a>1</a>
        <a>2</a>
        <a>3</a>
</root>
```

Figure 7.5: XML document that contains a root element with 3 children

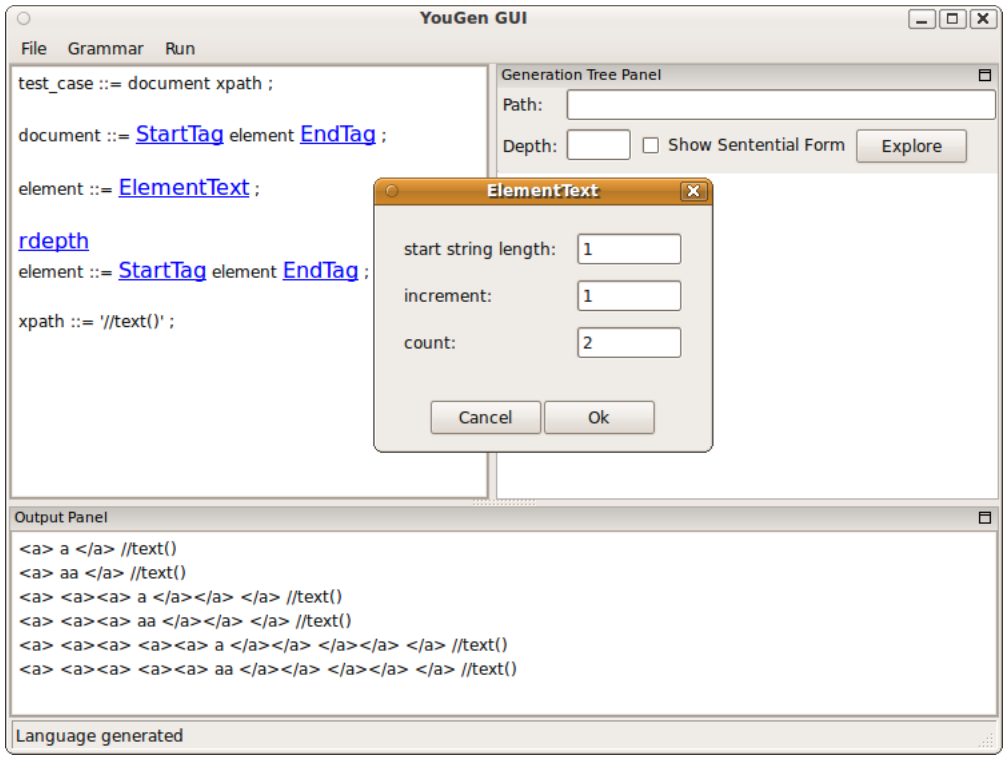

Figure 7.6: Dervish screenshot of first grammar in action

with element containing text of variable length and nested at variable depth and (2) a grammar to generate XML documents of variable length. The top left panel in Figure 7.6 shows the grammar used for generating XML documents with elements containing text of large length and XML documents of large depth. In this figure, the `ElementText` generator non-terminal text has been clicked, generating a dialog box showing the parameters that the tester may modify. This dialog box allows the testers to specify three parameters: *start string length*, *increment*, and *count*. The *start string length* parameter allows the testers to specify the start length of the text element. The *increment* parameter specifies the number of characters by which the next element text increases. The *count* parameter specifies the number of text elements to generate. For example, setting *start string length* to 1, *increment* to 1, and *count* to 2 allows 2 text elements to be generated, starting with a length of 1 character and increasing the length of the next element text by one character.

While the `ElementText` generator non-terminal allows the testers to specify ele-

ment text of varying length, the generator non-terminals `StartTag` and `EndTag`, and tag `rdepth` allow testers to specify the depth at which an element text is nested. The `StartTag` generates one terminal containing the `<a>` start tag a number of repeated times specified by the testers. The `EndTag` behaves similarly. For example, setting the `StartTag` and `EndTag` parameters to 1 for the rule containing `document` as non-terminal on the left-hand side, setting the `StartTag` and `EndTag` parameters to 2 for the rule containing `element` as non-terminal on the left-hand side, and `rdepth` to 2 generates XML documents of depth 1, 3 and 5.

The XPath expression `//text()`, shown in the last line of this grammar retrieves the element text of all nodes which are below the root node. Because the grammar creates XML documents with only one element text, one element text is expected to be returned when applying the XPath expression on each XML document generated.

The bottom panel of Figure 7.6 shows the six test cases generated from the setup discussed in the three previous paragraphs. One test case appears on each line, showing the XML document first, followed by the XPath expression.

The second grammar, shown in the top left panel of Figure 7.7, allows testers to generate XML documents of varying length. Specifically, this grammar was developed assuming the tester is interested in generating documents with a varying number of children, where each child element name is identical or distinct and each child element text consists of a number of characters specified by the tester. In this figure, the `Elements` generator non-terminal text has been clicked, bringing a dialog box showing the parameters that the tester may modify. The first three parameters, *start number of children*, *increment*, and *count*, allows the tester to generate XML documents of varying length. Here, setting *start number of children* to 1, *increment* to 1, and *count* to 5 allows 5 documents to be generated with a length of 1 to 5. The parameter *identical children name* allows the tester to specify if children element names are identical or distinct. The parameter *string length of text elements* allows the tester to specify the number of characters for the text elements of the children. In this example, the *output* panel in Figure 7.7 shows 5 documents generated with a length of 1 to 5, where the element name of each children is distinct and the text

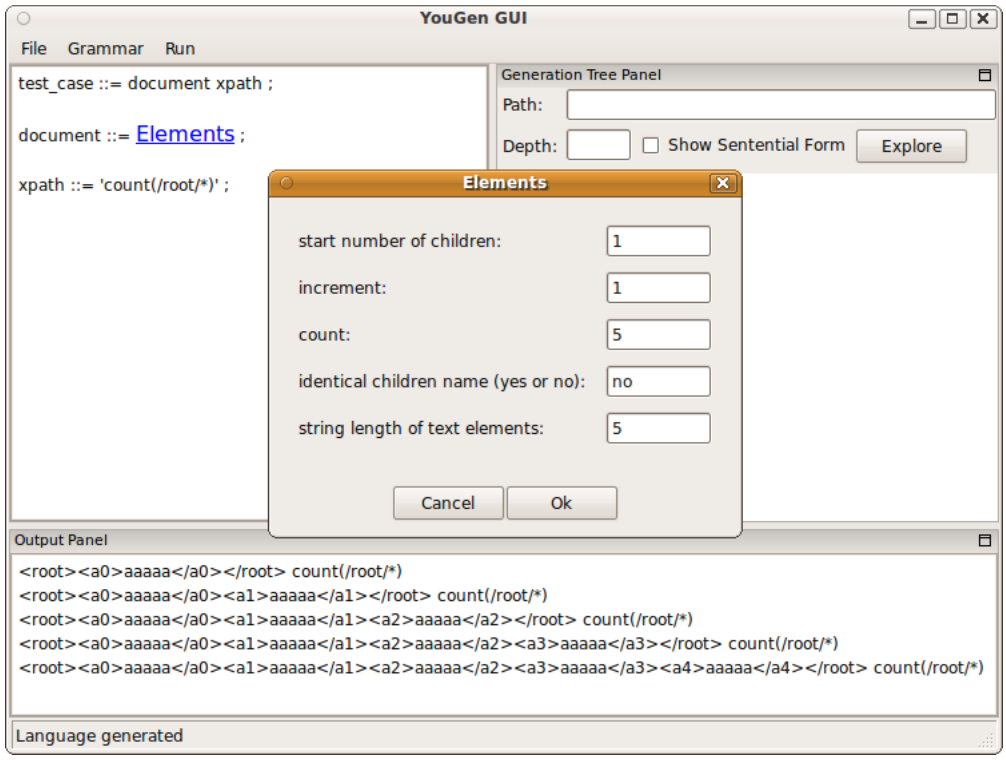

Figure 7.7: Dervish screenshot of second grammar in action

element consists of 5 characters.

The XPath expression `count(/root/*)`, shown in the last line of this grammar, is responsible for returning the number of children present in the XML document.

7.6 Test Execution

In this section, the focus is on explaining the embedded code for executing the test cases on the four interpreters. Figure 7.8 shows the text grammar and embedded code for generating XML documents with text elements of large length and of variable depth. For clarity, the embedded code for creating the generator non-terminals `StartTag`, `EndTag`, and `ElementText` is omitted. In this figure, we explain the three blocks of embedded code: `global_precode` (line 1 to 5), `postcode` (line 7 to 26), and `postcode` (line 31 to 34). The `global_precode` (line 1 to 5) is responsible for importing the `lib` module, which contains helper functions needed to execute the tests. The `global_precode` also declares two variables: `test_id` and `expected_output`. The

`postcode` (line 31 to 34) is responsible for assigning the expected output of each test case. For each test case, which consists of an XML document and an XPath expression, the `postcode` (line 7 to 26) is executed. First, the XML document and XPath expression generated from the grammar are assigned to variables `xml` and `xpath` (line 10 and 11). Next, a filename for the XML document and XPath expression is assigned to variables `xml_filename` and `xpath_filename` (line 13 and 14). Then, the helper function `WriteFile()` from the module `lib` (line 16 and 17) is invoked to write the XML document and the XPath expression to disk. Following this, each interpreter is run with the XML document and XPath expression (line 19 to 24). The function `getInterpreters()` from the `lib` module (line 19) yields the interpreter name and run command of each interpreter. The `program` variable contains the name of the interpreter and `command` contains the command for running the interpreter. The function `runTest()` from the `lib` module (line 20) executes the interpreter command with the XML document and XPath filenames. The actual output is captured and compared with the expected output (line 22). If the actual ouutput and expected output do not match, an error message is displayed in the *output* panel of Dervish specifying the test id and the interpreter on which the test case failed (line 23 and 24).

7.7 Test Results

The test results show that each one of the XPath interpreters has resource limitations. First, when the `element_text` in Figure 7.3 is replaced with a string of $94,332,507$ characters or more, VTD-XML fails to return the expected result and throws the exception `java.lang.OutOfMemoryError`. As opposed to VTD-XML, this test case does not reveal failures for Libxml, Lxml, and PyXML. XML documents with an element text of up to one hundred million characters were used as test cases and did not reveal failures for Libxml, Lxml, and PyXML. Second, when an XML document contains an element text nested at a depth greater or equal than 256, VTD-XML fails to parse the XML document and throws the exception `java.lang.OutOfMemoryError`. PyXML fails to return the expected result on XML

```
1  {global_precode
2  import lib
3  test_id = 1
4  expected_output = ''
5  }
6
7  {postcode
8       global test_id,expected_output
9
10      xml = flatten(s[0])
11      xpath = flatten(s[1])
12
13      xml_filename = str(test_id) + '_xml'
14      xpath_filename = str(test_id) + '_xpath'
15
16      lib.writeFile(xml_filename,xml)
17      lib.writeFile(xpath_filename,doc)
18
19      for program,command in lib.getInterpreters():
20              actual_output = lib.runTest(command,xml_filename,xpath_filename)
21
22              if actual_output.strip() != expected_output.strip():
23                      print 'test case ' + str(test_id) + ' failed: ' + \
24                       ' interpreter: ' + program
25      test_id += 1
26 }
27 test_case ::= document xpath;
28
29 document ::= StartTag(1) element EndTag(1);
30
31 {postcode
32      global expected_output
33      expected_output = flatten(s)
34 }
35 element ::= ElementText(1,1,5);
36
37 {rdepth 0}
38 element ::= StartTag(1) element EndTag(1);
39
40 xpath ::= '//text()';
```

Figure 7.8: Text of grammar generating element text of large length and at large depth in the XML document

documents with a depth greater or equal than 993. LibXML and Lxml fail to parse an XML document with a depth greater than $1025 = 2^{10} + 1$. Finally, failures in two of the interpreters were found for XML documents which contain a large number of children and where each child element text length is large. Specifically, when the `<root>` node in Figure 7.5 consists of $47,134$ or more children and each child element text is set to $10,000$ characters, VTD-XML fails to parse the XML document. When the number of children is greater than $68,800$ and each child element text consists of $10,000$ characters, PyXML also fails to parse the XML document. In contrast, no failures were revealed on Libxml2 and Lxml for these two test cases. However, XML documents consisted of up to one hundred thousand children.

7.8 Discussion

The test results show that the XPath interpreters have resource limitations such as the number of characters of an element text, the depth at which an element text may be nested, and the number of children that an element can have. What is more important is that this case study shows that Dervish can be used as a test bench to test XPath interpreters. First, the testers can manipulate the grammar tags and generator non-terminals to create the test cases which may help to reveal failures in one of the interpreters. Allowing the testers to manipulate grammar tags and generator non-terminals is important because it allows the tests to be focused on areas where the testers believe failures may occur. Therefore, this may help to reduce the number of tests needed to find failures in the code under test. Also, this case study shows that testers may use GBTG as an approach to automate the testing process. For instance, little manual effort is required by the testers to generate and execute test cases other than specifying the parameters to the tags and generator non-terminals and selecting the *Run* option from the menu bar. Finally, this case study shows that separating the tasks of grammar development and test control and execution is practical. For instance, the grammar developers may create grammars to be used and manipulated by testers.

Chapter 8
Related Work

This chapter reviews the work related to GBTG. We first present an overview of the domains in which GBTG has been applied. Then, we provide a summary of the earliest approaches used in developing test generators. Finally, we discuss the usability of GBTG tools.

8.1 Application domains of GBTG

GBTG has been used for more than 30 years in many application domains. The first domain in which GBTG has been used is compiler testing. In 1970, Hanford used a grammar to generate test cases for a PL/1 compiler [6]. His work is the earliest known application of context-free grammars to testing. Hanford inspired other researchers, such as Bird et al. [1], who later also applied GBTG to test compiler programs, graphical output applications and sort/merge programs. A major improvement to the work of Hanford is that they provided a mechanism to determine whether a test case has passed or failed by predicting its execution and comparing its execution with the predicted one at run time.

Years later, Sirer et al. [22] applied GBTG to the testing of the Java Virtual Machine implementations. Their work describe *lava*, a language for specifying grammars which can be used to construct complex test cases for testing JVMs. As a result of their work, faults were found in the Sun JDK1.0.2 and Microsoft Java virtual machine implementations.

Other domains in which GBTG has been used is firewall testing [10, 23] and XML applications [28].

Dervish is based on YouGen [23], a tool that combines GBTG and covering arrays. YouGen has been used in significant projects for the testing of firewalls [23], the generation of large XML documents and corresponding XPath queries [25], and the

testing of RSS clients for HMTL injection vulnerabilities [9]. More recently, YouGen has been applied to the testing of XML and XPath applications [17].

8.2 Approaches in developing test generators

Many authors have contributed to the development of GBTG tools and techniques. In this section, we summarize twelve important features of test generators which have been proposed or implemented in earlier work. We provide a short description of each feature and identify which YouGen feature corresponds to it, if applicable.

- *Counts:* restricts the use of a rule it is attached to, to the count specified. This feature was first developed by Hanford [6] and later used by Murali et al. [19]. *Counts* are also implemented in YouGen.

- *Weights:* serves to prioritize the selection of rule alternatives by attaching weights to the rules. This approach has been used in Hanford [6], Payne [20], and Bird et al. [1]. Weights are not part of the YouGen implementation.

- *Guards:* are boolean expressions which allow a rule to be selected for expansion if the expression evaluates to true. If the expression evaluates to false, an alternative rule is selected for expansion. It was first introduced by Duncan et al. [4] and was also found in the work of Homer et al. [11]. In YouGen, `precode` tags can be used as *guards*.

- *Actions:* are code fragments associated with a rule in the grammar. This feature is first found in Celentano et al. [2] and later in Duncan et al. [4]. In YouGen, the `precode` and `postcode` tags plays the role of *actions*.

- *Recursion limit:* restricts the number of times a recursive rule can be selected. This feature has been widely used and is found in the work of Hanford [6], Celentano et al. [2], Bird et al. [1]. In YouGen, the `rdepth` tag serves the same purpose as the *Recursion limit*.

- *Depth:* limits the traversal of a generation tree to a certain depth. It appears to have been introduced only recently in the work of Zaytsev [28] and also Lammel

et al. [15]. This feature is implemented in YouGen and is represented by the `depth` tag.

- *Balance control:* limits the variation in the depth of the non-terminals from a rule's right-hand side to which it is attached to, to the limit specified. Zaytsev [28] and Lammel et al. [15] have used this feature in their work. Balance controls are not implemented in YouGen.

- *Dependence control:* generates strings in the language based on combinations of the non-terminals and terminals appearing on the right hand side of a rule. It is also Zaytsev [28] and Lammel et al. [15] who introduced this feature. *Dependence control* are implemented in YouGen with the `cov` tag.

- *Terminal constructors:* are utility features used when a long sequence of terminal alternatives exist for a rule. Examples of this feature are found in Homer et al. [11] and Maurer [18]. *Terminal constructors* are implemented in YouGen. They are called generator non-terminals and always appear on the right-hand side of a rule.

- *Random traversal:* is a traversal order in which rules are randomly selected. This is by far the most used approach to GBTG and is used by Hanford [6], Bird et al. [1], Murali et al. [19], Homer et al. [11], Maurer [18]. Random traversal order is not part of the YouGen generation strategy.

- *Breadth-first traversal:* is a traversal order in which strings of the language are generated in the order of increasing depth. Zaytsev [28] and Lammel et al. [15] used this traversal approach in their work. Breadth-first traversal in not part of the YouGen generation strategy.

- *Dynamic grammars:* allows rules to be added to the grammar during generation to ensure contextual sensitivity of the generated tests. Hanford [6] appears to be first one to introduce this feature. Dynamic grammars are supported in YouGen and are implemented using `postcode` tags in the text grammar.

8.3 Usability of GBTG tools

Except for GUI capture and playback test tools, tools to automate the testing effort typically requires programming skills. Recently, work has been done to provide GBTG tools to testing practioners [24]. In their work, the authors present a YouGen controller which is a graphical user interface allowing testers to modify parts of grammars. Their work has shown that GBTG testing can be conducted without testers having strong programming skills. However, their tool does not take advantages of generation trees nor does it allows testers to invoke YouGen through the GUI to generate test cases. Therefore, our work extends the work on the YouGen Controller, and attempts, through a GUI, to provide ways to use generation trees and to allow testers to control the execution of test cases.

Chapter 9
Conclusions and Future Work

Although software testing is known to be a time-intensive task, it has had considerable impact on the success of many projects. Nowadays, organizations are faced with the pressure of providing quality software products within tight time constraints. To accomplish this goal, a practical solution is test automation. Test automation typically requires programming skills. Testers, who usually know a lot about the application under test, often do not have the programming skills to automate the testing effort. Their role is crucial, however, since they know many test cases likely to reveal errors in the application under test. Grammar-based test generation (GBTG) is an approach to test automation. GBTG can help testers create grammars to generate the test cases needed by the testers. However, creating grammars requires programming skills.

In this work, we present Dervish, a graphical user interface which provides the testers with the power to use GBTG tools without the need of programming skills. The results of this work indicate that Dervish can effectively provide testers with the support to use and learn GBTG. For instance, Dervish allows testers to modify parts of a grammar, to generate test cases, and to visualize generation trees. Since viewing generation trees can facilitate a tester's understanding of the language generated by a grammar, Dervish can also help testers understand complex grammars.

We have conducted three case studies to show the benefits of Dervish. The first case study showed how Dervish supports the learning of grammar-based test generation and how Dervish helps testers use GBTG tools such as YouGen. The second case study showed that Dervish can be used to help testers understand complex grammars. In this case study, we showed that generation tree exploration can effectively help testers understand how test cases are generated. Finally, in the third case study we showed that Dervish can be used as a test bench to test XPath interpreters. We

showed that through Dervish, testers can guide the testing effort and generate test cases to reveal errors in the interpreters under test.

9.1 Future Work

The results in this thesis form the basis for understanding how testers can use GBTG tools in their testing efforts without programming skills. Looking ahead, the results in this thesis also provide a foundation for future work.

9.1.1 Save Functionality

As future work, one could consider adding a functionality to save grammar files. This would allow the state of modified grammar files to be preserved and be reloaded in Dervish at a later moment.

9.1.2 Test Additional Applications

A large number of XML parsers, XML document type interpreters and XPath implementations exists in the industry and in the open-source community. Therefore, the testing effort could focus on using Dervish for testing more applications in this area.

9.1.3 Educational Use

Dervish could be used in an academic environment. For example, Dervish could be used as a learning tool support in a course teaching grammar-based test generation. Students could interact with Dervish to view and explore generation trees or modify grammar tag and generator non-terminal parameters. Interacting with Dervish could help students in understanding how the language of a grammar is generated. Also, Dervish could help students learn by examples since the GUI allows students to create different grammars by modifying tag and generator non-terminal parameters.

Bibliography

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

[2] A. Celentano, S. Crespi-Reghizzi, P. Vigna, C. Ghezzi, G. Granata, and F. Savoretti. Compiler testing using a sentence generator. *Software - Practice and Experience*, 10(11):897–918, 1980.

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

[4] A.G. Duncan and J.S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings 5th International Conference on Software Engineering*, pages 170–178. IEEE Press, 1981.

[5] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley, 1999.

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

[7] E.R. Harold and W.S. Means. *XML in a Nutshell*. O'Reilly & Associates, Inc., third edition, 2004.

[8] D. Hoffman, D. Ly-Gagnon, P. Strooper, and H. Wang. Grammar-based test generation with YouGen. 2010. Submitted to Software - Practice and Experience.

[9] D. Hoffman, H. Wang, M. Chang, and D. Ly-Gagnon. Grammar based testing of HTML injection vulnerabilities in RSS feeds. In *Proceedings of TAIC-PART 2009*, pages 105–110. IEEE Computer Society, 2009.

[10] D. Hoffman and K. Yoo. Blowtorch: a framework for firewall test automation. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 96–103, 2005.

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

[12] C.A. Jones and F.L. Drake. *Python and XML*. O'Reilly & Associates, Inc., 2001.

[13] R. Kaksonen. *A Functional Method for Assessing Protocol Implementation Security*. Technical Research Centre of Finland, 2001.

[14] M. Kay. *XPath 2.0 Programmer's Reference*. Wiley Publishing, Inc., 2004.

[15] 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.

[16] Lxml. a pythonic binding for the libxml2 and libxslt libraries. `http://codespeak.net/lxml/`. Last accessed, February 2010.

[17] C. MacNamara. Grammar-based testing of XML and XPath applications. BE Thesis, The University of Queensland, 2009.

[18] P.M. Maurer. The design and implementation of a grammar-based data generator. *Software - Practice and Experience*, 22(3):223–244, 1992.

[19] V. Murali and R.K. Shyamasundar. Sentence generator for a compiler for PT, a Pascal subset. *Software - Practice and Experience*, 13(9):857–869, 1983.

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

[21] PyXML. Python and XML processing. `http://sourceforge.net/projects/pyxml/`. Last accessed, February 2010.

[22] E.G. Sirer and B.N. Bershad. Using production grammars in software testing. In *2nd Conference on Domain-specific Languages*, pages 1–13. ACM Press, 1999.

[23] L. Sobotkiewicz. A new tool for grammar-based test case generation. Master's thesis, Univ. of Victoria, December 2008.

[24] J. K. Son. YouGen controller. Master's thesis, Univ. of Victoria, 2008.

[25] H. Wang. Grammar-based test generation for XPath queries. Master's thesis, Univ. of Victoria, 2008.

[26] XimpleWare. VTD-XML: The future of XML processing. `http://vtd-xml.sourceforge.net`. Last accessed, February 2010.

[27] XMLSoft. The XML C parser and toolkit of Gnome: Introduction. `http://www.xmlsoft.org/intro.html`. Last accessed, February 2010.

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